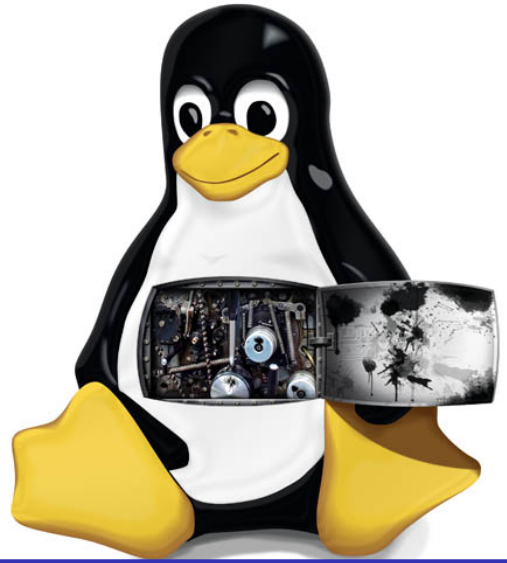


GPOS General Purpose Operating Systems

Linux Kernel

Alejandro Furfaro

25 de agosto de 2021

Temario

Toda una frase. . .

“El Número de instalaciones de UNIX ha llegado a 10, y se espera que este número aumente.”

UNIX Programmer’s Manual, 2nd. Edition. Junio 1972.

Bell Telephone Laboratories . . .

Bell Telephone Laboratories . . .

1965 Desarrollo de Multics junto con General Electric y el grupo MAC del MIT. Objetivo: Obtener un SO capaz de proveer acceso a una gran cantidad de usuarios concurrentes a un mismo computador.

Bell Telephone Laboratories . . .

1965 Desarrollo de Multics junto con General Electric y el grupo MAC del MIT. Objetivo: Obtener un SO capaz de proveer acceso a una gran cantidad de usuarios concurrentes a un mismo computador.

1969 Primer versión de Multics en un computador GE 645. BTL deja el proyecto por no alcanzar los objetivos propuestos

Bell Telephone Laboratories . . .

1965 Desarrollo de Multics junto con General Electric y el grupo MAC del MIT. Objetivo: Obtener un SO capaz de proveer acceso a una gran cantidad de usuarios concurrentes a un mismo computador.

1969 Primer versión de Multics en un computador GE 645. BTL deja el proyecto por no alcanzar los objetivos propuestos

1969 Ken Thompson y Dennis Ritchie trabajaban en la GE 645 en un juego “Space Travel”, una suerte de Arcade manejado desde teclado. Resultaba costoso en ciclos de computación y muy complejo de manejar.

Bell Telephone Laboratories . . .

- 1965 Desarrollo de Multics junto con General Electric y el grupo MAC del MIT. Objetivo: Obtener un SO capaz de proveer acceso a una gran cantidad de usuarios concurrentes a un mismo computador.
- 1969 Primer versión de Multics en un computador GE 645. BTL deja el proyecto por no alcanzar los objetivos propuestos
- 1969 Ken Thompson y Dennis Ritchie trabajaban en la GE 645 en un juego “Space Travel”, una suerte de Arcade manejado desde teclado. Resultaba costoso en ciclos de computación y muy complejo de manejar.
- 1969 Thompson encuentra una vieja PDP-7 de Digital, casi sin utilizar. Su capacidad gráfica y bajo costo computacional los motiva a escribir allí “Space Travel”. Requería cross compilar el código. La entrada utilizaba cinta de papel.

Bell Telephone Laboratories . . .

1969 Thompson y Ritchie, describen en un paper el diseño de un nuevo file system. Thompson lo implementa, junto con un kernel básico basado en un esquema de demanda de memoria por páginas, en la GE 645.

Bell Telephone Laboratories . . .

1969 Thompson y Ritchie, describen en un paper el diseño de un nuevo file system. Thompson lo implementa, junto con un kernel básico basado en un esquema de demanda de memoria por páginas, en la GE 645.

1969 Thompson y Ritchie re escriben el juego, con un kernel, el file system publicado, y un pequeño administrador de procesos. Objetivo: arrancar “Space Travel ”en forma automática.

Bell Telephone Laboratories . . .

1969 Thompson y Ritchie, describen en un paper el diseño de un nuevo file system. Thompson lo implementa, junto con un kernel básico basado en un esquema de demanda de memoria por páginas, en la GE 645.

1969 Thompson y Ritchie re escriben el juego, con un kernel, el file system publicado, y un pequeño administrador de procesos. Objetivo: arrancar “Space Travel ”en forma automática.

1969 Agregan algunas líneas mas de assembler. Portan el conjuto a la PDP-7, y concluyen que tienen aquel SO tan buscado. Lo llaman UNIX. Sencillo pero muy potente. Sus ideas innovadoras sentarían las bases de los Sistemas Operativos hasta hoy.

Bell Telephone Laboratories . . .

1969 Thompson y Ritchie, describen en un paper el diseño de un nuevo file system. Thompson lo implementa, junto con un kernel básico basado en un esquema de demanda de memoria por páginas, en la GE 645.

1969 Thompson y Ritchie re escriben el juego, con un kernel, el file system publicado, y un pequeño administrador de procesos. Objetivo: arrancar “Space Travel ”en forma automática.

1969 Agregan algunas líneas mas de assembler. Portan el conjuto a la PDP-7, y concluyen que tienen aquel SO tan buscado. Lo llaman UNIX. Sencillo pero muy potente. Sus ideas innovadoras sentarían las bases de los Sistemas Operativos hasta hoy.

1969 Presentación en BTL. “¿Porque en la PDP-7?, ¡Portenlo a la nueva PDP-11!”. . . . Pequeño detalle: estaba escrito íntegramente en assembler . . .

Bell Telephone Laboratories . . .

1971 Port de UNIX a PDP-11. Usa 16 Kbyes de memoria. 8 Kbytes para los programas de usuario, y 512Kbytes de storage.

Bell Telephone Laboratories . . .

1971 Port de UNIX a PDP-11. Usa 16 Kbyes de memoria. 8 Kbytes para los programas de usuario, y 512Kbytes de storage.

1971 Ritchie retoma un proyecto de un lenguaje llamado B, basado en BCPL. Le define tipos de datos y estructuras, y escribe un compilador. Lo llama C. . .

Bell Telephone Laboratories . . .

- 1971 Port de UNIX a PDP-11. Usa 16 Kbyes de memoria. 8 Kbytes para los programas de usuario, y 512Kbytes de storage.
- 1971 Ritchie retoma un proyecto de un lenguaje llamado B, basado en BCPL. Le define tipos de datos y estructuras, y escribe un compilador. Lo llama C. . .
- 1973 Se presenta la primer versión de UNIX escrito en C.

Bell Telephone Laboratories . . .

- 1971 Port de UNIX a PDP-11. Usa 16 Kbyes de memoria. 8 Kbytes para los programas de usuario, y 512Kbytes de storage.
- 1971 Ritchie retoma un proyecto de un lenguaje llamado B, basado en BCPL. Le define tipos de datos y estructuras, y escribe un compilador. Lo llama C. . .
- 1973 Se presenta la primer versión de UNIX escrito en C.
- 1974 Ritchie y Thompson publican en ACM “The UNIX Time-Sharing System”.

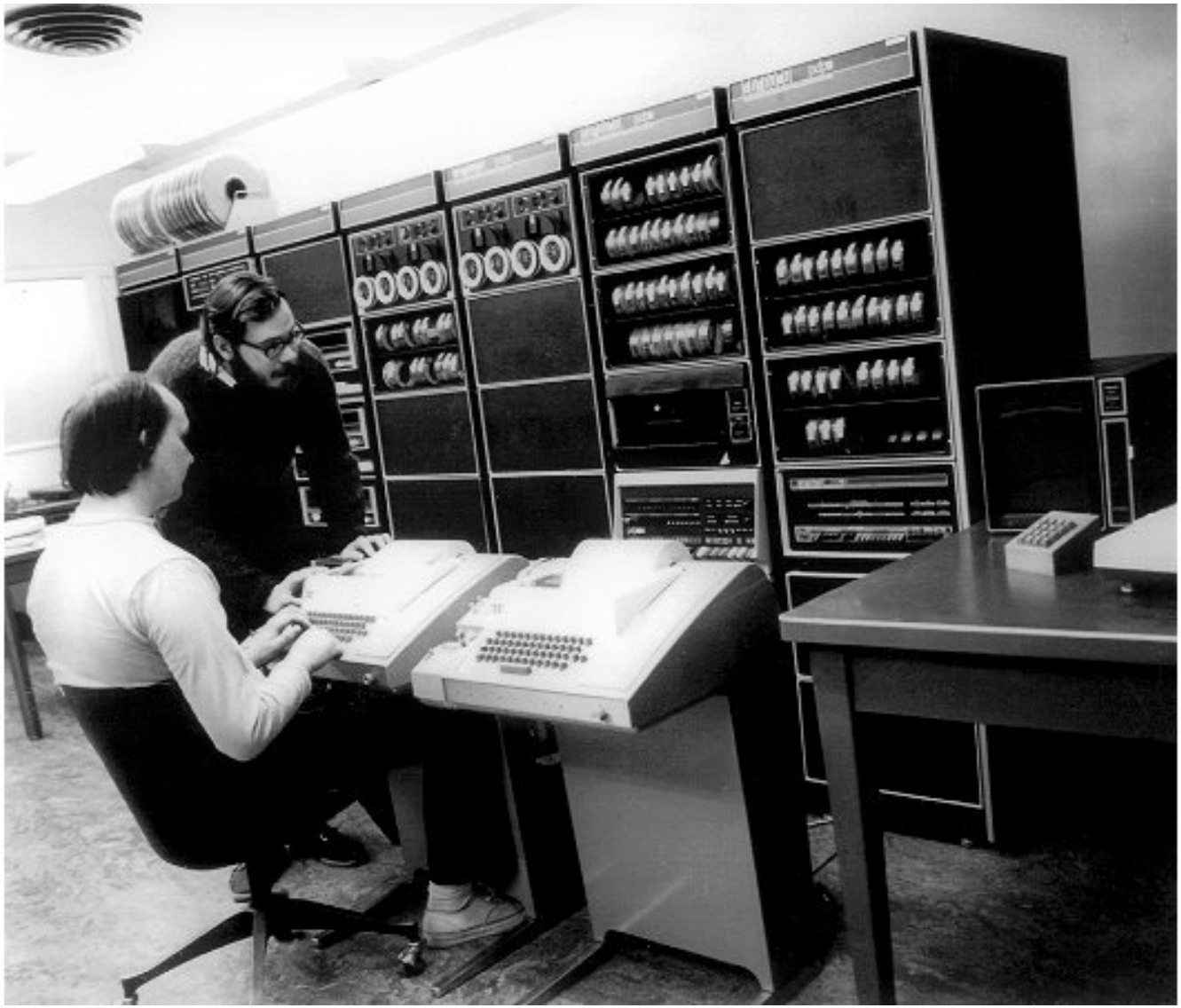
Bell Telephone Laboratories . . .

- 1971 Port de UNIX a PDP-11. Usa 16 Kbyes de memoria. 8 Kbytes para los programas de usuario, y 512Kbytes de storage.
- 1971 Ritchie retoma un proyecto de un lenguaje llamado B, basado en BCPL. Le define tipos de datos y estructuras, y escribe un compilador. Lo llama C. . .
- 1973 Se presenta la primer versión de UNIX escrito en C.
- 1974 Ritchie y Thompson publican en ACM “The UNIX Time-Sharing System”.
- 1974 Se abre el desarrollo de UNIX a Universidades, para impulsar su estudio en las carreras de grado y su uso en Investigación.

Bell Telephone Laboratories . . .

- 1971 Port de UNIX a PDP-11. Usa 16 Kbyes de memoria. 8 Kbytes para los programas de usuario, y 512Kbytes de storage.
- 1971 Ritchie retoma un proyecto de un lenguaje llamado B, basado en BCPL. Le define tipos de datos y estructuras, y escribe un compilador. Lo llama C. . .
- 1973 Se presenta la primer versión de UNIX escrito en C.
- 1974 Ritchie y Thompson publican en ACM “The UNIX Time-Sharing System”.
- 1974 Se abre el desarrollo de UNIX a Universidades, para impulsar su estudio en las carreras de grado y su uso en Investigación.
- 1977 500 instalaciones. 125 en Universidades, y 350 en empresas. Proliferan los sistemas abiertos para administración de procesos y automatización de oficinas. Ya hay versiones No-PDP.

Próceres. . .



Para que no queden dudas

Para que no queden dudas

1983 Turing Award “por su desarrollo teórico de Sistemas Operativos genéricos y específicamente por la implementación del sistema operativo UNIX ”.

Para que no queden dudas

1983 Turing Award “por su desarrollo teórico de Sistemas Operativos genéricos y específicamente por la implementación del sistema operativo UNIX ”.

1990 IEEE Richard W. Hamming Medal del IEEE, “por originar el Sistema Operativo UNIX y el Lenguaje de Programación C”.

Para que no queden dudas

1983 Turing Award “por su desarrollo teórico de Sistemas Operativos genéricos y específicamente por la implementación del sistema operativo UNIX ”.

1990 IEEE Richard W. Hamming Medal del IEEE, “por originar el Sistema Operativo UNIX y el Lenguaje de Programación C”.

1997 Fellows del Computer History Museum, “”por la co-creación del Sistema Operativo UNIX, y por el desarrollo del Lenguaje de Programación C”.

Para que no queden dudas

- 1983 Turing Award “por su desarrollo teórico de Sistemas Operativos genéricos y específicamente por la implementación del sistema operativo UNIX ”.
- 1990 IEEE Richard W. Hamming Medal del IEEE, “por originar el Sistema Operativo UNIX y el Lenguaje de Programación C”.
- 1997 Fellows del Computer History Museum, “”por la co-creación del Sistema Operativo UNIX, y por el desarrollo del Lenguaje de Programación C”.
- 1999 Medalla Nacional de Tecnología de 1998 de manos del Presidente de EEUU Bill Clinton “por co-inventar el Sistema Operativo UNIX y el lenguaje de programación C”.

Para que no queden dudas

2005 Industrial Research Institute Achievement Award en reconocimiento de su contribución a la ciencia y la tecnología y a la sociedad en general, con su desarrollo del Sistema Operativo UNIX. (D. Ritchie solo)

Para que no queden dudas

2005 Industrial Research Institute Achievement Award en reconocimiento de su contribución a la ciencia y la tecnología y a la sociedad en general, con su desarrollo del Sistema Operativo UNIX. (D. Ritchie solo)

2011 Japan Prize for Information and Communications por su trabajo en el desarrollo del Sistema Operativo UNIX.

Consolidando esfuerzos dispersos

Consolidando esfuerzos dispersos

- Entre 1977 y 1982, los Laboratorios Bell se abocaron a consolidar los esfuerzos de diferentes grupos de desarrollo en una única versión consolidada de UNIX, la cual se conoció comercialmente como System III.

Consolidando esfuerzos dispersos

- Entre 1977 y 1982, los Laboratorios Bell se abocaron a consolidar los esfuerzos de diferentes grupos de desarrollo en una única versión consolidada de UNIX, la cual se conoció comercialmente como System III.
- En poco tiempo se agregaron nuevas funcionalidades, entre otras cosas un novedoso y muy versatil conjunto de herramientas para Intercomunicación de Procesos (IPC’s), y se estabilizó la versión como System V.

Consolidando esfuerzos dispersos

- Entre 1977 y 1982, los Laboratorios Bell se abocaron a consolidar los esfuerzos de diferentes grupos de desarrollo en una única versión consolidada de UNIX, la cual se conoció comercialmente como System III.
- En poco tiempo se agregaron nuevas funcionalidades, entre otras cosas un novedoso y muy versatil conjunto de herramientas para Intercomunicación de Procesos (IPC’s), y se estabilizó la versión como System V.
- Enero de 1983: ATT anuncia oficialmente su soporte a System V.

Consolidando esfuerzos dispersos

- Entre 1977 y 1982, los Laboratorios Bell se abocaron a consolidar los esfuerzos de diferentes grupos de desarrollo en una única versión consolidada de UNIX, la cual se conoció comercialmente como System III.
- En poco tiempo se agregaron nuevas funcionalidades, entre otras cosas un novedoso y muy versatil conjunto de herramientas para Intercomunicación de Procesos (IPC’s), y se estabilizó la versión como System V.
- Enero de 1983: ATT anuncia oficialmente su soporte a System V.
- En paralelo al Universidad de Berkeley lanza su versión UNIX BSD 4.3. (**BSD** por **B**erkeley **S**oftware **D**itribution). Esta distribución de UNIX aun hoy sigue vigente (free BSD) y es un producto de libre distribución y muy sólido funcionamiento.

Las claves del éxito de UNIX

Las claves del éxito de UNIX

1. Escrito 99 % en un lenguaje específicamente diseñado para escribir Sistemas Operativos en reemplazo del assembler: C.

Las claves del éxito de UNIX

1. Escrito 99 % en un lenguaje específicamente diseñado para escribir Sistemas Operativos en reemplazo del assembler: C.
2. Portable entre diferentes plataformas de hardware debido a la portabilidad nativa del C.

Las claves del éxito de UNIX

1. Escrito 99 % en un lenguaje específicamente diseñado para escribir Sistemas Operativos en reemplazo del assembler: C.
2. Portable entre diferentes plataformas de hardware debido a la portabilidad nativa del C.
3. Su Arquitectura base de extrema simplicidad, lo hace poderoso, versátil e ideal para programadores.

Las claves del éxito de UNIX

1. Escrito 99 % en un lenguaje específicamente diseñado para escribir Sistemas Operativos en reemplazo del assembler: C.
2. Portable entre diferentes plataformas de hardware debido a la portabilidad nativa del C.
3. Su Arquitectura base de extrema simplicidad, lo hace poderoso, versátil e ideal para programadores.
4. Provee primitivas muy potentes que permiten construir programas complejos a partir de código relativamente simple.

Las claves del éxito de UNIX

1. Escrito 99 % en un lenguaje específicamente diseñado para escribir Sistemas Operativos en reemplazo del assembler: C.
2. Portable entre diferentes plataformas de hardware debido a la portabilidad nativa del C.
3. Su Arquitectura base de extrema simplicidad, lo hace poderoso, versátil e ideal para programadores.
4. Provee primitivas muy potentes que permiten construir programas complejos a partir de código relativamente simple.
5. Introduce el concepto de stream de bytes que permite tratar entidades de diferente origen y características bajo una simple interfaz común. Se conoce como el paradigma “everything is a file”.

Las claves del éxito de UNIX

6. Provee una visión de máquina transparente a los detalles de hardware.

Las claves del éxito de UNIX

6. Provee una visión de máquina transparente a los detalles de hardware.
7. Provee un entorno de programación y ejecución multitarea y multiusuario, de bajo costo computacional y alta estabilidad y seguridad.

Las claves del éxito de UNIX

6. Provee una visión de máquina transparente a los detalles de hardware.
7. Provee un entorno de programación y ejecución multitarea y multiusuario, de bajo costo computacional y alta estabilidad y seguridad.
8. La potencia de cada programa no se debe al programa en sí mismo, sino a su relación con el resto de los programas alojados en el file system

Las claves del éxito de UNIX

6. Provee una visión de máquina transparente a los detalles de hardware.
7. Provee un entorno de programación y ejecución multitarea y multiusuario, de bajo costo computacional y alta estabilidad y seguridad.
8. La potencia de cada programa no se debe al programa en sí mismo, sino a su relación con el resto de los programas alojados en el file system
9. Junto con las herramientas de programación y utilitarios que trae junto con el sistema, y al agregarse como protocolo de conectividad standard la suite TCP/IP, un System V o BSD 4.3 forman un ecosistema de desarrollo de muy alta productividad cuando se lo sabe utilizar.

Estandarizar o vivir integrando esfuerzos dispersos

Estandarizar o vivir integrando esfuerzos dispersos

- System III y System V dejaron una clara enseñanza: Cuando un sistema se hace popular, y sus fuentes están disponibles, todo el mundo va a querer mejorarlo, potenciarlo y adaptarlo.

Estandarizar o vivir integrando esfuerzos dispersos

- System III y System V dejaron una clara enseñanza: Cuando un sistema se hace popular, y sus fuentes están disponibles, todo el mundo va a querer mejorarlo, potenciarlo y adaptarlo.
- Las necesidades y objetivos de cada grupo de desarrolladores nunca es la misma.

Estandarizar o vivir integrando esfuerzos dispersos

- System III y System V dejaron una clara enseñanza: Cuando un sistema se hace popular, y sus fuentes están disponibles, todo el mundo va a querer mejorarlo, potenciarlo y adaptarlo.
- Las necesidades y objetivos de cada grupo de desarrolladores nunca es la misma.
- Rápidamente aparecieron versiones diferentes de UNIX.

Estandarizar o vivir integrando esfuerzos dispersos

- System III y System V dejaron una clara enseñanza: Cuando un sistema se hace popular, y sus fuentes están disponibles, todo el mundo va a querer mejorarlo, potenciarlo y adaptarlo.
- Las necesidades y objetivos de cada grupo de desarrolladores nunca es la misma.
- Rápidamente aparecieron versiones diferentes de UNIX.
- Si no se ordena, hay que trabajar como lo hizo ATT en consolidar esfuerzos bajo el nombre System III, pero de manera permanente

Estandarizar o vivir integrando esfuerzos dispersos

- System III y System V dejaron una clara enseñanza: Cuando un sistema se hace popular, y sus fuentes están disponibles, todo el mundo va a querer mejorarlo, potenciarlo y adaptarlo.
- Las necesidades y objetivos de cada grupo de desarrolladores nunca es la misma.
- Rápidamente aparecieron versiones diferentes de UNIX.
- Si no se ordena, hay que trabajar como lo hizo ATT en consolidar esfuerzos bajo el nombre System III, pero de manera permanente
- La otra opción es escribir un estandard y crear un comité científico profesional que vele por su actualización y cumplimiento.

Estandarizar o vivir integrando esfuerzos dispersos

- System III y System V dejaron una clara enseñanza: Cuando un sistema se hace popular, y sus fuentes están disponibles, todo el mundo va a querer mejorarlo, potenciarlo y adaptarlo.
- Las necesidades y objetivos de cada grupo de desarrolladores nunca es la misma.
- Rápidamente aparecieron versiones diferentes de UNIX.
- Si no se ordena, hay que trabajar como lo hizo ATT en consolidar esfuerzos bajo el nombre System III, pero de manera permanente
- La otra opción es escribir un estandard y crear un comité científico profesional que vele por su actualización y cumplimiento.
- Así nació **P**ortable **O**perating **S**ystem **I**nterface: **POSIX**

Estandarizar o vivir integrando esfuerzos dispersos

- System III y System V dejaron una clara enseñanza: Cuando un sistema se hace popular, y sus fuentes están disponibles, todo el mundo va a querer mejorarlo, potenciarlo y adaptarlo.
- Las necesidades y objetivos de cada grupo de desarrolladores nunca es la misma.
- Rápidamente aparecieron versiones diferentes de UNIX.
- Si no se ordena, hay que trabajar como lo hizo ATT en consolidar esfuerzos bajo el nombre System III, pero de manera permanente
- La otra opción es escribir un estandard y crear un comité científico profesional que vele por su actualización y cumplimiento.
- Así nació **P**ortable **O**perating **S**ystem **I**nterface: **POSIX**
- ¿Y la **X** del final de donde viene?. . .

Estandarizar o vivir integrando esfuerzos dispersos

- System III y System V dejaron una clara enseñanza: Cuando un sistema se hace popular, y sus fuentes están disponibles, todo el mundo va a querer mejorarlo, potenciarlo y adaptarlo.
- Las necesidades y objetivos de cada grupo de desarrolladores nunca es la misma.
- Rápidamente aparecieron versiones diferentes de UNIX.
- Si no se ordena, hay que trabajar como lo hizo ATT en consolidar esfuerzos bajo el nombre System III, pero de manera permanente
- La otra opción es escribir un estandard y crear un comité científico profesional que vele por su actualización y cumplimiento.
- Así nació **P**ortable **O**perating **S**ystem **I**nterface: **POSIX**
- ¿Y la **X** del final de donde viene?. . .
- . . . y . . . UNI**X**

Inicios de POSIX

Inicios de POSIX

- POSIX fue inicialmente abordado en el Portable Applications Standards Committee de la Computer Society del IEEE

Inicios de POSIX

- POSIX fue inicialmente abordado en el Portable Applications Standards Committee de la Computer Society del IEEE
- Se propuso estandarizar no solo el API sino también comandos del shell, scripting language, y hasta ciertos programas como awk, make etc.

Inicios de POSIX

- POSIX fue inicialmente abordado en el Portable Applications Standards Committee de la Computer Society del IEEE
- Se propuso estandarizar no solo el API sino también comandos del shell, scripting language, y hasta ciertos programas como awk, make etc.
- A posteriori se sumó OpenGroup.

Inicios de POSIX

- POSIX fue inicialmente abordado en el Portable Applications Standards Committee de la Computer Society del IEEE
- Se propuso estandarizar no solo el API sino también comandos del shell, scripting language, y hasta ciertos programas como awk, make etc.
- A posteriori se sumó OpenGroup.
- De este modo un sistema no necesariamente debe llamarse UNIX para ser POSIX.

Inicios de POSIX

- POSIX fue inicialmente abordado en el Portable Applications Standards Committee de la Computer Society del IEEE
- Se propuso estandarizar no solo el API sino también comandos del shell, scripting language, y hasta ciertos programas como awk, make etc.
- A posteriori se sumó OpenGroup.
- De este modo un sistema no necesariamente debe llamarse UNIX para ser POSIX.
- Simplemente, debe respetar este stantadrd.

Inicios de POSIX

- POSIX fue inicialmente abordado en el Portable Applications Standards Committee de la Computer Society del IEEE
- Se propuso estandarizar no solo el API sino también comandos del shell, scripting language, y hasta ciertos programas como awk, make etc.
- A posteriori se sumó OpenGroup.
- De este modo un sistema no necesariamente debe llamarse UNIX para ser POSIX.
- Simplemente, debe respetar este stantadrd.
- Con el tiempo y la vigencia y solidez creciente de UNIX muchos fabricantes se apegaron al estandard, incluidos algunos Sistemas Operativos como QNX que inicialmente fueron propietarios, pero que al pasar a POSIX incrementaron su nivel de uso.

Sistemas Fully POSIX-compliant

- A/UX
- AIX
- BSD/OS
- DSPnano
- HP/UX
- INTEGRUTY
- IRIX
- LinxOS
- MPE/IX
- OS X Mavericks
- QNX
- RTEMS (POSIX 1003.13-2003 Profile 52)
- Solaris
- True64
- Unison RTOS
- unixWare

Sistemas Mostly POSIX-compliant

Sistemas POSIX no certificados:

- Be OS (y luego Haiku)
- FreeBSD
- Contiki
- Darwin (Core de OS X e iOS)
- illumos
- Linux
- MINIX
- NetBSD
- Nucleous RTOS
- OpenBSD
- OpenSolaris
- PikeOS (RTOS for embedded systems with optional PSE51 and PSE52 partitions)
- RTEMS (API basado en IEEE Std. 1003.13-2003 PSE52)
- Sanos
- SkyOS
- Syllable
- VSTa
- VxWorks
- NuttX

Primeros años

Hasta 1997 POSIX comprendía varios estándares

POSIX.1 (IEEE Std 1003.1-1988). Define Core Services (incorpora el Estandard ANSI C)

- Creación y Control de Procesos
- Señales
- Excepciones de Punto Flotante
- Violaciones de Segmento / Memoria
- Instrucciones Ilegales
- Errores de Bus
- Timers
- Operaciones de Archivos y Directorios
- Pipes
- Biblioteca C (Standard C)
- Interfaz y Control de Puertos de E/S
- Disparo de Procesos

Primeros años

POSIX.1b Real-time extensions (IEEE Std 1003.1b-1993)

- Scheduling con Prioridad
- Señales Real-Time
- Clocks y Timers
- Semáforos
- Pasaje de Mensajes
- Shared Memory
- E/S Asincrónica y Sincrónica
- Interfaz de Locking de Memoria

Primeros años

POSIX.1c Threads extensions (IEEE Std 1003.1c-1995)

- Creación, Control, y Cleanup de Threads
- Scheduling de Thread
- Sincronización de Threads
- Manejo de Señales

POSIX.2 Shell y Utilidades (IEEE Std 1003.2-1992)

- Interprete de Comandos
- Programas Utilitarios

Revisiones

Luego de 1997 se hicieron revisiones a cargo de Austin Group, bajo el nombre genérico de Single UNIX Specification. Luego de ser formalmente aprobadas por ISO volvieron a ser un estándard POSIX

POSIX.1-2001 o IEEE Std 1003.1-2001 equates to the Single UNIX Specification version 3

- Base Definitions
- System Interfaces y Headers
- Comandos y Utilidades

POSIX.1-2004 a.k.a. IEEE Std 1003.1-2004. Es una actualización menor del POSIX.1-2001. Solo dos correcciones técnicas

POSIX.1-2008 a.k.a. IEEE Std 1003.1-2008. Es la versión actual.

Portabilidad

Portabilidad

- POSIX deriva de UNIX que fue concebido bajo el concepto de portabilidad

Portabilidad

- POSIX deriva de UNIX que fue concebido bajo el concepto de portabilidad
- Este atributo permite desarrollar aplicaciones que pueden correr en plataformas de hardware heterogénea sin necesidad de ser reescritas.

Portabilidad

- POSIX deriva de UNIX que fue concebido bajo el concepto de portabilidad
- Este atributo permite desarrollar aplicaciones que pueden correr en plataformas de hardware heterogénea sin necesidad de ser reescritas.
- Solo se requiere compilar en cada plataforma los mismos fuentes.

Portabilidad

- POSIX deriva de UNIX que fue concebido bajo el concepto de portabilidad
- Este atributo permite desarrollar aplicaciones que pueden correr en plataformas de hardware heterogénea sin necesidad de ser reescritas.
- Solo se requiere compilar en cada plataforma los mismos fuentes.
- O “cross compilar”. Es decir utilizar un cross compilador para generar con él en una plataforma de hardware, código apto para ser ejecutado en otra plataforma diferente de Hardware.

Portabilidad

- POSIX deriva de UNIX que fue concebido bajo el concepto de portabilidad
- Este atributo permite desarrollar aplicaciones que pueden correr en plataformas de hardware heterogénea sin necesidad de ser reescritas.
- Solo se requiere compilar en cada plataforma los mismos fuentes.
- O “cross compilar”. Es decir utilizar un cross compilador para generar con él en una plataforma de hardware, código apto para ser ejecutado en otra plataforma diferente de Hardware.
- Tal es el caso del paquete **`binutils-arm-linux-gnueabi`** que contiene el ensamblador, compilador, y el resto del toolchain para generar código de procesadores ARM en una típica PC con procesador de Intel.

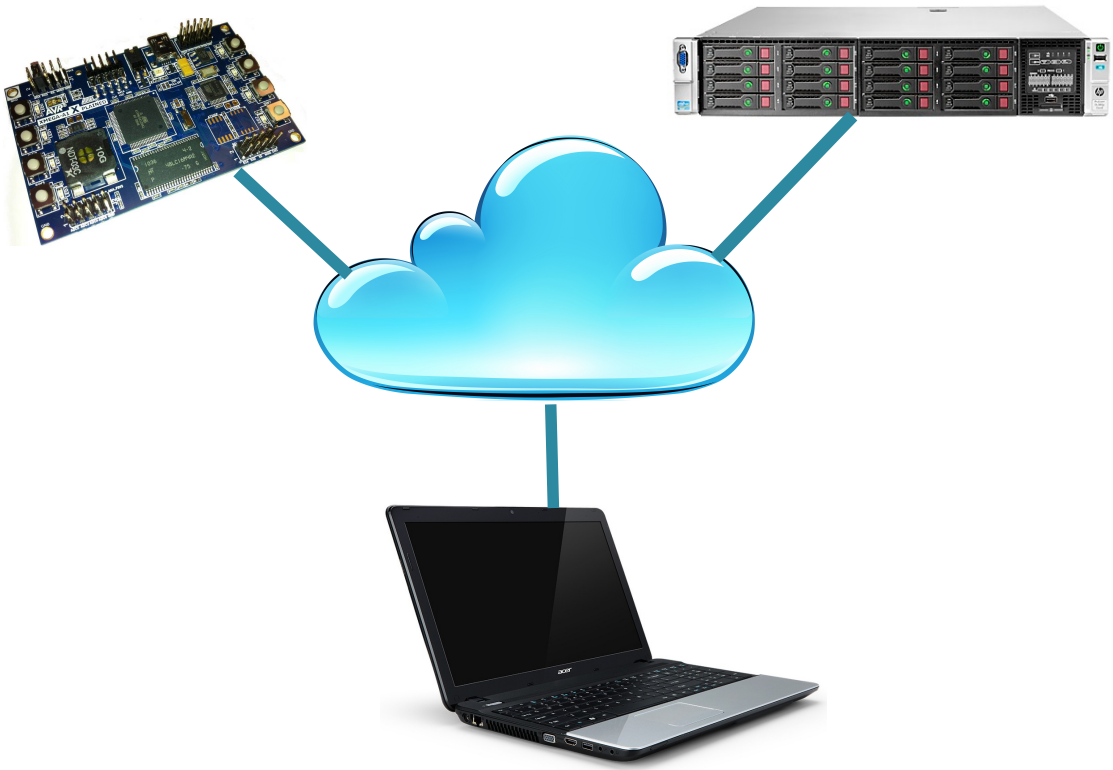
Portabilidad

- POSIX deriva de UNIX que fue concebido bajo el concepto de portabilidad
- Este atributo permite desarrollar aplicaciones que pueden correr en plataformas de hardware heterogénea sin necesidad de ser reescritas.
- Solo se requiere compilar en cada plataforma los mismos fuentes.
- O “cross compilar”. Es decir utilizar un cross compilador para generar con él en una plataforma de hardware, código apto para ser ejecutado en otra plataforma diferente de Hardware.
- Tal es el caso del paquete **`binutils-arm-linux-gnueabi`** que contiene el ensamblador, compilador, y el resto del toolchain para generar código de procesadores ARM en una típica PC con procesador de Intel.

```
sudo apt install binutils-arm-linux-gnueabi
```

Portabilidad: Caso de Aplicación

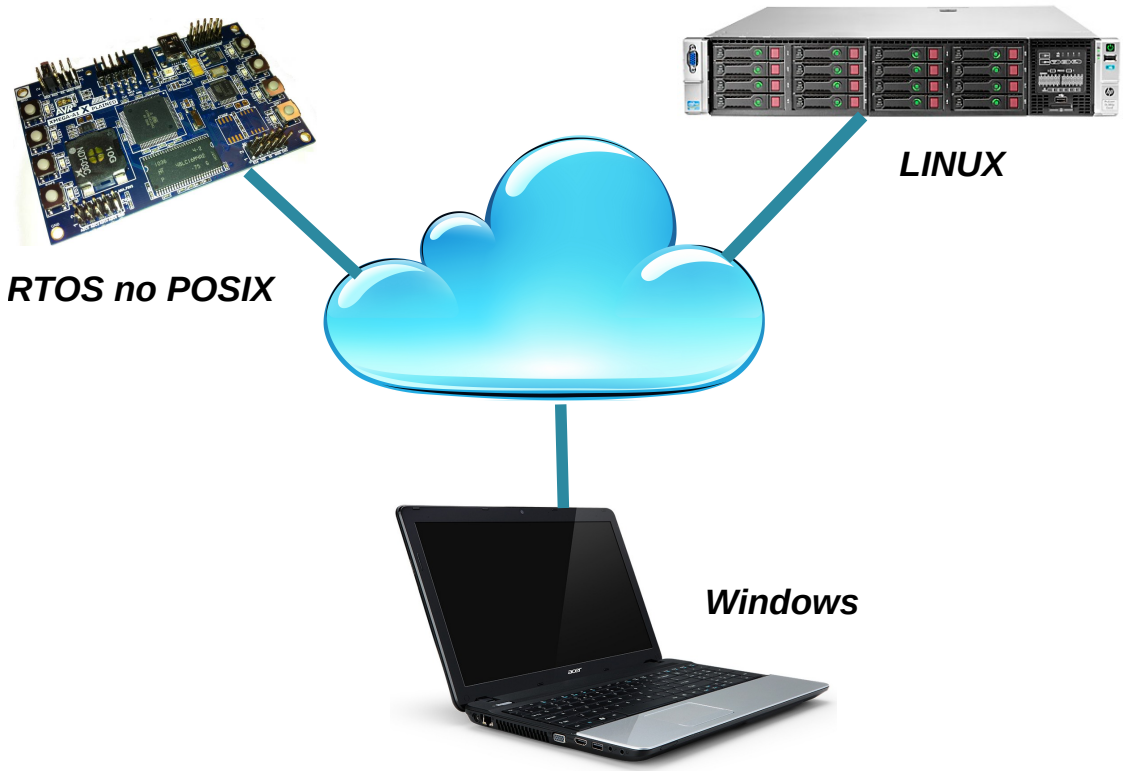
Sistema de control distribuido. En los tres nodos, tenemos funciones cliente y servidor sobre TCP/IP:



La pregunta es: ¿Que Sistema Operativo usamos en cada nodo?

Portabilidad: Caso de Aplicación

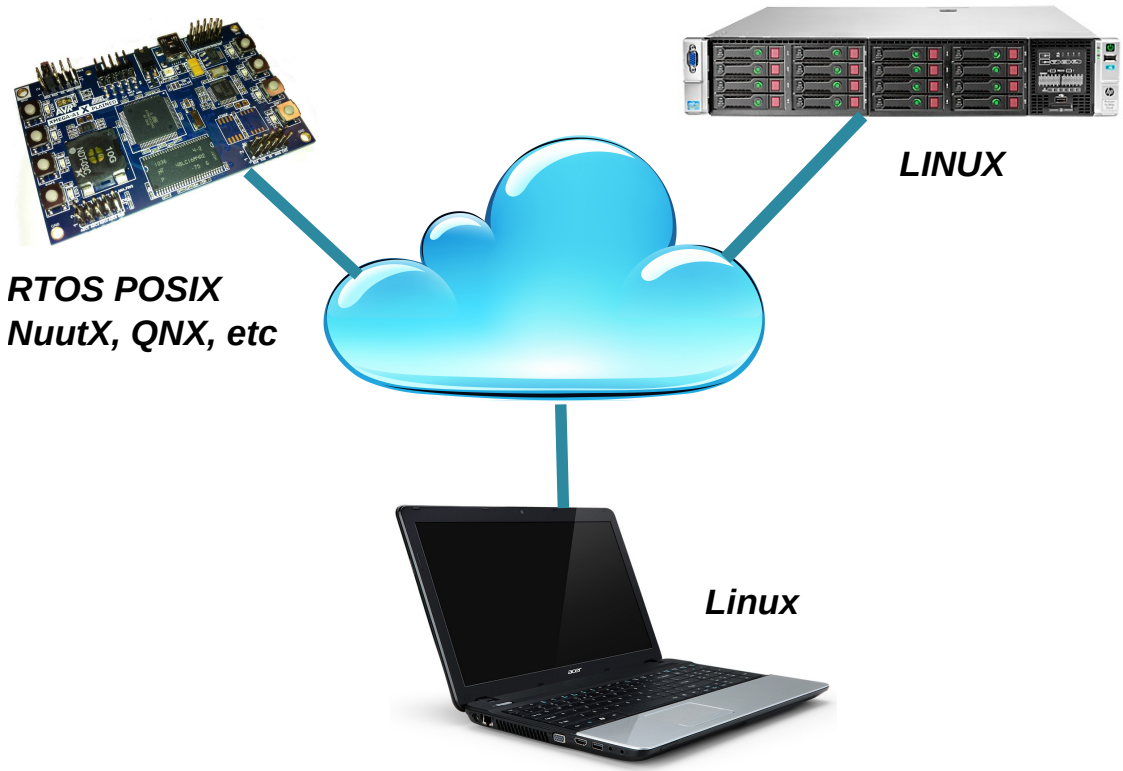
Sistema de control distribuido. En los tres nodos, tenemos funciones cliente y servidor sobre TCP/IP:



Opción 1: Un microcontrolador chico con un RTOS no POSIX, Windows en la notebook, y en el server Linux. Resultado tres fuentes diferentes para una misma aplicación

Portabilidad: Caso de Aplicación

Sistema de control distribuido. En los tres nodos, tenemos funciones cliente y servidor sobre TCP/IP:



Opción 2: Microcontrolador con RTOS POSIX, notebook con Ubuntu, Centos, Fedora o cualquier otra distribución, y un server con un Debian estable. Copiamos el mismo fuente en cada sistema y lo compilamos.

Conclusiones

Conclusiones

- Hay un conjunto de funciones cuya especificación es idéntica, para las funciones de cliente y servidor.

Conclusiones

- Hay un conjunto de funciones cuya especificación es idéntica, para las funciones de cliente y servidor.
- Si trabajamos con Sistemas Operativos heterogéneos que no guardan una estandarización común necesitamos escribir tres juegos de programas fuente para las mismas funciones.

Conclusiones

- Hay un conjunto de funciones cuya especificación es idéntica, para las funciones de cliente y servidor.
- Si trabajamos con Sistemas Operativos heterogéneos que no guardan una estandarización común necesitamos escribir tres juegos de programas fuente para las mismas funciones.
- Mayor cantidad de Horas Hombre para desarrollar tres versiones distintas del mismo grupo de funciones. Mas líneas de código =>mas errores. Tres puntos de falla. Mayor time to market.

Conclusiones

- Hay un conjunto de funciones cuya especificación es idéntica, para las funciones de cliente y servidor.
- Si trabajamos con Sistemas Operativos heterogéneos que no guardan una estandarización común necesitamos escribir tres juegos de programas fuente para las mismas funciones.
- Mayor cantidad de Horas Hombre para desarrollar tres versiones distintas del mismo grupo de funciones. Mas líneas de código =>mas errores. Tres puntos de falla. Mayor time to market.
- Mantenimiento y evolución: Tres juegos de fuentes a modificar cada vez que se agrega una funcionalidad. Un bug requiere determinar en cual de las tres partes está el problema antes de intervenir.

Conclusiones

- Hay un conjunto de funciones cuya especificación es idéntica, para las funciones de cliente y servidor.
- Si trabajamos con Sistemas Operativos heterogéneos que no guardan una estandarización común necesitamos escribir tres juegos de programas fuente para las mismas funciones.
- Mayor cantidad de Horas Hombre para desarrollar tres versiones distintas del mismo grupo de funciones. Mas líneas de código =>mas errores. Tres puntos de falla. Mayor time to market.
- Mantenimiento y evolución: Tres juegos de fuentes a modificar cada vez que se agrega una funcionalidad. Un bug requiere determinar en cual de las tres partes está el problema antes de intervenir.
- Si las tres Unidades adhieren a un estándar común, hay un solo juego de fuentes y un solo equipo de desarrollo para todo el sistema. Disminuyen la complejidad del testing, el time to market, y el costo de desarrollo y de Mantenimiento.

Tendencias

Tendencias

- En 2010, en los debates académicos del ámbito nacional, la opinión mayoritaria apoyaba la conveniencia de mantener la enseñanza de microcontroladores en base a modelos de 8 bits “porque la industria todavía los usa”. En el mundo se los consideraba obsoletos.

Tendencias

- En 2010, en los debates académicos del ámbito nacional, la opinión mayoritaria apoyaba la conveniencia de mantener la enseñanza de microcontroladores en base a modelos de 8 bits “porque la industria todavía los usa”. En el mundo se los consideraba obsoletos.
- En 2015 los fabricantes comenzaron a retirar su línea de descuentos en microcontroladores de 8 bits, de modo que pasaron a ser mas caros que los de 32 bits.

Tendencias

- En 2010, en los debates académicos del ámbito nacional, la opinión mayoritaria apoyaba la conveniencia de mantener la enseñanza de microcontroladores en base a modelos de 8 bits “porque la industria todavía los usa”. En el mundo se los consideraba obsoletos.
- En 2015 los fabricantes comenzaron a retirar su línea de descuentos en microcontroladores de 8 bits, de modo que pasaron a ser mas caros que los de 32 bits.
- Es lógico. Las nuevas líneas de producción desplazan a las mas antiguas. Un fabricante jamas sostiene indefinidamente una línea obsoleta. Eso requiere espacio mano de obra y recursos.

Tendencias

- En 2010, en los debates académicos del ámbito nacional, la opinión mayoritaria apoyaba la conveniencia de mantener la enseñanza de microcontroladores en base a modelos de 8 bits “porque la industria todavía los usa”. En el mundo se los consideraba obsoletos.
- En 2015 los fabricantes comenzaron a retirar su línea de descuentos en microcontroladores de 8 bits, de modo que pasaron a ser mas caros que los de 32 bits.
- Es lógico. Las nuevas líneas de producción desplazan a las mas antiguas. Un fabricante jamas sostiene indefinidamente una línea obsoleta. Eso requiere espacio mano de obra y recursos.
- Nuestro trabajo es anticiparnos y descartar antes.

Tendencias

- ¿En cuanto tiempo mas un procesador de 1 dolar va a contener suficientes recursos para dejar de ser un limitante en cuanto al sistema operativo que pueda o no tener?.

Tendencias

- ¿En cuanto tiempo mas un procesador de 1 dolar va a contener suficientes recursos para dejar de ser un limitante en cuanto al sistema operativo que pueda o no tener?.
- Miren el microcontrolador Cortex M7. Tiene memoria cache. Además dividida en Instrucciones y Datos, con buses separados hacia la CPU. Esto es estructura Hardvard. Pipeline superescalar....

Tendencias

- ¿En cuanto tiempo mas un procesador de 1 dolar va a contener suficientes recursos para dejar de ser un limitante en cuanto al sistema operativo que pueda o no tener?.
- Miren el microcontrolador Cortex M7. Tiene memoria cache. Además dividida en Instrucciones y Datos, con buses separados hacia la CPU. Esto es estructura Hardvard. Pipeline superescalar....
- Vale la pena preguntarse... ¿Es un microcontrolador?...

Tendencias

- ¿En cuanto tiempo mas un procesador de 1 dolar va a contener suficientes recursos para dejar de ser un limitante en cuanto al sistema operativo que pueda o no tener?.
- Miren el microcontrolador Cortex M7. Tiene memoria cache. Además dividida en Instrucciones y Datos, con buses separados hacia la CPU. Esto es estructura Hardvard. Pipeline superescalar....
- Vale la pena preguntarse... ¿Es un microcontrolador?...
- Si. Pero no responde a los estereotipos existentes.

Tendencias

- ¿En cuanto tiempo mas un procesador de 1 dolar va a contener suficientes recursos para dejar de ser un limitante en cuanto al sistema operativo que pueda o no tener?.
- Miren el microcontrolador Cortex M7. Tiene memoria cache. Además dividida en Instrucciones y Datos, con buses separados hacia la CPU. Esto es estructura Hardvard. Pipeline superescalar....
- Vale la pena preguntarse... ¿Es un microcontrolador?...
- Si. Pero no responde a los estereotipos existentes.
- Y vendrán cosas mas interesantes seguramente. (Y disruptivas!)

Tendencias

- ¿En cuanto tiempo mas un procesador de 1 dolar va a contener suficientes recursos para dejar de ser un limitante en cuanto al sistema operativo que pueda o no tener?.
- Miren el microcontrolador Cortex M7. Tiene memoria cache. Además dividida en Instrucciones y Datos, con buses separados hacia la CPU. Esto es estructura Hardvard. Pipeline superescalar....
- Vale la pena preguntarse... ¿Es un microcontrolador?...
- Si. Pero no responde a los estereotipos existentes.
- Y vendrán cosas mas interesantes seguramente. (Y disruptivas!)
- Tratar de ver hacia adelante siempre, sin pre conceptos, parece ser la mejor alternativa.

Everything is a file

Everything is a file

- Una de las innovaciones que introdujo UNIX es que muchos dispositivos de E/S, se tratan como byte streams.

Everything is a file

- Una de las innovaciones que introdujo UNIX es que muchos dispositivos de E/S, se tratan como byte streams.
- Esto hace que un pequeño conjunto de operaciones sobre archivos tanto del API como del shell se utilicen para manipular E/S

Everything is a file

- Una de las innovaciones que introdujo UNIX es que muchos dispositivos de E/S, se tratan como byte streams.
- Esto hace que un pequeño conjunto de operaciones sobre archivos tanto del API como del shell se utilicen para manipular E/S
- Todo lo que se necesita es la descripción del dispositivo como un nodo del File System.

Everything is a file

- Una de las innovaciones que introdujo UNIX es que muchos dispositivos de E/S, se tratan como byte streams.
- Esto hace que un pequeño conjunto de operaciones sobre archivos tanto del API como del shell se utilicen para manipular E/S
- Todo lo que se necesita es la descripción del dispositivo como un nodo del File System.
- En general encontramos en el directorio /dev todos los nodos que describen devices.

Everything is a file

- Una de las innovaciones que introdujo UNIX es que muchos dispositivos de E/S, se tratan como byte streams.
- Esto hace que un pequeño conjunto de operaciones sobre archivos tanto del API como del shell se utilicen para manipular E/S
- Todo lo que se necesita es la descripción del dispositivo como un nodo del File System.
- En general encontramos en el directorio /dev todos los nodos que describen devices.
- Su implementación corre por cuenta del device driver correspondiente.

Everything is a file

- De este modo accedemos a un device de E/S como si fuese un archivo

  open Obtenemos un file descriptor que será utilizado en delante para referenciarlo

  read Leemos en un buffer el byte stream que provee el device (a la velocidad del mismo)

  write Enviamos información presente en un buffer al device

  ioctl Accedemos a su configuración (depende de cada device)

  close Una vez que terminamos de trabajar con él, devolvemos el file descriptor y liberamos el recurso

Origen de Linux

Origen de Linux

- En 1991, un estudiante de la Universidad de Helsinki llamado Linus Torvalds, estaba ávido por encontrar una versión de UNIX robusta y con fuentes disponibles e hizo lo que cualquier estudiante de computación inquieto y con vocación: comenzó a escribir su propio Sistema Operativo. Lo publicó a fines de 1991 bajo la GNU General Public Licence (GPL).

Origen de Linux

- En 1991, un estudiante de la Universidad de Helsinki llamado Linus Torvalds, estaba ávido por encontrar una versión de UNIX robusta y con fuentes disponibles e hizo lo que cualquier estudiante de computación inquieto y con vocación: comenzó a escribir su propio Sistema Operativo. Lo publicó a fines de 1991 bajo la GNU General Public Licence (GPL).
- Rápidamente se sumó un gran número de hackers y desarrolladores que comenzaron a mejorar corregir y aumentar su código y actualmente es un sistema operativo robusto, de interés comercial (Muchos fabricantes de Hardware ofrece soluciones basadas en LINUX), desarrollado por un equipo de desarrolladores a través de Internet.

Linux Hoy

Linux Hoy

- Sistema Operativo Unix-like, con soporte a POSIX.

Linux Hoy

- Sistema Operativo Unix-like, con soporte a POSIX.
- Kernel monolítico: Imagen única de código.

Linux Hoy

- Sistema Operativo Unix-like, con soporte a POSIX.
- Kernel monolítico: Imagen única de código.
- Diseñado bajo el concepto Lightweight Processes (LWP)

Linux Hoy

- Sistema Operativo Unix-like, con soporte a POSIX.
- Kernel monolítico: Imagen única de código.
- Diseñado bajo el concepto Lightweight Processes (LWP)
- Preemptive Kernel: Hasta la versión 2.4 el kernel no puede intercalar arbitrariamente flujos de ejecución mientras está en modo privilegiado (PL=00). Desde la versión 2.6 el kernel puede ser compilado con diferentes opciones de preemption y convertirse en un Hard Real Time Operating System si se lo requiere.

Linux Hoy

- Sistema Operativo Unix-like, con soporte a POSIX.
- Kernel monolítico: Imagen única de código.
- Diseñado bajo el concepto Lightweight Processes (LWP)
- Preemptive Kernel: Hasta la versión 2.4 el kernel no puede intercalar arbitrariamente flujos de ejecución mientras está en modo privilegiado (PL=00). Desde la versión 2.6 el kernel puede ser compilado con diferentes opciones de preemption y convertirse en un Hard Real Time Operating System si se lo requiere.
- Soporta SMP (Symmetric Multi Processing)

Linux Hoy

- Sistema Operativo Unix-like, con soporte a POSIX.
- Kernel monolítico: Imagen única de código.
- Diseñado bajo el concepto Lightweight Processes (LWP)
- Preemptive Kernel: Hasta la versión 2.4 el kernel no puede intercalar arbitrariamente flujos de ejecución mientras está en modo privilegiado (PL=00). Desde la versión 2.6 el kernel puede ser compilado con diferentes opciones de preemption y convertirse en un Hard Real Time Operating System si se lo requiere.
- Soporta SMP (Symmetric Multi Processing)
- Soporta un amplia diversidad de File Systems (IBM AIX, SGI Irix, FAT32, NTFS,etc.)

Linux Hoy

- Sistema Operativo Unix-like, con soporte a POSIX.
- Kernel monolítico: Imagen única de código.
- Diseñado bajo el concepto Lightweight Processes (LWP)
- Preemptive Kernel: Hasta la versión 2.4 el kernel no puede intercalar arbitrariamente flujos de ejecución mientras está en modo privilegiado (PL=00). Desde la versión 2.6 el kernel puede ser compilado con diferentes opciones de preemption y convertirse en un Hard Real Time Operating System si se lo requiere.
- Soporta SMP (Symmetric Multi Processing)
- Soporta un amplia diversidad de File Systems (IBM AIX, SGI Irix, FAT32, NTFS,etc.)
- Las herramientas de construcción del kernel, permiten customizar imágenes a la medida del hardware de base y con los componentes mínimos necesarios de modo de obtener sistemas muy compactos.

El Kernel

El Kernel

- Es el Sistema Operativo propiamente dicho

El Kernel

- Es el Sistema Operativo propiamente dicho
- Típicamente se encarga de:

El Kernel

- Es el Sistema Operativo propiamente dicho
- Típicamente se encarga de:
  1. Manejar las interrupciones de hardware.

El Kernel

- Es el Sistema Operativo propiamente dicho
- Típicamente se encarga de:
  1. Manejar las interrupciones de hardware.
  2. Asignar el tiempo de CPU a cada uno de los procesos que estén en ejecución en un instante dado.

El Kernel

- Es el Sistema Operativo propiamente dicho
- Típicamente se encarga de:
  1. Manejar las interrupciones de hardware.
  2. Asignar el tiempo de CPU a cada uno de los procesos que estén en ejecución en un instante dado.
  3. Gestionar la memoria para manejar los diferentes espacios de direcciones de loso procesos.

El Kernel

- Es el Sistema Operativo propiamente dicho
- Típicamente se encarga de:
  1. Manejar las interrupciones de hardware.
  2. Asignar el tiempo de CPU a cada uno de los procesos que estén en ejecución en un instante dado.
  3. Gestionar la memoria para manejar los diferentes espacios de direcciones de loso procesos.
  4. Gestionar servicios de sistema, como networking, Intercomunicación de procesos, etc.

El Kernel

- Es el Sistema Operativo propiamente dicho
- Típicamente se encarga de:
  1. Manejar las interrupciones de hardware.
  2. Asignar el tiempo de CPU a cada uno de los procesos que estén en ejecución en un instante dado.
  3. Gestionar la memoria para manejar los diferentes espacios de direcciones de loso procesos.
  4. Gestionar servicios de sistema, como networking, Intercomunicación de procesos, etc.
- Trabaja en un estado de elevado nivel comparado con las aplicaciones gracias a las funciones de protección de memoria y recursos que poseen los procesadores modernos.

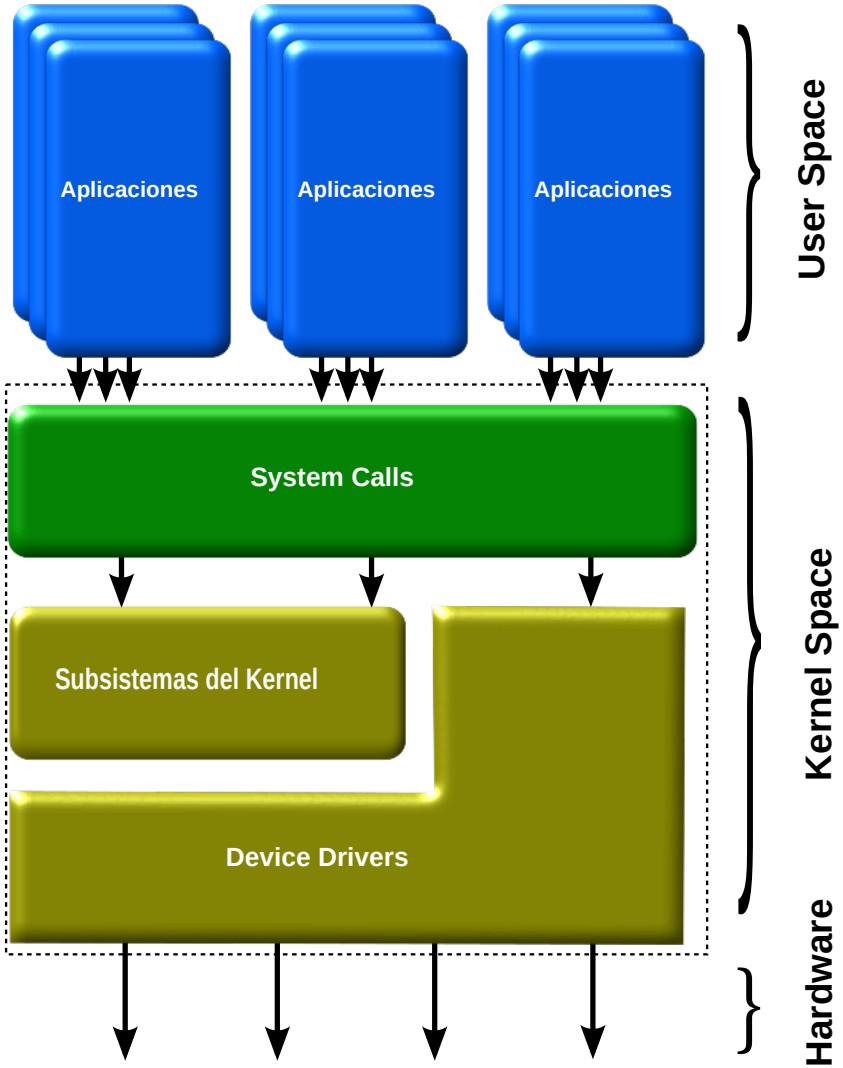
El Kernel

- Es el Sistema Operativo propiamente dicho
- Típicamente se encarga de:
  1. Manejar las interrupciones de hardware.
  2. Asignar el tiempo de CPU a cada uno de los procesos que estén en ejecución en un instante dado.
  3. Gestionar la memoria para manejar los diferentes espacios de direcciones de loso procesos.
  4. Gestionar servicios de sistema, como networking, Intercomunicación de procesos, etc.
- Trabaja en un estado de elevado nivel comparado con las aplicaciones gracias a las funciones de protección de memoria y recursos que poseen los procesadores modernos.
- Tiene un área propia de memoria y acceso completo al hardware.

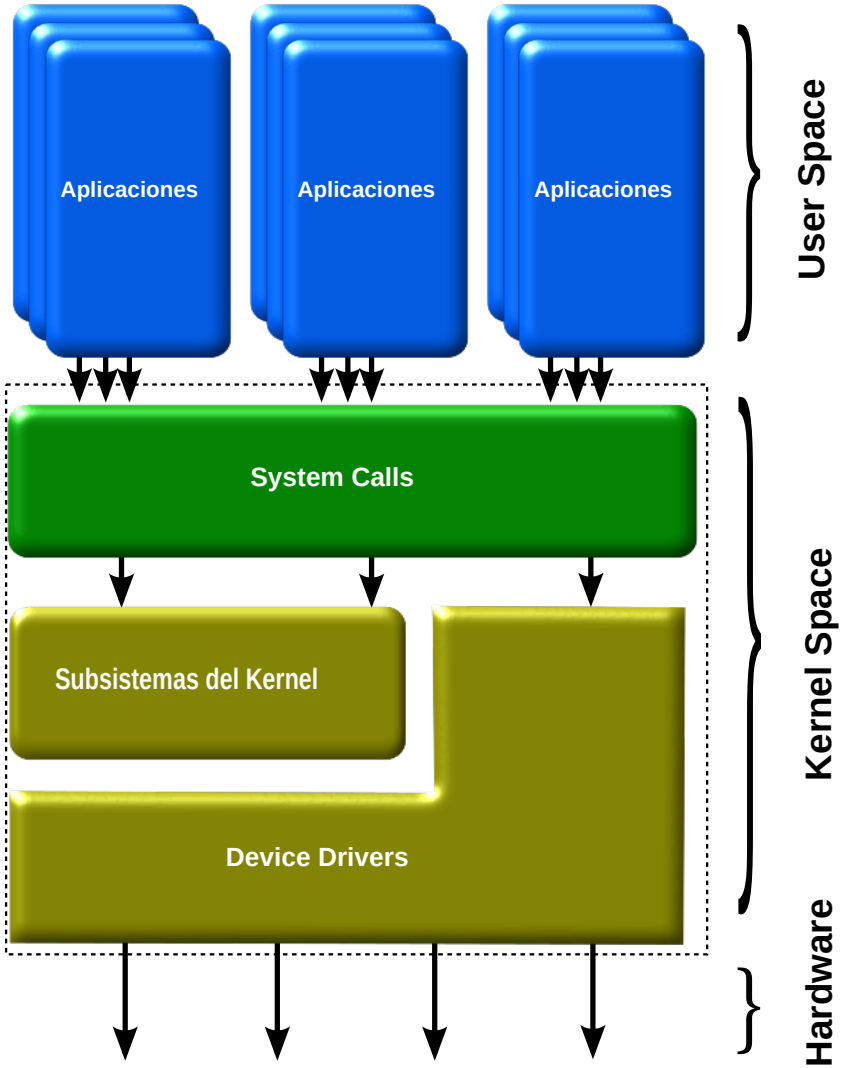
El Kernel

- Es el Sistema Operativo propiamente dicho
- Típicamente se encarga de:
  1. Manejar las interrupciones de hardware.
  2. Asignar el tiempo de CPU a cada uno de los procesos que estén en ejecución en un instante dado.
  3. Gestionar la memoria para manejar los diferentes espacios de direcciones de loso procesos.
  4. Gestionar servicios de sistema, como networking, Intercomunicación de procesos, etc.
- Trabaja en un estado de elevado nivel comparado con las aplicaciones gracias a las funciones de protección de memoria y recursos que poseen los procesadores modernos.
- Tiene un área propia de memoria y acceso completo al hardware.
- A este estado de sistema y su espacio de memoria se lo suele referir como *kernel space*. Y por otra parte al espacio de memoria y estado de las aplicaciones se lo refiere como *user space*.

El Kernel

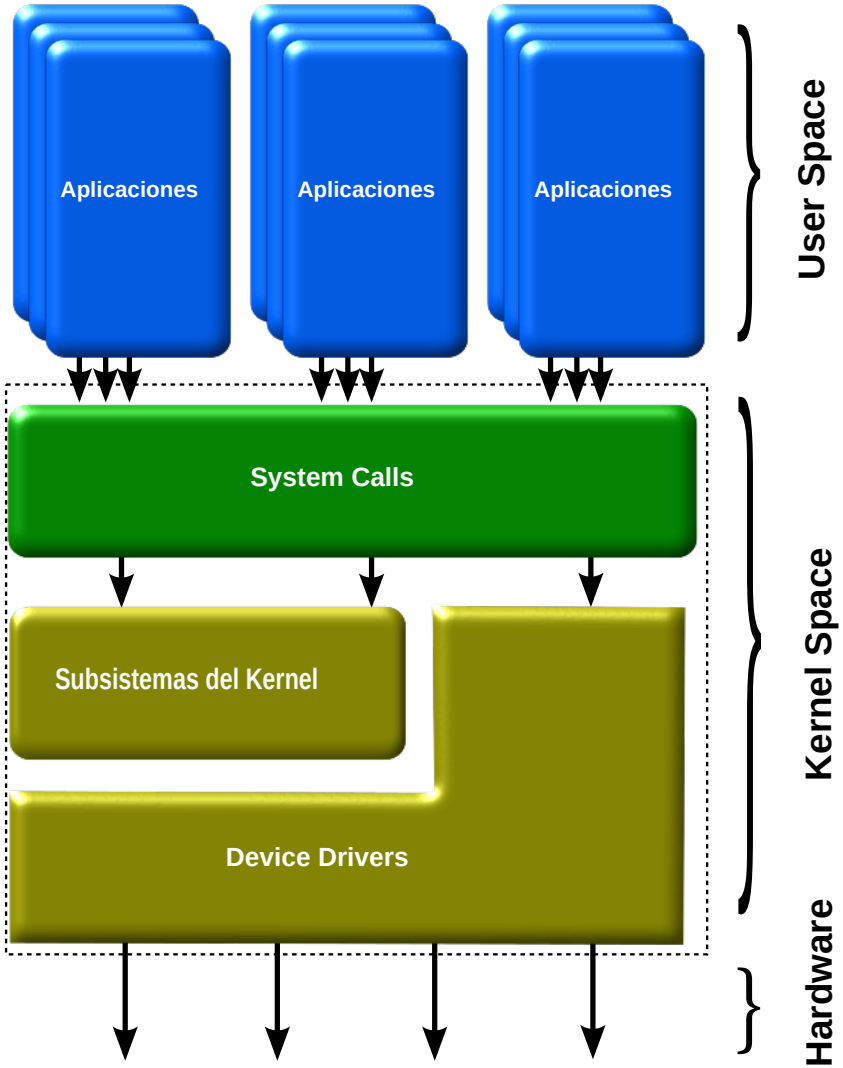

Aplicaciones
Aplicaciones
Aplicaciones
User Space
System Calls
Subsistemas del Kernel
Device Drivers
Kernel Space
Hardware

El Kernel



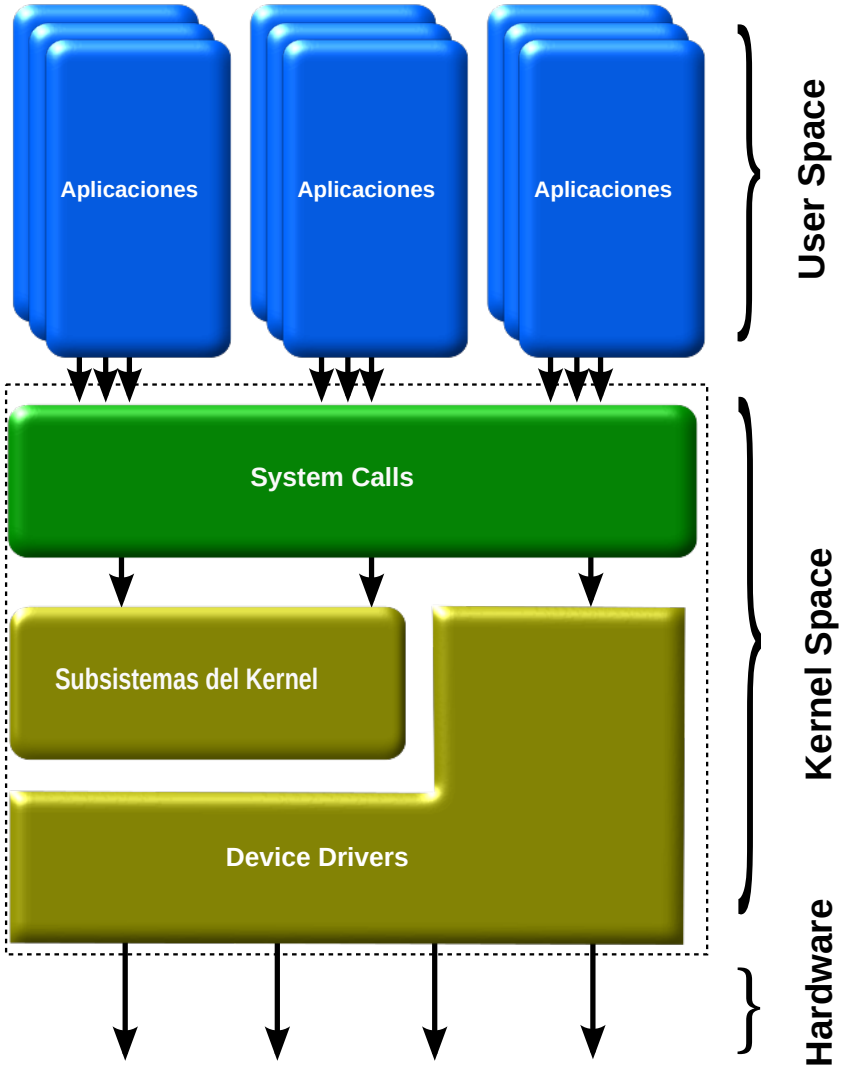
- Las aplicaciones ejecutan en espacios de direccionamiento propios (protección de memoria por hardware), y el kernel tiene su propio espacio de direccionamiento

El Kernel



- Las aplicaciones ejecutan en espacios de direccionamiento propios (protección de memoria por hardware), y el kernel tiene su propio espacio de direccionamiento
- Las aplicaciones se comunican con el kernel por medio de *System Calls*.

El Kernel



- Las aplicaciones ejecutan en espacios de direccionamiento propios (protección de memoria por hardware), y el kernel tiene su propio espacio de direccionamiento
- Las aplicaciones se comunican con el kernel por medio de *System Calls*.
- Luego el kernel resuelve los servicios requeridos accediendo al hardware o mediante algoritmos propios y entrega resultados.

¿Que es un proceso?

¿Que es un proceso?

- Los objetos que conforman la imagen de un programa (código, pila, y datos) están originalmente en algún medio masivo de almacenamiento.

¿Que es un proceso?

- Los objetos que conforman la imagen de un programa (código, pila, y datos) están originalmente en algún medio masivo de almacenamiento.
- La ***text section***), la ***data section*** que contiene las variables globales, y demás secciones componen esta imagen.

¿Que es un proceso?

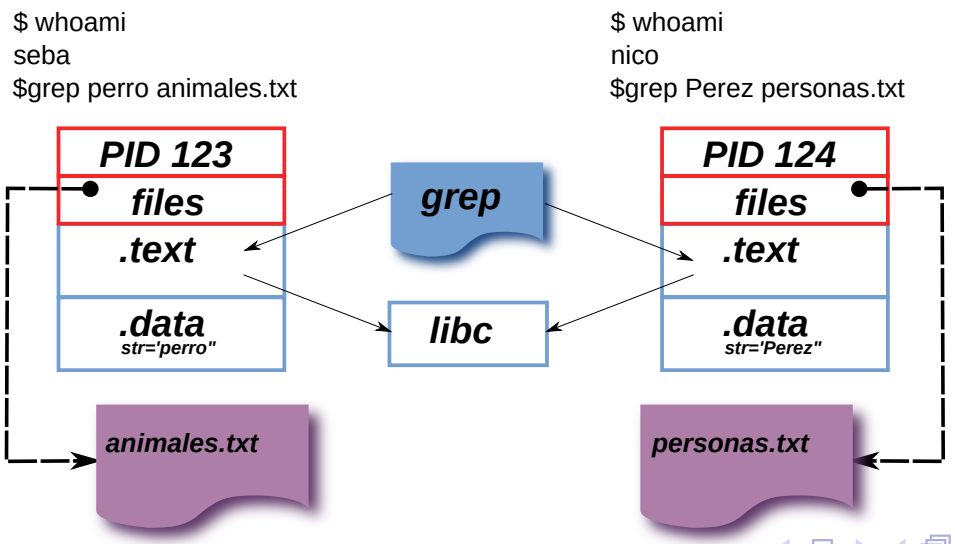
- Los objetos que conforman la imagen de un programa (código, pila, y datos) están originalmente en algún medio masivo de almacenamiento.
- La ***text section***), la ***data section*** que contiene las variables globales, y demás secciones componen esta imagen.
- Pero un proceso es bastante mas que eso.

¿Que es un proceso?

- Los objetos que conforman la imagen de un programa (código, pila, y datos) están originalmente en algún medio masivo de almacenamiento.
- La ***text section***), la ***data section*** que contiene las variables globales, y demás secciones componen esta imagen.
- Pero un proceso es bastante mas que eso.
- Es la instancia de ejecución de un programa en la memoria del sistema.

¿Que es un proceso?

- Los objetos que conforman la imagen de un programa (código, pila, y datos) están originalmente en algún medio masivo de almacenamiento.
- La ***text section***), la ***data section*** que contiene las variables globales, y demás secciones componen esta imagen.
- Pero un proceso es bastante mas que eso.
- Es la instancia de ejecución de un programa en la memoria del sistema.



¿Que mas es un proceso?

Un proceso involucra mucha información además de su código y sus datos:

¿Que mas es un proceso?

Un proceso involucra mucha información además de su código y sus datos:

- Los file descriptors de los archivos abiertos por el programa,

¿Que mas es un proceso?

Un proceso involucra mucha información además de su código y sus datos:

- Los file descriptors de los archivos abiertos por el programa,
- Las señales pendientes de atención,

¿Que mas es un proceso?

Un proceso involucra mucha información además de su código y sus datos:

- Los file descriptors de los archivos abiertos por el programa,
- Las señales pendientes de atención,
- El estado interno del kernel para ese proceso,

¿Que mas es un proceso?

Un proceso involucra mucha información además de su código y sus datos:

- Los file descriptors de los archivos abiertos por el programa,
- Las señales pendientes de atención,
- El estado interno del kernel para ese proceso,
- El contexto de la CPU (valores de cada registro de la CPU),

¿Que mas es un proceso?

Un proceso involucra mucha información además de su código y sus datos:

- Los file descriptors de los archivos abiertos por el programa,
- Las señales pendientes de atención,
- El estado interno del kernel para ese proceso,
- El contexto de la CPU (valores de cada registro de la CPU),
- El espacio de direcciones de memoria,

¿Que mas es un proceso?

Un proceso involucra mucha información además de su código y sus datos:

- Los file descriptors de los archivos abiertos por el programa,
- Las señales pendientes de atención,
- El estado interno del kernel para ese proceso,
- El contexto de la CPU (valores de cada registro de la CPU),
- El espacio de direcciones de memoria,
- Estará ligado al usuario que lo ejecuta (tendrá sus permisos, sus variables de entorno, su HOME directory, etc)

¿Que mas es un proceso?

Un proceso involucra mucha información además de su código y sus datos:

- Los file descriptors de los archivos abiertos por el programa,
- Las señales pendientes de atención,
- El estado interno del kernel para ese proceso,
- El contexto de la CPU (valores de cada registro de la CPU),
- El espacio de direcciones de memoria,
- Estará ligado al usuario que lo ejecuta (tendrá sus permisos, sus variables de entorno, su HOME directory, etc)
- La prioridad de ejecución (el % de tiempo de CPU que el kernel le asignará para su ejecución)...

¿Que mas es un proceso?

Un proceso involucra mucha información además de su código y sus datos:

- Los file descriptors de los archivos abiertos por el programa,
- Las señales pendientes de atención,
- El estado interno del kernel para ese proceso,
- El contexto de la CPU (valores de cada registro de la CPU),
- El espacio de direcciones de memoria,
- Estará ligado al usuario que lo ejecuta (tendrá sus permisos, sus variables de entorno, su HOME directory, etc)
- La prioridad de ejecución (el % de tiempo de CPU que el kernel le asignará para su ejecución)...
- ...entre muchas otras cosas.

Bloque de control de Proceso (BCP)

Bloque de control de Proceso (BCP)

- Toda la información de control del proceso debe agruparse en forma de un bloque asociado al proceso en ejecución.

Bloque de control de Proceso (BCP)

- Toda la información de control del proceso debe agruparse en forma de un bloque asociado al proceso en ejecución.
- Por lo genera se trata de una estructura de datos que el kernel crea en su memoria (***kernel space***).

Bloque de control de Proceso (BCP)

- Toda la información de control del proceso debe agruparse en forma de un bloque asociado al proceso en ejecución.
- Por lo genera se trata de una estructura de datos que el kernel crea en su memoria (***kernel space***).
- Genéricamente se la denomina **P**rocess **C**ontrol **B**lock (**PCB**), o **BCP** si lo mencionamos en castellano.

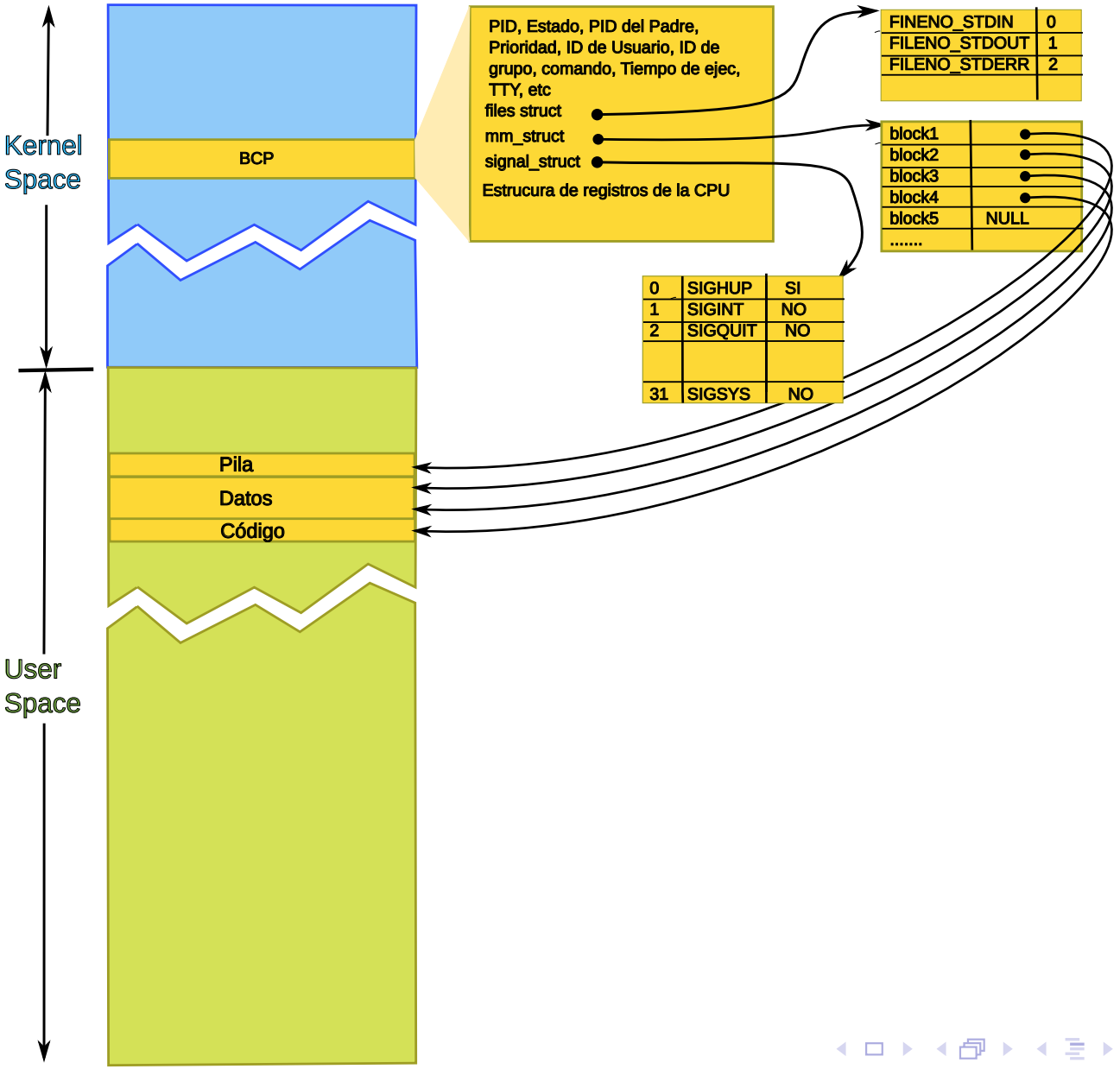
Bloque de control de Proceso (BCP)

- Toda la información de control del proceso debe agruparse en forma de un bloque asociado al proceso en ejecución.
- Por lo genera se trata de una estructura de datos que el kernel crea en su memoria (***kernel space***).
- Genéricamente se la denomina **P**rocess **C**ontrol **B**lock (**PCB**), o **BCP** si lo mencionamos en castellano.
- En el user space están las secciones de código, datos y pila del proceso.

Bloque de control de Proceso (BCP)

- Toda la información de control del proceso debe agruparse en forma de un bloque asociado al proceso en ejecución.
- Por lo genera se trata de una estructura de datos que el kernel crea en su memoria (***kernel space***).
- Genéricamente se la denomina **P**rocess **C**ontrol **B**lock (**PCB**), o **BCP** si lo mencionamos en castellano.
- En el user space están las secciones de código, datos y pila del proceso.
- Por lo general los GPOS mapean estas secciones en áreas de memoria exclusivas de cada proceso.

Bloque de Control de Proceso



Descriptor de proceso de Linux

Descriptor de proceso de Linux

- Linux utiliza una estructura definida como `struct task_struc`

Descriptor de proceso de Linux

- Linux utiliza una estructura definida como `struct task_struc`
- Definida en `<kernel\sched.h>`

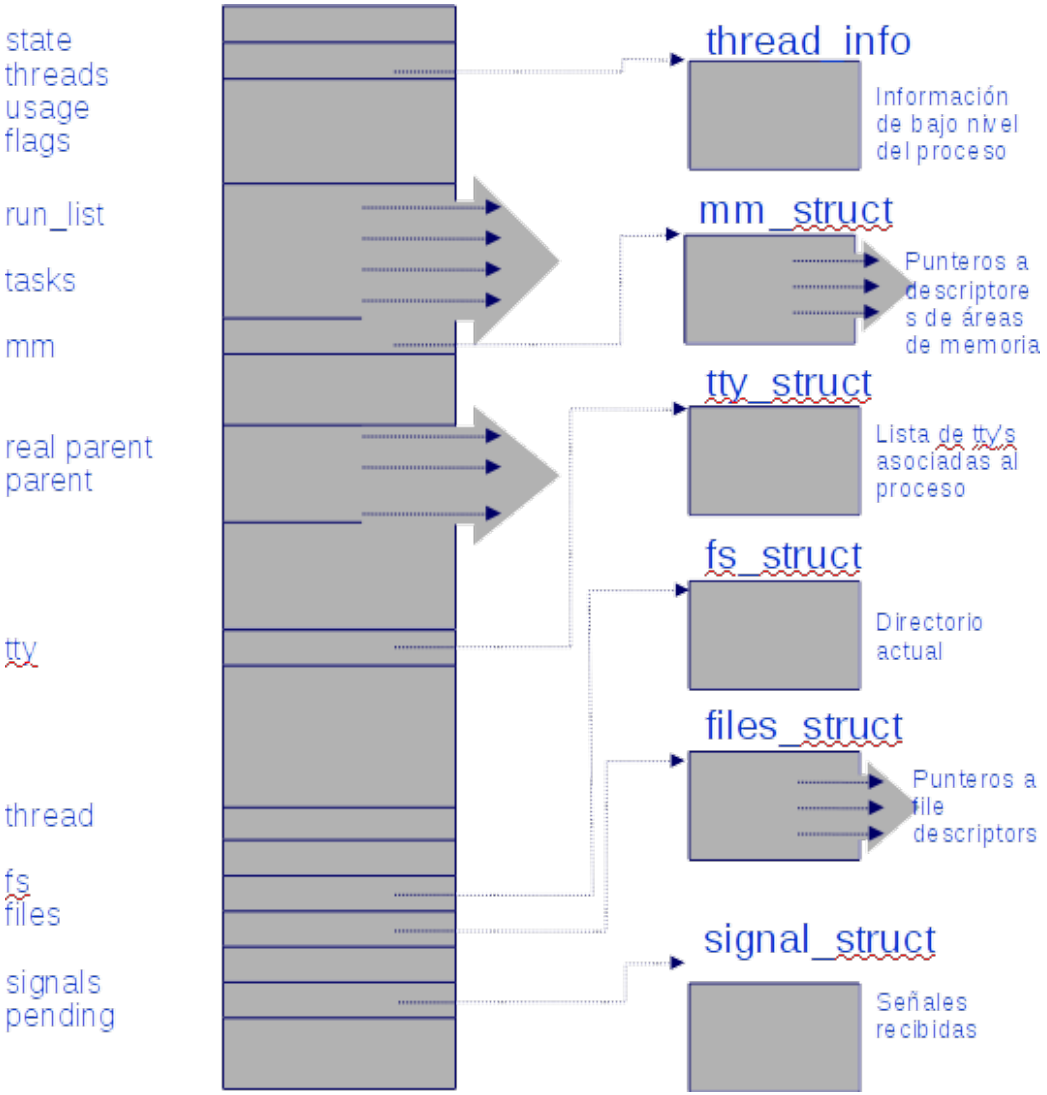
Descriptor de proceso de Linux

- Linux utiliza una estructura definida como `struct task_struc`
- Definida en `<kernel\sched.h>`
- Es una señora estructura: Aproximadamente 1.7 kilobytes en una máquina de 32 bits.

Descriptor de proceso de Linux

- Linux utiliza una estructura definida como `struct task_struc`
- Definida en `<kernel\sched.h>`
- Es una señora estructura: Aproximadamente 1.7 kilobytes en una máquina de 32 bits.
- Tiene toda la información que tiene y necesita el kernel para controlar el proceso.

Descriptor de proceso de Linux



Descriptor de proceso de Linux

Descriptor de proceso de Linux

- volatile long state; Estado del proceso

Descriptor de proceso de Linux

- volatile long state; Estado del proceso
- struct thread_info *thread_info; struct thread_info es una estructura que contiene el contexto de la CPU.

Descriptor de proceso de Linux

- volatile long state; Estado del proceso
- struct thread_info *thread_info; struct thread_info es una estructura que contiene el contexto de la CPU.
- unsigned int cpu; CPU que ejecuta actualmente el proceso

Descriptor de proceso de Linux

- volatile long state; Estado del proceso
- struct thread_info *thread_info; struct thread_info es una estructura que contiene el contexto de la CPU.
- unsigned int cpu; CPU que ejecuta actualmente el proceso
- struct files_struct *files; Array de file descriptors de archivos abiertos por el proceso.

Descriptor de proceso de Linux

- volatile long state; Estado del proceso
- struct thread_info *thread_info; struct thread_info es una estructura que contiene el contexto de la CPU.
- unsigned int cpu; CPU que ejecuta actualmente el proceso
- struct files_struct *files; Array de file descriptors de archivos abiertos por el proceso.
- struct signal_struct *signal; Estructura de señales asociadas al proceso

Descriptor de proceso de Linux

- volatile long state; Estado del proceso
- struct thread_info *thread_info; struct thread_info es una estructura que contiene el contexto de la CPU.
- unsigned int cpu; CPU que ejecuta actualmente el proceso
- struct files_struct *files; Array de file descriptors de archivos abiertos por el proceso.
- struct signal_struct *signal; Estructura de señales asociadas al proceso
- struct sighand_struct *sighand; Estructura de punteros a función de los manejadores de señales

Descriptor de proceso de Linux

- struct sigpending pending; Señales enviadas al proceso y aún no procesadas.

Descriptor de proceso de Linux

- struct sigpending pending; Señales enviadas al proceso y aún no procesadas.
- struct rlimit rlim[RLIM_NLIMITS]; Estructura que contiene valores limite del proceso (detalles en el comando **`ulimit -a`**)

Descriptor de proceso de Linux

- struct sigpending pending; Señales enviadas al proceso y aún no procesadas.
- struct rlimit rlim[RLIM_NLIMITS]; Estructura que contiene valores limite del proceso (detalles en el comando **`ulimit -a`**)
- spinlock_t alloc_lock; Spin lock para segurizar operaciones de alojamiento o liberación de memoria, accesos en general al file system, y asignación de terminales.

Descriptor de proceso de Linux

- struct sigpending pending; Señales enviadas al proceso y aún no procesadas.
- struct rlimit rlim[RLIM_NLIMITS]; Estructura que contiene valores limite del proceso (detalles en el comando **`ulimit -a`**)
- spinlock_t alloc_lock; Spin lock para segurizar operaciones de alojamiento o liberación de memoria, accesos en general al file system, y asignación de terminales.
- spinlock_t proc_lock; Spin lock empeado en anidamiento de procesos por ejemplo.

Descriptor de proceso de Linux

- struct sigpending pending; Señales enviadas al proceso y aún no procesadas.
- struct rlimit rlim[RLIM_NLIMITS]; Estructura que contiene valores limite del proceso (detalles en el comando **`ulimit -a`**)
- spinlock_t alloc_lock; Spin lock para segurizar operaciones de alojamiento o liberación de memoria, accesos en general al file system, y asignación de terminales.
- spinlock_t proc_lock; Spin lock empeado en anidamiento de procesos por ejemplo.
- spinlock_t switch_lock; Spin lock para lockear la estructura durante el context switch

¿No es Linux un kernel monolítico?

¿No es Linux un kernel monolítico?

- En ocasiones es útil (y necesario) que el kernel realice algunas operaciones en segundo plano.

¿No es Linux un kernel monolítico?

- En ocasiones es útil (y necesario) que el kernel realice algunas operaciones en segundo plano.
- El kernel realiza estas operaciones mediante procesos de threads estándar, que existen solamente en el *kernel space* .

¿No es Linux un kernel monolítico?

- En ocasiones es útil (y necesario) que el kernel realice algunas operaciones en segundo plano.
- El kernel realiza estas operaciones mediante procesos de threads estándar, que existen solamente en el *kernel space* .
- La fundamental diferencia es que los kernel threads no tienen un espacio de direcciones. Esto significa que para cada kernel thread se verifica `struct task_struc.mm=NULL`

¿No es Linux un kernel monolítico?

- En ocasiones es útil (y necesario) que el kernel realice algunas operaciones en segundo plano.
- El kernel realiza estas operaciones mediante procesos de threads estándar, que existen solamente en el *kernel space* .
- La fundamental diferencia es que los kernel threads no tienen un espacio de direcciones. Esto significa que para cada kernel thread se verifica `struct task_struc.mm=NULL`
- Nunca cambian al user-space.

¿No es Linux un kernel monolítico?

- En ocasiones es útil (y necesario) que el kernel realice algunas operaciones en segundo plano.
- El kernel realiza estas operaciones mediante procesos de threads estándar, que existen solamente en el *kernel space* .
- La fundamental diferencia es que los kernel threads no tienen un espacio de direcciones. Esto significa que para cada kernel thread se verifica `struct task_struc.mm=NULL`
- Nunca cambian al user-space.
- Se schedulan y son interrumpibles (preemptables) como cualquier proceso normal. Se ven con ps -ef

¿No es Linux un kernel monolítico?

- En ocasiones es útil (y necesario) que el kernel realice algunas operaciones en segundo plano.
- El kernel realiza estas operaciones mediante procesos de threads estándar, que existen solamente en el *kernel space* .
- La fundamental diferencia es que los kernel threads no tienen un espacio de direcciones. Esto significa que para cada kernel thread se verifica `struct task_struc.mm=NULL`
- Nunca cambian al user-space.
- Se schedulan y son interrumpibles (preemptables) como cualquier proceso normal. Se ven con ps -ef
- Linux delega tareas en diferentes kernel threads. En particular las flush y las ksoftirqd.

¿No es Linux un kernel monolítico?

- En ocasiones es útil (y necesario) que el kernel realice algunas operaciones en segundo plano.
- El kernel realiza estas operaciones mediante procesos de threads estándar, que existen solamente en el *kernel space* .
- La fundamental diferencia es que los kernel threads no tienen un espacio de direcciones. Esto significa que para cada kernel thread se verifica `struct task_struc.mm=NULL`
- Nunca cambian al user-space.
- Se schedulan y son interrumpibles (preemptables) como cualquier proceso normal. Se ven con ps -ef
- Linux delega tareas en diferentes kernel threads. En particular las flush y las ksoftirqd.
- Se los crean durante el arranque del sistema, por init o por otros kernel threads.

Identificación de los procesos

Identificación de los procesos

- Cada proceso tiene un Descriptor que lo define unívocamente.

Identificación de los procesos

- Cada proceso tiene un Descriptor que lo define unívocamente.
- Los punteros a los descriptores de proceso sirven para identificar-los. Son Números de 32 bits

Identificación de los procesos

- Cada proceso tiene un Descriptor que lo define unívocamente.
- Los punteros a los descriptores de proceso sirven para identificar-los. Son Números de 32 bits
- Los UNIX identifican a los procesos con un Process ID (PID). Los PID van desde 0 a 32767.

Identificación de los procesos

- Cada proceso tiene un Descriptor que lo define unívocamente.
- Los punteros a los descriptores de proceso sirven para identificar-los. Son Números de 32 bits
- Los UNIX identifican a los procesos con un Process ID (PID). Los PID van desde 0 a 32767.
- El Process ID corresponde al campo **`pid`** en el descriptor de proceso (**`task_struct->pid`**)

Estados de un proceso

Estados de un proceso

- Se definen en **`task_struct->state`**

Estados de un proceso

- Se definen en **`task_struct->state`**
- Valores posibles (flags)

Estados de un proceso

- Se definen en **`task_struct->state`**
- Valores posibles (flags)

```
#define  TASK_RUNNING            0
#define  TASK_INTERRUPTIBLE      1
#define  TASK_UNINTERRUPTIBLE    2
#define  TASK_STOPPED            4
#define  TASK_TRACED             8
#define  EXIT_ZOMBIE            16
#define  EXIT_DEAD              32
```

Descriptor de Proceso: Alocando memoria

Descriptor de Proceso: Alocando memoria

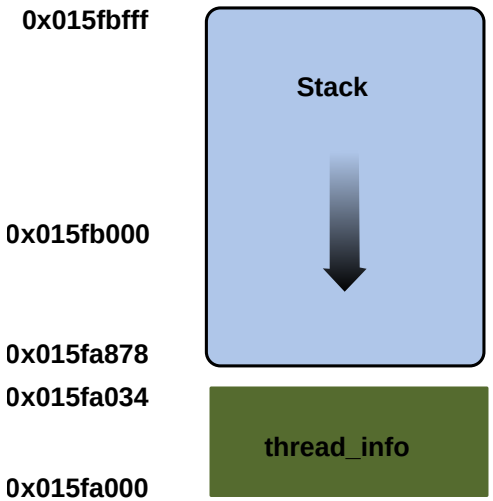
- En Kernel Mode, los procesos usan un Stack del Segmento de Datos del Kernel. El uso de este stack no es intensivo.

Descriptor de Proceso: Alocando memoria

- En Kernel Mode, los procesos usan un Stack del Segmento de Datos del Kernel. El uso de este stack no es intensivo.
- Los stacks expanden hacia la base de memoria (decrementa **`esp`**).

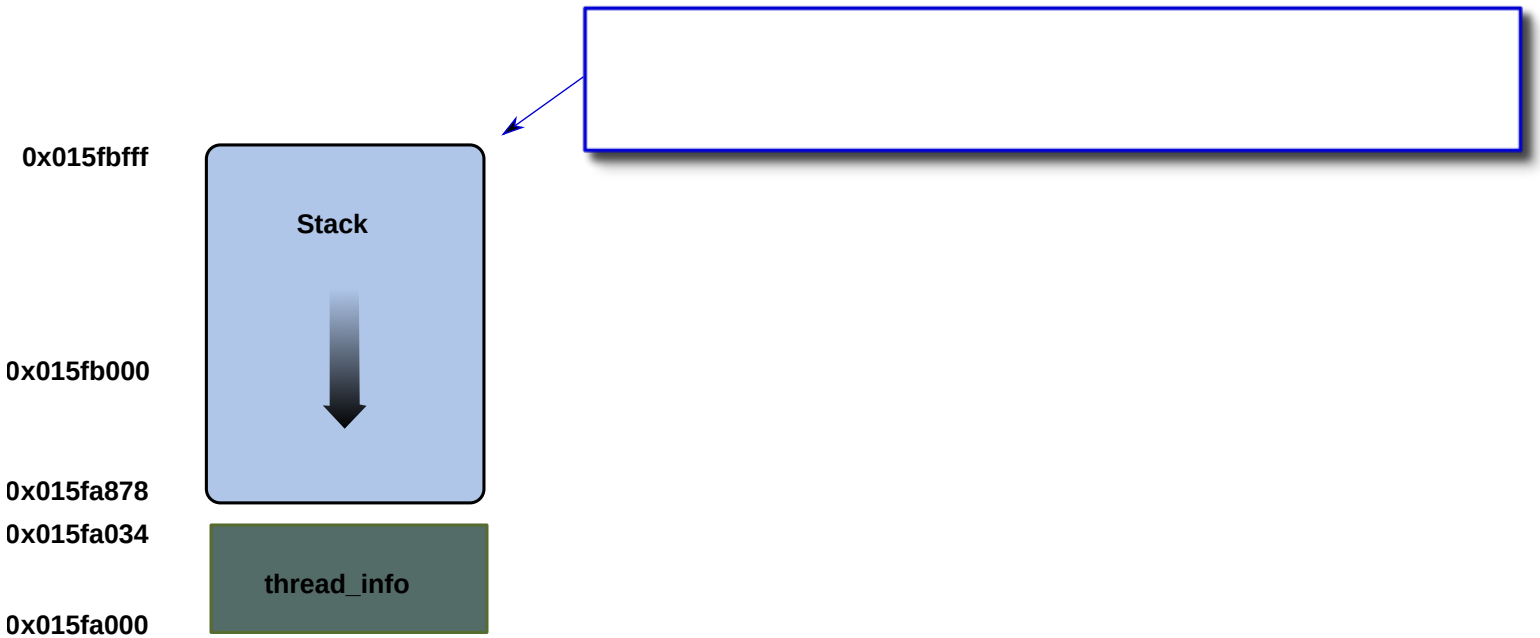
Descriptor de Proceso: Alocando memoria

- En Kernel Mode, los procesos usan un Stack del Segmento de Datos del Kernel. El uso de este stack no es intensivo.
- Los stacks expanden hacia la base de memoria (decrementa **`esp`**).
- Conociendo el registro **`esp`** se deduce el comienzo del Descriptor de Proceso.



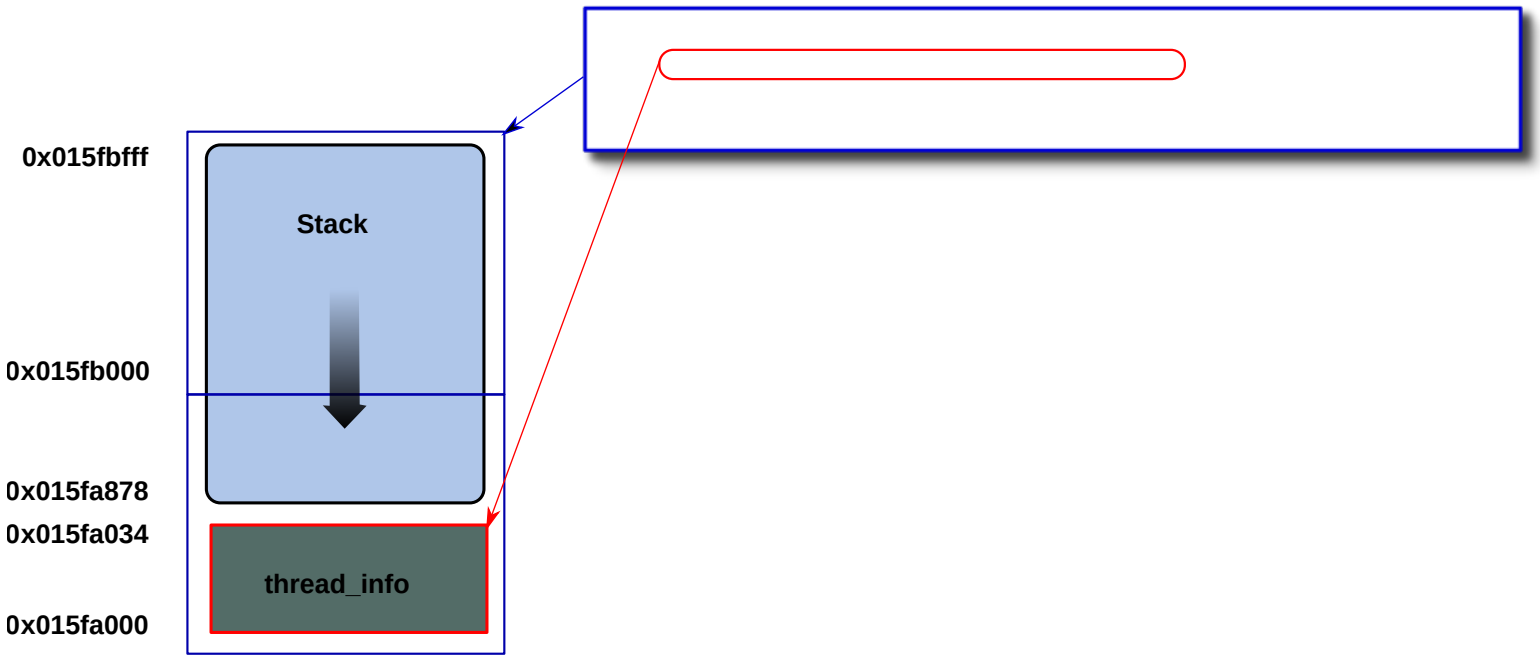
Descriptor de Proceso: Alocando memoria

- En Kernel Mode, los procesos usan un Stack del Segmento de Datos del Kernel. El uso de este stack no es intensivo.
- Los stacks expanden hacia la base de memoria (decrementa **`esp`**).
- Conociendo el registro **`esp`** se deduce el comienzo del Descriptor de Proceso.



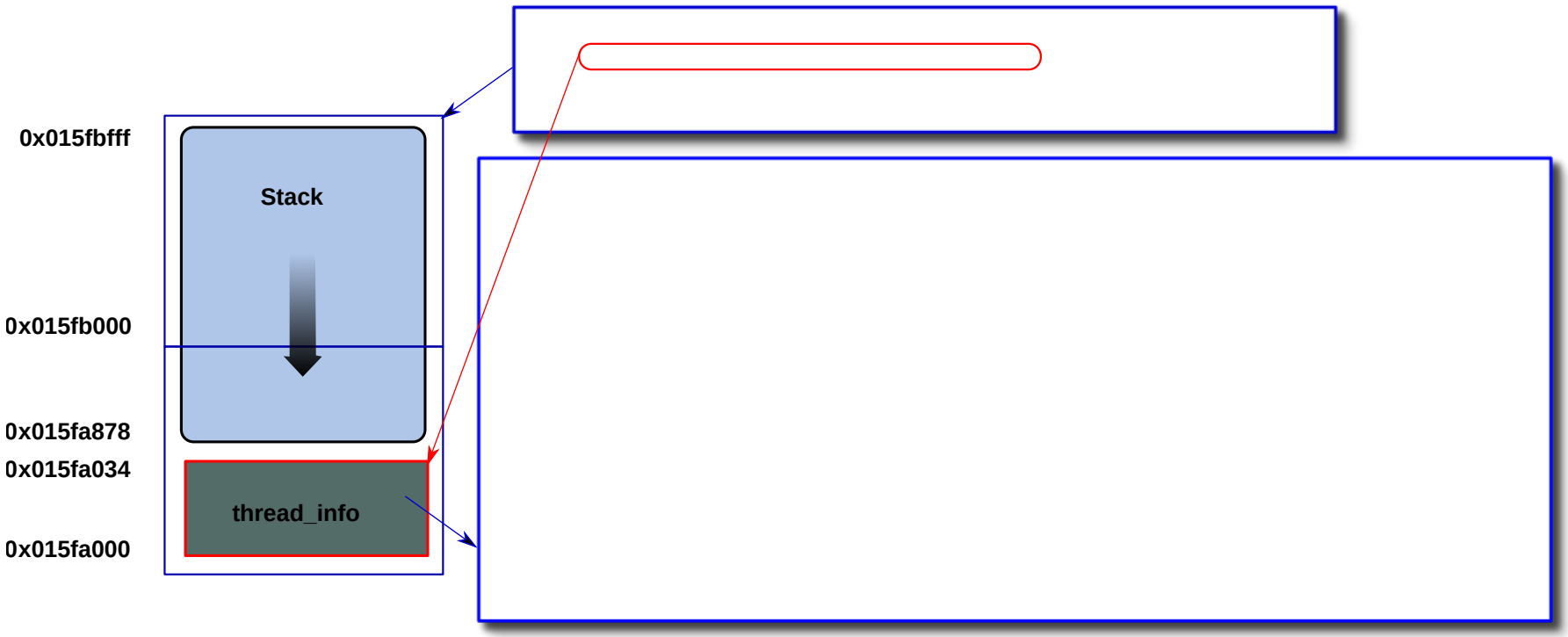
Descriptor de Proceso: Alocando memoria

- En Kernel Mode, los procesos usan un Stack del Segmento de Datos del Kernel. El uso de este stack no es intensivo.
- Los stacks expanden hacia la base de memoria (decrementa **`esp`**).
- Conociendo el registro **`esp`** se deduce el comienzo del Descriptor de Proceso.



Descriptor de Proceso: Alocando memoria

- En Kernel Mode, los procesos usan un Stack del Segmento de Datos del Kernel. El uso de este stack no es intensivo.
- Los stacks expanden hacia la base de memoria (decrementa **`esp`**).
- Conociendo el registro **`esp`** se deduce el comienzo del Descriptor de Proceso.



Ubicando la estructura thread

Ubicando la estructura thread

- Sirve para que el kernel pueda ubicar el inicio del Descriptor de Proceso. El código es:

Ubicando la estructura thread

- Sirve para que el kernel pueda ubicar el inicio del Descriptor de Proceso. El código es:

```
movl $0xffffe000,% ecx ;/* or 0xfffff000 for 4KB stacks */
andl %esp,% ecx
movl %ecx,p
```

Ubicando la estructura thread

- Sirve para que el kernel pueda ubicar el inicio del Descriptor de Proceso. El código es:

```
movl $0xffffe000,% ecx ;/* or 0xfffff000 for 4KB stacks */
andl %esp,% ecx
movl %ecx,p
```

- A la salida de este código, **p** contiene un puntero a **struc thread_i**

Ubicando la estructura thread

- Sirve para que el kernel pueda ubicar el inicio del Descriptor de Proceso. El código es:

```
movl $0xffffe000,% ecx ;/* or 0xfffff000 for 4KB stacks */
andl %esp,% ecx
movl %ecx,p
```

- A la salida de este código, **p** contiene un puntero a **struc thread_i**
- El primer miembro de **thread_info** es el puntero a **task_struct**.

Ubicando la estructura thread

- Sirve para que el kernel pueda ubicar el inicio del Descriptor de Proceso. El código es:

```
movl $0xffffe000,% ecx ;/* or 0xfffff000 for 4KB stacks */
andl %esp,% ecx
movl %ecx,p
```

- A la salida de este código, **p** contiene un puntero a **struc thread_i**
- El primer miembro de **thread_info** es el puntero a **task_struct**.
- Por compatibilidad con los kernels anteriores se dispone del alias a la macro **current**

Ubicando la estructura thread

- Sirve para que el kernel pueda ubicar el inicio del Descriptor de Proceso. El código es:

```
movl $0xffffe000,% ecx ;/* or 0xfffff000 for 4KB stacks */
andl %esp,% ecx
movl %ecx,p
```

- A la salida de este código, **p** contiene un puntero a **struc thread_i**
- El primer miembro de **thread_info** es el puntero a **task_struct**.
- Por compatibilidad con los kernels anteriores se dispone del alias a la macro **current**
- **current->pid** es una sentencia de lo mas común en todo el código del kernel para obtener el pid del proceso, por ejemplo.

Visualización de Procesos

alejandro@DarkSideOfTheMoon: ~/git/TDIII/slides/Arquitectura-Linux/LinuxArchIntro

File Edit View Search Terminal Help

```
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 56608 -      07:32 ?        00:00:19 /sbin/init
1 S root           2       0  0  80   0 -     0 -      07:32 ?        00:00:00 [kthreadd]
1 I root           4       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/0:0H]
1 I root           6       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [mm_percpu_wq]
1 S root           7       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/0]
1 I root           8       2  0  80   0 -     0 -      07:32 ?        00:00:35 [rcu_sched]
1 I root           9       2  0  80   0 -     0 -      07:32 ?        00:00:00 [rcu_bh]
1 S root          10       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/0]
5 S root          11       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/0]
1 S root          12       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/0]
1 S root          13       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/1]
5 S root          14       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/1]
1 S root          15       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/1]
1 S root          16       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/1]
1 I root          18       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/1:0H]
1 S root          19       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/2]
5 S root          20       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/2]
1 S root          21       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/2]
1 S root          22       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/2]
1 I root          24       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/2:0H]
1 S root          25       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/3]
5 S root          26       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/3]
1 S root          27       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/3]
1 S root          28       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/3]
1 I root          30       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/3:0H]
1 S root          31       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/4]
5 S root          32       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/4]
1 S root          33       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/4]
```

Visualización de Procesos

alejandro@DarkSideOfTheMoon: ~/git/TDIII/slides/Arquitectura-Linux/LinuxArchIntro

File Edit View Search Terminal Help

```
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 56608 -      07:32 ?        00:00:19 /sbin/init
1 S root           2       0  0  80   0 -     0 -      07:32 ?        00:00:00 [kthreadd]
1 I root           4       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/0:0H]
1 I root           6       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [mm_percpu_wq]
1 S root           7       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/0]
1 I root           8       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [rcu_sched]
1 I root           9       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [rcu_bh]
1 S root          10       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/0]
5 S root          11       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/0]
1 S root          12       2  0 -40   - -     0 -      07:32 ?        00:00:00 [cpuhp/0]
1 S root          13       2  0 -40   - -     0 -      07:32 ?        00:00:00 [cpuhp/1]
5 S root          14       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/1]
1 S root          15       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/1]
1 S root          16       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/1]
1 I root          18       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/1:0H]
1 S root          19       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/2]
5 S root          20       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/2]
1 S root          21       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/2]
1 S root          22       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/2]
1 I root          24       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/2:0H]
1 S root          25       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/3]
5 S root          26       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/3]
1 S root          27       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/3]
1 S root          28       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/3]
1 I root          30       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/3:0H]
1 S root          31       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/4]
5 S root          32       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/4]
1 S root          33       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/4]
```

F = Flags. Es la suma de los siguientes valores posibles:
1: Se creó con fork pero sin excec
4:Utilizado con permisos de root

Visualización de Procesos

alejandro@DarkSideOfTheMoon: ~/git/TDIII/slides/Arquitectura-Linux/LinuxArchIntro

File Edit View Search Terminal Help

```
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 56608 -      07:32 ?        00:00:19 /sbin/init
1 S root           2       0  0  80   0 -     0 -      07:32 ?        00:00:00 [kthreadd]
1 I root           4       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/0:0H]
1 I root           6       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [mm_percpu_wq]
1 S root
1 I root
1 I root
1 S root           1
5 S root           1
1 S root           1
1 S root           1
5 S root           1
1 S root           1
1 S root           1
1 I root           1
1 S root           1
5 S root           2
1 S root           2
1 S root           2
1 I root           2
1 S root           2
5 S root          26
1 S root          27       2  0 -40   -  -     0 -      07:32 ?        00:00:00 [migration/3]
1 S root          28       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/3]
1 I root          30       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/3:0H]
1 S root          31       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/4]
5 S root          32       2  0 -40   -  -     0 -      07:32 ?        00:00:00 [watchdog/4]
1 S root          33       2  0 -40   -  -     0 -      07:32 ?        00:00:00 [migration/4]
```

S = state.

- ***D*** *Uninterruptible sleep*
- ***R*** *Running or runnable (en run queue)*
- ***S*** *Interruptible sleep (esperando que se complete un evento)*
- ***T*** *Stopped, por señal SIGTOP o si está en siendo Traced.*
- ***W*** *paging (inválido desde kernel 2.6.xx)*
- ***X*** *dead (nunca deberíamos verlo)*
- ***Z*** *Defunct ("zombie")*

Visualización de Procesos

alejandro@DarkSideOfTheMoon: ~/git/TDIII/slides/Arquitectura-Linux/LinuxArchIntro

File Edit View Search Terminal Help

```
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 56608 -      07:32 ?        00:00:19 /sbin/init
1 S root           2       0  0  80   0 -     0 -      07:32 ?        00:00:00 [kthreadd]
1 I root           4       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/0:0H]
1 I root           6       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [mm_percpu_wq]
1 S root           7       2  0
1 I root           8       2  0
1 I root           9       2  0
1 S root          10       2  0
5 S root          11       2  0
1 S root          12       2  0
1 S root          13       2  0
5 S root          14       2  0 -40   -  -     0 -      07:32 ?        00:00:00 [watchdog/1]
1 S root          15       2  0 -40   -  -     0 -      07:32 ?        00:00:00 [migration/1]
1 S root          16       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/1]
1 I root          18       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/1:0H]
1 S root          19       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/2]
5 S root          20       2  0 -40   -  -     0 -      07:32 ?        00:00:00 [watchdog/2]
1 S root          21       2  0 -40   -  -     0 -      07:32 ?        00:00:00 [migration/2]
1 S root          22       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/2]
1 I root          24       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/2:0H]
1 S root          25       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/3]
5 S root          26       2  0 -40   -  -     0 -      07:32 ?        00:00:00 [watchdog/3]
1 S root          27       2  0 -40   -  -     0 -      07:32 ?        00:00:00 [migration/3]
1 S root          28       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/3]
1 I root          30       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/3:0H]
1 S root          31       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/4]
5 S root          32       2  0 -40   -  -     0 -      07:32 ?        00:00:00 [watchdog/4]
1 S root          33       2  0 -40   -  -     0 -      07:32 ?        00:00:00 [migration/4]
```

UID: User ID. Es quien ejecutó el proceso

PID: Process ID

PPID: Parent Process ID

Visualización de Procesos

alejandro@DarkSideOfTheMoon: ~/git/TDIII/slides/Arquitectura-Linux/LinuxArchIntro

File Edit View Search Terminal Help

```
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 56608 -      07:32 ?        00:00:19 /sbin/init
1 S root           2       0  0  80   0 -     0 -      07:32 ?        00:00:00 [kthreadd]
1 I root           4       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/0:0H]
1 I root           6       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [mm_percpu_wq]
1 S root           7       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/0]
1 I root           8       2  0  80   0 -     0 -      07:32 ?        00:00:00 [rcu_sched]
1 I root           9       2  0  80   0 -     0 -      07:32 ?        00:00:00 [rcu_bh]
1 S root          10       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/0]
5 S root          11       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/0]
1 S root          12       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/1]
1 S root          13       2  0  80   0 -     0 -      07:32 ?        00:00:00 [watchdog/1]
5 S root          14       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/1]
1 S root          15       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/1]
1 S root          16       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/1]
1 I root          18       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/1:0H]
1 S root          19       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/2]
5 S root          20       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/2]
1 S root          21       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/2]
1 S root          22       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/2]
1 I root          24       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/2:0H]
1 S root          25       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/3]
5 S root          26       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/3]
1 S root          27       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/3]
1 S root          28       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/3]
1 I root          30       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/3:0H]
1 S root          31       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/4]
5 S root          32       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/4]
1 S root          33       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/4]
```

C: Uso de CPU. Es el porcentaje de uso de CPU sobre el tiempo de vida del proceso

Visualización de Procesos

alejandro@DarkSideOfTheMoon: ~/git/TDIII/slides/Arquitectura-Linux/LinuxArchIntro

File Edit View Search Terminal Help

```
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 56608 -      07:32 ?        00:00:19 /sbin/init
1 S root           2       0  0  80   0 -     0 -      07:32 ?        00:00:00 [kthreadd]
1 I root           4       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/0:0H]
1 I root           6       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [mm_percpu_wq]
1 S root           7       2  0  8
1 I root           8       2  0  8
1 I root           9       2  0  8
1 S root          10       2  0 -4
5 S root          11       2  0 -4
1 S root          12       2  0  8
1 S root          13       2  0  8
5 S root          14       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/1]
1 S root          15       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/1]
1 S root          16       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/1]
1 I root          18       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/1:0H]
1 S root          19       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/2]
5 S root          20       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/2]
1 S root          21       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/2]
1 S root          22       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/2]
1 I root          24       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/2:0H]
1 S root          25       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/3]
5 S root          26       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/3]
1 S root          27       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/3]
1 S root          28       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/3]
1 I root          30       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/3:0H]
1 S root          31       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/4]
5 S root          32       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/4]
1 S root          33       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/4]
```

PRI: Prioridad. A mayor valor menor prioridad

NI: Valor nice. (-19 a 20). nice modifica la prioridad desde el shell

Visualización de Procesos

alejandro@DarkSideOfTheMoon: ~/git/TDIII/slides/Arquitectura-Linux/LinuxArchIntro

File Edit View Search Terminal Help

```
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 56608 -      07:32 ?        00:00:19 /sbin/init
1 S root           2       0  0  80   0 -     0 -      07:32 ?        00:00:00 [kthreadd]
1 I root           4       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/0:0H]
1 I root           6       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [mm_percpu_wq]
1 S root           7       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/0]
1 I root           8       2  0  80   0 -     0 -      07:32 ?        00:00:00 [rcu_sched]
1 I root           9       2  0  80   0 -     0 -      07:32 ?        00:00:00 [rcu_bh]
1 S root          10       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/0]
5 S root          11       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/0]
1 S root          12       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/1]
1 S root          13       2  0  80   0 -     0 -      07:32 ?        00:00:00 [watchdog/1]
5 S root          14       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/1]
1 S root          15       2  0 -40   - -     0 -      07:32 ?        00:00:00 [ksoftirqd/1]
1 S root          16       2  0  80   0 -     0 -      07:32 ?        00:00:00 [kworker/1:0]
1 I root          18       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/1:0H]
1 S root          19       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/2]
5 S root          20       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/2]
1 S root          21       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/2]
1 S root          22       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/2]
1 I root          24       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/2:0H]
1 S root          25       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/3]
5 S root          26       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/3]
1 S root          27       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/3]
1 S root          28       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/3]
1 I root          30       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/3:0H]
1 S root          31       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/4]
5 S root          32       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/4]
1 S root          33       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/4]
```

ADDR: Dirección de memoria del proceso.

SZ: tamaño en páginas físicas de la imagen core del proceso. Esto incluye el espacio para secciones text, data, y stack.

Visualización de Procesos

alejandro@DarkSideOfTheMoon: ~/git/TDIII/slides/Arquitectura-Linux/LinuxArchIntro

File Edit View Search Terminal Help

```
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 56608 -      07:32 ?        00:00:19 /sbin/init
1 S root           2       0  0  80   0 -     0 -      07:32 ?        00:00:00 [kthreadd]
1 I root           4       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/0:0H]
1 I root           6       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [mm_percpu_wq]
1 S root           7       2  0  8
1 I root           8       2  0  8
1 I root           9       2  0  8
1 S root          10       2  0 -4
5 S root          11       2  0 -4
1 S root          12       2  0  8
1 S root          13       2  0  8
5 S root          14       2  0 -4
1 S root          15       2  0 -4
1 S root          16       2  0  8
1 I root          18       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/1:0H]
1 S root          19       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/2]
5 S root          20       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/2]
1 S root          21       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/2]
1 S root          22       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/2]
1 I root          24       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/2:0H]
1 S root          25       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/3]
5 S root          26       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/3]
1 S root          27       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/3]
1 S root          28       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/3]
1 I root          30       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/3:0H]
1 S root          31       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/4]
5 S root          32       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/4]
1 S root          33       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/4]
```

WCHAN: Nombre de la función del kernel en la que el proceso está durmiendo y – si el proceso está Running.

Visualización de Procesos

alejandro@DarkSideOfTheMoon: ~/git/TDIII/slides/Arquitectura-Linux/LinuxArchIntro

File Edit View Search Terminal Help

```
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 - 56608 -      07:32 ?        00:00:19 /sbin/init
1 S root           2       0  0  80   0 -     0 -      07:32 ?        00:00:00 [kthreadd]
1 I root           4       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/0:0H]
1 I root           6       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [mm_percpu_wq]
1 S root           7       2  0  8
1 I root           8       2  0  8
1 I root           9       2  0  8
1 S root          10       2  0 -4
5 S root          11       2  0 -4
1 S root          12       2  0  8
1 S root          13       2  0  8
5 S root          14       2  0 -4
1 S root          15       2  0 -4
1 S root          16       2  0  8
1 I root          18       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [rcu_par_gp/1:0H]
1 S root          19       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/2]
5 S root          20       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/2]
1 S root          21       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/2]
1 S root          22       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/2]
1 I root          24       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/2:0H]
1 S root          25       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/3]
5 S root          26       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/3]
1 S root          27       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/3]
1 S root          28       2  0  80   0 -     0 -      07:32 ?        00:00:00 [ksoftirqd/3]
1 I root          30       2  0  60 -20 -     0 -      07:32 ?        00:00:00 [kworker/3:0H]
1 S root          31       2  0  80   0 -     0 -      07:32 ?        00:00:00 [cpuhp/4]
5 S root          32       2  0 -40   - -     0 -      07:32 ?        00:00:00 [watchdog/4]
1 S root          33       2  0 -40   - -     0 -      07:32 ?        00:00:00 [migration/4]
```

***STIME:** Fecha de creación.*

***TTY:** Terminal asociada.*

***TIME:** Tiempo de CPU acumulado(por el kernel y el proceso)hh:mm:ss.*

***CMD:** comando ejecutado para crear el proceso.*

Creación de procesos

Creación de procesos

- En POSIX un proceso se crea mediante la syscall `fork()`.

Creación de procesos

- En POSIX un proceso se crea mediante la syscall `fork()`.
- El proceso creado es el duplicado del que invoca a `fork()` para crearlo, y es además su proceso hijo.

Creación de procesos

- En POSIX un proceso se crea mediante la syscall `fork()`.
- El proceso creado es el duplicado del que invoca a `fork()` para crearlo, y es además su proceso hijo.
- El proceso padre continúa su ejecución justo luego de `fork()`

Creación de procesos

- En POSIX un proceso se crea mediante la syscall `fork()`.
- El proceso creado es el duplicado del que invoca a `fork()` para crearlo, y es además su proceso hijo.
- El proceso padre continúa su ejecución justo luego de `fork()`
- El proceso hijo *inicia* su ejecución en ese mismo punto pero en una instancia (proceso) diferente.

Creación de procesos

- En POSIX un proceso se crea mediante la syscall `fork()`.
- El proceso creado es el duplicado del que invoca a `fork()` para crearlo, y es además su proceso hijo.
- El proceso padre continúa su ejecución justo luego de `fork()`
- El proceso hijo *inicia* su ejecución en ese mismo punto pero en una instancia (proceso) diferente.
- `fork()` retorna dos veces, una para cada proceso y devuelve valores diferentes de acuerdo a quien retorne.

Creación de procesos

- En POSIX un proceso se crea mediante la syscall **`fork()`**.
- El proceso creado es el duplicado del que invoca a **`fork()`** para crearlo, y es además su proceso hijo.
- El proceso padre continúa su ejecución justo luego de **`fork()`**
- El proceso hijo *inicia* su ejecución en ese mismo punto pero en una instancia (proceso) diferente.
- **`fork()`** retorna dos veces, una para cada proceso y devuelve valores diferentes de acuerdo a quien retorne.
- En Linux **`fork()`** se implementa con **`clone()`**

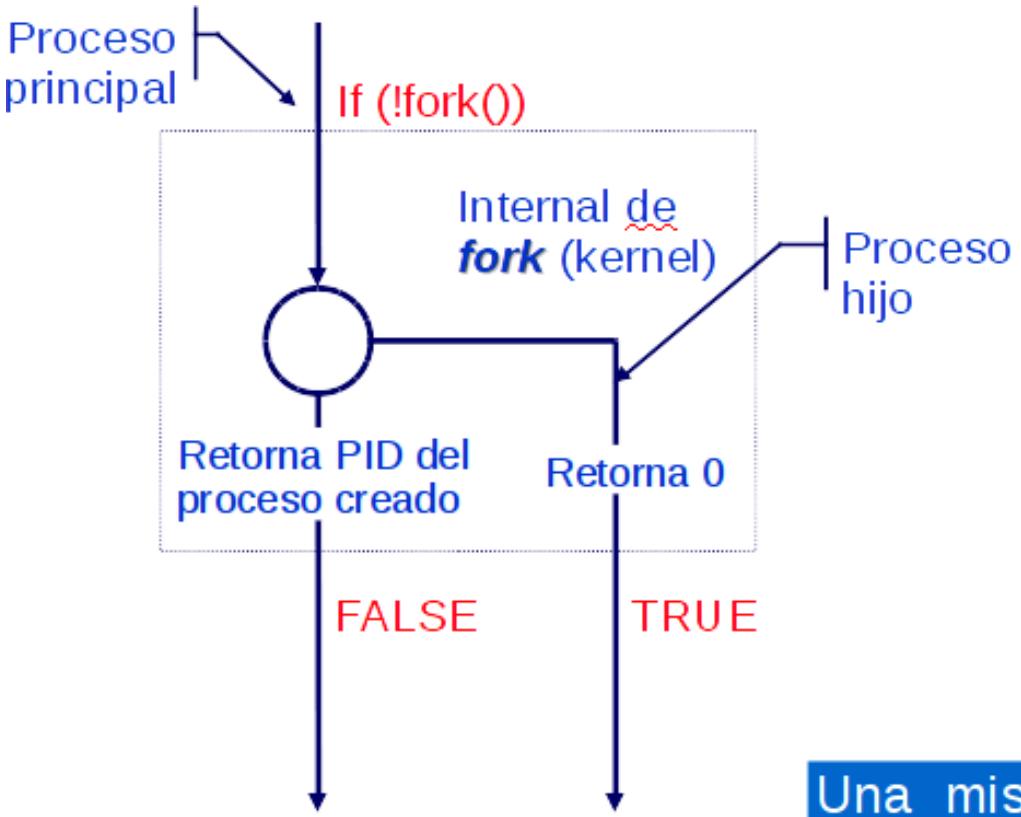
Creación de procesos

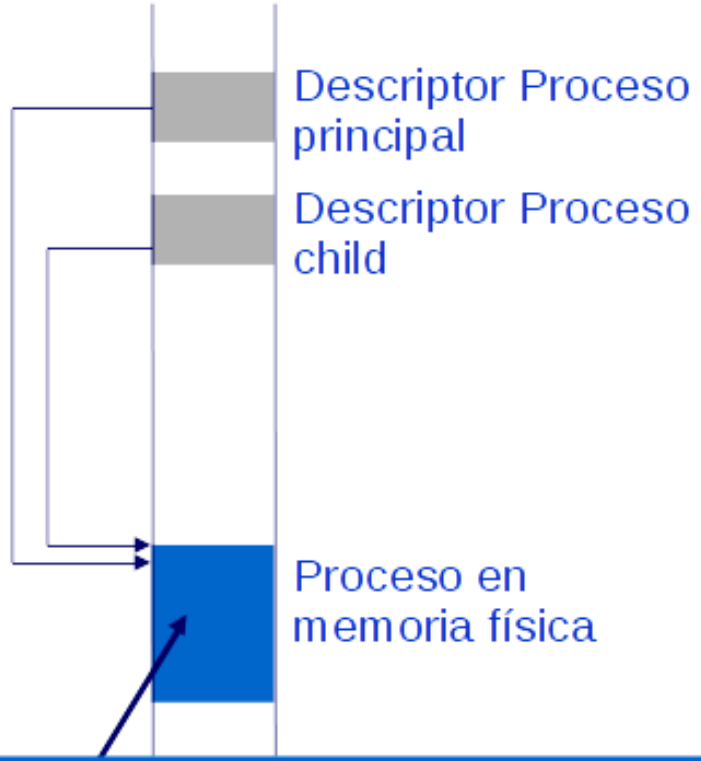
- En POSIX un proceso se crea mediante la syscall **`fork()`**.
- El proceso creado es el duplicado del que invoca a **`fork()`** para crearlo, y es además su proceso hijo.
- El proceso padre continúa su ejecución justo luego de **`fork()`**
- El proceso hijo *inicia* su ejecución en ese mismo punto pero en una instancia (proceso) diferente.
- **`fork()`** retorna dos veces, una para cada proceso y devuelve valores diferentes de acuerdo a quien retorne.
- En Linux **`fork()`** se implementa con **`clone()`**
- Si se desea ejecutar un programa diferente en lugar de la copia del mismo programa, la familia de funciones **`exec()`** se encargará de reemplazar el código, los datos, y la pila del proceso duplicado, por los del programa deseado.

Creación de procesos: fork ()

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  int main(void)
5  {
6    pid_t new_pid;
7    new_pid = fork();
8    switch(new_pid)
9    {
10     case -1 :    /* Error */
11         perror ("fork:");
12         exit (1);
13         break;
14     case 0 :     /* Proceso hijo*/
15         printf ("Hola Mundo!: Soy el proceso hijo!. Mi pid es: % d.\n",
               getpid());
16         break;
17     default :    /* Proceso padre*/
18         printf ("Hola Mundo!: Soy el proceso padre!. Mi hijo es % d.\n"
               ,new_pid);
19         break;
20   }
21 }
```

Creación de procesos: fork ()





Una misma copia física del código y los datos en memoria apuntada por dos descriptores de proceso diferentes.
Lightweight Process: Una copia = Dos procesos.

forking

forking

- En Linux, la syscall `fork()` se construye en base a un procedimiento conocido como ***copy-on-write-pages***

forking

- En Linux, la syscall `fork()` se construye en base a un procedimiento conocido como ***copy-on-write-pages***
- En principio no se duplica su espacio de ejecución. Esto es: ambos procesos (padre e hijo) comparten el mismo espacio de direccionamiento para código, datos y pila.

forking

- En Linux, la syscall `fork()` se construye en base a un procedimiento conocido como ***copy-on-write-pages***
- En principio no se duplica su espacio de ejecución. Esto es: ambos procesos (padre e hijo) comparten el mismo espacio de direccionamiento para código, datos y pila.
- Cuando se ejecuta el código de la syscall `fork()`, el kernel solo crea una estructura de tablas de página para el nuevo proceso que es el duplicado fiel de las del proceso invocado.

forking

- En Linux, la syscall `fork()` se construye en base a un procedimiento conocido como ***copy-on-write-pages***
- En principio no se duplica su espacio de ejecución. Esto es: ambos procesos (padre e hijo) comparten el mismo espacio de direccionamiento para código, datos y pila.
- Cuando se ejecuta el código de la syscall `fork()`, el kernel solo crea una estructura de tablas de página para el nuevo proceso que es el duplicado fiel de las del proceso invocado.
- Si el child de inmediato invoca a una función de la familia `exec()`, el kernel le crea su propia estructura de páginas eliminando los duplicados con su padre (es decir, le crea un espacio de direccionamiento separado del de su poroceso padre).

forking

- El único costo de memoria que implica `fork()` es duplicar (literalmente) las tablas de página del proceso padre, asignandole la copia al hijo, y crear un `task_struct` , es decir allocar en memoria una nueva, en la que gran parte de sus miembros se copian desde de la estructura `task_struct` del padre.

forking

- El único costo de memoria que implica `fork()` es duplicar (literalmente) las tablas de página del proceso padre, asignandole la copia al hijo, y crear un `task_struct` , es decir allocar en memoria una nueva, en la que gran parte de sus miembros se copian desde de la estructura `task_struct` del padre.
- De este modo cada proceso ejecuta el mismo bloque de código.

forking

- El único costo de memoria que implica `fork()` es duplicar (literalmente) las tablas de página del proceso padre, asignandole la copia al hijo, y crear un `task_struct` , es decir allocar en memoria una nueva, en la que gran parte de sus miembros se copian desde de la estructura `task_struct` del padre.
- De este modo cada proceso ejecuta el mismo bloque de código.
- En el caso del código esto tiene algún sentido, considerando que en cada BCP se mantiene siempre una estructura con los valores que tenían los registros de la CPU la última vez que el proceso fue suspendido por el kernel.

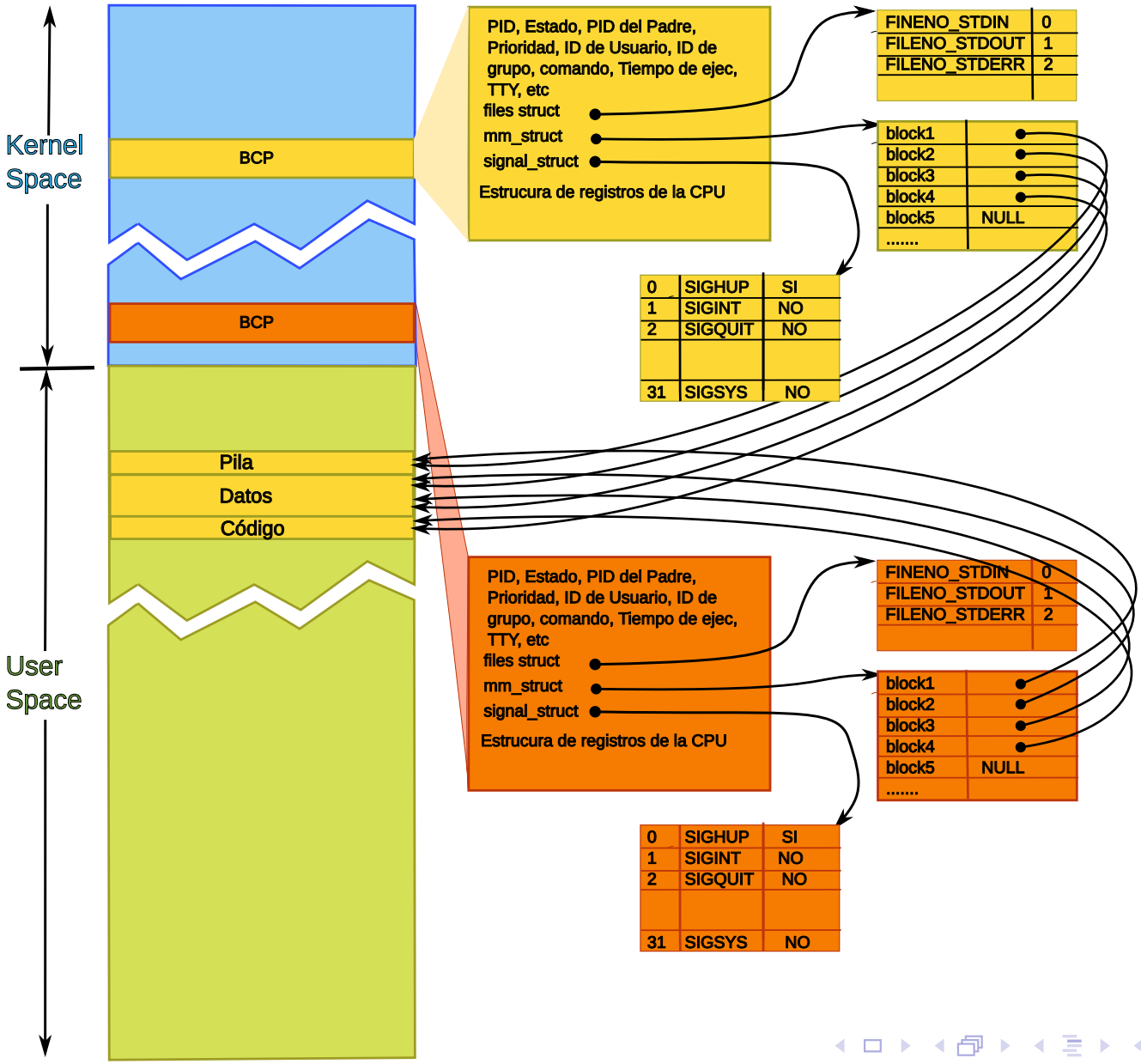
forking

- El único costo de memoria que implica `fork()` es duplicar (literalmente) las tablas de página del proceso padre, asignandole la copia al hijo, y crear un `task_struct` , es decir allocar en memoria una nueva, en la que gran parte de sus miembros se copian desde de la estructura `task_struct` del padre.
- De este modo cada proceso ejecuta el mismo bloque de código.
- En el caso del código esto tiene algún sentido, considerando que en cada BCP se mantiene siempre una estructura con los valores que tenían los registros de la CPU la última vez que el proceso fue suspendido por el kernel.
- Además el proceso padre le transfiere a su child sus recursos (file descriptors), variables, manejadores de señales, y demás.

forking

- El único costo de memoria que implica `fork()` es duplicar (literalmente) las tablas de página del proceso padre, asignandole la copia al hijo, y crear un `task_struct` , es decir allocar en memoria una nueva, en la que gran parte de sus miembros se copian desde de la estructura `task_struct` del padre.
- De este modo cada proceso ejecuta el mismo bloque de código.
- En el caso del código esto tiene algún sentido, considerando que en cada BCP se mantiene siempre una estructura con los valores que tenían los registros de la CPU la última vez que el proceso fue suspendido por el kernel.
- Además el proceso padre le transfiere a su child sus recursos (file descriptors), variables, manejadores de señales, y demás.
- La situación es la siguiente:

Imagen de memoria de los procesos padre e hijo



forking

forking

- Como respuesta a `fork()`, el kernel invoca a `do_fork()`, definida en `kernel/fork.c`

forking

- Como respuesta a `fork()`, el kernel invoca a `do_fork()`, definida en `kernel/fork.c`
- A su vez `do_fork()` invoca a `copy_process()`, que dá los pasos principales, que consisten en:

forking

- Como respuesta a **`fork()`**, el kernel invoca a **`do_fork()`**, definida en **`kernel/fork.c`**
- A su vez **`do_fork()`** invoca a **`copy_process()`**, que dá los pasos principales, que consisten en:
- Invocar **`dup_task_struct()`** Crea el stack del kernel, la estructura **`task_info`**, y la **`task_struct`** para el nuevo proceso.

forking

- Como respuesta a `fork()`, el kernel invoca a `do_fork()`, definida en `kernel/fork.c`
- A su vez `do_fork()` invoca a `copy_process()`, que dá los pasos principales, que consisten en:
- Invocar `dup_task_struct()` Crea el stack del kernel, la estructura `task_info`, y la `task_struct` para el nuevo proceso.
- Controla que con el nuevo child no se supere el límite de cantidad máxima de procesos del usuario que ejecuta el padre.

forking

- Como respuesta a **`fork()`**, el kernel invoca a **`do_fork()`**, definida en **`kernel/fork.c`**
- A su vez **`do_fork()`** invoca a **`copy_process()`**, que dá los pasos principales, que consisten en:
- Invocar **`dup_task_struct()`** Crea el stack del kernel, la estructura **`task_info`**, y la **`task_struct`** para el nuevo proceso.
- Controla que con el nuevo child no se supere el límite de cantidad máxima de procesos del usuario que ejecuta el padre.
- Modificar aquellos miembros de **`task_struct`** que le dan identidad al nuevo proceso (la mayor parte permanece sin cambios).

forking

- Como respuesta a **`fork()`**, el kernel invoca a **`do_fork()`**, definida en **`kernel/fork.c`**
- A su vez **`do_fork()`** invoca a **`copy_process()`**, que dá los pasos principales, que consisten en:
- Invocar **`dup_task_struct()`** Crea el stack del kernel, la estructura **`task_info`**, y la **`task_struct`** para el nuevo proceso.
- Controla que con el nuevo child no se supere el límite de cantidad máxima de procesos del usuario que ejecuta el padre.
- Modificar aquellos miembros de **`task_struct`** que le dan identidad al nuevo proceso (la mayor parte permanece sin cambios).
- Pone al chid en estado **`TASK_UNINTERRUPTIBLE`** para asegurar la atomicidad de lo que sigue.

forking

- Invoca la función **`copy_flags`** para actualizar el miembro **`flags`** de **`task_struct`** . El flag **`PF_SUPERPRIV`** se limpia (denota cuando una tarea tiene privilegios de superuser, **`PF_FORKNOEXEC`** se pone en 1 para indicar que no se ejecutó **`exec()`** sino **`fork()`**.

forking

- Invoca la función **`copy_flags`** para actualizar el miembro **`flags`** de **`task_struct`** . El flag **`PF_SUPERPRIV`** se limpia (denota cuando una tarea tiene privilegios de superuser, **`PF_FORKNOEXEC`** se pone en 1 para indicar que no se ejecutó **`exec()`** sino **`fork()`**.
- Invoca la función **`alloc_pid()`**, que busca el primer número de PID disponible, y lo asigna al miembro **`pid`** de **`task_struct`** .

forking

- Invoca la función **`copy_flags`** para actualizar el miembro **`flags`** de **`task_struct`** . El flag **`PF_SUPERPRIV`** se limpia (denota cuando una tarea tiene privilegios de superuser, **`PF_FORKNOEXEC`** se pone en 1 para indicar que no se ejecutó **`exec()`** sino **`fork()`**.
- Invoca la función **`alloc_pid()`**, que busca el primer número de PID disponible, y lo asigna al miembro **`pid`** de **`task_struct`** .
- Linux posee una syscall propia llamada **`clone()`** que no es POSIX pero hace lo mismo que **`fork()`** con algunas particulariades como por ejemplo pasar ciertos flags para darle mas flexibilidad.

forking

- Invoca la función **copy_flags** para actualizar el miembro **flags** de **task_struct** . El flag **PF_SUPERPRIV** se limpia (denota cuando una tarea tiene privilegios de superuser, **PF_FORKNOEXEC** se pone en 1 para indicar que no se ejecutó **exec()** sino **fork()**.
- Invoca la función **alloc_pid()**, que busca el primer número de PID disponible, y lo asigna al miembro **pid** de **task_struct** .
- Linux posee una syscall propia llamada **clone()** que no es POSIX pero hace lo mismo que **fork()** con algunas particulariades como por ejemplo pasar ciertos flags para darle mas flexibilidad.
- Dependiendo de esos flags **copy_process**, duplica o comparte los file descriptor, información del file system, handlers de señales, espacio de memoria, y el namespace.

forking

- Invoca la función **copy_flags** para actualizar el miembro **flags** de **task_struct** . El flag **PF_SUPERPRIV** se limpia (denota cuando una tarea tiene privilegios de superuser, **PF_FORKNOEXEC** se pone en 1 para indicar que no se ejecutó **exec()** sino **fork()**.
- Invoca la función **alloc_pid()**, que busca el primer número de PID disponible, y lo asigna al miembro **pid** de **task_struct** .
- Linux posee una syscall propia llamada **clone()** que no es POSIX pero hace lo mismo que **fork()** con algunas particulariades como por ejemplo pasar ciertos flags para darle mas flexibilidad.
- Dependiendo de esos flags **copy_process**, duplica o comparte los file descriptor, información del file system, handlers de señales, espacio de memoria, y el namespace.
- **fork()** no utiliza flags de modo que el default es compartir estos datos.

forking

- Retorna a `do_fork()` el puntero al nuevo `task_struct` .

forking

- Retorna a `do_fork()` el puntero al nuevo `task_struct` .
- `do_fork()` , despierta al proceso child y lo lanza a ejecutar, deliberadamente antes que al padre.

forking

- Retorna a `do_fork()` el puntero al nuevo `task_struct` .
- `do_fork()` , despierta al proceso child y lo lanza a ejecutar, deliberadamente antes que al padre.
- Si el nuevo proceso llama a `exec()` entonces se elimina cualquier rastro de `copy-on-write` por si el padre recupera su ejecución antes y utiliza algún recurso.

Copy On Write

Copy On Write

- Luego de `fork()` los procesos padre e hijo comparten el mismo espacio de direccionamiento.

Copy On Write

- Luego de `fork()` los procesos padre e hijo comparten el mismo espacio de direccionamiento.
- Esta propiedad, para el código nadie duda que es óptima.

Copy On Write

- Luego de `fork()` los procesos padre e hijo comparten el mismo espacio de direccionamiento.
- Esta propiedad, para el código nadie duda que es óptima.
- ¿El criterio aplicado al código es igualmente válido con los datos y la pila?

Copy On Write

- Luego de `fork()` los procesos padre e hijo comparten el mismo espacio de direccionamiento.
- Esta propiedad, para el código nadie duda que es óptima.
- ¿El criterio aplicado al código es igualmente válido con los datos y la pila?
- Respuesta: obviamente NO.

Copy On Write

- Luego de `fork()` los procesos padre e hijo comparten el mismo espacio de direccionamiento.
- Esta propiedad, para el código nadie duda que es óptima.
- ¿El criterio aplicado al código es igualmente válido con los datos y la pila?
- Respuesta: obviamente NO.
- Cada vez que uno los procesos con parentesco modifica una variable local, estática o global, el kernel detecta el acceso y le crea una copia propia al proceso que haga la modificación.

Copy On Write

- Luego de `fork()` los procesos padre e hijo comparten el mismo espacio de direccionamiento.
- Esta propiedad, para el código nadie duda que es óptima.
- ¿El criterio aplicado al código es igualmente válido con los datos y la pila?
- Respuesta: obviamente NO.
- Cada vez que uno los procesos con parentesco modifica una variable local, estática o global, el kernel detecta el acceso y le crea una copia propia al proceso que haga la modificación.
- Este mecanismo se llama ***Copy on write***.

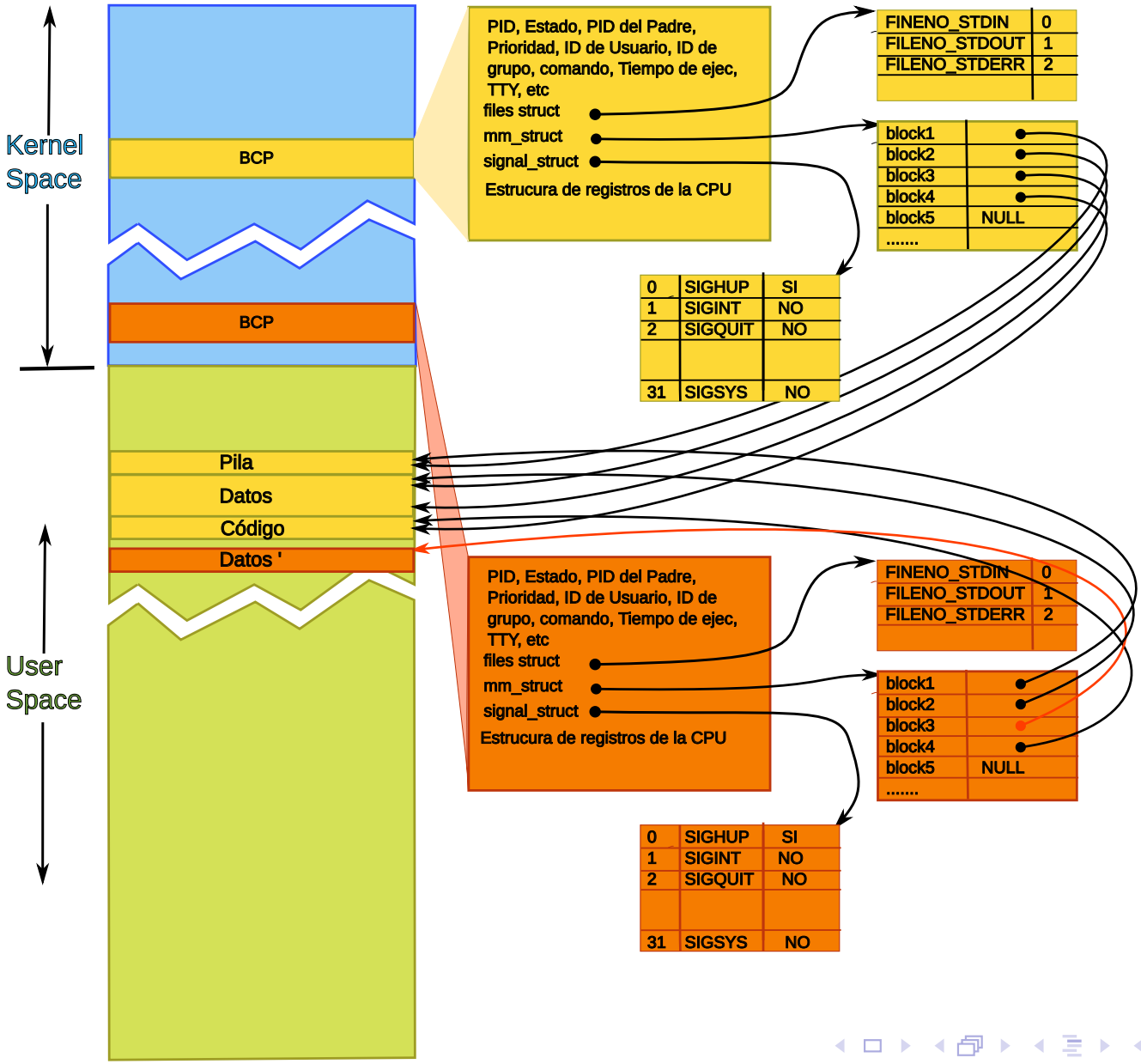
Copy On Write

- Luego de `fork()` los procesos padre e hijo comparten el mismo espacio de direccionamiento.
- Esta propiedad, para el código nadie duda que es óptima.
- ¿El criterio aplicado al código es igualmente válido con los datos y la pila?
- Respuesta: obviamente NO.
- Cada vez que uno los procesos con parentesco modifica una variable local, estática o global, el kernel detecta el acceso y le crea una copia propia al proceso que haga la modificación.
- Este mecanismo se llama ***Copy on write***.
- Este criterio se conoce como ***Lightweight processes*** (como lo mencionamos anteriormente).

Copy On Write

- Luego de `fork()` los procesos padre e hijo comparten el mismo espacio de direccionamiento.
- Esta propiedad, para el código nadie duda que es óptima.
- ¿El criterio aplicado al código es igualmente válido con los datos y la pila?
- Respuesta: obviamente NO.
- Cada vez que uno los procesos con parentesco modifica una variable local, estática o global, el kernel detecta el acceso y le crea una copia propia al proceso que haga la modificación.
- Este mecanismo se llama ***Copy on write***.
- Este criterio se conoce como ***Lightweight processes*** (como lo mencionamos anteriormente).
- Consiste en al inicio solo crear las estructuras del kernel (BCP), y compartir todo lo posible del espacio de direcciones en modo User.

Copy On Write



vfork ()

vfork ()

- `vfork()` es una función de POSIX que en geneal funciona como `fork()`, con las siguientes diferencias:

vfork ()

- `vfork()` es una función de POSIX que en geneal funciona como `fork()`, con las siguientes diferencias:
- `vfork()` no hace una copia de las tablas de página del proceso sino que el child ejecuta en el mismo espacio de direcciones del padre.

vfork ()

- `vfork()` es una función de POSIX que en geneal funciona como `fork()`, con las siguientes diferencias:
- `vfork()` no hace una copia de las tablas de página del proceso sino que el child ejecuta en el mismo espacio de direcciones del padre.
- El padre queda bloqueado (`TASK_INTERRUPTIBLE`) hasta que el hijo ejecute `exec()` o eventualmente `exit()`

vfork ()

- **vfork()** es una función de POSIX que en geneal funciona como **fork()**, con las siguientes diferencias:
- **vfork()** no hace una copia de las tablas de página del proceso sino que el child ejecuta en el mismo espacio de direcciones del padre.
- El padre queda bloqueado (**TASK_INTERRUPTIBLE**) hasta que el hijo ejecute **exec()** o eventualmente **exit()**
- El child no puede escribir en el espacio de direccionamiento. Si lo hace el resultado es imprevisible.

Diagrama de estados de un proceso

Diagrama de estados de un proceso

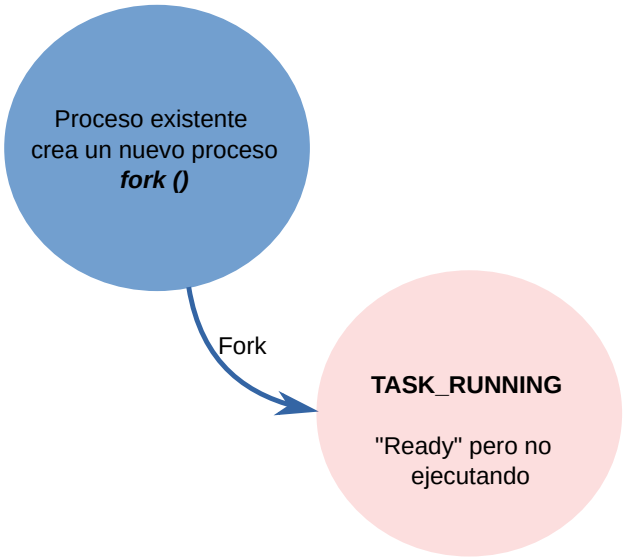


Diagrama de estados de un proceso

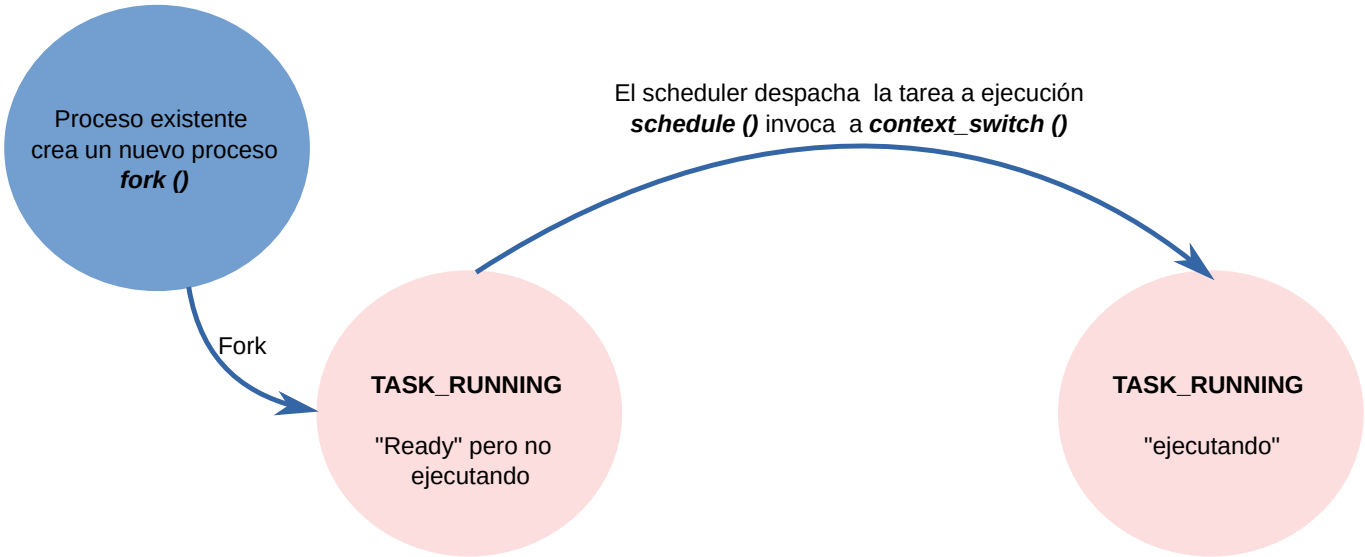


Diagrama de estados de un proceso

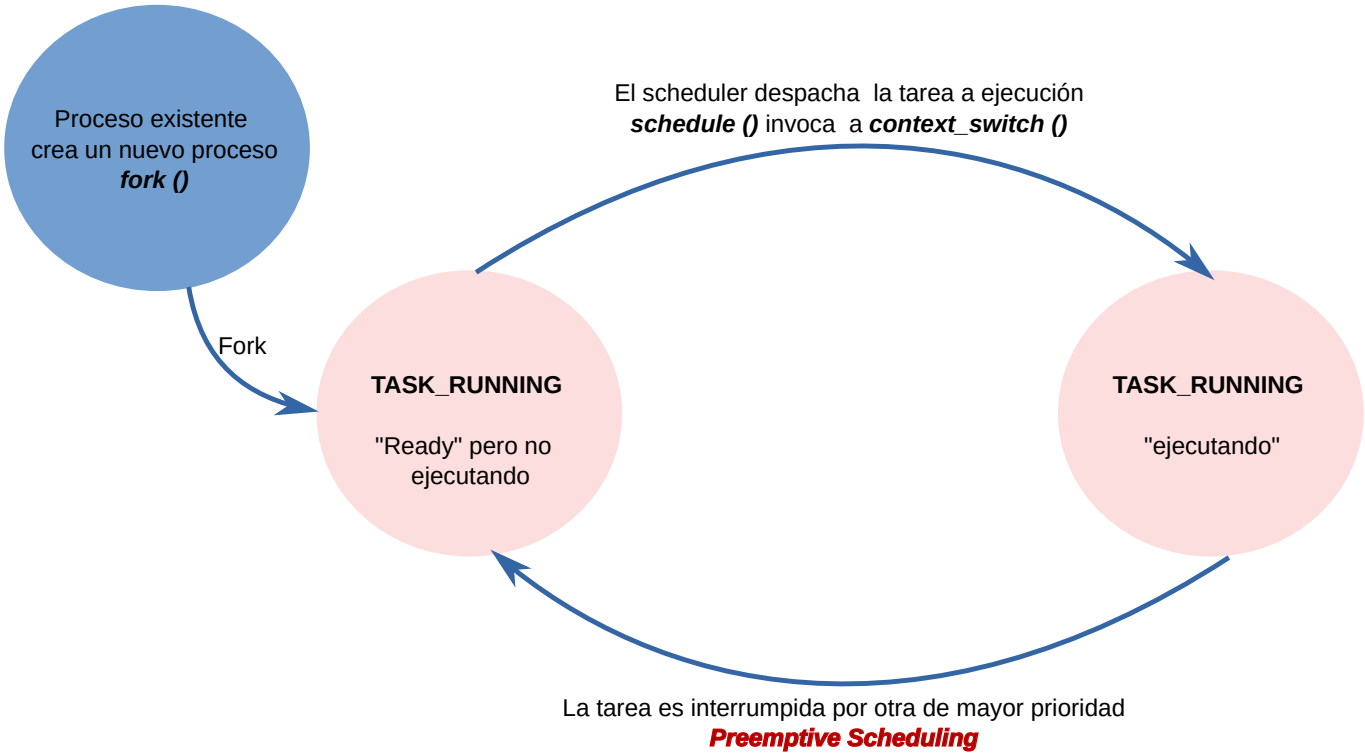


Diagrama de estados de un proceso

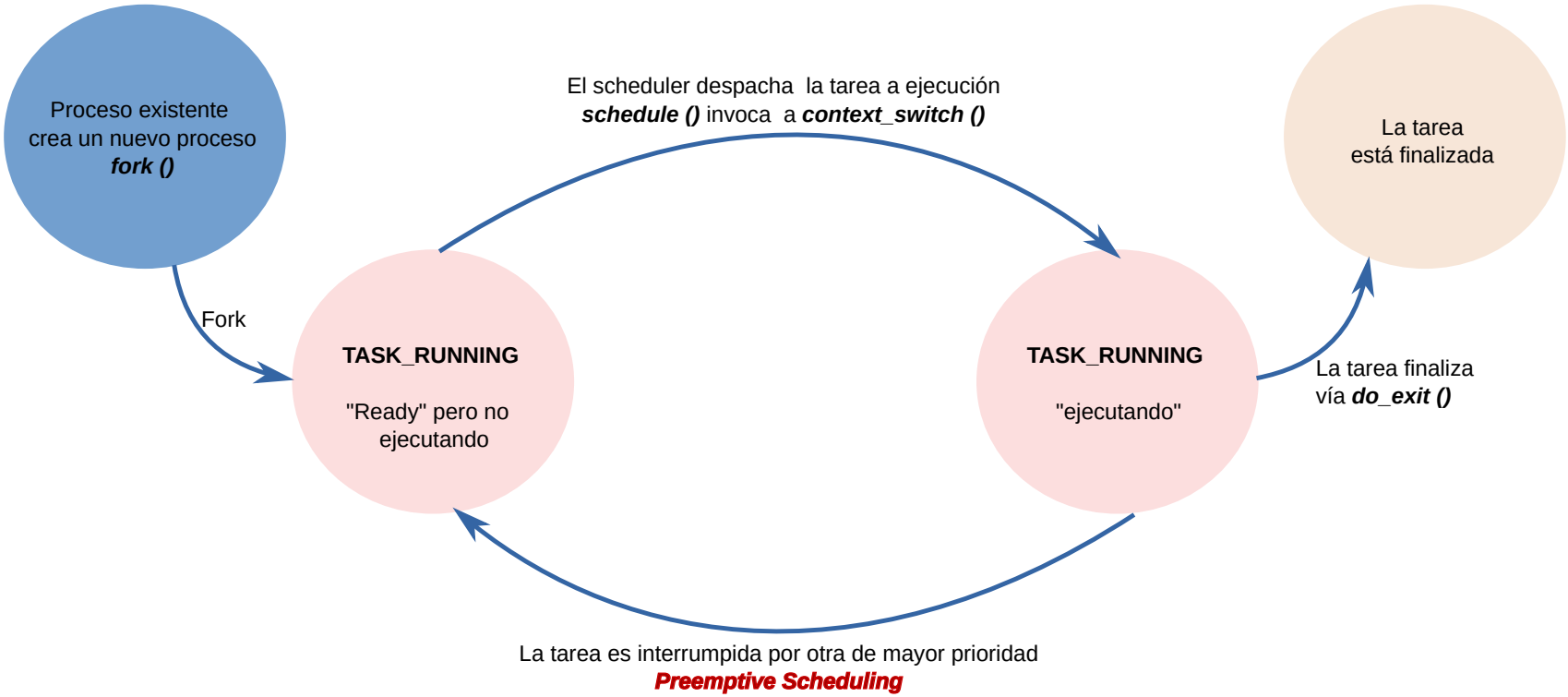


Diagrama de estados de un proceso

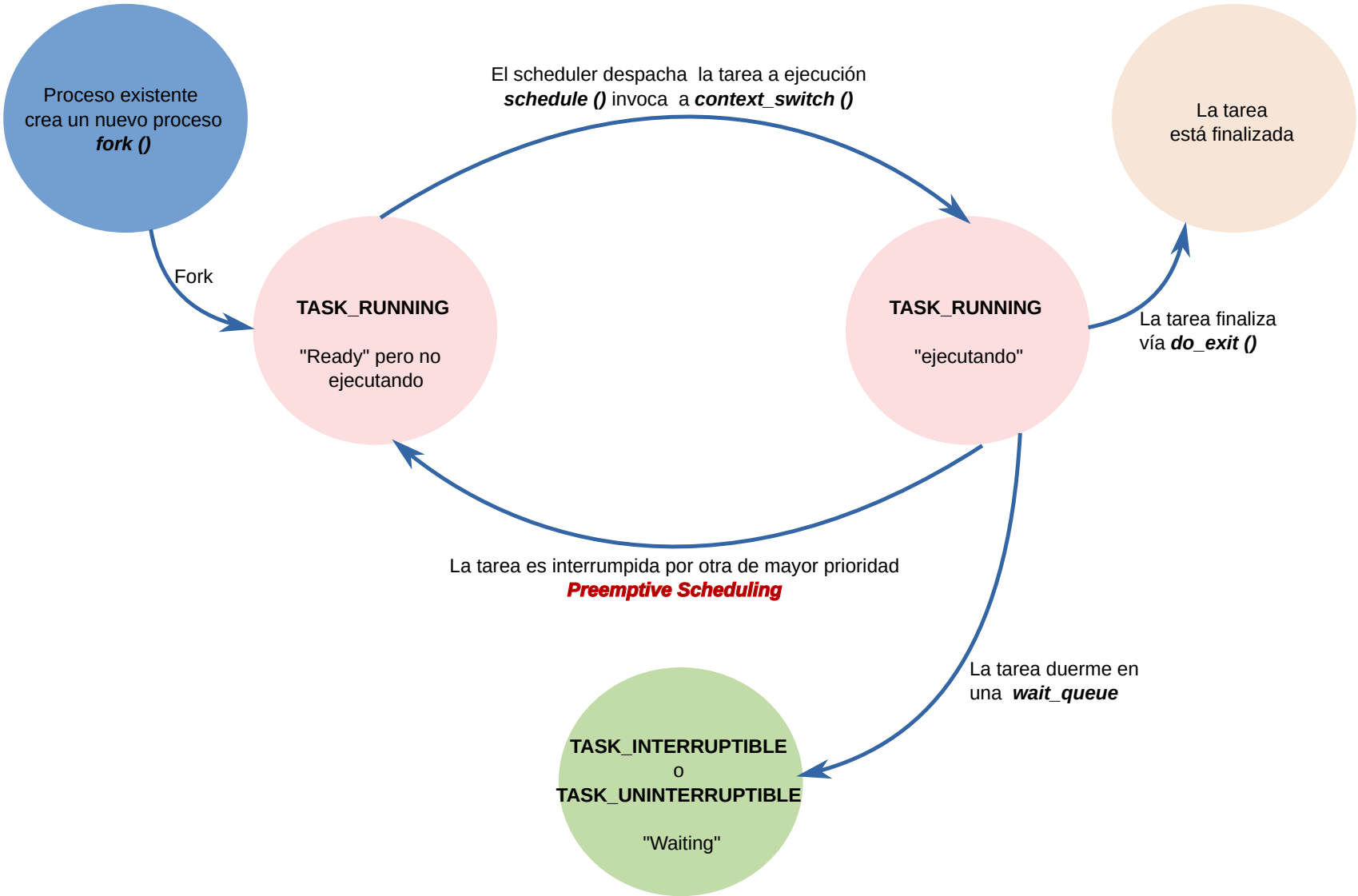
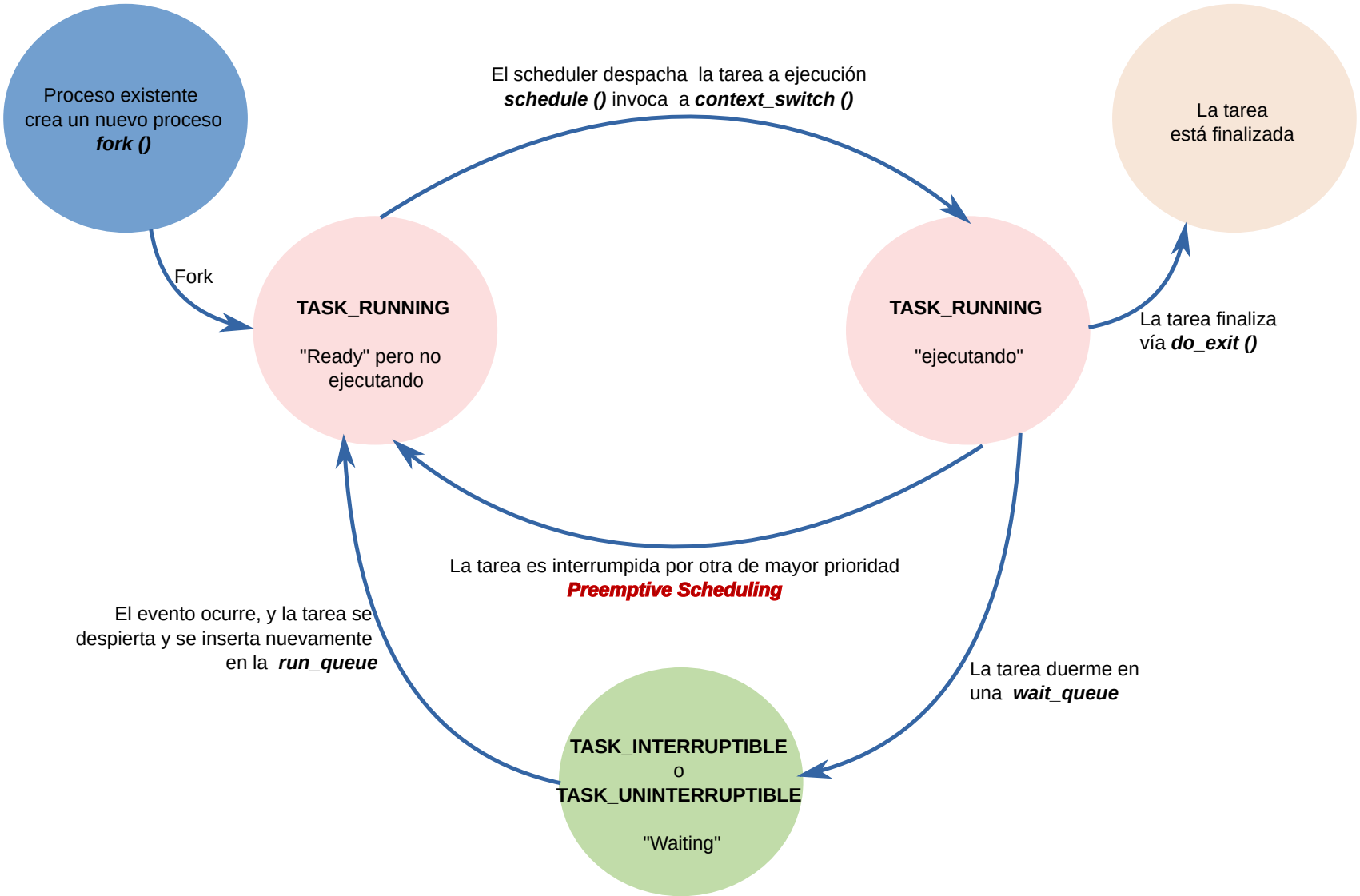


Diagrama de estados de un proceso



Terminación de un proceso

Terminación de un proceso

- Cuando un proceso finaliza su ejecución, normalmente a través de la función `exit()`, el kernel libera sus recursos y notifica al proceso padre de la finalización.

Terminación de un proceso

- Cuando un proceso finaliza su ejecución, normalmente a través de la función `exit()`, el kernel libera sus recursos y notifica al proceso padre de la finalización.

- En general si el proceso se desarrolla a través de `exit()` es conducido de manera segura por el kernel.

Terminación de un proceso

- Cuando un proceso finaliza su ejecución, normalmente a través de la función `exit()`, el kernel libera sus recursos y notifica al proceso padre de la finalización.
- En general si el proceso se desarrolla a través de `exit()` es conducido de manera segura por el kernel.
- Pero un proceso puede finalizar de manera involuntaria. Por ejemplo si recibe alguna señal o excepción que no puede manejar o ignorar.

Terminación de un proceso

- Cuando un proceso finaliza su ejecución, normalmente a través de la función **`exit()`**, el kernel libera sus recursos y notifica al proceso padre de la finalización.
- En general si el proceso se desarrolla a través de **`exit()`** es conducido de manera segura por el kernel.
- Pero un proceso puede finalizar de manera involuntaria. Por ejemplo si recibe alguna señal o excepción que no puede manejar o ignorar.
- No obstante el proceso está prácticamente conducido en cualquier caso por la función **`do_exit()`** definida en `<kernel/exit.c>`.

Terminación de un proceso

- Cuando un proceso finaliza su ejecución, normalmente a través de la función **`exit()`**, el kernel libera sus recursos y notifica al proceso padre de la finalización.
- En general si el proceso se desarrolla a través de **`exit()`** es conducido de manera segura por el kernel.
- Pero un proceso puede finalizar de manera involuntaria. Por ejemplo si recibe alguna señal o excepción que no puede manejar o ignorar.
- No obstante el proceso está prácticamente conducido en cualquier caso por la función **`do_exit()`** definida en `<kernel/exit.c>`.
- A continuación la operatoria que realiza **`do_exit()`**.

Actividades de `do_exit ()`

Actividades de `do_exit ()`

- Setea flag **`PF_EXITING`** en el miembro **`flags`** de **`task_struct`** .

Actividades de `do_exit ()`

- Setea flag **`PF_EXITING`** en el miembro **`flags`** de **`task_struct`** .
- Llama a **`del_timer_sync()`** para remover cualquier timer del kernel que eventualmente pueda estar asociado al proceso. Al retorno de ésta función el kernel asegura que el proceso no está encolado en ningún timer, y que no tiene handlers asociados a ningún timer.

Actividades de `do_exit ()`

- Setea flag **`PF_EXITING`** en el miembro **`flags`** de **`task_struct`** .
- Llama a **`del_timer_sync()`** para remover cualquier timer del kernel que eventualmente pueda estar asociado al proceso. Al retorno de ésta función el kernel asegura que el proceso no está encolado en ningún timer, y que no tiene handlers asociados a ningún timer.
- Si el proceso trabaja con ***BSD accounting***[1] llama a la función **`acct_update_integrals()`** para escribir la información de accounting.

[1] BSD tiene un mecanismo de seguridad llamado process accounting que loguea para seguimiento por parte del superusuario los recursos utilizados por un proceso y su asignación entre diferentes usuarios.

Actividades de `do_exit ()`

- Setea flag **`PF_EXITING`** en el miembro **`flags`** de **`task_struct`** .
- Llama a **`del_timer_sync()`** para remover cualquier timer del kernel que eventualmente pueda estar asociado al proceso. Al retorno de ésta función el kernel asegura que el proceso no está encolado en ningún timer, y que no tiene handlers asociados a ningún timer.
- Si el proceso trabaja con ***BSD accounting***[1] llama a la función **`acct_update_integrals()`** para escribir la información de accounting.
- Llama a **`exit_mm()`** para liberar la estructura de **`mm_struct`** asignada a **`task_struct`** .

[1] BSD tiene un mecanismo de seguridad llamado process accounting que loguea para seguimiento por parte del superusuario los recursos utilizados por un proceso y su asignación entre diferentes usuarios.

Actividades de `do_exit ()`

- Setea flag **`PF_EXITING`** en el miembro **`flags`** de **`task_struct`** .
- Llama a **`del_timer_sync()`** para remover cualquier timer del kernel que eventualmente pueda estar asociado al proceso. Al retorno de ésta función el kernel asegura que el proceso no está encolado en ningún timer, y que no tiene handlers asociados a ningún timer.
- Si el proceso trabaja con ***BSD accounting***[1] llama a la función **`acct_update_integrals()`** para escribir la información de accounting.
- Llama a **`exit_mm()`** para liberar la estructura de **`mm_struct`** asignada a **`task_struct`** .
- Llama a **`exit_sem()`** para remover al proceso de cualquier semáforo en el que pueda estar encolado.

[1] BSD tiene un mecanismo de seguridad llamado process accounting que loguea para seguimiento por parte del superusuario los recursos utilizados por un proceso y su asignación entre diferentes usuarios.

Actividades de `do_exit ()`

- Llama a **`exit_files()`** y **`exit_fs()`** para decrementar la cuenta de uso de cada objeto relacionado con cada file descriptor y file sistem respectivamente. Por cada contador que llegue a cero el objeto es destruido ya que ningún otro proceso lo está utilizando.

Actividades de `do_exit ()`

- Llama a `exit_files()` y `exit_fs()` para decrementar la cuenta de uso de cada objeto relacionado con cada file descriptor y file sistem respectivamente. Por cada contador que llegue a cero el objeto es destruido ya que ningún otro proceso lo está utilizando.
- Escribe en el miembro `exit_code` de `task_struct` , el código de finalización del proceso provisto por `exit()` o por cualquier mecanismo del kernel que haya forzado su finalización. El sentido es para que el padre eventualmente lo pueda recuperar de aquí.

Actividades de `do_exit ()`

- Llama a **`exit_notify()`** para enviar al proceso padre la señal **`SIGCHLD`**, y “reparentar”eventualmente al o los procesos hijos del que está terminando a cualquier thread del mismo thread group o en su defecto al proceso **`init`**, y pone **`EXIT_ZOMBIE`** estado del proceso en el miembro **`exit_state`** en **`task_struct`** .

Actividades de `do_exit ()`

- Llama a **`exit_notify()`** para enviar al proceso padre la señal **`SIGCHLD`**, y “reparentar”eventualmente al o los procesos hijos del que está terminando a cualquier thread del mismo thread group o en su defecto al proceso **`init`**, y pone **`EXIT_ZOMBIE`** estado del proceso en el miembro **`exit_state`** en **`task_struct`** .

- **`do_exit()`** llama a **`schedule()`** para conmutar a un nuevo proceso. El proceso que está finalizando nunca mas será schedulable, por lo tanto éste es el último código que ejecuta este proceso. **`do_exit()`** nunca retorna.

Remoción del Descriptor de Proceso

- Una vez que `do_exit()` termina, `task_struct` (el Descriptor de Proceso) aún permanece.

Remoción del Descriptor de Proceso

- Una vez que `do_exit()` termina, `task_struct` (el Descriptor de Proceso) aún permanece.
- El proceso aún está en estado `EXIT_ZOMBIE` y no es ejecutable nunca mas.

Remoción del Descriptor de Proceso

- Una vez que `do_exit()` termina, `task_struct` (el Descriptor de Proceso) aún permanece.
- El proceso aún está en estado `EXIT_ZOMBIE` y no es ejecutable nunca mas.
- `task_struct` será desalojada solo cuando el proceso padre haya obtenido al información de finalización de su child o haya efectuado alguna actividad que evidencie que no interesa ésta información.

Remoción del Descriptor de Proceso

- Una vez que `do_exit()` termina, `task_struct` (el Descriptor de Proceso) aún permanece.
- El proceso aún está en estado `EXIT_ZOMBIE` y no es ejecutable nunca mas.
- `task_struct` será desalojada solo cuando el proceso padre haya obtenido al información de finalización de su child o haya efectuado alguna actividad que evidencie que no interesa ésta información.
- El familia de funciones `wait()` está implementada mediante la syscall `wait4()`.

Remoción del Descriptor de Proceso

- Una vez que `do_exit()` termina, `task_struct` (el Descriptor de Proceso) aún permanece.
- El proceso aún está en estado `EXIT_ZOMBIE` y no es ejecutable nunca mas.
- `task_struct` será desalojada solo cuando el proceso padre haya obtenido al información de finalización de su child o haya efectuado alguna actividad que evidencie que no interesa ésta información.
- El familia de funciones `wait()` está implementada mediante la syscall `wait4()`.
- El comportamiento es bloquear al proceso invocante hasta que su proceso hijo ejecute `exit()`.

Remoción del Descriptor de Proceso

- En ese momento el proceso invocante obtiene el PID del child finalizado, y un puntero a su código de finalización (aquel que quedó almacenado en el miembro **`exit_code`** de **`task_struct`** del child finalizado).

Remoción del Descriptor de Proceso

- En ese momento el proceso invocante obtiene el PID del child finalizado, y un puntero a su código de finalización (aquel que quedó almacenado en el miembro **`exit_code`** de **`task_struct`** del child finalizado).
- En este momento el kernel invoca a **`release_task()`**.

Remoción del Descriptor de Proceso

- En ese momento el proceso invocante obtiene el PID del child finalizado, y un puntero a su código de finalización (aquel que quedó almacenado en el miembro **`exit_code`** de **`task_struct`** del child finalizado).
- En este momento el kernel invoca a **`release_task()`**.
- **`release_task()`** llama a **`__exit_signal()`**, que a su vez invoca a **`__unhash_process()`**, que a su vez llama a **`detach_pid()`**. Esta última se ocupa de remover al proceso del pidhash y de la task_list.

Remoción del Descriptor de Proceso

- En ese momento el proceso invocante obtiene el PID del child finalizado, y un puntero a su código de finalización (aquel que quedó almacenado en el miembro **`exit_code`** de **`task_struct`** del child finalizado).
- En este momento el kernel invoca a **`release_task()`**.
- **`release_task()`** llama a **`__exit_signal()`**, que a su vez invoca a **`__unhash_process()`**, que a su vez llama a **`detach_pid()`**. Esta última se ocupa de remover al proceso del pidhash y de la task_list.
- **`__exit_signal()`** libera los recursos restantes utilizados por el proceso ahora terminado y finaliza las estadísticas y el accounting.

Remoción del Descriptor de Proceso

- Si el proceso era el último miembro de un grupo de subprocesos y el líder de grupo esta **`TASK_ZOMBIE`**, **`release_task ()`** notifica al padre del líder.

Remoción del Descriptor de Proceso

- Si el proceso era el último miembro de un grupo de subprocesos y el líder de grupo esta **`TASK_ZOMBIE`**, **`release_task ()`** notifica al padre del líder.
- En este punto, el descriptor del proceso y todos los recursos que pertenecen únicamente al proceso han sido liberados.

Remoción del Descriptor de Proceso

- Si el proceso era el último miembro de un grupo de subprocesos y el líder de grupo esta **`TASK_ZOMBIE`**, **`release_task ()`** notifica al padre del líder.
- En este punto, el descriptor del proceso y todos los recursos que pertenecen únicamente al proceso han sido liberados.
- **`release_task ()`** llama a **`put_task_struct ()`** para liberar las páginas que contienen la pila del kernel del proceso y la estructura **`thread_info`** y desasignar la ***slab cache*** que contiene **`task_struc`**
.

El problema de los procesos huérfanos

- Cuando un proceso padre finaliza antes que el hijo, surge el problema de reparentar al child a un nuevo proceso, o de otro modo cuando el child finalice quedaría **`TASK_ZOMBIE`** para siempre, consumiendo memoria del sistema.

El problema de los procesos huérfanos

- Cuando un proceso padre finaliza antes que el hijo, surge el problema de reparentar al child a un nuevo proceso, o de otro modo cuando el child finalice quedaría **`TASK_ZOMBIE`** para siempre, consumiendo memoria del sistema.
- La solución es reparentar al childen con otro proceso dentro del thread group al que pertenece, o si no queda ninguno, reparentarlo a **`init`**.

El problema de los procesos huérfanos

- Cuando un proceso padre finaliza antes que el hijo, surge el problema de reparentar al child a un nuevo proceso, o de otro modo cuando el child finalice quedaría **`TASK_ZOMBIE`** para siempre, consumiendo memoria del sistema.
- La solución es reparentar al childen con otro proceso dentro del thread group al que pertenece, o si no queda ninguno, reparentarlo a **`init`**.
- La función **`do_exit()`** llama a **`exit_notify()`**, que a su vez llama a **`forget_original_parent()`**, que finalmente llama a **`find_new_reaper()`** para realizar el reparentamiento.

¿Que es un thread?

¿Que es un thread?

- Los Threads son abstracciones de programación mas modernas que los procesos.

¿Que es un thread?

- Los Threads son abstracciones de programación mas modernas que los procesos.
- Proporcionan múltiples flujos de ejecución dentro de un mismo programa (proceso).

¿Que es un thread?

- Los Threads son abstracciones de programación mas modernas que los procesos.
- Proporcionan múltiples flujos de ejecución dentro de un mismo programa (proceso).
- Comparten entre si una serie de recursos que los procesos no comparten.

¿Que es un thread?

- Los Threads son abstracciones de programación mas modernas que los procesos.
- Proporcionan múltiples flujos de ejecución dentro de un mismo programa (proceso).
- Comparten entre si una serie de recursos que los procesos no comparten.
- Permiten implementar programación concurrente de manera mas ágil que los procesos y si el sistema es multicore (as usual), habilitan paralelismo 100 % real.

¿Que es un thread?

- Los Threads son abstracciones de programación mas modernas que los procesos.
- Proporcionan múltiples flujos de ejecución dentro de un mismo programa (proceso).
- Comparten entre si una serie de recursos que los procesos no comparten.
- Permiten implementar programación concurrente de manera mas ágil que los procesos y si el sistema es multicore (as usual), habilitan paralelismo 100 % real.
- El Kernel de Linux implementa Threads de manera única.

¿Que es un thread?

- Los Threads son abstracciones de programación mas modernas que los procesos.
- Proporcionan múltiples flujos de ejecución dentro de un mismo programa (proceso).
- Comparten entre si una serie de recursos que los procesos no comparten.
- Permiten implementar programación concurrente de manera mas ágil que los procesos y si el sistema es multicore (as usual), habilitan paralelismo 100 % real.
- El Kernel de Linux implementa Threads de manera única.
- No crea ninguna estructura particular para controlar threads.

¿Que es un thread?

- Los Threads son abstracciones de programación mas modernas que los procesos.
- Proporcionan múltiples flujos de ejecución dentro de un mismo programa (proceso).
- Comparten entre si una serie de recursos que los procesos no comparten.
- Permiten implementar programación concurrente de manera mas ágil que los procesos y si el sistema es multicore (as usual), habilitan paralelismo 100 % real.
- El Kernel de Linux implementa Threads de manera única.
- No crea ninguna estructura particular para controlar threads.
- De hecho cada Thread posee su propia `task_struct` .

¿Que es un thread?

- Los Threads son abstracciones de programación mas modernas que los procesos.
- Proporcionan múltiples flujos de ejecución dentro de un mismo programa (proceso).
- Comparten entre si una serie de recursos que los procesos no comparten.
- Permiten implementar programación concurrente de manera mas ágil que los procesos y si el sistema es multicore (as usual), habilitan paralelismo 100 % real.
- El Kernel de Linux implementa Threads de manera única.
- No crea ninguna estructura particular para controlar threads.
- De hecho cada Thread posee su propia `task_struct` .
- En Linux un Thread termina siendo un proceso que tiene la particularidad de compartir un conjunto de recursos con otros procesos (threads del mismo programa principal).

¿Que es un thread?

- Emplear la misma estructura de un proceso común para controlar a un Thread, diferencia a Linux de otros Sistemas Operativos, como Solaris o Windows que implementan estructuras de control mas simples que las de los procesos.

¿Que es un thread?

- Emplear la misma estructura de un proceso común para controlar a un Thread, diferencia a Linux de otros Sistemas Operativos, como Solaris o Windows que implementan estructuras de control mas simples que las de los procesos.
- Estos sistemas califican a un Thread como “LightweightProcess ”, ya que responden a la teoría general de Threads en la que la estructura de control necesaria es mas pequeña.

¿Que es un thread?

- Emplear la misma estructura de un proceso común para controlar a un Thread, diferencia a Linux de otros Sistemas Operativos, como Solaris o Windows que implementan estructuras de control mas simples que las de los procesos.
- Estos sistemas califican a un Thread como “LightweightProcess ”, ya que responden a la teoría general de Threads en la que la estructura de control necesaria es mas pequeña.
- Linux por su parte define como “Lightweight Process ”a todos sus procesos debido a la práctica implementación COW ya descripta.

¿Que es un thread?

- Emplear la misma estructura de un proceso común para controlar a un Thread, diferencia a Linux de otros Sistemas Operativos, como Solaris o Windows que implementan estructuras de control mas simples que las de los procesos.
- Estos sistemas califican a un Thread como “LightweightProcess ”, ya que responden a la teoría general de Threads en la que la estructura de control necesaria es mas pequeña.
- Linux por su parte define como “Lightweight Process ”a todos sus procesos debido a la práctica implementación COW ya descripta.
- Para Linux la única diferencia entre un thread y un proceso es la cantidad de recursos que comparten.

¿Que es un thread?

- Emplear la misma estructura de un proceso común para controlar a un Thread, diferencia a Linux de otros Sistemas Operativos, como Solaris o Windows que implementan estructuras de control mas simples que las de los procesos.
- Estos sistemas califican a un Thread como “LightweightProcess ”, ya que responden a la teoría general de Threads en la que la estructura de control necesaria es mas pequeña.
- Linux por su parte define como “Lightweight Process ”a todos sus procesos debido a la práctica implementación COW ya descripta.
- Para Linux la única diferencia entre un thread y un proceso es la cantidad de recursos que comparten.
- Por lo tanto en Linux un proceso consta como mínimo de un thread.

¿Que es un thread?

- Emplear la misma estructura de un proceso común para controlar a un Thread, diferencia a Linux de otros Sistemas Operativos, como Solaris o Windows que implementan estructuras de control mas simples que las de los procesos.
- Estos sistemas califican a un Thread como “LightweightProcess ”, ya que responden a la teoría general de Threads en la que la estructura de control necesaria es mas pequeña.
- Linux por su parte define como “Lightweight Process ”a todos sus procesos debido a la práctica implementación COW ya descripta.
- Para Linux la única diferencia entre un thread y un proceso es la cantidad de recursos que comparten.
- Por lo tanto en Linux un proceso consta como mínimo de un thread.
- Como se verá Linux finalmente maneja Threads en el scheduler, y los trata como un tipo especial de proceso.

Creación de Threads

- La creación de un Thread en Linux es similar a la de un proceso normal, excepto en los argumentos con que se invoca a la función `clone()`.

Creación de Threads

- La creación de un Thread en Linux es similar a la de un proceso normal, excepto en los argumentos con que se invoca a la función `clone()`.
- Cuando se crea un Thread `clone()` es invocada con una serie de Flags que indican cuales recursos serán compartidos con los demás threads derivados del mismo proceso original.

Creación de Threads

- La creación de un Thread en Linux es similar a la de un proceso normal, excepto en los argumentos con que se invoca a la función `clone()`.
- Cuando se crea un Thread `clone()` es invocada con una serie de Flags que indican cuales recursos serán compartidos con los demás threads derivados del mismo proceso original.
- En el caso de un thread `clone()` recibe en el argumento flags `CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD | CLONE_SYSVSEM | CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID`

Creación de Threads

- La creación de un Thread en Linux es similar a la de un proceso normal, excepto en los argumentos con que se invoca a la función **`clone()`**.
- Cuando se crea un Thread **`clone()`** es invocada con una serie de Flags que indican cuales recursos serán compartidos con los demás threads derivados del mismo proceso original.
- En el caso de un thread **`clone()`** recibe en el argumento flags **`CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD | CLONE_SYSVSEM | CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID`**
- En cambio, si **`clone()`** se invoca desde un **`fork()`** flags recibe **`CLONE_CHILD_CLEARTID | CLONE_CHILD_SETTID | SIGCHLD`**

Creación de Threads

- La creación de un Thread en Linux es similar a la de un proceso normal, excepto en los argumentos con que se invoca a la función **`clone()`**.
- Cuando se crea un Thread **`clone()`** es invocada con una serie de Flags que indican cuales recursos serán compartidos con los demás threads derivados del mismo proceso original.
- En el caso de un thread **`clone()`** recibe en el argumento flags **`CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD | CLONE_SYSVSEM | CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID`**
- En cambio, si **`clone()`** se invoca desde un **`fork()`** flags recibe **`CLONE_CHILD_CLEARTID | CLONE_CHILD_SETTID | SIGCHLD`**
- El detalle de lo que hace cada flag se puede consultar en ***man clone***.

Creación de Threads

- **`CLONE_CHILD_CLEARTID`** Limpia el Child Thread ID en la locación de memoria *ctid* del child cuando éste termina, y desbloquea esa área de memoria.

Creación de Threads

- **`CLONE_CHILD_CLEARTID`** Limpia el Child Thread ID en la locación de memoria *ctid* del child cuando éste termina, y desbloquea esa área de memoria.
- **`CLONE_CHILD_SETTID`** Almacena el Child Thread ID en la locación de memoria *ctid* del child, antes de que **`clone()`** finalice.

Multitasking

Hay dos clases de Multitasking:

Multitasking

Hay dos clases de Multitasking:

1. Multitasking Cooperativo. Un proceso ejecuta hasta que voluntariamente decide ceder la CPU a otro proceso o al kernel. El acto de suspenderse a si mismo se denomina *yielding*.

Multitasking

Hay dos clases de Multitasking:

1. Multitasking Cooperativo. Un proceso ejecuta hasta que voluntariamente decide ceder la CPU a otro proceso o al kernel. El acto de suspenderse a si mismo se denomina *yielding*.

2. Multitasking Expropiativo[1]. El scheduler decide cuando un proceso termina su ejecución y cuando otro proceso retoma o comienza su ejecución. El tiempo que un proceso ejecuta hasta ser interrumpido por el scheduler se llama *time slice*.

[1] Suele leerse el Argentinismo “Preemptivo”. Preemption en inglés refleja el acto de suspender arbitrariamente un proceso que está ejecutando.

Multitasking

Hay dos clases de Multitasking:

1. Multitasking Cooperativo. Un proceso ejecuta hasta que voluntariamente decide ceder la CPU a otro proceso o al kernel. El acto de suspenderse a si mismo se denomina *yielding*.

2. Multitasking Expropiativo[1]. El scheduler decide cuando un proceso termina su ejecución y cuando otro proceso retoma o comienza su ejecución. El tiempo que un proceso ejecuta hasta ser interrumpido por el scheduler se llama *time slice*.

Linux como la mayoría de los Sistemas Operativos moderno trabaja con Multitasking Expropiativo (Preemptive Multitasking).

[1] Suele leerse el Argentinismo “Preemptivo”. Preemption en inglés refleja el acto de suspender arbitrariamente un proceso que está ejecutando.

El scheduler de Linux

El scheduler de Linux

- Desde 1991 hasta la versión de kernel 2.4, el scheduler de Linux fue sumamente sencillo.

El scheduler de Linux

- Desde 1991 hasta la versión de kernel 2.4, el scheduler de Linux fue sumamente sencillo.
- Se lo ajustó de acuerdo a la posibilidad de ejecutar mas procesos, o de soportar Multiprocesamiento Simétrico.

El scheduler de Linux

- Desde 1991 hasta la versión de kernel 2.4, el scheduler de Linux fue sumamente sencillo.
- Se lo ajustó de acuerdo a la posibilidad de ejecutar mas procesos, o de soportar Multiprocesamiento Simétrico.
- Durante el desarrollo del kernel 2.5 (que se liberó como el 2.6) el diseño del scheduler fue uno de los puntos a revisar.

El scheduler de Linux

- Desde 1991 hasta la versión de kernel 2.4, el scheduler de Linux fue sumamente sencillo.
- Se lo ajustó de acuerdo a la posibilidad de ejecutar mas procesos, o de soportar Multiprocesamiento Simétrico.
- Durante el desarrollo del kernel 2.5 (que se liberó como el 2.6) el diseño del scheduler fue uno de los puntos a revisar.
- El nuevo scheduler (big-o), implementado en el kernel 2.6 utiliza un algoritmo de tiempo constante para calcular el *time slice*, y una cola de procesos (runqueue) modificada como un arreglo de listas por valor de prioridad y además una runqueue por procesador.

El scheduler de Linux

- Desde 1991 hasta la versión de kernel 2.4, el scheduler de Linux fue sumamente sencillo.
- Se lo ajustó de acuerdo a la posibilidad de ejecutar mas procesos, o de soportar Multiprocesamiento Simétrico.
- Durante el desarrollo del kernel 2.5 (que se liberó como el 2.6) el diseño del scheduler fue uno de los puntos a revisar.
- El nuevo scheduler (big-o), implementado en el kernel 2.6 utiliza un algoritmo de tiempo constante para calcular el *time slice*, y una cola de procesos (runqueue) modificada como un arreglo de listas por valor de prioridad y además una runqueue por procesador.
- Implementa lo que se llamó Rotating Staircase Deadline, es decir, una rotación escalonada del Plazo de expiración del timeslice, concepto conocido como *fair scheduling* (equitativo).

El scheduler de Linux

- Desde 1991 hasta la versión de kernel 2.4, el scheduler de Linux fue sumamente sencillo.
- Se lo ajustó de acuerdo a la posibilidad de ejecutar mas procesos, o de soportar Multiprocesamiento Simétrico.
- Durante el desarrollo del kernel 2.5 (que se liberó como el 2.6) el diseño del scheduler fue uno de los puntos a revisar.
- El nuevo scheduler (big-o), implementado en el kernel 2.6 utiliza un algoritmo de tiempo constante para calcular el *time slice*, y una cola de procesos (runqueue) modificada como un arreglo de listas por valor de prioridad y además una runqueue por procesador.
- Implementa lo que se llamó Rotating Staircase Deadline, es decir, una rotación escalonada del Plazo de expiración del timeslice, concepto conocido como *fair scheduling* (equitativo).
- A partir de la versión 2.6.23 el sceduler evolucionó a su configuración actual: CFS (Compltetly Fair Scheduler)

Política de scheduling

Política de scheduling

- Establece que hará el scheduler y cuando lo hará.

Política de scheduling

- Establece que hará el scheduler y cuando lo hará.
- Los procesos pueden ser **I/O-Bound** o **CPU-Bound**. No son categorías mutuamente excluyentes.

Política de scheduling

- Establece que hará el scheduler y cuando lo hará.
- Los procesos pueden ser **I/O-Bound** o **CPU-Bound**. No son categorías mutuamente excluyentes.
- Los procesos se clasifican en tres categorías:

Política de scheduling

- Establece que hará el scheduler y cuando lo hará.
- Los procesos pueden ser **I/O-Bound** o **CPU-Bound**. No son categorías mutuamente excluyentes.
- Los procesos se clasifican en tres categorías:

Interactivos Interactúan constantemente con el usuario, y toman buena parte de su tiempo en esperar ingresos desde el teclado o reaccionar ante acciones del mouse. Generalmente están **`TASK_INTERRUPTIBLE`**, pero cuando se produce el evento deben reaccionar rápidamente (50 a 150 mseg de demora típicamente).

Política de scheduling

- Establece que hará el scheduler y cuando lo hará.
- Los procesos pueden ser **I/O-Bound** o **CPU-Bound**. No son categorías mutuamente excluyentes.
- Los procesos se clasifican en tres categorías:

Interactivos Interactúan constantemente con el usuario, y toman buena parte de su tiempo en esperar ingresos desde el teclado o reaccionar ante acciones del mouse. Generalmente están `TASK_INTERRUPTIBLE`, pero cuando se produce el evento deben reaccionar rápidamente (50 a 150 mseg de demora típicamente).

Batch No tienen interacción con el usuario. A menudo corren en background. Son aptos para ser “penalizados”por el scheduler.

Política de scheduling

- Establece que hará el scheduler y cuando lo hará.
- Los procesos pueden ser **I/O-Bound** o **CPU-Bound**. No son categorías mutuamente excluyentes.
- Los procesos se clasifican en tres categorías:

Interactivos Interactúan constantemente con el usuario, y toman buena parte de su tiempo en esperar ingresos desde el teclado o reaccionar ante acciones del mouse. Generalmente están `TASK_INTERRUPTIBLE`, pero cuando se produce el evento deben reaccionar rápidamente (50 a 150 mseg de demora típicamente).

Batch No tienen interacción con el usuario. A menudo corren en background. Son aptos para ser “penalizados”por el scheduler.

Real-time Imponen requisitos de scheduling muy estrictos. Nunca deben ser bloqueados por procesos de menor prioridad, y requieren tener un tiempo de respuesta garantizado con una mínima varianza.

Prioridad de un proceso

Prioridad de un proceso

- Se define en base a la importancia de un proceso y su necesidad de tiempo de CPU.

Prioridad de un proceso

- Se define en base a la importancia de un proceso y su necesidad de tiempo de CPU.
- Es un número con el cual el scheduler decide como tratar al proceso.

Prioridad de un proceso

- Se define en base a la importancia de un proceso y su necesidad de tiempo de CPU.
- Es un número con el cual el scheduler decide como tratar al proceso.
- El kernel de Linux maneja dos rangos de prioridades:

Prioridad de un proceso

- Se define en base a la importancia de un proceso y su necesidad de tiempo de CPU.
- Es un número con el cual el scheduler decide como tratar al proceso.
- El kernel de Linux maneja dos rangos de prioridades:
- Ambos espacios de prioridad son disjuntos.

Prioridad de un proceso: `nice value`

Prioridad de un proceso: `nice value`

- *nice value*: En UNIX es el rango de valores típicos del comando **`nice`**.

Prioridad de un proceso: `nice value`

- *nice value*: En UNIX es el rango de valores típicos del comando **`nice`**.
- Varía de -20 a +19. El default es 0. A mayor valor numérico menor prioridad (¿donde escuché esto antes?...).

Prioridad de un proceso: `nice value`

- *nice value*: En UNIX es el rango de valores típicos del comando **`nice`**.
- Varía de -20 a +19. El default es 0. A mayor valor numérico menor prioridad (¿donde escuché esto antes?...).
- Preciera que un proceso es "*nice*"con los demás, conforme su prioridad es menor.

Prioridad de un proceso: `nice value`

- *nice value*: En UNIX es el rango de valores típicos del comando **`nice`**.
- Varía de -20 a +19. El default es 0. A mayor valor numérico menor prioridad (¿donde escuché esto antes?...).
- Preciera que un proceso es "*nice*"con los demás, conforme su prioridad es menor.
- El comando `ps -el` lo muestra en la columna NI.

Prioridad de un proceso: `nice value`

```
alejandro@DarkSideOfTheMoon:~/git/TDIII/slides/Arquitectura-Linux/LinuxArchIntro$ ps -el
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  80   0 - 56627 -      ?        00:05:14 systemd
1 S     0     2     0  0  80   0 -     0 -      ?        00:00:00 kthreadd
1 I     0     4     2  0  60 -20 -     0 -      ?        00:00:00 kworker/0:0H
1 I     0     6     2  0  60 -20 -     0 -      ?        00:00:00 mm_percpu_wq
4 S    33  1708     1  0  80   0 - 75925 -      ?        00:00:00 Main
5 S     0  1724     2  0  70 -10 -     0 -      ?        00:00:00 krfcommd
4 S   117  1730     1  0  80   0 - 80898 -      ?        00:00:02 colord
4 S   125  1765     1  0  80   0 -  2077 -      ?        00:00:42 vnstatd
4 S     0  1784     1  0  80   0 - 18075 -      ?        00:00:00 sshd
4 S     0  1798     1  0  80   0 - 498333 -     ?        00:16:36 containerd
4 S     0  1858     1  0  80   0 - 47442 -      ?        00:00:00 unattended-upgr
4 S     0  2031     1  0  80   0 - 75900 -      ?        00:00:00 gdm3
4 S     0  2040  2031  0  80   0 - 64242 -      ?        00:00:00 gdm-session-wor
4 S   121  2067     1  0  80   0 - 19336 -      ?        00:00:00 systemd
0 S  1000 12514 12040  0  99  19 - 189436 poll_s tty2     00:07:00 tracker-miner-f
0 S  1000 12528 12461  0  80   0 - 98365 poll_s tty2     00:00:00 slack
0 S  1000 12529 12040  0  99  19 - 67193591 poll_s tty2  00:00:04 baloo_file
0 S   121  2113  2067  0  80   0 - 12593 -      ?        00:00:00 dbus-daemon
0 S   121  2116  2097  0  80   0 - 138622 -     tty1     00:00:00 gnome-session-b
0 S   121  2123  2067  0  80   0 - 87355 -      ?        00:00:00 at-spi-bus-laun
0 S   121  2128  2123  0  80   0 - 12481 -      ?        00:00:00 dbus-daemon
0 S   121  2130  2067  0  80   0 - 55198 -      ?        00:00:00 at-spi2-registr
0 S   121  2144  2116  0  80   0 - 1061867 -    tty1     00:01:33 gnome-shell
4 S     0  2151     1  0  80   0 - 95997 -      ?        00:01:01 upowerd
0 S   121  2192  2067  0  80   0 - 396083 -     ?        00:00:00 pulseaudio
4 S   109  2193     1  0  81   1 - 45877 -      ?        00:00:09 rtkit-daemon
0 S   121  2208  2192  0  80   0 - 28435 -      ?        00:00:00 gconf-helper
```

Prioridad Real Time de un proceso

Prioridad Real Time de un proceso

- *prioridad real time*: Es un rango configurable, que por default va desde 0 a 99.

Prioridad Real Time de un proceso

- *prioridad real time*: Es un rango configurable, que por default va desde 0 a 99.
- El mayor valor numérico implica mayor prioridad (¿Porqué habrá que aclararlo? :)).

Prioridad Real Time de un proceso

- *prioridad real time*: Es un rango configurable, que por default va desde 0 a 99.
- El mayor valor numérico implica mayor prioridad (¿Porqué habrá que aclararlo? :)).
- La especificación POSIX.1b, establece los requisitos para implementar la prioridad real time. Linux basa su implementación en este standard.

Prioridad Real Time de un proceso

- *prioridad real time*: Es un rango configurable, que por default va desde 0 a 99.
- El mayor valor numérico implica mayor prioridad (¿Porqué habrá que aclararlo? :)).
- La especificación POSIX.1b, establece los requisitos para implementar la prioridad real time. Linux basa su implementación en este standard.
- El comando **`ps`** está preparado para mostrarla A continuación el comando, la muestra en la columna RTPRIO ('-'significa que no tiene prioridad real time).

Prioridad Real Time de un proceso

- *prioridad real time*: Es un rango configurable, que por default va desde 0 a 99.
- El mayor valor numérico implica mayor prioridad (¿Porqué habrá que aclararlo? :)).
- La especificación POSIX.1b, establece los requisitos para implementar la prioridad real time. Linux basa su implementación en este standard.
- El comando **`ps`** está preparado para mostrarla A continuación el comando, la muestra en la columna RTPRIO ('-'significa que no tiene prioridad real time).

```
ps -eo state,uid,pid,ppid,rtprio,time,comm
```

Prioridad Real Time de un proceso

```
alejandro@DarkSideOfTheMoon:~$ ps -eo state,uid,pid,ppid,rtprio,time,comm
S   UID   PID  PPID RTPRIO     TIME COMMAND
S     0     1     0      -  00:05:14 systemd
S     0     2     0      -  00:00:00 kthreadd
S     0   225     2      1  00:01:40 i915/signal:0
S     0   226     2      1  00:00:12 i915/signal:1
S     0   227     2      1  00:00:11 i915/signal:2
S     0   228     2      1  00:00:11 i915/signal:4
I     0     9     2      -  00:00:00 rcu_bh
S     0    10     2     99  00:00:02 migration/0
S     0    11     2     99  00:00:01 watchdog/0
S     0    12     2      -  00:00:00 cpuhp/0
S     0    13     2      -  00:00:00 cpuhp/1
S     0    14     2     99  00:00:01 watchdog/1
S     0    15     2     99  00:00:01 migration/1
S     0    16     2      -  00:00:10 ksoftirqd/1
S     0   885     2     50  00:00:57 irq/51-DLL06E4:
S     0    19     2      -  00:00:00 cpuhp/2
S     0    20     2     99  00:00:01 watchdog/2
S     0    21     2     99  00:00:02 migration/2
```

Completely Fair Scheduling

Completely Fair Scheduling

- Implementado a partir de la versión 2.6.23 del kernel de LINUX.

Completely Fair Scheduling

- Implementado a partir de la versión 2.6.23 del kernel de LINUX.
- Difiere del resto de los schedulers en lo que se refiere al criterio de asignación de la CPU.

Completely Fair Scheduling

- Implementado a partir de la versión 2.6.23 del kernel de LINUX.
- Difiere del resto de los schedulers en lo que se refiere al criterio de asignación de la CPU.
- En lugar de asignar un ***timeslice***, garantiza un porcentaje mínimo de CPU a cada proceso.

Completely Fair Scheduling

- Implementado a partir de la versión 2.6.23 del kernel de LINUX.
- Difiere del resto de los schedulers en lo que se refiere al criterio de asignación de la CPU.
- En lugar de asignar un ***timeslice***, garantiza un porcentaje mínimo de CPU a cada proceso.
- Ejemplo: un editor de texto, y un encoder de video.

Completely Fair Scheduling

- Implementado a partir de la versión 2.6.23 del kernel de LINUX.
- Difiere del resto de los schedulers en lo que se refiere al criterio de asignación de la CPU.
- En lugar de asignar un ***timeslice***, garantiza un porcentaje mínimo de CPU a cada proceso.
- Ejemplo: un editor de texto, y un encoder de video.
- El editor está generalmente **`TASK_INTERRUPTIBLE`**, pero requiere atención inmediata (es interactivo) si el usuario pulsa una tecla.

Completely Fair Scheduling

- Implementado a partir de la versión 2.6.23 del kernel de LINUX.
- Difiere del resto de los schedulers en lo que se refiere al criterio de asignación de la CPU.
- En lugar de asignar un ***timeslice***, garantiza un porcentaje mínimo de CPU a cada proceso.
- Ejemplo: un editor de texto, y un encoder de video.
- El editor está generalmente **`TASK_INTERRUPTIBLE`**, pero requiere atención inmediata (es interactivo) si el usuario pulsa una tecla.
- El encoder accede a disco y video, pero la mayor parte del tiempo la consume utilizando la CPU para ejecutar el codec.

Completely Fair Scheduling

- CFS le **garantiza** un mínimo de 50 % de la CPU a cada uno.

Completely Fair Scheduling

- CFS le **garantiza** un mínimo de 50 % de la CPU a cada uno.
- El editor corre con mayor prioridad. Así al pulsar la tecla interrumpe (preempt) al encoder, presenta el caracter asociado a la tecla, y vuelve a dormir, usando por lejos menos del 50 % de la CPU.

Completely Fair Scheduling

- CFS le **garantiza** un mínimo de 50 % de la CPU a cada uno.
- El editor corre con mayor prioridad. Así al pulsar la tecla interrumpe (preempt) al encoder, presenta el caracter asociado a la tecla, y vuelve a dormir, usando por lejos menos del 50 % de la CPU.
- El encoder trabaja sabiendo que el 50 % del tiempo de CPU lo tiene asegurado, pero si requiere mas y el scheduler tiene disponible, le será asignado.

Completely Fair Scheduling

- CFS le **garantiza** un mínimo de 50 % de la CPU a cada uno.
- El editor corre con mayor prioridad. Así al pulsar la tecla interrumpe (preempt) al encoder, presenta el caracter asociado a la tecla, y vuelve a dormir, usando por lejos menos del 50 % de la CPU.
- El encoder trabaja sabiendo que el 50 % del tiempo de CPU lo tiene asegurado, pero si requiere mas y el scheduler tiene disponible, le será asignado.
- Con esta política ambos procesos coexisten con una eficiencia de ejecución óptima.

Problemas del scheduling pre CFS

Problemas del scheduling pre CFS

- Mapear ***nice values*** en ***timeslices*** . No hay un valor óptimo. Supongamos 100 mseg para nice=0, 5mseg para nice=20. El tiempo que se tarda en recorrer la lista de procesos en ejecución es variable en función de las prioridades de los procesos involucrados.

Problemas del scheduling pre CFS

- Mapear ***nice values*** en ***timeslices*** . No hay un valor óptimo. Supongamos 100 mseg para nice=0, 5mseg para nice=20. El tiempo que se tarda en recorrer la lista de procesos en ejecución es variable en función de las prioridades de los procesos involucrados.
- Incrementos y decrementos de ***nice value*** . Tomando un valor constante, cuanto mayor es la prioridad de dos procesos, menor es porcentualmente el efecto de asignar a uno de ellos un decremento para ampliar su prioridad. En cambio en procesos de baja prioridad un delta de nice implica un incremento porcentualmente muy alto.

Problemas del scheduling pre CFS

- La base de tiempos para activar el scheduler es el timer tick que puede programarse para interrumpir como mínimo cada 1 mseg. Si se quisiera 0.1 mseg. ya no es posible, y no hay otros recursos de hardware standard que lo puedan sustituir.

Problemas del scheduling pre CFS

- La base de tiempos para activar el scheduler es el timer tick que puede programarse para interrumpir como mínimo cada 1 mseg. Si se quisiera 0.1 mseg. ya no es posible, y no hay otros recursos de hardware standard que lo puedan sustituir.
- Un programa interactivo que se ejecuta con un alto nice value, puede haber sido schedulado y al poco de suspenderse recibe otro evento, e interrumpe a los demás, pudiendo derivar en un uso no equitativo de la CPU.

El Algoritmo de scheduling

El Algoritmo de scheduling

- Se basa el el modelo de procesador Perfectamente Multitasking. En éste, cada proceso recibe 1/n del tiempo de CPU, siendo n la cantidad de procesos en estado de ejecución.

El Algoritmo de scheduling

- Se basa el el modelo de procesador Perfectamente Multitasking. En éste, cada proceso recibe 1/n del tiempo de CPU, siendo n la cantidad de procesos en estado de ejecución.
- En vez de asignar un ***timeslice***, calcula cuanto tiempo puede correr en función de la cantidad de procesos en estado de ejecución.

El Algoritmo de scheduling

- Se basa el el modelo de procesador Perfectamente Multitasking. En éste, cada proceso recibe 1/n del tiempo de CPU, siendo n la cantidad de procesos en estado de ejecución.
- En vez de asignar un ***timeslice***, calcula cuanto tiempo puede correr en función de la cantidad de procesos en estado de ejecución.
- CFS usa el ***nice value*** para calcular el “peso”del proceso.

El Algoritmo de scheduling

- Se basa el el modelo de procesador Perfectamente Multitasking. En éste, cada proceso recibe 1/n del tiempo de CPU, siendo n la cantidad de procesos en estado de ejecución.
- En vez de asignar un ***timeslice***, calcula cuanto tiempo puede correr en función de la cantidad de procesos en estado de ejecución.
- CFS usa el ***nice value*** para calcular el “peso”del proceso.
- Cada proceso ejecutará durante un tiempo igual a la razón entre su peso y la suma de los pesos de todos los procesos en estado **`TASK_RUNNING`**.

El Algoritmo de scheduling

El Algoritmo de scheduling

- La base del scheduler está definida en `<kernel\sched \sched.h>` y su código en los diferentes fuentes de ese mismo directorio.

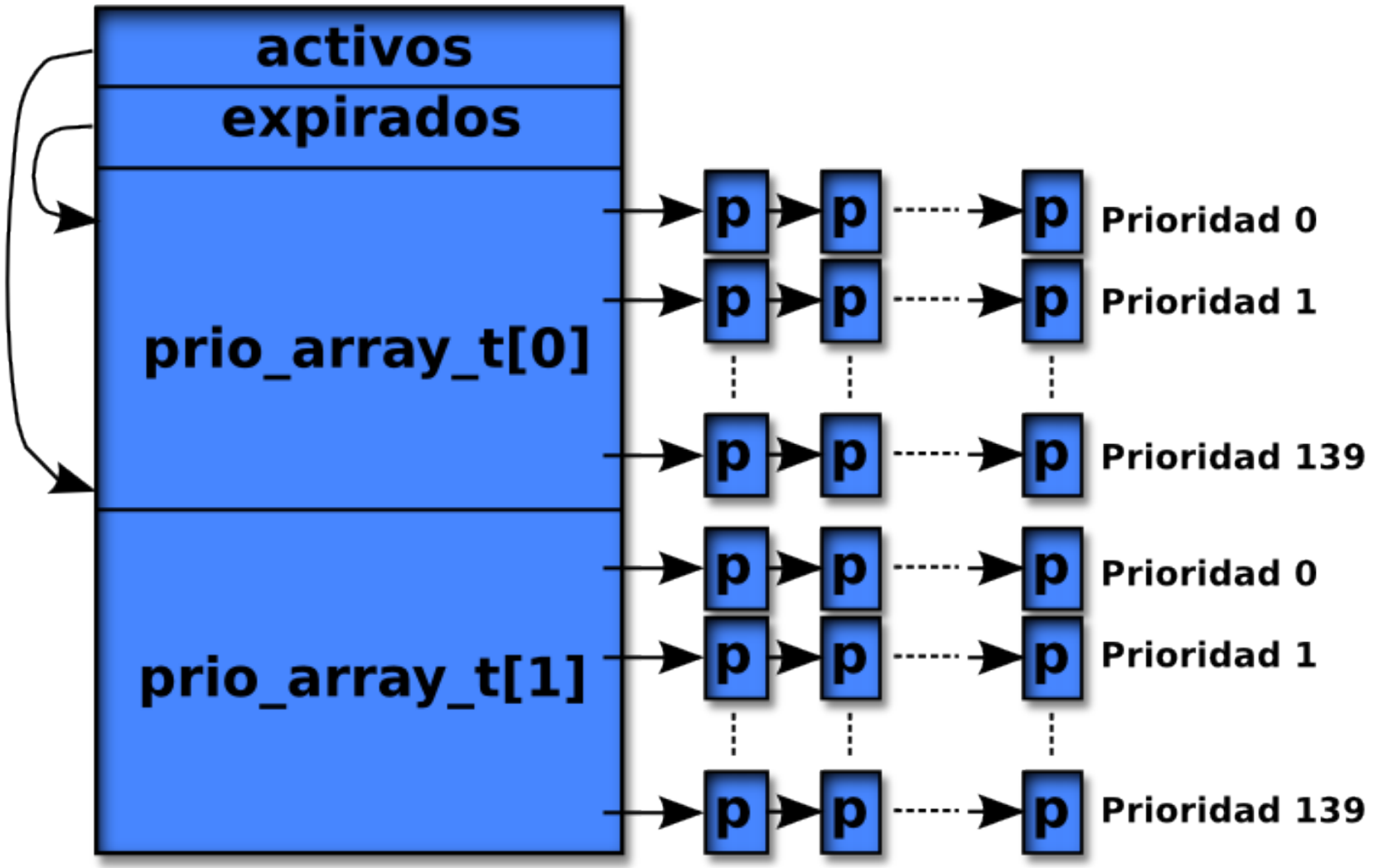
El Algoritmo de scheduling

- La base del scheduler está definida en `<kernel\sched \sched.h>` y su código en los diferentes fuentes de ese mismo directorio.
- Luego se han incluido las que se llaman Clases de Scheduling.

El Algoritmo de scheduling

- La base del scheduler está definida en `<kernel\sched \sched.h>` y su código en los diferentes fuentes de ese mismo directorio.
- Luego se han incluido las que se llaman Clases de Scheduling.
- La clase Completely Fair Scheduling tiene su código en `<kernel\sch`

La ¿lista? de procesos



El kernel en LINUX maneja todas las interrupciones

El kernel en LINUX maneja todas las interrupciones

- Cada Interrupción tiene asociado un ISR (Interrupt Service Routine), escrita en C.

El kernel en LINUX maneja todas las interrupciones

- Cada Interrupción tiene asociado un ISR (Interrupt Service Routine), escrita en C.
- El kernel maneja las interrupciones de hardware que llegan a través del Controlador desde los dispositivos de hardware.

El kernel en LINUX maneja todas las interrupciones

- Cada Interrupción tiene asociado un ISR (Interrupt Service Routine), escrita en C.
- El kernel maneja las interrupciones de hardware que llegan a través del Controlador desde los dispositivos de hardware.
- También actúa ante las excepciones.

El kernel en LINUX maneja todas las interrupciones

- Cada Interrupción tiene asociado un ISR (Interrupt Service Routine), escrita en C.
- El kernel maneja las interrupciones de hardware que llegan a través del Controlador desde los dispositivos de hardware.
- También actúa ante las excepciones.
- Además el ingreso a las System Calls es mediante una Software Interrupt.

El kernel en LINUX maneja todas las interrupciones

- Cada Interrupción tiene asociado un ISR (Interrupt Service Routine), escrita en C.
- El kernel maneja las interrupciones de hardware que llegan a través del Controlador desde los dispositivos de hardware.
- También actúa ante las excepciones.
- Además el ingreso a las System Calls es mediante una Software Interrupt.
- Linux no maneja tareas por interrupción de modo que en la IDT solo hay

Top Halves y Bottom Halves

Normalmente esperamos de un handler de interrupción una ejecución muy rápida de modo que su trabajo no degrade la performance del sistema.

Top Halves y Bottom Halves

Normalmente esperamos de un handler de interrupción una ejecución muy rápida de modo que su trabajo no degrade la performance del sistema.
¿Que pasa cuando necesitamos además realizar una gran cantidad de trabajo?. Ambos objetivos parecen incompatibles
Linux lo resuelve dividiendo el trabajo en mitades (en inglés halves):

Top Halves y Bottom Halves

Normalmente esperamos de un handler de interrupción una ejecución muy rápida de modo que su trabajo no degrade la performance del sistema.

¿Que pasa cuando necesitamos además realizar una gran cantidad de trabajo?. Ambos objetivos parecen incompatibles

Linux lo resuelve dividiendo el trabajo en mitades (en inglés halves):

Top Halve Resuelve la cuestión crítica: Leer el dato (si se trata de una recepción), enviar el fin de la interrupción y de ser necesario resetear el hardware.

Top Halves y Bottom Halves

Normalmente esperamos de un handler de interrupción una ejecución muy rápida de modo que su trabajo no degrade la performance del sistema.
¿Que pasa cuando necesitamos además realizar una gran cantidad de trabajo?. Ambos objetivos parecen incompatibles
Linux lo resuelve dividiendo el trabajo en mitades (en inglés halves):

Top Halve Resuelve la cuestión crítica: Leer el dato (si se trata de una recepción), enviar el fin de la interrupción y de ser necesario resetear el hardware.

Bottom Halve Corre ya con las interrupciones habilitadas y en un momento mas oportuno para completar otros aspectos de la tarea.

Top Halves y Bottom Halves

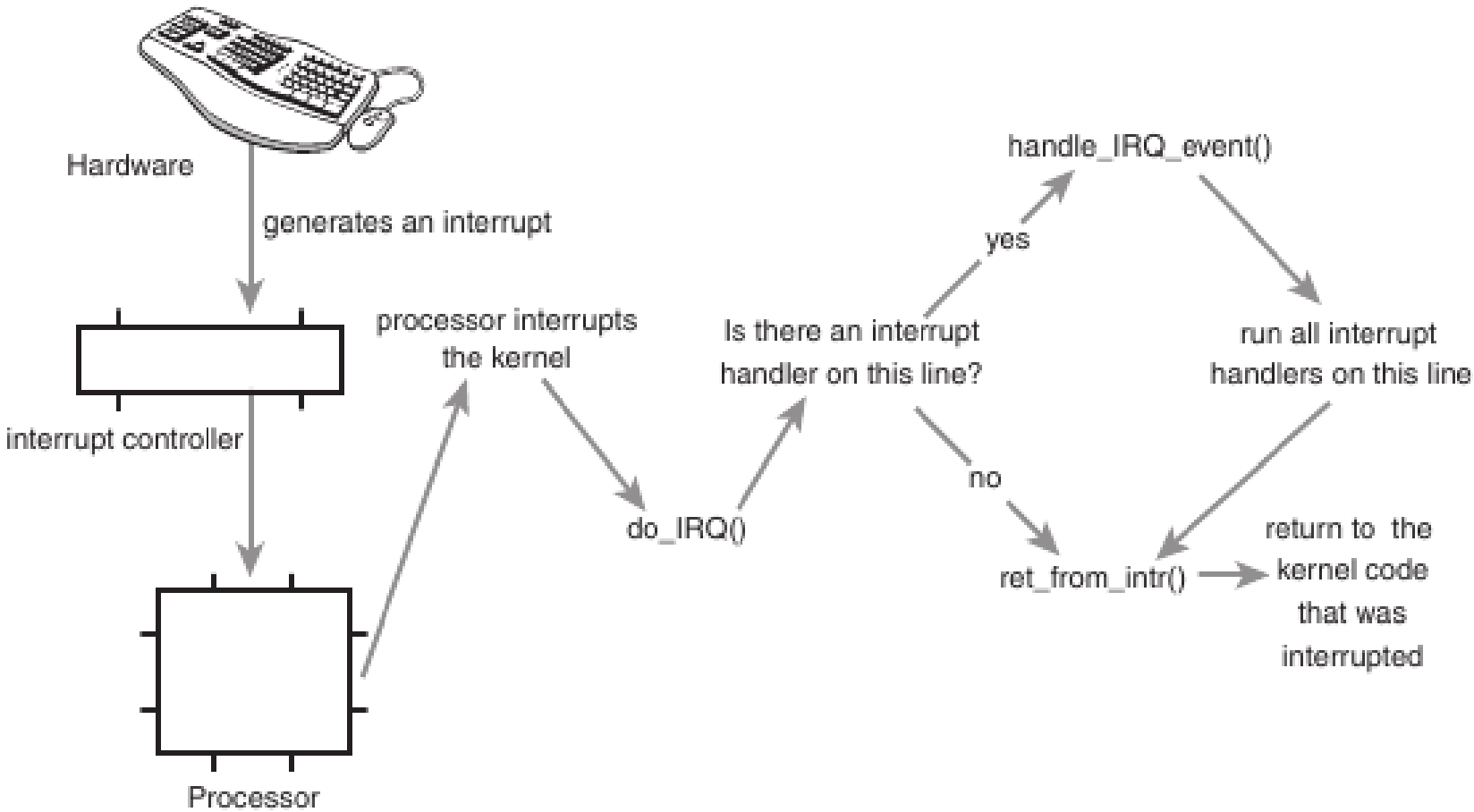
Normalmente esperamos de un handler de interrupción una ejecución muy rápida de modo que su trabajo no degrade la performance del sistema.
¿Que pasa cuando necesitamos además realizar una gran cantidad de trabajo?. Ambos objetivos parecen incompatibles
Linux lo resuelve dividiendo el trabajo en mitades (en inglés halves):

Top Halve Resuelve la cuestión crítica: Leer el dato (si se trata de una recepción), enviar el fin de la interrupción y de ser necesario resetear el hardware.

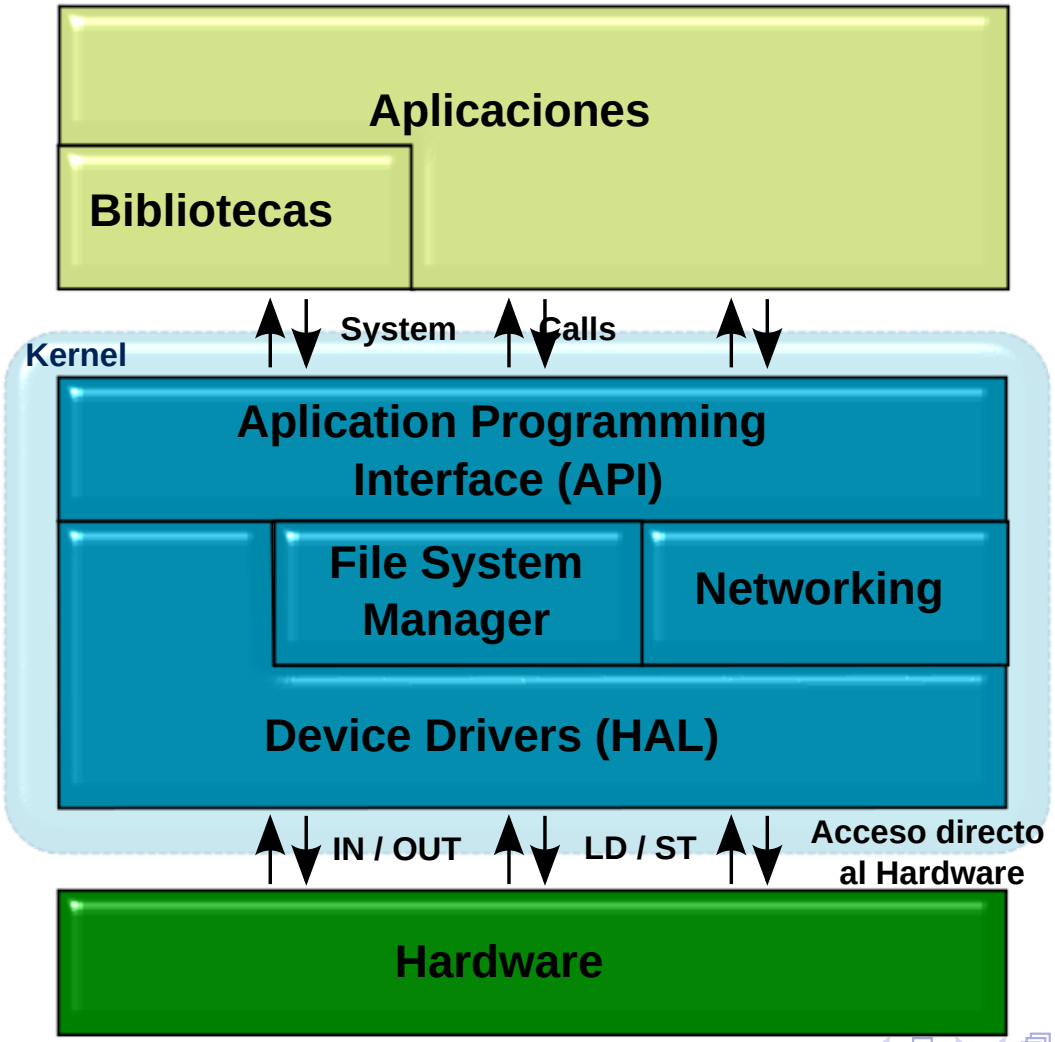
Bottom Halve Corre ya con las interrupciones habilitadas y en un momento mas oportuno para completar otros aspectos de la tarea.

Típico ejemplo: en una placa de red es necesario bajar el paquete a la memoria del sistema antes de que sea pisado con un nuevo paquete entrante, y poner la placa de red ready para seguir trabajando. El análisis en sí de la estructura del paquete recibido puede postergarse.

Manejo de una interrupción (sharing incluido)



Estructura de un Sistema Operativo



1. Descarga de la imagen del kernel

1. Descarga de la imagen del kernel

- Una vez que U-boot, coreboot, o el boot loader que utilicemos finaliza su tarea, se tiene la CPU el hardware y la memoria configurados, establecidos y listos para comenzar a operar.

1. Descarga de la imagen del kernel

- Una vez que U-boot, coreboot, o el boot loader que utilicemos finaliza su tarea, se tiene la CPU el hardware y la memoria configurados, establecidos y listos para comenzar a operar.
- Si hay un Disco RAM Inicial (Initrd) el gestor de arranque también debe cargarlo en la memoria. Initrd es un file system temporal que el Kernel utiliza durante el inicio del sistema para generar todas las acciones previas que permitan luego cargar el Root File System (sin el cual ningún sistema “UNIX like” puede trabajar).

1. Descarga de la imagen del kernel

- Una vez que U-boot, coreboot, o el boot loader que utilicemos finaliza su tarea, se tiene la CPU el hardware y la memoria configurados, establecidos y listos para comenzar a operar.
- Si hay un Disco RAM Inicial (Initrd) el gestor de arranque también debe cargarlo en la memoria. Initrd es un file system temporal que el Kernel utiliza durante el inicio del sistema para generar todas las acciones previas que permitan luego cargar el Root File System (sin el cual ningún sistema “UNIX like” puede trabajar).
- El siguiente paso es cargar el kernel de Linux, para lo cual se requiere ubicar el archivo que contiene su imagen dentro de alguno de los medios de almacenamiento.

1. Descarga de la imagen del kernel

- Una vez que U-boot, coreboot, o el boot loader que utilicemos finaliza su tarea, se tiene la CPU el hardware y la memoria configurados, establecidos y listos para comenzar a operar.
- Si hay un Disco RAM Inicial (Initrd) el gestor de arranque también debe cargarlo en la memoria. Initrd es un file system temporal que el Kernel utiliza durante el inicio del sistema para generar todas las acciones previas que permitan luego cargar el Root File System (sin el cual ningún sistema “UNIX like” puede trabajar).
- El siguiente paso es cargar el kernel de Linux, para lo cual se requiere ubicar el archivo que contiene su imagen dentro de alguno de los medios de almacenamiento.
- Seguidamente se la descomprime y se distribuyen las diferentes secciones acorde a como se las ha definido durante la compilación, información que se encuentra en el encabezado ELF del archivo que contiene la imagen de Linux

Pasaje de argumentos para configuración

Pasaje de argumentos para configuración

- El Kernel de Linux soporta la transferencia de argumentos por parte del programa de arranque (U-Boot por ejemplo).

Pasaje de argumentos para configuración

- El Kernel de Linux soporta la transferencia de argumentos por parte del programa de arranque (U-Boot por ejemplo).
- Para eso Linux dispone de medios genéricos para la transferencia de argumentos al Kernel a través de cualquier plataforma.

Pasaje de argumentos para configuración

- El Kernel de Linux soporta la transferencia de argumentos por parte del programa de arranque (U-Boot por ejemplo).
- Para eso Linux dispone de medios genéricos para la transferencia de argumentos al Kernel a través de cualquier plataforma.
- Normalmente el gestor de arranque configura un área de la memoria inicializándola con determinadas estructuras de datos estipuladas

Pasaje de argumentos para configuración

- El Kernel de Linux soporta la transferencia de argumentos por parte del programa de arranque (U-Boot por ejemplo).
- Para eso Linux dispone de medios genéricos para la transferencia de argumentos al Kernel a través de cualquier plataforma.
- Normalmente el gestor de arranque configura un área de la memoria inicializándola con determinadas estructuras de datos estipuladas
- Estas estructuras serán identificadas por el Kernel

Pasaje de argumentos para configuración

- El Kernel de Linux soporta la transferencia de argumentos por parte del programa de arranque (U-Boot por ejemplo).
- Para eso Linux dispone de medios genéricos para la transferencia de argumentos al Kernel a través de cualquier plataforma.
- Normalmente el gestor de arranque configura un área de la memoria inicializándola con determinadas estructuras de datos estipuladas
- Estas estructuras serán identificadas por el Kernel
- Solo es necesario cargar los valores deseados en estas estructuras.

Salto al punto de entrada del Kernel

Salto al punto de entrada del Kernel

- Cuando se compila el Kernel con ayuda del Linker define su punto de entrada.

Salto al punto de entrada del Kernel

- Cuando se compila el Kernel con ayuda del Linker define su punto de entrada.
- Una vez que el gestor de arranque salta al kernel, finaliza su tarea.

Salto al punto de entrada del Kernel

- Cuando se compila el Kernel con ayuda del Linker define su punto de entrada.
- Una vez que el gestor de arranque salta al kernel, finaliza su tarea.
- El Kernel entra a disponer del espacio en memoria utilizado por sistema de arranque, (debe tenerse en cuenta este detalle cuando se diseña el mapa de memoria del sistema).

Arranque del kernel

Consta de dos partes:

Arranque del kernel

Consta de dos partes:

1. **Inicialización de la plataforma (Board) y la CPU**.
   Es tal vez la mas compleja ya que lidia con todo el bajo nivel de hardware, tanto de la CPU como de los dispositivos de soporte de E/S básicos.

Arranque del kernel

Consta de dos partes:

1. **Inicialización de la plataforma (Board) y la CPU**.
   Es tal vez la mas compleja ya que lidia con todo el bajo nivel de hardware, tanto de la CPU como de los dispositivos de soporte de E/S básicos.

2. **Inicialización del Sub-sistema**.
   Se ocupa de poner en marcha los módulos básicos del sistema operativo que se apoyan en los recursos inicializados en el ítem anterior

Arranque del kernel: Board / CPU

Arranque del kernel: Board / CPU

- Si portamos una plataforma genérica para empezar un desarrollo (Ejemplo BleagleBone Black), esta fase está resuelta en el Board Support Package (BSP)

Arranque del kernel: Board / CPU

- Si portamos una plataforma genérica para empezar un desarrollo (Ejemplo BleagleBone Black), esta fase está resuelta en el Board Support Package (BSP)
- Si lo que estamos desarrollando es un sistema embebido personalizado (a partir de algún sistema genérico), esta fase consiste en adaptar el código del BSP para portarlo a nuestro desarrollo.

Arranque del kernel: Board / CPU

- Si portamos una plataforma genérica para empezar un desarrollo (Ejemplo BleagleBone Black), esta fase está resuelta en el Board Support Package (BSP)
- Si lo que estamos desarrollando es un sistema embebido personalizado (a partir de algún sistema genérico), esta fase consiste en adaptar el código del BSP para portarlo a nuestro desarrollo.
- El primer escenario es el preliminar en un proyecto al inicio.

Arranque del kernel: Board / CPU

- Si portamos una plataforma genérica para empezar un desarrollo (Ejemplo BleagleBone Black), esta fase está resuelta en el Board Support Package (BSP)
- Si lo que estamos desarrollando es un sistema embebido personalizado (a partir de algún sistema genérico), esta fase consiste en adaptar el código del BSP para portarlo a nuestro desarrollo.
- El primer escenario es el preliminar en un proyecto al inicio.
- El segundo escenario es nuestro trabajo habitualmente cuando ya tenemos definido el hardware definitivo de un proyecto.Este nunca es igual al del board de desarrollo y por lo tanto hay que portar el BSP genérico a nuestra plataforma específica

Arranque del kernel: Board / CPU

- Si portamos una plataforma genérica para empezar un desarrollo (Ejemplo BleagleBone Black), esta fase está resuelta en el Board Support Package (BSP)
- Si lo que estamos desarrollando es un sistema embebido personalizado (a partir de algún sistema genérico), esta fase consiste en adaptar el código del BSP para portarlo a nuestro desarrollo.
- El primer escenario es el preliminar en un proyecto al inicio.
- El segundo escenario es nuestro trabajo habitualmente cuando ya tenemos definido el hardware definitivo de un proyecto.Este nunca es igual al del board de desarrollo y por lo tanto hay que portar el BSP genérico a nuestra plataforma específica
- La inicialización de la plataforma / CPU se compone de los pasos siguientes:

Arranque del kernel: Board / CPU

Arranque del kernel: Board / CPU

1. El punto de entrada del Kernel se trata de una rutina escrita en assembler. Depende específicamente de la plataforma. Su nombre varía pero normalmente se encuentra en un archivo llamado `head.s`. Realiza las siguientes actividades.

Arranque del kernel: Board / CPU

1. El punto de entrada del Kernel se trata de una rutina escrita en assembler. Depende específicamente de la plataforma. Su nombre varía pero normalmente se encuentra en un archivo llamado `head.s`. Realiza las siguientes actividades.
    1. Activar el MMU del procesador. Muchos de los gestores de arranque no activan la MMU o la manejan con identity mapping, por lo tanto la dirección virtual concuerda con la física. Como el kernel se compila con la direcciones virtuales, al activar la MMU el Kernel utiliza direcciones virtuales sin incurrir en errores.

Arranque del kernel: Board / CPU

1. El punto de entrada del Kernel se trata de una rutina escrita en assembler. Depende específicamente de la plataforma. Su nombre varía pero normalmente se encuentra en un archivo llamado `head.s`. Realiza las siguientes actividades.
    1. Activar el MMU del procesador. Muchos de los gestores de arranque no activan la MMU o la manejan con identity mapping, por lo tanto la dirección virtual concuerda con la física. Como el kernel se compila con la direcciones virtuales, al activar la MMU el Kernel utiliza direcciones virtuales sin incurrir en errores.
    2. De acuerdo a la plataforma de Hardware lo siguiente es activar la memoria cache (en caso que exista).

Arranque del kernel: Board / CPU

1. El punto de entrada del Kernel se trata de una rutina escrita en assembler. Depende específicamente de la plataforma. Su nombre varía pero normalmente se encuentra en un archivo llamado `head.s`. Realiza las siguientes actividades.
   1. Activar el MMU del procesador. Muchos de los gestores de arranque no activan la MMU o la manejan con identity mapping, por lo tanto la dirección virtual concuerda con la física. Como el kernel se compila con la direcciones virtuales, al activar la MMU el Kernel utiliza direcciones virtuales sin incurrir en errores.
   2. De acuerdo a la plataforma de Hardware lo siguiente es activar la memoria cache (en caso que exista).
   3. Inicializa el BSS (block started by symbol), segmento de datos no inicializados en cero.

Arranque del kernel: Board / CPU

1. El punto de entrada del Kernel se trata de una rutina escrita en assembler. Depende específicamente de la plataforma. Su nombre varía pero normalmente se encuentra en un archivo llamado `head.s`. Realiza las siguientes actividades.
    1. Activar el MMU del procesador. Muchos de los gestores de arranque no activan la MMU o la manejan con identity mapping, por lo tanto la dirección virtual concuerda con la física. Como el kernel se compila con la direcciones virtuales, al activar la MMU el Kernel utiliza direcciones virtuales sin incurrir en errores.
    2. De acuerdo a la plataforma de Hardware lo siguiente es activar la memoria cache (en caso que exista).
    3. Inicializa el BSS (block started by symbol), segmento de datos no inicializados en cero.
    4. Configurar la pila

Arranque del kernel: Board / CPU

1. El punto de entrada del Kernel se trata de una rutina escrita en assembler. Depende específicamente de la plataforma. Su nombre varía pero normalmente se encuentra en un archivo llamado `head.s`. Realiza las siguientes actividades.
   1. Activar el MMU del procesador. Muchos de los gestores de arranque no activan la MMU o la manejan con identity mapping, por lo tanto la dirección virtual concuerda con la física. Como el kernel se compila con la direcciones virtuales, al activar la MMU el Kernel utiliza direcciones virtuales sin incurrir en errores.
   2. De acuerdo a la plataforma de Hardware lo siguiente es activar la memoria cache (en caso que exista).
   3. Inicializa el BSS (block started by symbol), segmento de datos no inicializados en cero.
   4. Configurar la pila
   5. Invoca a la función **`start_kernel()`** (init/main.c), que se encarga de funciones algo mas complejas e inicia un proceso con PID=0 que se encarga del resto del arranque.

Arranque del kernel: Board / CPU

Arranque del kernel: Board / CPU

2. Se invoca desde el proceso PID=0 a la función `setup_arch()`, que realiza la inicialización de la plataforma/CPU específica, asegurando que las demás inicializaciones puedan ser invocadas de forma segura. Es altamente dependiente de la plataforma. Entre otras cosas se encarga de:

Arranque del kernel: Board / CPU

2. Se invoca desde el proceso PID=0 a la función `setup_arch()`, que realiza la inicialización de la plataforma/CPU específica, asegurando que las demás inicializaciones puedan ser invocadas de forma segura. Es altamente dependiente de la plataforma. Entre otras cosas se encarga de:

    1. Inicializaciones específicas del procesador

Arranque del kernel: Board / CPU

2. Se invoca desde el proceso PID=0 a la función `setup_arch()`, que realiza la inicialización de la plataforma/CPU específica, asegurando que las demás inicializaciones puedan ser invocadas de forma segura. Es altamente dependiente de la plataforma. Entre otras cosas se encarga de:

    1. Inicializaciones específicas del procesador
    2. Inicializaciones específicas del Board.

Arranque del kernel: Board / CPU

2. Se invoca desde el proceso PID=0 a la función `setup_arch()`, que realiza la inicialización de la plataforma/CPU específica, asegurando que las demás inicializaciones puedan ser invocadas de forma segura. Es altamente dependiente de la plataforma. Entre otras cosas se encarga de:

    1. Inicializaciones específicas del procesador
    2. Inicializaciones específicas del Board.
    3. Analiza los parámetros pasados al kernel

Arranque del kernel: Board / CPU

2. Se invoca desde el proceso PID=0 a la función `setup_arch()`, que realiza la inicialización de la plataforma/CPU específica, asegurando que las demás inicializaciones puedan ser invocadas de forma segura. Es altamente dependiente de la plataforma. Entre otras cosas se encarga de:

    1. Inicializaciones específicas del procesador
    2. Inicializaciones específicas del Board.
    3. Analiza los parámetros pasados al kernel
    4. Verifica la configuración del RAMDisk hecha por el programa de arranque, para que el kernel pueda montarlo como Root FS.

Arranque del kernel: Board / CPU

2. Se invoca desde el proceso PID=0 a la función `setup_arch()`, que realiza la inicialización de la plataforma/CPU específica, asegurando que las demás inicializaciones puedan ser invocadas de forma segura. Es altamente dependiente de la plataforma. Entre otras cosas se encarga de:

    1. Inicializaciones específicas del procesador
    2. Inicializaciones específicas del Board.
    3. Analiza los parámetros pasados al kernel
    4. Verifica la configuración del RAMDisk hecha por el programa de arranque, para que el kernel pueda montarlo como Root FS.
    5. Llama a las funciones Bootmem. Estas definen la memoria inicial que el Kernel puede reservar para diferentes propósitos (por ejemplo el manejo de DMA si es necesario) antes de que el código de paginación organice toda la memoria.

Arranque del kernel: Board / CPU

2. Se invoca desde el proceso PID=0 a la función `setup_arch()`, que realiza la inicialización de la plataforma/CPU específica, asegurando que las demás inicializaciones puedan ser invocadas de forma segura. Es altamente dependiente de la plataforma. Entre otras cosas se encarga de:

    1. Inicializaciones específicas del procesador
    2. Inicializaciones específicas del Board.
    3. Analiza los parámetros pasados al kernel
    4. Verifica la configuración del RAMDisk hecha por el programa de arranque, para que el kernel pueda montarlo como Root FS.
    5. Llama a las funciones Bootmem. Estas definen la memoria inicial que el Kernel puede reservar para diferentes propósitos (por ejemplo el manejo de DMA si es necesario) antes de que el código de paginación organice toda la memoria.
    6. Llama a la función de inicialización de paginación que toma el control del resto de la memoria y la administra en base a Páginas.

Arranque del kernel: Board / CPU

Arranque del kernel: Board / CPU

3. Inicializa las Excepciones, función `trap_init()`.Setea los manejadores de excepciones referentes al Kernel específico. Previo a esto si ocurre una excepción la respuesta depende de la configuración de la plataforma.

Arranque del kernel: Board / CPU

3. Inicializa las Excepciones, función `trap_init()`.Setea los manejadores de excepciones referentes al Kernel específico. Previo a esto si ocurre una excepción la respuesta depende de la configuración de la plataforma.

4. Inicializa el proceso del manejo de interrupciones,función `init_IRQ()`. Inicializa el controlador y los descriptores de interrupciones (estos son estructuras de datos empleadas por el BSP para asignar a cada fuente de interrupción su subrutina de servicio). **Las interrupciones no se encuentran habilitadas en este punto**. Los drivers contienen el código de habilitación de las interrupciones, que se ejecutará durante su inicialización pero son cargados a posteriori de esta función. Por ejemplo la inicialización del timer se asegurará de que su interrupción es habilitada.

Arranque del kernel: Board / CPU

Arranque del kernel: Board / CPU

5. Inicialización de Timers, función `time_init()`. Inicializa la señal del temporizador de Hardware para se comience a generar la señal de reloj con la cual funciona el sistema, es decir el Timer Tick.

Arranque del kernel: Board / CPU

5. Inicialización de Timers, función `time_init()`. Inicializa la señal del temporizador de Hardware para se comience a generar la señal de reloj con la cual funciona el sistema, es decir el Timer Tick.
6. Inicialización de la consola, función `console_init()`. Inicializa el dispositivo serial como una consola. Una vez esta está activada, todos los mensajes de arranque son presentados en la pantalla del sistema si la posee, de lo contrario será enviada por el puerto serial. Para imprimir un mensaje desde el Kernel se emplea la función `printk()`.

Arranque del kernel: Board / CPU

5. Inicialización de Timers, función **`time_init()`**. Inicializa la señal del temporizador de Hardware para se comience a generar la señal de reloj con la cual funciona el sistema, es decir el Timer Tick.
6. Inicialización de la consola, función **`console_init()`**. Inicializa el dispositivo serial como una consola. Una vez esta está activada, todos los mensajes de arranque son presentados en la pantalla del sistema si la posee, de lo contrario será enviada por el puerto serial. Para imprimir un mensaje desde el Kernel se emplea la función **`printk()`**.
7. Calculando ciclos de espera para la plataforma, función **`calibrate_delay`** Cuando el kernel necesita implementar esperas de microsegundos utiliza la función **`udelay()`**. Para esto el kernel tiene que conocer con exactitud el número de ciclos de reloj por microsegundo . Este función calibra el número de ciclos de espera, en base a las interrupciones del timer asegurando que los ciclos de espera trabajan de forma uniforme para cualquier plataforma.

Arranque del kernel: Inicialización del Sub-sistema

Arranque del kernel: Inicialización del Sub-sistema

1. La mayoría de las inicializaciones del subsistema son realizadas durante la ejecución de la función `start_kernel()`. Comprende:

Arranque del kernel: Inicialización del Sub-sistema

1. La mayoría de las inicializaciones del subsistema son realizadas durante la ejecución de la función `start_kernel()`. Comprende:
    1. Inicialización del Scheduler.

Arranque del kernel: Inicialización del Sub-sistema

1. La mayoría de las inicializaciones del subsistema son realizadas durante la ejecución de la función `start_kernel()`. Comprende:
    1. Inicialización del Scheduler.
    2. Inicialización del administrador de memoria.

Arranque del kernel: Inicialización del Sub-sistema

1. La mayoría de las inicializaciones del subsistema son realizadas durante la ejecución de la función `start_kernel()`. Comprende:
    1. Inicialización del Scheduler.
    2. Inicialización del administrador de memoria.
    3. Inicialización del VFS.

Arranque del kernel: Inicialización del Sub-sistema

1. La mayoría de las inicializaciones del subsistema son realizadas durante la ejecución de la función `start_kernel()`. Comprende:
   1. Inicialización del Scheduler.
   2. Inicialización del administrador de memoria.
   3. Inicialización del VFS.
2. Una vez que ésta finaliza, el Kernel crea otro proceso llamado proceso de inicio que realiza el resto de la inicialización: drivers, llamadas de inicio, carga del root FS y salto al espacio de usuario.

Arranque del kernel: Inicialización del Sub-sistema

1. La mayoría de las inicializaciones del subsistema son realizadas durante la ejecución de la función `start_kernel()`. Comprende:
   1. Inicialización del Scheduler.
   2. Inicialización del administrador de memoria.
   3. Inicialización del VFS.
2. Una vez que ésta finaliza, el Kernel crea otro proceso llamado proceso de inicio que realiza el resto de la inicialización: drivers, llamadas de inicio, carga del root FS y salto al espacio de usuario.
3. Este proceso es el que se convierte en el proceso 1.

Inicialización de Drivers

- Se realiza después de que la administración de memoria y de procesos se encuentran activos,
- Se realiza en el contexto del proceso de inicio.

Carga del root File System

Carga del root File System

- Es el punto de origen desde el cual se pueden cargar el resto de los file systems y nodos.

Carga del root File System

- Es el punto de origen desde el cual se pueden cargar el resto de los file systems y nodos.
- Se trata de un elemento indispensable

Carga del root File System

- Es el punto de origen desde el cual se pueden cargar el resto de los file systems y nodos.
- Se trata de un elemento indispensable
- El registro de arranque se puede establecer en el momento de compilar el kernel, o al cargarlo en la memoria pasándole como argumento “root=”.

Carga del root File System

- Es el punto de origen desde el cual se pueden cargar el resto de los file systems y nodos.
- Se trata de un elemento indispensable
- El registro de arranque se puede establecer en el momento de compilar el kernel, o al cargarlo en la memoria pasándole como argumento “root=”.
- El bloque que contiene la descripción del FS puede estar en:

Carga del root File System

- Es el punto de origen desde el cual se pueden cargar el resto de los file systems y nodos.
- Se trata de un elemento indispensable
- El registro de arranque se puede establecer en el momento de compilar el kernel, o al cargarlo en la memoria pasándole como argumento “root=”.
- El bloque que contiene la descripción del FS puede estar en:
  - RAM Disk (initrd). Mapea un disco en la memoria RAM. Fundamental cuando no se tiene listo el drver de flash.
  - Network File System.
  - Flash

Carga del root File System

- Es el punto de origen desde el cual se pueden cargar el resto de los file systems y nodos.
- Se trata de un elemento indispensable
- El registro de arranque se puede establecer en el momento de compilar el kernel, o al cargarlo en la memoria pasándole como argumento “root=”.
- El bloque que contiene la descripción del FS puede estar en:
  - RAM Disk (initrd). Mapea un disco en la memoria RAM. Fundamental cuando no se tiene listo el drver de flash.
  - Network File System.
  - Flash
- El primero se usa para compilaciones de debuging, y los dos restantes para compilaciones de producción

Carga del root File System: *Initrd*

Carga del root File System: *Initrd*

- A pesar de su versatilidad durante la construcción del sistema para debuging, el RAM Disk es utilizado como root durante el proceso de arranque (Initrd).

Carga del root File System: *Initrd*

- A pesar de su versatilidad durante la construcción del sistema para debuging, el RAM Disk es utilizado como root durante el proceso de arranque (Initrd).
- Además puede utilizarse para cargar un File System generando una imagen de éste y luego replicándola en el medio de storage una vez cargado el driver de éste.

Carga del root File System: *Initrd*

- A pesar de su versatilidad durante la construcción del sistema para debuging, el RAM Disk es utilizado como root durante el proceso de arranque (Initrd).
- Además puede utilizarse para cargar un File System generando una imagen de éste y luego replicándola en el medio de storage una vez cargado el driver de éste.
- Especialmente práctico en las primeras etapas de diseño de Linux Embebido cuando no se cuenta con un driver para Flash pero las aplicaciones se encuentran listas para hacer pruebas

Carga del root File System: *Initrd*

- A pesar de su versatilidad durante la construcción del sistema para debuging, el RAM Disk es utilizado como root durante el proceso de arranque (Initrd).
- Además puede utilizarse para cargar un File System generando una imagen de éste y luego replicándola en el medio de storage una vez cargado el driver de éste.
- Especialmente práctico en las primeras etapas de diseño de Linux Embebido cuando no se cuenta con un driver para Flash pero las aplicaciones se encuentran listas para hacer pruebas
- Para que el Kernel cargue Initrd, se lo debe configurar durante la compilación con la opción CONFIG_BVLK_DEV_INITRD.

Llamada a Initcall y libera la memoria inicial

Llamada a Initcall y libera la memoria inicial

- Si se observa el script del enlazador de cualquier arquitectura, este tendrá una sección de arranque, el inicio y fin de esta sección se marca con: __init_begin e __init_end

Llamada a Initcall y libera la memoria inicial

- Si se observa el script del enlazador de cualquier arquitectura, este tendrá una sección de arranque, el inicio y fin de esta sección se marca con: __init_begin e __init_end
- La idea de este espacio es el de contener código y datos que puedan ser desechados una vez el booteado el sistema. Ej: funciones de inicialización de drivers. Se agrupa todo el código para que quede disponible al liberarse un chunk importante de memoria que pueda paginarse.

Llamada a Initcall y libera la memoria inicial

- Si se observa el script del enlazador de cualquier arquitectura, este tendrá una sección de arranque, el inicio y fin de esta sección se marca con: __init_begin e __init_end
- La idea de este espacio es el de contener código y datos que puedan ser desechados una vez el booteado el sistema. Ej: funciones de inicialización de drivers. Se agrupa todo el código para que quede disponible al liberarse un chunk importante de memoria que pueda paginarse.
- Las funciones que se quieren invocar durante la inicialización se declaran con la directiva __initcall

Mover al espacio de Usuario.

Mover al espacio de Usuario.

- El Kernel que se ejecuta en el contexto del proceso de arranque salta al espacio de usuario llamando la función execve reemplazando su propio código de inicialización (no el resto!!!) con el ejecutable de la imagen de un programa especial conocido como ***init***.

Mover al espacio de Usuario.

- El Kernel que se ejecuta en el contexto del proceso de arranque salta al espacio de usuario llamando la función execve reemplazando su propio código de inicialización (no el resto!!!) con el ejecutable de la imagen de un programa especial conocido como ***init***.

- Este ejecutable normalmente se encuentra en la Raíz en el archivo /sbin.