Sistemas Operativos

Alejandro Furfaro

28 de septiembre de 2020

Temario

Temario

¿Que hace un sistema operativo?

¿Que hace un sistema operativo?

- CPU, memoria, dispositivos de Entrada Salida. . . Sin software, son meros circuitos electrónicos.

¿Que hace un sistema operativo?

- CPU, memoria, dispositivos de Entrada Salida. . . Sin software, son meros circuitos electrónicos.
- Firefox, gcc, Sublime Text, Slack, Telegram, vlc, spotify, Document Viewer. . . Aplicaciones muy útiles. Pero si todas juntas quieren acceder al hardware. . . caos.

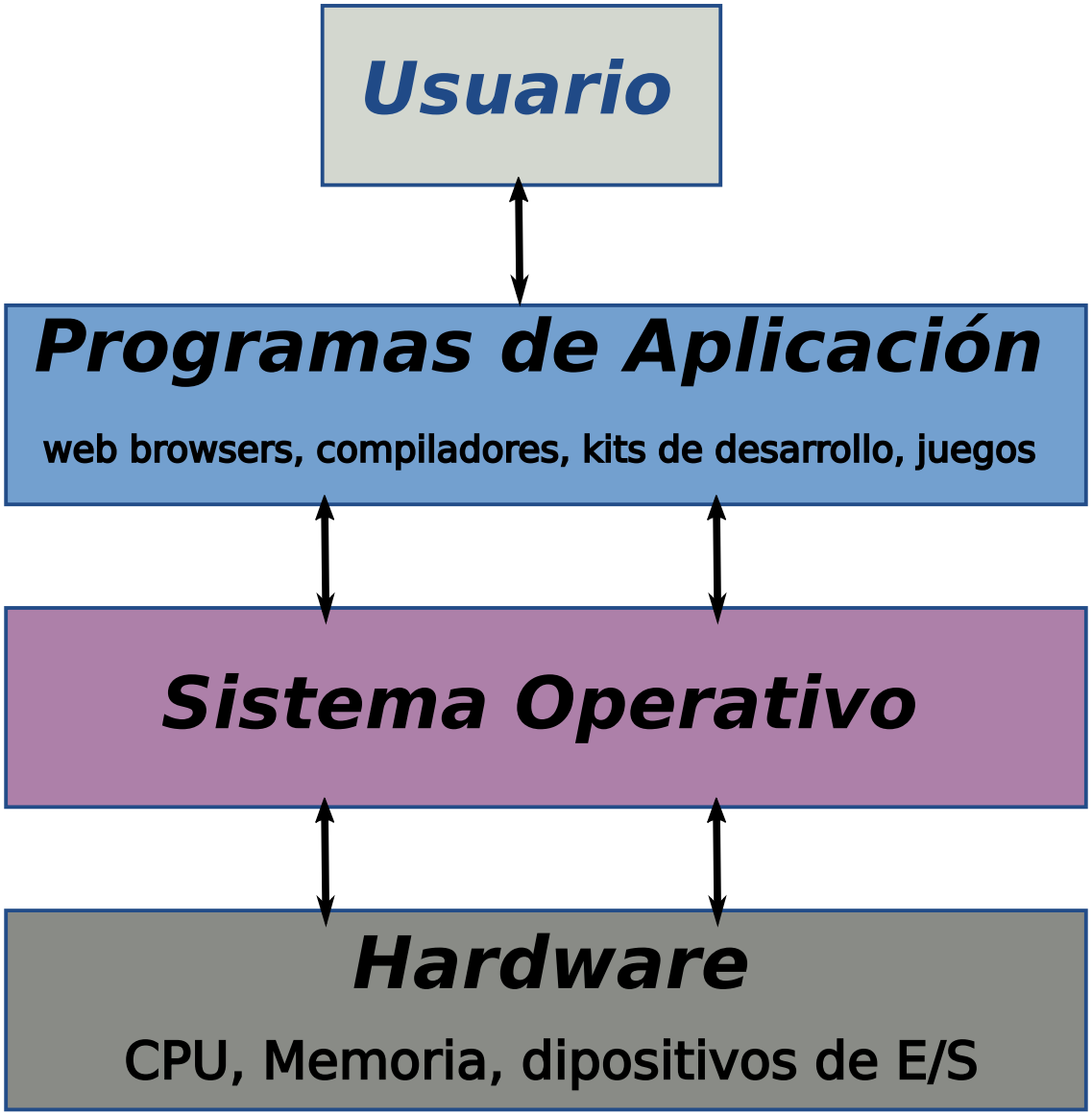
¿Que hace un sistema operativo?

- CPU, memoria, dispositivos de Entrada Salida. . . Sin software, son meros circuitos electrónicos.
- Firefox, gcc, Sublime Text, Slack, Telegram, vlc, spotify, Document Viewer. . . Aplicaciones muy útiles. Pero si todas juntas quieren acceder al hardware. . . caos.
- Deberían ponerse de acuerdo entre las diferentes aplicaciones en todo momento para acceder a áreas de memoria sin interferirse una a otras, para configurar el hardware con determinadas propiedades y utilizarlo en forma serializada también.

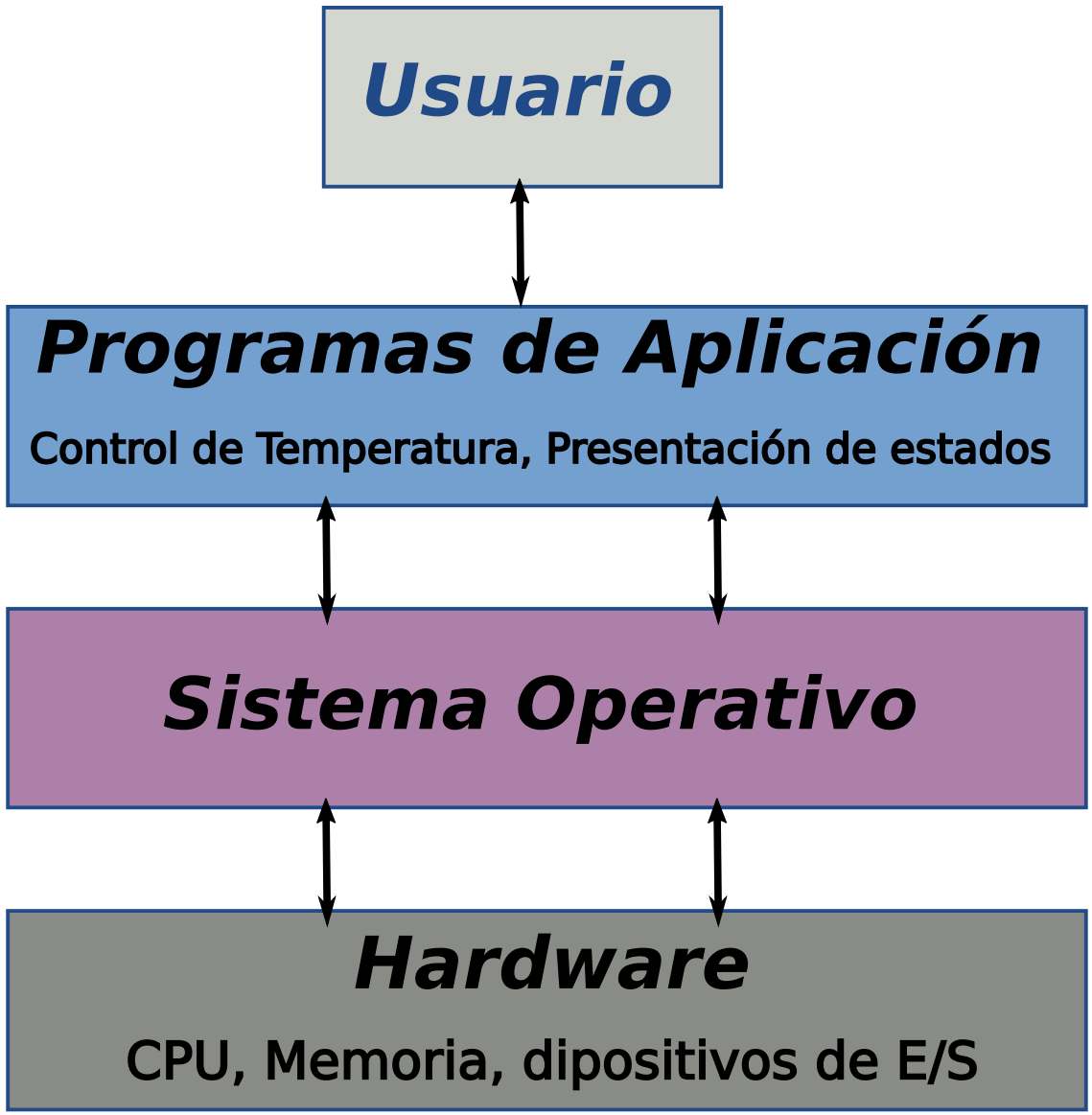
¿Que hace un sistema operativo?

- CPU, memoria, dispositivos de Entrada Salida. . . Sin software, son meros circuitos electrónicos.
- Firefox, gcc, Sublime Text, Slack, Telegram, vlc, spotify, Document Viewer. . . Aplicaciones muy útiles. Pero si todas juntas quieren acceder al hardware. . . caos.
- Deberían ponerse de acuerdo entre las diferentes aplicaciones en todo momento para acceder a áreas de memoria sin interferirse una a otras, para configurar el hardware con determinadas propiedades y utilizarlo en forma serializada también.
- ¿Hace falta mas para darse cuenta que se necesita una capa tipo “man in the middle”que intermedie entre el hardware y las aplicaciones?

Arquitectura de un sistema de extremo a extremo



Otra versión mas “embedded like”



Primeras conclusiones

No importan tanto las características del hardware. Si lo que vamos a desarrollar está organizado en forma de diferentes tareas que ejecutan en forma concurrente, necesitamos una pieza de software que administre la forma en que los recursos de hardware se van a a asignar a cada tarea (o proceso).
De otra manera no existe protección entre tareas, y no se puede proveer sincronización entre los diferentes procesos para acceder a un mismo recurso, por citar solo las calamidades mas evidentes.

Rangos de Sistemas Operativos

Rangos de Sistemas Operativos

- La escala del desarrollo del Sistema Operativo depende de las funciones que se esperan del sistema de cómputo a desarrollar y de las posibilidades del Hardware.

Rangos de Sistemas Operativos

- La escala del desarrollo del Sistema Operativo depende de las funciones que se esperan del sistema de cómputo a desarrollar y de las posibilidades del Hardware.
- Podemos implementar un kernel bare metal, muy simple con algunas tareas estáticas, con escasa o ninguna protección mas que la confiabilidad de nuestro código, hasta un Sistema de alto desempeño como Linux, pasando por todos los estadios intermedios

Componentes de un Sistema Operativo

Componentes de un Sistema Operativo

- Un sistema Operativo es básicamente un administrador de recursos.

Componentes de un Sistema Operativo

- Un sistema Operativo es básicamente un administrador de recursos.
- Los principales recursos del sistema son la CPU, la Memoria, los dispositivos de entrada salida, y en caso de que haya algún sistema de storage, éste se convierte también en un recurso a administrar.

Componentes de un Sistema Operativo

- Un sistema Operativo es básicamente un administrador de recursos.
- Los principales recursos del sistema son la CPU, la Memoria, los dispositivos de entrada salida, y en caso de que haya algún sistema de storage, éste se convierte también en un recurso a administrar.
- Los usuarios de esos recursos son los procesos o tareas. Dicho de manera muy rudimentaria los programas que se están ejecutando en la memoria del sistema.

Diagrama General de un Sistema Operativo

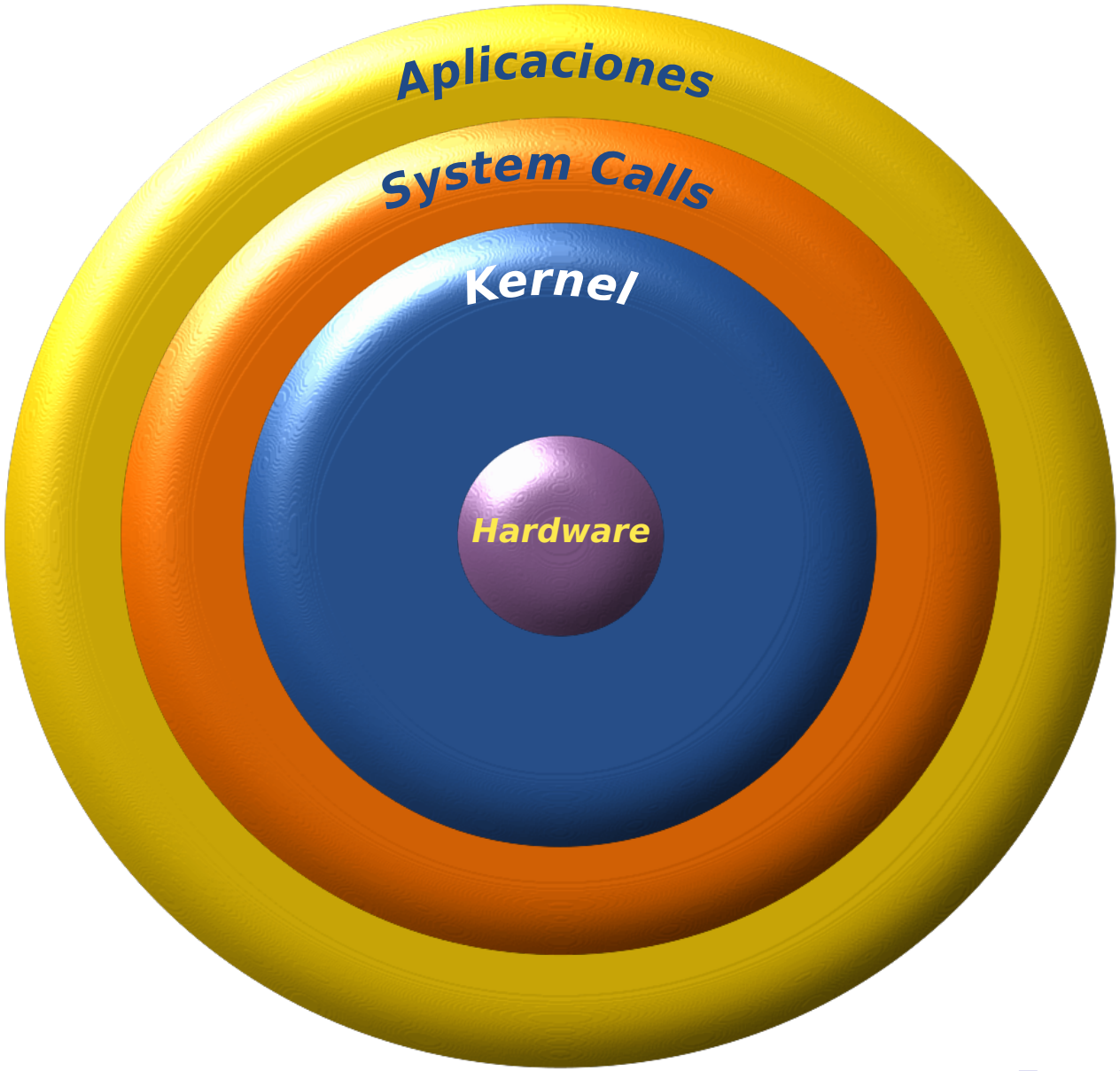


Diagrama General de un Sistema Operativo

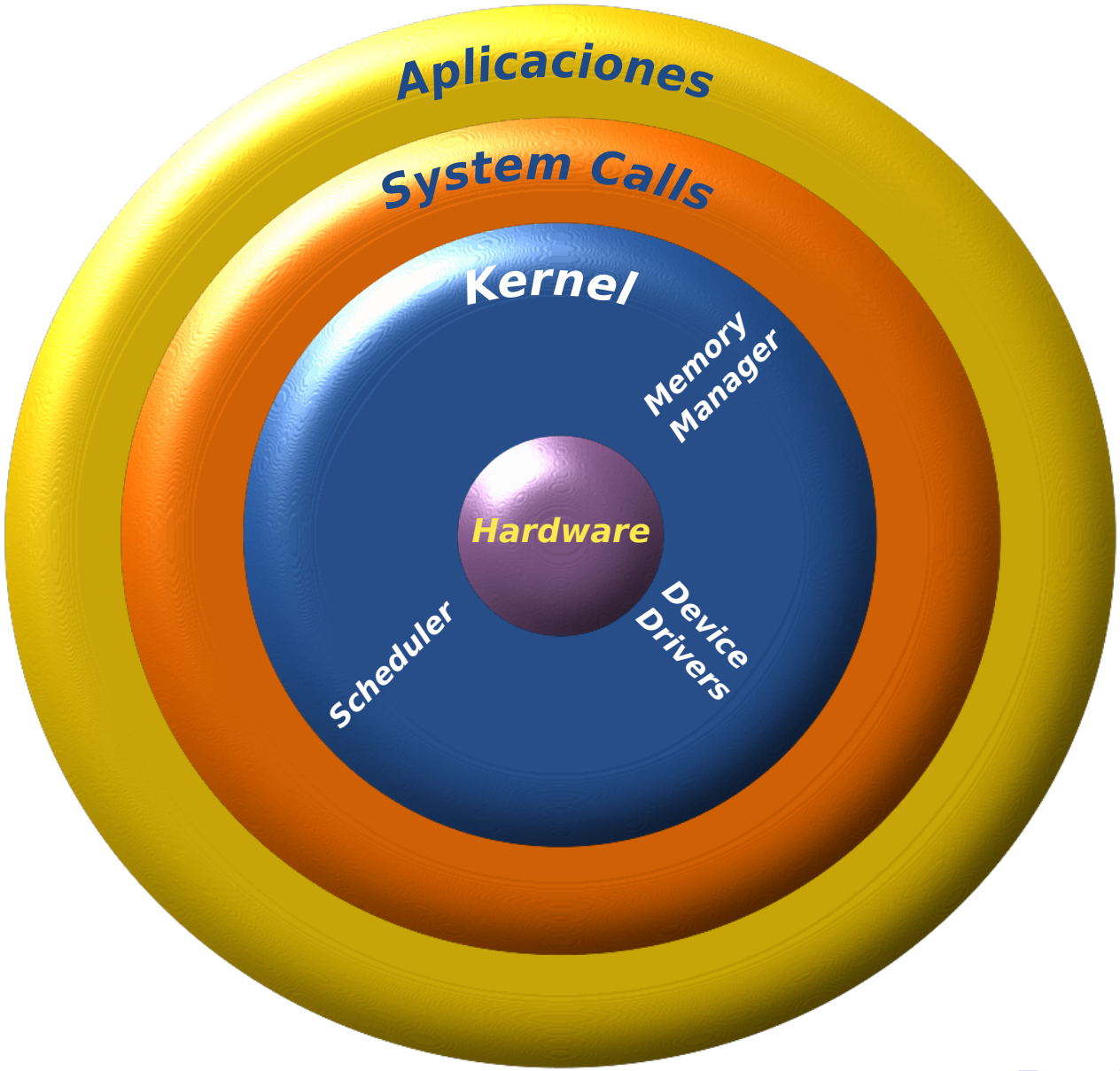
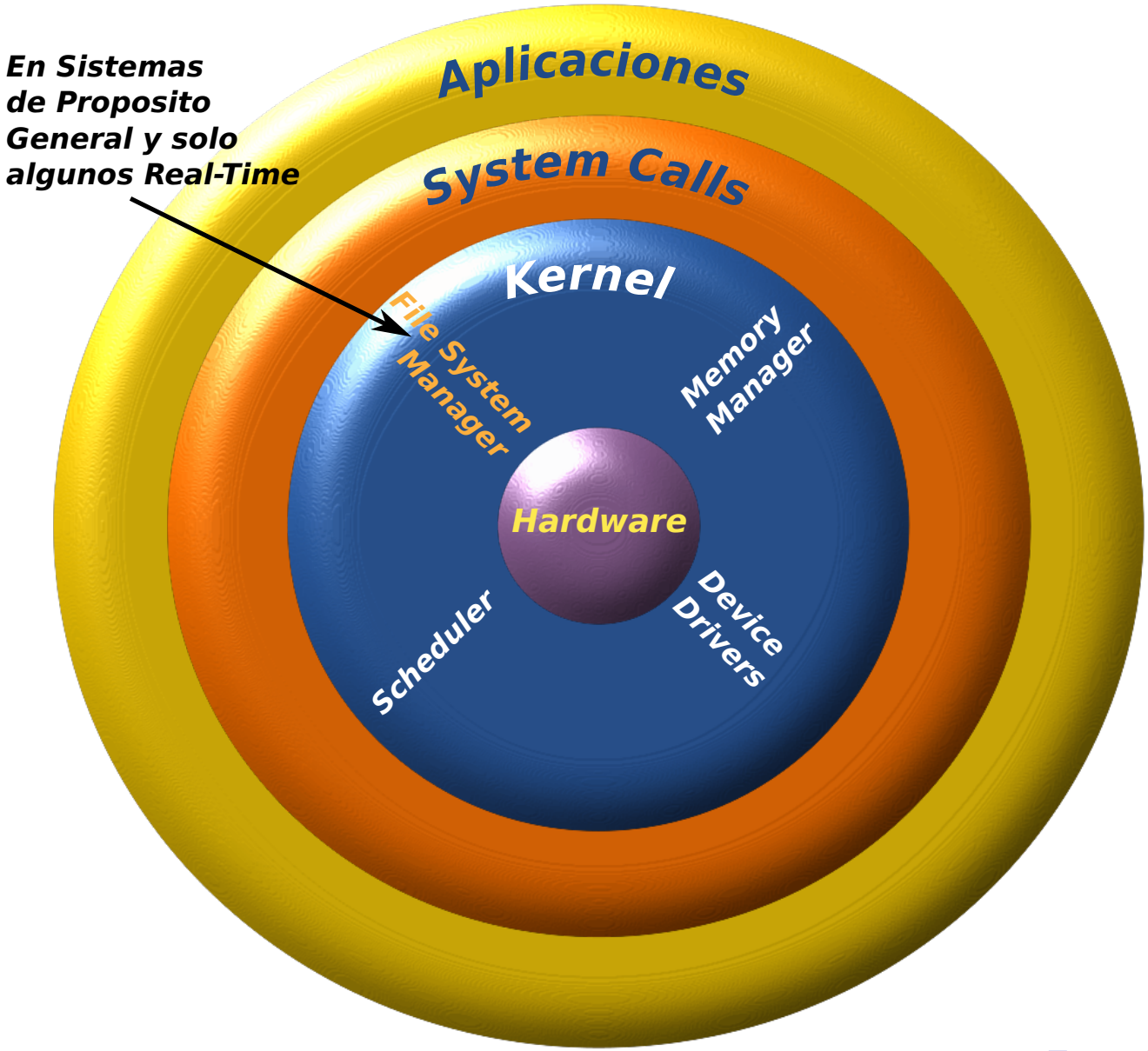


Diagrama General de un Sistema Operativo



Temario

Scheduling de Procesos

Scheduling de Procesos

- Hemos dicho que los procesos son los principales usuarios de los recursos del computador administrados bajo el Sistema Operativo.

Scheduling de Procesos

- Hemos dicho que los procesos son los principales usuarios de los recursos del computador administrados bajo el Sistema Operativo.
- Un proceso necesita como mínimo para iniciar su ejecución dos elementos sin los cuales no le es posible operar: CPU, y Memoria.

Scheduling de Procesos

- Hemos dicho que los procesos son los principales usuarios de los recursos del computador administrados bajo el Sistema Operativo.
- Un proceso necesita como mínimo para iniciar su ejecución dos elementos sin los cuales no le es posible operar: CPU, y Memoria.
- En un sistema multitasking hay una cantidad de procesos en condiciones de ser ejecutados. Esta cantidad normalmente es mayor que la cantidad de CPUs que tiene el sistema.

Scheduling de Procesos

- Hemos dicho que los procesos son los principales usuarios de los recursos del computador administrados bajo el Sistema Operativo.
- Un proceso necesita como mínimo para iniciar su ejecución dos elementos sin los cuales no le es posible operar: CPU, y Memoria.
- En un sistema multitasking hay una cantidad de procesos en condiciones de ser ejecutados. Esta cantidad normalmente es mayor que la cantidad de CPUs que tiene el sistema.
- El scheduler es el módulo del sistema operativo encargado de administrar la ejecución de los procesos, es decir es quien les asigna uno de los recursos fundamentales: la CPU.

Scheduling de Procesos

- Hemos dicho que los procesos son los principales usuarios de los recursos del computador administrados bajo el Sistema Operativo.
- Un proceso necesita como mínimo para iniciar su ejecución dos elementos sin los cuales no le es posible operar: CPU, y Memoria.
- En un sistema multitasking hay una cantidad de procesos en condiciones de ser ejecutados. Esta cantidad normalmente es mayor que la cantidad de CPUs que tiene el sistema.
- El scheduler es el módulo del sistema operativo encargado de administrar la ejecución de los procesos, es decir es quien les asigna uno de los recursos fundamentales: la CPU.

El Scheduler determina cuando y sobre cual de las CPUs ejecutarán cada uno de los procesos a fin de lograr el mejor rendimiento posible del sistema.

Aclarando términos asociados a scheduling

Aclarando términos asociados a scheduling

En computación el término **I/O-Bound**, se refiere a actividades de cómputo tales que la mayor parte de su tiempo de ejecución está ligado a actividades de Entrada Salida. En general se trata de procesos con mucha interacción de usuario y de los que se requiere un tiempo de respuesta muy bajo.

Aclarando términos asociados a scheduling

En computación el término **I/O-Bound**, se refiere a actividades de cómputo tales que la mayor parte de su tiempo de ejecución está ligado a actividades de Entrada Salida. En general se trata de procesos con mucha interacción de usuario y de los que se requiere un tiempo de respuesta muy bajo.

Por su parte el término **CPU-Bound** se aplica a actividades de cómputo con procesamiento intensivo y cuyo principal componente de tiempo de ejecución es la CPU, como por ejemplo, compilación de programas, algoritmos de ordenamiento de arreglos de memoria muy grande, o de compresión, por citar algunos ejemplos comunes.

Aclarando términos asociados a scheduling

El término **Tiempo de Respuesta**, es la medida de cuan rápido el sistema procesa un evento (como por ejemplo la pulsación de una tecla, o el cambio de estado en una entrada, o la llegada de un byte por una interfaz de comunicación serie).

Aclarando términos asociados a scheduling

El término **Tiempo de Respuesta**, es la medida de cuan rápido el sistema procesa un evento (como por ejemplo la pulsación de una tecla, o el cambio de estado en una entrada, o la llegada de un byte por una interfaz de comunicación serie).

Por su parte el término **trhoughput** (o rendimiento en castellano), se refiere al número de procesos que se completan en la unidad de tiempo.

Aclarando términos asociados a scheduling

En un **Round-Robin scheduling**, un proceso ejecuta durante un cierto lapso cuando le llega su turno y de no haber finalizado su ejecución al cabo de ese lapso es suspendido.

Aclarando términos asociados a scheduling

En un **Round-Robin scheduling**, un proceso ejecuta durante un cierto lapso cuando le llega su turno y de no haber finalizado su ejecución al cabo de ese lapso es suspendido.

Por su parte en un **dynamic-priority scheduling** (o en castellano planificado por prioridad dinámica), cada proceso tiene una prioridad que cambia dinámicamente, y el sistema trata de ejecutar siempre al proceso de mayor prioridad.

Aclarando términos asociados a scheduling

Preemption, significa que la CPU puede serle quitada a un proceso en ejecución en cualquier momento (en ocasiones aún si el proceso ejecuta en modo kernel).

Aclarando términos asociados a scheduling

Preemption, significa que la CPU puede serle quitada a un proceso en ejecución en cualquier momento (en ocasiones aún si el proceso ejecuta en modo kernel).

Non-Preemption significa que un proceso ejecutando en una CPU seguirá en poder de la CPU hasta que el decida por sí mismo entregarla, ya sea porque finalizó su ejecución, o porque fue bloqueado o ingresó a un estado de `sleep`.

Aclarando términos asociados a scheduling

Un sistema **Real-Time**, es capaz de responder a cada uno de los eventos que controla (interrupciones por ejemplo) en un tiempo mínimo, o dentro de un límite de tiempo especificado.

Aclarando términos asociados a scheduling

Un sistema **Real-Time**, es capaz de responder a cada uno de los eventos que controla (interrupciones por ejemplo) en un tiempo mínimo, o dentro de un límite de tiempo especificado.

Por su parte, un sistema **Time-Sharing** los diferentes procesos ejecutan dentro de un time frame mínimo garantizado de modo que todos reciban un tiempo de CPU considerado “justo”.

Objetivos del Scheduling de procesos

Objetivos del Scheduling de procesos

- Alta utilización de los recursos del sistema, en especial el tiempo de CPU.

Objetivos del Scheduling de procesos

- Alta utilización de los recursos del sistema, en especial el tiempo de CPU.
- Tiempo de respuesta mínimo a los eventos que debe atender en ***Real-Time***.

Objetivos del Scheduling de procesos

- Alta utilización de los recursos del sistema, en especial el tiempo de CPU.
- Tiempo de respuesta mínimo a los eventos que debe atender en ***Real-Time***.
- Asegurar que aquellos eventos Real-Time, sean procesados dentro del límite de tiempo especificado

Objetivos del Scheduling de procesos

- Alta utilización de los recursos del sistema, en especial el tiempo de CPU.
- Tiempo de respuesta mínimo a los eventos que debe atender en ***Real-Time***.
- Asegurar que aquellos eventos Real-Time, sean procesados dentro del límite de tiempo especificado
- Asignación justa (***fearness***) de la CPU para todos los procesos, para asegurar el máximo ***throughput***.

Objetivos del Scheduling de procesos

- Alta utilización de los recursos del sistema, en especial el tiempo de CPU.
- Tiempo de respuesta mínimo a los eventos que debe atender en ***Real-Time***.
- Asegurar que aquellos eventos Real-Time, sean procesados dentro del límite de tiempo especificado
- Asignación justa (***fearness***) de la CPU para todos los procesos, para asegurar el máximo ***throughput***.

Puede verse a simple vista que satisfacer todos estos objetivos resulta en algunos casos contradictorio. Pretender Tiempo de respuesta mínimo a los eventos Real-Time y máximo rendimiento claramente es un claro ejemplo de ello.

Políticas de Scheduling

Políticas de Scheduling

Definición

La ***política de scheduling*** es un conjunto de reglas, que se establecen de acuerdo con el perfil del Sistema para alcanzar ***algunas*** de las metas deseables.

Políticas de Scheduling

Definición

La ***política de scheduling*** es un conjunto de reglas, que se establecen de acuerdo con el perfil del Sistema para alcanzar ***algunas*** de las metas deseables.

Así, para el caso de un Sistema Operativo de Propósito General, la política de scheduling normalmente apunta a alcanzar un rendimiento global para el sistema que resulte aceptable intentando mediante soluciones de compromiso balancear lo mejor posible los conflictos entre los diferentes objetivos.

Políticas de Scheduling

Definición

La ***política de scheduling*** es un conjunto de reglas, que se establecen de acuerdo con el perfil del Sistema para alcanzar ***algunas*** de las metas deseables.

Así, para el caso de un Sistema Operativo de Propósito General, la política de scheduling normalmente apunta a alcanzar un rendimiento global para el sistema que resulte aceptable intentando mediante soluciones de compromiso balancear lo mejor posible los conflictos entre los diferentes objetivos.

Para el caso de Sistemas Operativos Embedded y Real-Time, se pone todo el foco en proporcionar el tiempo de respuesta mas rápido posible a los eventos y garantizar el tiempo de ejecución de las funciones de respuesta a éstos por debajo del máximo límite tolerable.

Algoritmo de Scheduling

Algoritmo de Scheduling

Definición

El ***algoritmo de scheduling*** es un conjunto de métodos, que implementan las ***políticas de Scheduling*** definidas para un Sistema Operativo.

Algoritmo de Scheduling

Definición

El ***algoritmo de scheduling*** es un conjunto de métodos, que implementan las ***políticas de Scheduling*** definidas para un Sistema Operativo.

El conjunto de estructuras de datos y funciones de código empleadas por el Kernel para implementar las ***Políticas de Scheduling*** se denominan ***Scheduler de Procesos*** de Sistema Operativo.

Algoritmo de Scheduling

Definición

El ***algoritmo de scheduling*** es un conjunto de métodos, que implementan las ***políticas de Scheduling*** definidas para un Sistema Operativo.

El conjunto de estructuras de datos y funciones de código empleadas por el Kernel para implementar las ***Políticas de Scheduling*** se denominan ***Scheduler de Procesos*** de Sistema Operativo.

En general el código del Scheduler de Procesos está distribuido en diferentes módulos del Sistema Operativo. Su parte mas identificable normalmente es el Handler de Interrupción de algún timer. Pero cada vez que se tiene que ir a esperar por un evento y el proceso se pone a dormir y cuando el evento llega se vuelve a activar, son claros ejemplos de esta distribución de la función de Scheduling.

Temario

Procesos en Sistemas Embedded

Procesos en Sistemas Embedded

- **Creación**: Por eventos externos, o de manera periódica

Procesos en Sistemas Embedded

- **Creación**: Por eventos externos, o de manera periódica
- **Prioridad**: Por lo general Estática. Acorde a la importancia de la tarea que tienen asociada.

Procesos en Sistemas Embedded

- **Creación**: Por eventos externos, o de manera periódica
- **Prioridad**: Por lo general Estática. Acorde a la importancia de la tarea que tienen asociada.
- **Objetivos de Scheduling**: Garantizar tiempo de respuesta rápido, y tiempo de ejecución mínimo.

Procesos en Sistemas Embedded

- **Creación**: Por eventos externos, o de manera periódica
- **Prioridad**: Por lo general Estática. Acorde a la importancia de la tarea que tienen asociada.
- **Objetivos de Scheduling**: Garantizar tiempo de respuesta rápido, y tiempo de ejecución mínimo.
- **Política de Scheduling**: Ejecutan en base a prioridad, y round-robin si hay varios procesos con el mismo nivel de prioridad.

Procesos en Sistemas Embedded

- **Creación**: Por eventos externos, o de manera periódica
- **Prioridad**: Por lo general Estática. Acorde a la importancia de la tarea que tienen asociada.
- **Objetivos de Scheduling**: Garantizar tiempo de respuesta rápido, y tiempo de ejecución mínimo.
- **Política de Scheduling**: Ejecutan en base a prioridad, y round-robin si hay varios procesos con el mismo nivel de prioridad.
- **Espacio de Direccionamiento**: Por lo general ejecutan en el mismo espacio de direccionamiento.

Procesos en Sistemas Embedded

- **Creación**: Por eventos externos, o de manera periódica
- **Prioridad**: Por lo general Estática. Acorde a la importancia de la tarea que tienen asociada.
- **Objetivos de Scheduling**: Garantizar tiempo de respuesta rápido, y tiempo de ejecución mínimo.
- **Política de Scheduling**: Ejecutan en base a prioridad, y round-robin si hay varios procesos con el mismo nivel de prioridad.
- **Espacio de Direccionamiento**: Por lo general ejecutan en el mismo espacio de direccionamiento.
- **Preemption**: Preemption en estas condiciones es muy complejo. Manejar concurrencia en espacio de memoria compartido, permite que a un proceso se le quite la CPU en cualquier momento. Si esto ocurre en medio de una modificación de un objeto compartido, y otro proceso lo utiliza, éste se corrompe. Se requiere implementar protección de objetos compartidos.

Temario

Sleep - Wake Up

Sleep - Wake Up

- Origen: UNIX

Sleep - Wake Up

- Origen: UNIX
- Cuando un proceso debe esperar la ocurrencia de un evento, o la disponibilidad de un recurso actualmente no disponible se pone a dormir para suspender su ejecución y liberar la CPU. Esta podrá eventualmente ser asignada a otro proceso.

Sleep - Wake Up

- Origen: UNIX
- Cuando un proceso debe esperar la ocurrencia de un evento, o la disponibilidad de un recurso actualmente no disponible se pone a dormir para suspender su ejecución y liberar la CPU. Esta podrá eventualmente ser asignada a otro proceso.
- Cuando el recurso se pone disponible el proceso en poder de la CPU la cede al proceso que espera el recurso.

Sleep - Wake Up

- Origen: UNIX
- Cuando un proceso debe esperar la ocurrencia de un evento, o la disponibilidad de un recurso actualmente no disponible se pone a dormir para suspender su ejecución y liberar la CPU. Esta podrá eventualmente ser asignada a otro proceso.
- Cuando el recurso se pone disponible el proceso en poder de la CPU la cede al proceso que espera el recurso.
- Si esperaba un evento, la CPU le es devuelta al proceso dormido desde el handler de la interrupción que maneja el evento.

Sleep - Wake Up

- Origen: UNIX
- Cuando un proceso debe esperar la ocurrencia de un evento, o la disponibilidad de un recurso actualmente no disponible se pone a dormir para suspender su ejecución y liberar la CPU. Esta podrá eventualmente ser asignada a otro proceso.
- Cuando el recurso se pone disponible el proceso en poder de la CPU la cede al proceso que espera el recurso.
- Si esperaba un evento, la CPU le es devuelta al proceso dormido desde el handler de la interrupción que maneja el evento.
- Limitaciones

Sleep - Wake Up

- Origen: UNIX
- Cuando un proceso debe esperar la ocurrencia de un evento, o la disponibilidad de un recurso actualmente no disponible se pone a dormir para suspender su ejecución y liberar la CPU. Esta podrá eventualmente ser asignada a otro proceso.
- Cuando el recurso se pone disponible el proceso en poder de la CPU la cede al proceso que espera el recurso.
- Si esperaba un evento, la CPU le es devuelta al proceso dormido desde el handler de la interrupción que maneja el evento.
- Limitaciones
  - ✓ Funcionan bien en sistema UniProcesador, debido a que hay una sola CPU para dormir y despertar al proceso.

Sleep - Wake Up

- Origen: UNIX
- Cuando un proceso debe esperar la ocurrencia de un evento, o la disponibilidad de un recurso actualmente no disponible se pone a dormir para suspender su ejecución y liberar la CPU. Esta podrá eventualmente ser asignada a otro proceso.
- Cuando el recurso se pone disponible el proceso en poder de la CPU la cede al proceso que espera el recurso.
- Si esperaba un evento, la CPU le es devuelta al proceso dormido desde el handler de la interrupción que maneja el evento.
- Limitaciones
  - ✓ Funcionan bien en sistema UniProcesador, debido a que hay una sola CPU para dormir y despertar al proceso.
  - ✓ En sistemas Multiprocesador la secuencia **`Sleep`** primero, **`WakeUp`** después, puede ejecutarse en dos CPUs diferentes con lo cual no puede asegurarse.

Semáforos

Semáforos

- Creados por Edsger Dijkstra en 1962, mejoran las limitaciones del mecanismo **`Sleep WakeUp`**

Semáforos

- Creados por Edsger Dijkstra en 1962, mejoran las limitaciones del mecanismo **`Sleep WakeUp`**
- Un semáforo es una variable abstracta.

Semáforos

- Creados por Edsger Dijkstra en 1962, mejoran las limitaciones del mecanismo **`Sleep WakeUp`**
- Un semáforo es una variable abstracta.
- Se puede implementar una estructura del tipo:

```
1 struct {
2     int spinlock;    // fundamental en sistemas MP
3     int val;         // valor inicial del semáforo
4     int *sem_queue; // Cola de procesos que bloquea el semáforo
5 }sem;
```

Semáforos

- Creados por Edsger Dijkstra en 1962, mejoran las limitaciones del mecanismo **`Sleep WakeUp`**
- Un semáforo es una variable abstracta.
- Se puede implementar una estructura del tipo:

```
1 struct {
2     int spinlock;    // fundamental en sistemas MP
3     int val;         // valor inicial del semáforo
4     int *sem_queue; // Cola de procesos que bloquea el semáforo
5 }sem;
```

- **spinlock**: Asegura que la operación del semáforo es atómica (solo un proceso en el sistema lo puede tomar y liberar en un momento determinado). Ideal en sistemas Multiprocesador.

Semáforos

- Creados por Edsger Dijkstra en 1962, mejoran las limitaciones del mecanismo **`Sleep WakeUp`**
- Un semáforo es una variable abstracta.
- Se puede implementar una estructura del tipo:

```
1 struct {
2     int spinlock;    // fundamental en sistemas MP
3     int val;         // valor inicial del semáforo
4     int *sem_queue; // Cola de procesos que bloquea el semáforo
5 }sem;
```

- **spinlock**: Asegura que la operación del semáforo es atómica (solo un proceso en el sistema lo puede tomar y liberar en un momento determinado). Ideal en sistemas Multiprocesador.
- **val**: Valor del semáforo = Cantidad de unidades disponibles del recurso. Operación standard: **`sem.val++`** y **`sem.val--`**. En semáforos binarios vale **`0`** o **`1`**.

Semáforos

- Creados por Edsger Dijkstra en 1962, mejoran las limitaciones del mecanismo **`Sleep WakeUp`**
- Un semáforo es una variable abstracta.
- Se puede implementar una estructura del tipo:

```
1 struct {
2     int spinlock;    // fundamental en sistemas MP
3     int val;         // valor inicial del semáforo
4     int *sem_queue; // Cola de procesos que bloquea el semáforo
5 }sem;
```

- **spinlock**: Asegura que la operación del semáforo es atómica (solo un proceso en el sistema lo puede tomar y liberar en un momento determinado). Ideal en sistemas Multiprocesador.
- **val**: Valor del semáforo = Cantidad de unidades disponibles del recurso. Operación standard: **`sem.val++`** y **`sem.val--`**. En semáforos binarios vale **`0`** o **`1`**.
- **sem_queue**: FIFO en la que se encolan los procesos bloqueados en el semáforo.

Operaciones con Semáforos

- Se conocen genéricamente como **`V`** y **`P`**. **`V`** incrementa el valor del semáforo y **`P`** lo decrementa.

Operaciones con Semáforos

- Se conocen genéricamente como `V` y `P`. `V` incrementa el valor del semáforo y `P` lo decrementa.
- El semáforo contiene la cantidad de unidades de recursos disponibles. Si hay recursos `P`, devuelve el control al proceso, de otro modo lo pone a dormir, para que no consuma CPU, y lo agrega a la cola de procesos bloqueados en el semáforo.

Operaciones con Semáforos

- Se conocen genéricamente como **`V`** y **`P`**. **`V`** incrementa el valor del semáforo y **`P`** lo decrementa.
- El semáforo contiene la cantidad de unidades de recursos disponibles. Si hay recursos **`P`**, devuelve el control al proceso, de otro modo lo pone a dormir, para que no consuma CPU, y lo agrega a la cola de procesos bloqueados en el semáforo.

```
1  int P (struct sem * s){
2      int CPSR_prev =
           interrupts_off();
3      s->val--;
4      if (s->val < 0)
5          block (s);
6      interrupts_on(CPSR_prev);
7  }
8  void block (struct sem * s) {
9      current->status=SLEEP;
10     add_queue(s->sem_queue,
           current);
11     schedule();
12 }
```

Operaciones con Semáforos

- Se conocen genéricamente como **V** y **P**. **V** incrementa el valor del semáforo y **P** lo decrementa.
- El semáforo contiene la cantidad de unidades de recursos disponibles. Si hay recursos **P**, devuelve el control al proceso, de otro modo lo pone a dormir, para que no consuma CPU, y lo agrega a la cola de procesos bloqueados en el semáforo.

```
1  int P (struct sem * s){
2      int CPSR_prev =
           interrupts_off();
3      s->val--;
4      if (s->val < 0)
5          block (s);
6      interrupts_on(CPSR_prev);
7  }
8  void block (struct sem * s) {
9      current->status=SLEEP;
10     add_queue(s->sem_queue,
           current);
11     schedule();
12 }
```

```
1  int V (struct sem * s){
2      int CPSR_prev = interrupts_off();
3      s->val++;
4      if (s->val <= 0)
5          signal (s);
6      interrupts_on(CPSR_prev);
7  }
8  void signal (struct sem * s) {
9      task_PCB * p = extract_queue(s->
           sem_queue);
10     p->status=RUNNING;
11     add_queue (sched_queue,p);
12 }
```

Operaciones con Semáforos

- P y V son operaciones atómicas.

Operaciones con Semáforos

- P y V son operaciones atómicas.
- Pero como las interrupciones se manejan desde el registro **`CPSR`** las funciones preservan este registro ya que se suspende el proceso.

Operaciones con Semáforos

- P y V son operaciones atómicas.
- Pero como las interrupciones se manejan desde el registro **CPSR** las funciones preservan este registro ya que se suspende el proceso.

```
1  interrupts_off:
2      mrs r0,CPSR
3      mov r4,r0          // Preserva CPSR Original
4      orr r4,#0x80
5      mov CPSR,r4
6      mov pc,lr          // R0 devuelve el CPSR Original
7
8  interrupts_on:
9      mov CPSR,r0        //Restituye el CPSR Original
10     mov pc,lr
```

Semáforos Binarios

Semáforos Binarios

- En 1965 Dijkstra los presentó como una variante de semáforos que solo toman dos estados: `'1'=FREE` y `'0'=OCCUPIED`

Semáforos Binarios

- En 1965 Dijkstra los presentó como una variante de semáforos que solo toman dos estados: **'1'=FREE** y **'0'=OCCUPIED**
- Las funciones V y P son levemente diferentes:

```
1 int P (struct sem * s){
2     int CPSR_prev =
          interrupts_off();
3     if (s->val == 1)
4         s->val = 0;
5     else
6         block (s);
7     interrupts_on(CPSR_prev);
8 }
```

Semáforos Binarios

- En 1965 Dijkstra los presentó como una variante de semáforos que solo toman dos estados: **'1'=FREE** y **'0'=OCCUPIED**
- Las funciones V y P son levemente diferentes:

```
1 int P (struct sem * s){
2     int CPSR_prev =
          interrupts_off();
3     if (s->val == 1)
4         s->val = 0;
5     else
6         block (s);
7     interrupts_on(CPSR_prev);
8 }
```

```
1 int V (struct sem * s){
2     int CPSR_prev =
          interrupts_off();
3     if (s->val == 0)
4         s->val = 1;
5     else
6         signal (s);
7     interrupts_on();
8 }
```

Aplicaciones de semáforos

Aplicaciones de semáforos

- **Protección de regiones criticas de código**. Se debe garantizar que un solo proceso por vez ejecute esa región. Típica aplicación de un semáforo binario.

Aplicaciones de semáforos

- **Protección de regiones criticas de código**. Se debe garantizar que un solo proceso por vez ejecute esa región. Típica aplicación de un semáforo binario.

✓ Valor inicial del semáforo: **`FREE`**

Aplicaciones de semáforos

- **Protección de regiones criticas de código**. Se debe garantizar que un solo proceso por vez ejecute esa región. Típica aplicación de un semáforo binario.

✓ Valor inicial del semáforo: **`FREE`**

✓ En pseudocódigo:

Aplicaciones de semáforos

- **Protección de regiones criticas de código**. Se debe garantizar que un solo proceso por vez ejecute esa región. Típica aplicación de un semáforo binario.

✓ Valor inicial del semáforo: **`FREE`**

✓ En pseudocódigo:

```
proceso:
        struct sem *s;
        sem_init(s,FREE);
        P(s);
        //Aquí va el código crítico
        V(s);
```

- **Mutex lock**. Un ***Mutex*** es un semáforo que contiene un campo adicional, ***owner***, que identifica al proceso que lo toma.

Aplicaciones de semáforos

- **Protección de regiones criticas de código**. Se debe garantizar que un solo proceso por vez ejecute esa región. Típica aplicación de un semáforo binario.

✓ Valor inicial del semáforo: **`FREE`**

✓ En pseudocódigo:

```
proceso:
        struct sem *s;
        sem_init(s,FREE);
        P(s);
        //Aquí va el código crítico
        V(s);
```

- **Mutex lock**. Un ***Mutex*** es un semáforo que contiene un campo adicional, ***owner***, que identifica al proceso que lo toma.

✓ El campo ***owner*** inicialmente vale 0.

Aplicaciones de semáforos

- **Protección de regiones criticas de código**. Se debe garantizar que un solo proceso por vez ejecute esa región. Típica aplicación de un semáforo binario.

✓ Valor inicial del semáforo: **`FREE`**

✓ En pseudocódigo:

```
proceso:
        struct sem *s;
        sem_init(s,FREE);
        P(s);
        //Aquí va el código crítico
        V(s);
```

- **Mutex lock**. Un ***Mutex*** es un semáforo que contiene un campo adicional, ***owner***, que identifica al proceso que lo toma.

✓ El campo ***owner*** inicialmente vale 0.

✓ Función: **`mutex_lock()`**. Si ***owner*** es 0 se completa con su ID, sino se agrega a la cola de procesos bloqueados.

Aplicaciones de semáforos

- **Protección de regiones criticas de código**. Se debe garantizar que un solo proceso por vez ejecute esa región. Típica aplicación de un semáforo binario.

✓ Valor inicial del semáforo: **`FREE`**

✓ En pseudocódigo:

```
proceso:
        struct sem *s;
        sem_init(s,FREE);
        P(s);
        //Aquí va el código crítico
        V(s);
```

- **Mutex lock**. Un ***Mutex*** es un semáforo que contiene un campo adicional, ***owner***, que identifica al proceso que lo toma.

✓ El campo ***owner*** inicialmente vale 0.

✓ Función: **`mutex_lock()`**. Si ***owner*** es 0 se completa con su ID, sino se agrega a la cola de procesos bloqueados.

✓ Solo el ***owner*** puede devolver el ***Mutex***.

Wait for Interrupt y Semáforos

Cuando un proceso ejecuta `P(s)` y encuentra el semáforo en 0, ingresa a una Cola de Espera. Este es el comportamiento de la cola FIFO de procesos de la estructura de un semáforo.
Significa que un semáforo se comporta como un evento externo tal como una interrupción de hardware, a los efectos del estado del proceso.
Lo que no debe perderse de vista es que cuando el evento ocurre, seguramente estará ejecutándose cualquier otro proceso ya que el proceso bloqueado en el semáforo no será visible para el scheduler. Éste será invocado desde la función signal desde el proceso que libera el semáforo para que el proceso bloqueado pueda volver a ejecutar.

Aplicaciones de semáforos

Aplicaciones de semáforos

- **Problema Productor – Consumidor.**

Aplicaciones de semáforos

- **`Problema Productor – Consumidor`**.

✓ Un conjunto de procesos Productores y otro conjunto consumidores de datos comparten un conjunto finito de buffers.

Aplicaciones de semáforos

- **`Problema Productor – Consumidor`**.

✓ Un conjunto de procesos Productores y otro conjunto consumidores de datos comparten un conjunto finito de buffers.

✓ **Buffer Full**: Cuando un productor coloca un dato en un buffer.

Aplicaciones de semáforos

- **`Problema Productor – Consumidor`**.

✓ Un conjunto de procesos Productores y otro conjunto consumidores de datos comparten un conjunto finito de buffers.

✓ **Buffer Full**: Cuando un productor coloca un dato en un buffer.

✓ **Buffer Empty**: Un consumidor lee el dato de un buffer (lectura extractora).

Aplicaciones de semáforos

- **`Problema Productor – Consumidor`**.

✓ Un conjunto de procesos Productores y otro conjunto consumidores de datos comparten un conjunto finito de buffers.

✓ **Buffer Full**: Cuando un productor coloca un dato en un buffer.

✓ **Buffer Empty**: Un consumidor lee el dato de un buffer (lectura extractora).

✓ Un productor al intentar escribir bloquea si no hay **Buffer Empty**

Aplicaciones de semáforos

- **`Problema Productor – Consumidor`**.

✓ Un conjunto de procesos Productores y otro conjunto consumidores de datos comparten un conjunto finito de buffers.

✓ **Buffer Full**: Cuando un productor coloca un dato en un buffer.

✓ **Buffer Empty**: Un consumidor lee el dato de un buffer (lectura extractora).

✓ Un productor al intentar escribir bloquea si no hay **Buffer Empty**

✓ Un consumidor bloquea si no hay ningún **Buffer Full**.

Aplicaciones de semáforos

- **`Problema Productor – Consumidor`**.

✓ Un conjunto de procesos Productores y otro conjunto consumidores de datos comparten un conjunto finito de buffers.

✓ **Buffer Full**: Cuando un productor coloca un dato en un buffer.

✓ **Buffer Empty**: Un consumidor lee el dato de un buffer (lectura extractora).

✓ Un productor al intentar escribir bloquea si no hay **Buffer Empty**

✓ Un consumidor bloquea si no hay ningún **Buffer Full**.

✓ En resumidas cuentas, ambos esperan eventos.

Aplicaciones de semáforos

Aplicaciones de semáforos

- **Problema Escritor - Lector.**

Aplicaciones de semáforos

- **`Problema Escritor - Lector`**.

✓ Un conjunto de procesos Productores y otro conjunto consumidores de datos comparten un objeto de datos común. Ej: un archivo, una variable, un segmento de memoria.

Aplicaciones de semáforos

- **`Problema Escritor - Lector`**.

✓ Un conjunto de procesos Productores y otro conjunto consumidores de datos comparten un objeto de datos común. Ej: un archivo, una variable, un segmento de memoria.

✓ Un escritor activo debe excluir a los demás escritores de actuar.

Aplicaciones de semáforos

- **`Problema Escritor - Lector`**.

✓ Un conjunto de procesos Productores y otro conjunto consumidores de datos comparten un objeto de datos común. Ej: un archivo, una variable, un segmento de memoria.

✓ Un escritor activo debe excluir a los demás escritores de actuar.

✓ Los lectores pueden operar en forma concurrente, siempre que no haya un escritor activo.

Aplicaciones de semáforos

- **`Problema Escritor - Lector`.**

✓ Un conjunto de procesos Productores y otro conjunto consumidores de datos comparten un objeto de datos común. Ej: un archivo, una variable, un segmento de memoria.

✓ Un escritor activo debe excluir a los demás escritores de actuar.

✓ Los lectores pueden operar en forma concurrente, siempre que no haya un escritor activo.

✓ Además, ningún proceso (escritor o lector) puede esperar indefinidamente (**Starve Condition**, algo así como “morir de hambre”)

Aplicaciones de semáforos

- **`Problema Escritor - Lector`**.

✓ Un conjunto de procesos Productores y otro conjunto consumidores de datos comparten un objeto de datos común. Ej: un archivo, una variable, un segmento de memoria.

✓ Un escritor activo debe excluir a los demás escritores de actuar.

✓ Los lectores pueden operar en forma concurrente, siempre que no haya un escritor activo.

✓ Además, ningún proceso (escritor o lector) puede esperar indefinidamente (**Starve Condition**, algo así como “morir de hambre”)

✓ Una forma de evitar **Starve Condition** es manejar colas de tipo FIFO para escritores y lectores.

Ventajas de los semáforos

Ventajas de los semáforos

- Los Semáforos combinan un contador, los testean y deciden en base al resultado.

Ventajas de los semáforos

- Los Semáforos combinan un contador, los testean y deciden en base al resultado.
- Todo dentro de una operación indivisible.

Ventajas de los semáforos

- Los Semáforos combinan un contador, los testean y deciden en base al resultado.
- Todo dentro de una operación indivisible.
- La operación V desbloquea solo al proceso que espera en el primer lugar de la cola del semáforo (si la cola no tiene procesos no pasa nada).

Ventajas de los semáforos

- Los Semáforos combinan un contador, los testean y deciden en base al resultado.
- Todo dentro de una operación indivisible.
- La operación V desbloquea solo al proceso que espera en el primer lugar de la cola del semáforo (si la cola no tiene procesos no pasa nada).
- Luego de la operación P en un semáforo, se garantiza que el proceso tenga el recurso.

Ventajas de los semáforos

- Los Semáforos combinan un contador, los testean y deciden en base al resultado.
- Todo dentro de una operación indivisible.
- La operación V desbloquea solo al proceso que espera en el primer lugar de la cola del semáforo (si la cola no tiene procesos no pasa nada).
- Luego de la operación P en un semáforo, se garantiza que el proceso tenga el recurso.
- No debe intentar recuperar el recurso como en el caso de **`Sleep/Wa`**

Ventajas de los semáforos

- Los Semáforos combinan un contador, los testean y deciden en base al resultado.
- Todo dentro de una operación indivisible.
- La operación V desbloquea solo al proceso que espera en el primer lugar de la cola del semáforo (si la cola no tiene procesos no pasa nada).
- Luego de la operación P en un semáforo, se garantiza que el proceso tenga el recurso.
- No debe intentar recuperar el recurso como en el caso de **`Sleep/Wa`**
- El valor del semáforo registra el número de veces que ocurrió el evento

Ventajas de los semáforos

- Los Semáforos combinan un contador, los testean y deciden en base al resultado.
- Todo dentro de una operación indivisible.
- La operación V desbloquea solo al proceso que espera en el primer lugar de la cola del semáforo (si la cola no tiene procesos no pasa nada).
- Luego de la operación P en un semáforo, se garantiza que el proceso tenga el recurso.
- No debe intentar recuperar el recurso como en el caso de **`Sleep/Wa`**
- El valor del semáforo registra el número de veces que ocurrió el evento
- A diferencia de **`Sleep/Wakeup`**, que sigue el orden sleep-first-wakeup-later, los procesos pueden ejecutar operaciones P/V en semáforos en cualquier orden.

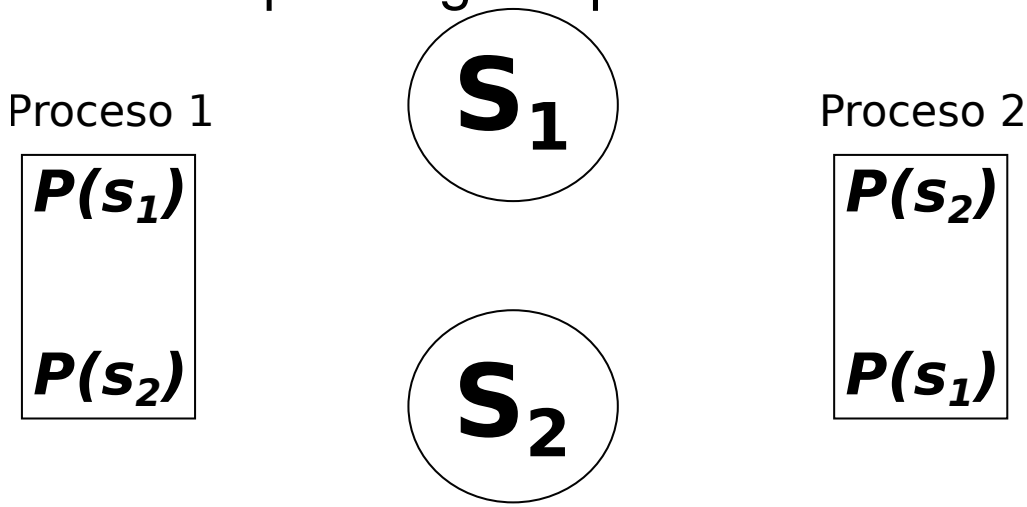
Límites y precauciones

Límites y precauciones

- **`Dead Lock`**:Es una situación en la cual un conjunto de procesos esperan mutuamente cualquiera de los otros permanentemente, resultando que ninguno pueda continuar.

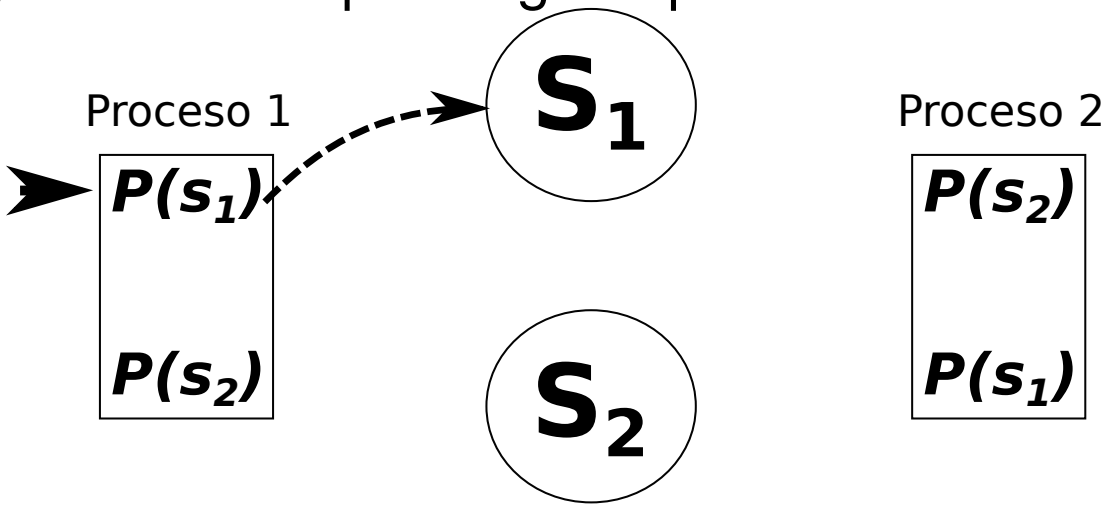
Límites y precauciones

- **Dead Lock**:Es una situación en la cual un conjunto de procesos esperan mutuamente cualquiera de los otros permanentemente, resultando que ninguno pueda continuar.



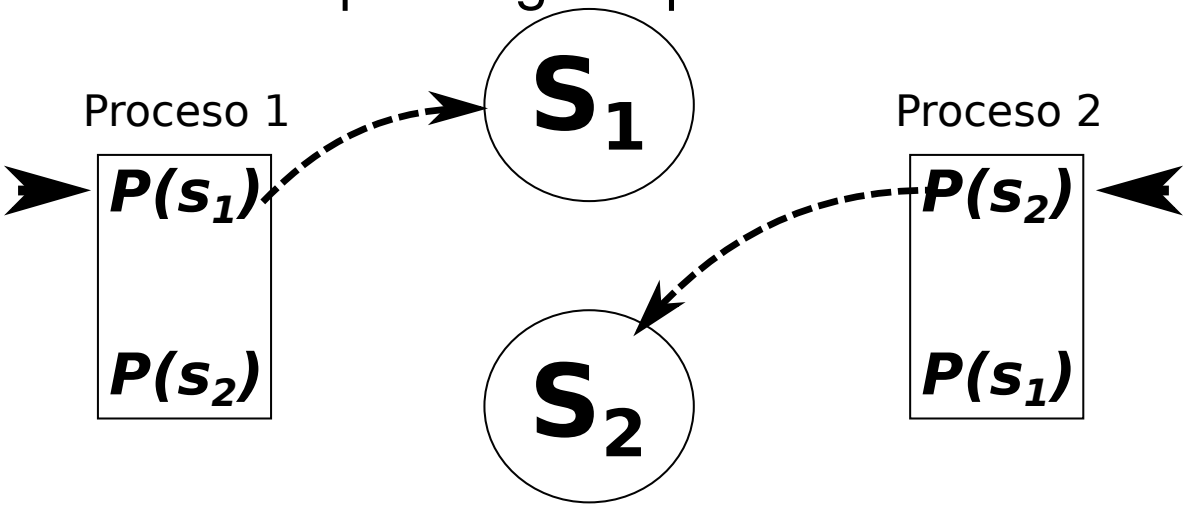
Límites y precauciones

- **Dead Lock**:Es una situación en la cual un conjunto de procesos esperan mutuamente cualquiera de los otros permanentemente, resultando que ninguno pueda continuar.



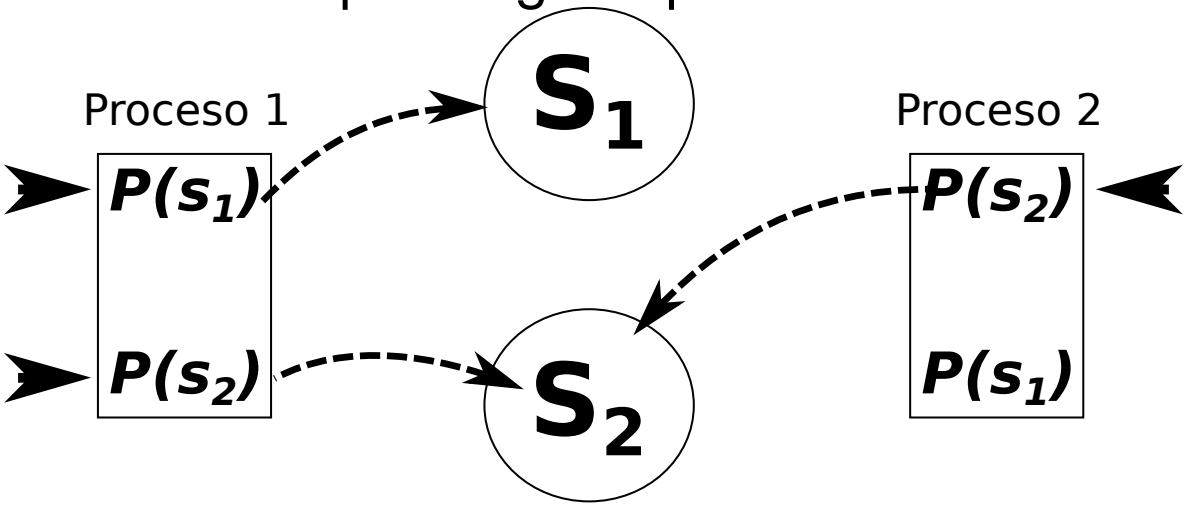
Límites y precauciones

- **`Dead Lock`**:Es una situación en la cual un conjunto de procesos esperan mutuamente cualquiera de los otros permanentemente, resultando que ninguno pueda continuar.



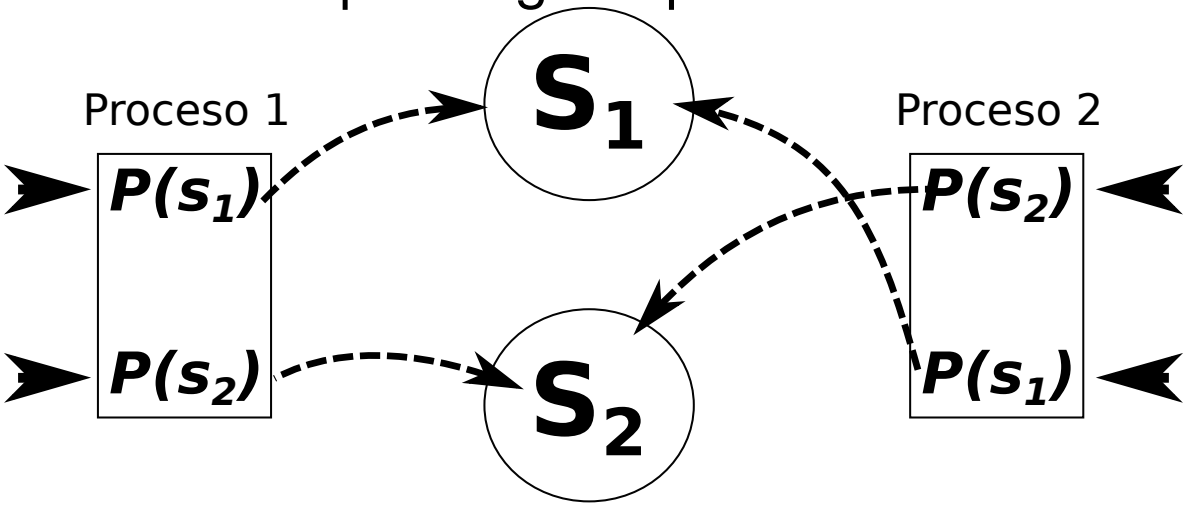
Límites y precauciones

- **`Dead Lock`**:Es una situación en la cual un conjunto de procesos esperan mutuamente cualquiera de los otros permanentemente, resultando que ninguno pueda continuar.



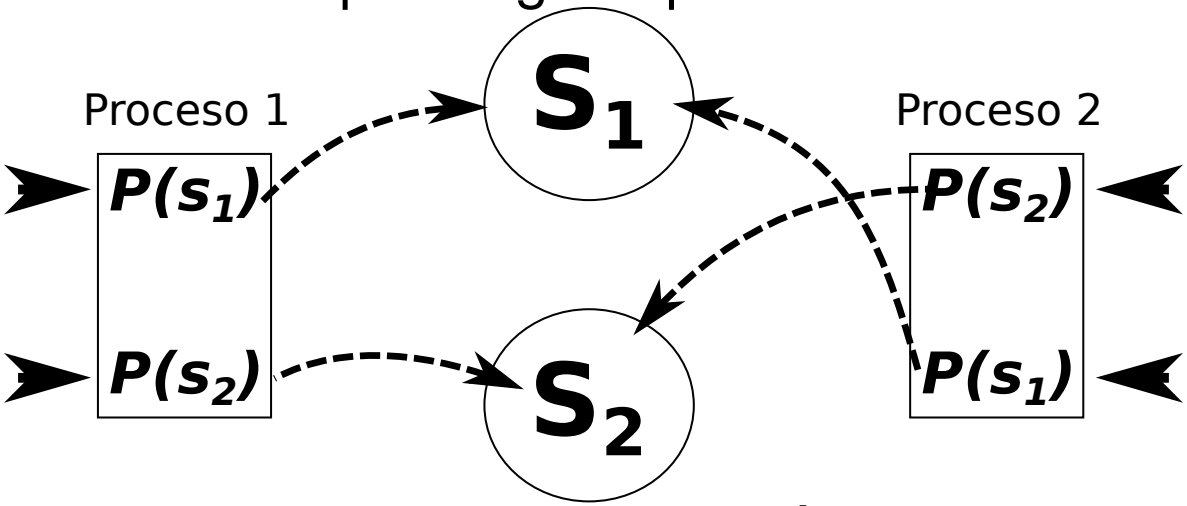
Límites y precauciones

- **`Dead Lock`**:Es una situación en la cual un conjunto de procesos esperan mutuamente cualquiera de los otros permanentemente, resultando que ninguno pueda continuar.



Límites y precauciones

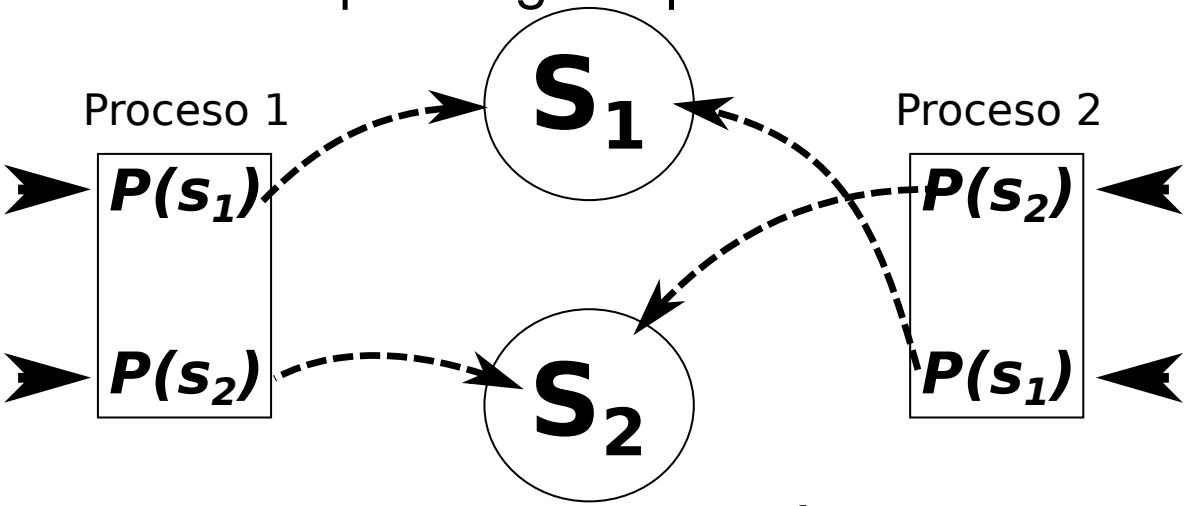
- **`Dead Lock`**:Es una situación en la cual un conjunto de procesos esperan mutuamente cualquiera de los otros permanentemente, resultando que ninguno pueda continuar.



- Es el principal riesgo de utilizar semáforos incorrectamente.

Límites y precauciones

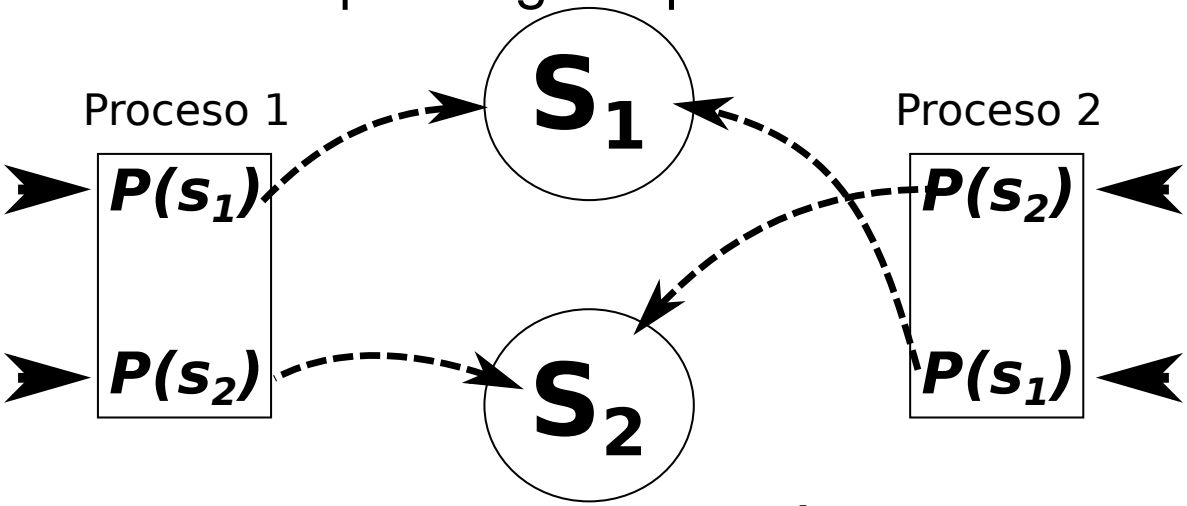
- **Dead Lock**:Es una situación en la cual un conjunto de procesos esperan mutuamente cualquiera de los otros permanentemente, resultando que ninguno pueda continuar.



- Es el principal riesgo de utilizar semáforos incorrectamente.
- Existen diferentes estrategias para evitar **Dead Locks**

Límites y precauciones

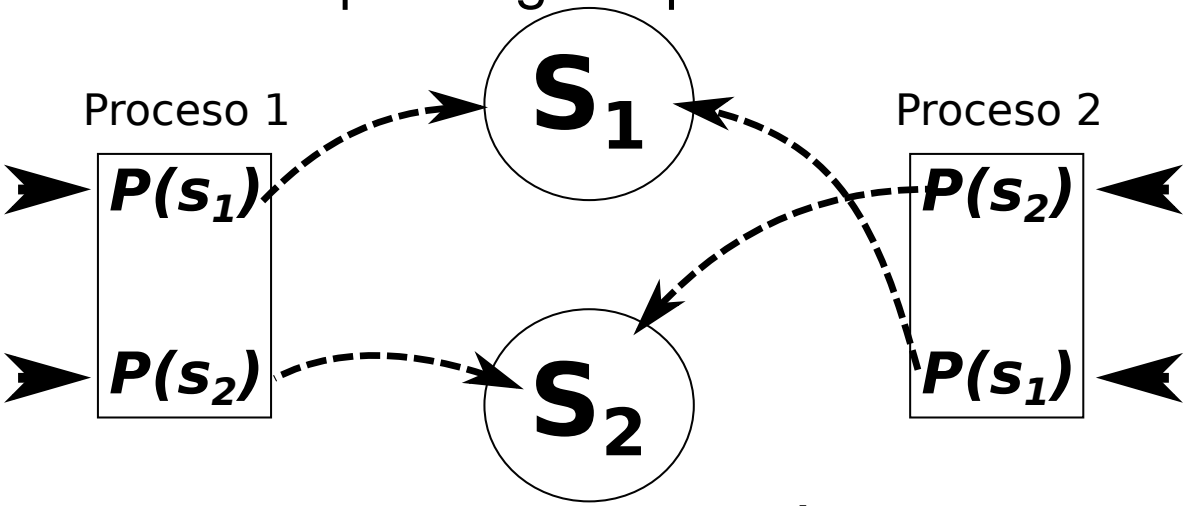
- **`Dead Lock`**:Es una situación en la cual un conjunto de procesos esperan mutuamente cualquiera de los otros permanentemente, resultando que ninguno pueda continuar.



- Es el principal riesgo de utilizar semáforos incorrectamente.
- Existen diferentes estrategias para evitar **`Dead Locks`**
  - ***Dead Lock Detection and Recovery***

Límites y precauciones

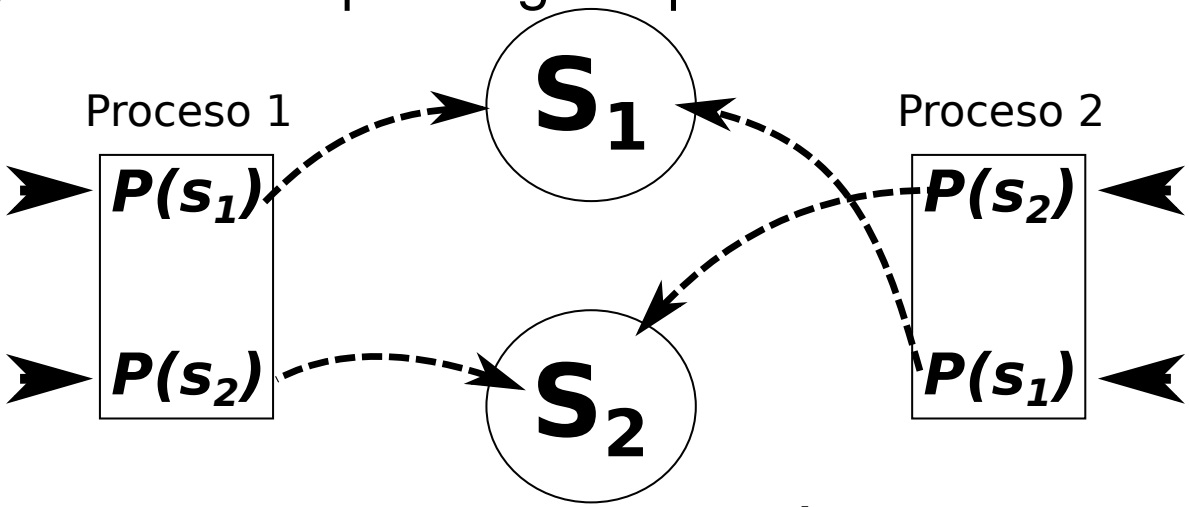
- **`Dead Lock`**:Es una situación en la cual un conjunto de procesos esperan mutuamente cualquiera de los otros permanentemente, resultando que ninguno pueda continuar.



- Es el principal riesgo de utilizar semáforos incorrectamente.
- Existen diferentes estrategias para evitar **`Dead Locks`**
  - ***Dead Lock Detection and Recovery***
  - ***Dead Lock Avoidance***

Límites y precauciones

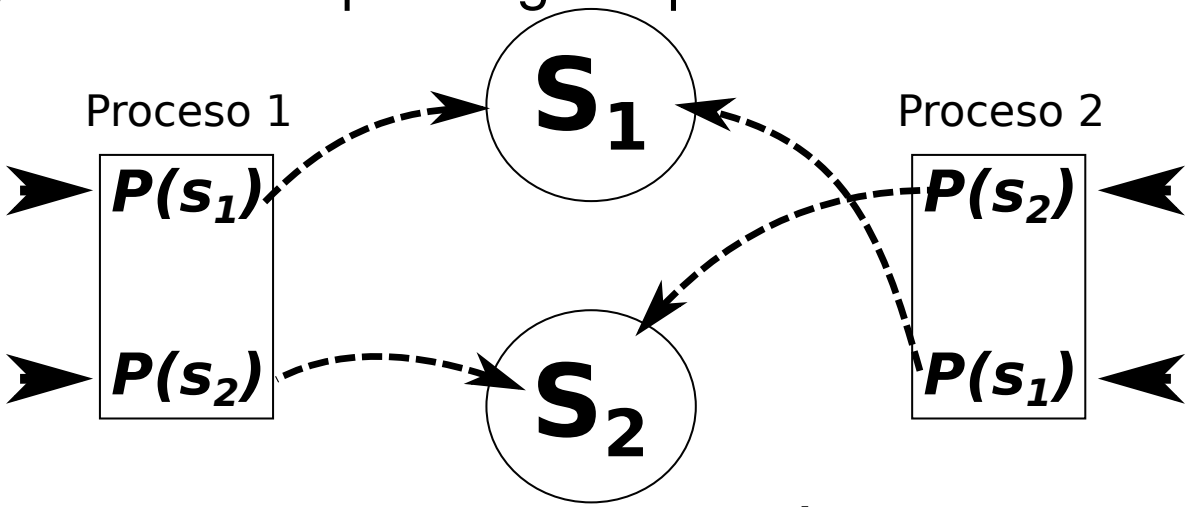
- **`Dead Lock`**:Es una situación en la cual un conjunto de procesos esperan mutuamente cualquiera de los otros permanentemente, resultando que ninguno pueda continuar.



- Es el principal riesgo de utilizar semáforos incorrectamente.
- Existen diferentes estrategias para evitar **`Dead Locks`**
  - ***Dead Lock Detection and Recovery***
  - ***Dead Lock Avoidance***
  - ***Dead Lock Prevention***

Límites y precauciones

- **`Dead Lock`**:Es una situación en la cual un conjunto de procesos esperan mutuamente cualquiera de los otros permanentemente, resultando que ninguno pueda continuar.



- Es el principal riesgo de utilizar semáforos incorrectamente.
- Existen diferentes estrategias para evitar **`Dead Locks`**
  - ***Dead Lock Detection and Recovery***
  - ***Dead Lock Avoidance***
  - ***Dead Lock Prevention***
- Esta última es la que se emplea generalmente en los Sistemas Operativos

Condiciones para un dead lock

Condiciones para un dead lock

- **`Exclusión mutua`**: Cada proceso reclama acceso exclusivo al recurso.

Condiciones para un dead lock

- **`Exclusión mutua`**: Cada proceso reclama acceso exclusivo al recurso.
- **`No Preemption`**: Una vez que un proceso consigue el recurso no tiene existe forma de liberarlo hasta completar su uso.

Condiciones para un dead lock

- **`Exclusión mutua`**: Cada proceso reclama acceso exclusivo al recurso.
- **`No Preemption`**: Una vez que un proceso consigue el recurso no tiene existe forma de liberarlo hasta completar su uso.
- **`Alojamiento Parcial`**: Cada proceso mantiene la exclusividad de los recursos que ya tiene, mientras solicita acceso a mas recursos.

Condiciones para un dead lock

- **`Exclusión mutua`**: Cada proceso reclama acceso exclusivo al recurso.
- **`No Preemption`**: Una vez que un proceso consigue el recurso no tiene existe forma de liberarlo hasta completar su uso.
- **`Alojamiento Parcial`**: Cada proceso mantiene la exclusividad de los recursos que ya tiene, mientras solicita acceso a mas recursos.
- **`Espera circular`**: Se conoce como *Circular Wait Condition*, y consiste en que cada proceso mantiene en su poder uno a mas recursos que son requeridos por uno o mas procesos siguientes en la cadena.

Prevención de dead locks

Prevención de dead locks

- Para diseñar un sistema libre de **dead locks**, hay que asegurar que en cada momento no se cumpla como mínimo una de las cuatro condiciones anteriores.

Prevención de dead locks

- Para diseñar un sistema libre de **dead locks**, hay que asegurar que en cada momento no se cumpla como mínimo una de las cuatro condiciones anteriores.
- Esto implica restricciones en la forma en que se puede requerir acceso a los recursos.

Prevención de dead locks

- Para diseñar un sistema libre de **dead locks**, hay que asegurar que en cada momento no se cumpla como mínimo una de las cuatro condiciones anteriores.
- Esto implica restricciones en la forma en que se puede requerir acceso a los recursos.
- La condición de Exclusión Mutua no puede negarse. Ejemplo: Si el recurso es un archivo, necesariamente un proceso puede ganar acceso para modificarlo cuando el resto no lo esté accediendo.

Prevención de dead locks

- Para diseñar un sistema libre de **dead locks**, hay que asegurar que en cada momento no se cumpla como mínimo una de las cuatro condiciones anteriores.
- Esto implica restricciones en la forma en que se puede requerir acceso a los recursos.
- La condición de Exclusión Mutua no puede negarse. Ejemplo: Si el recurso es un archivo, necesariamente un proceso puede ganar acceso para modificarlo cuando el resto no lo esté accediendo.
- A continuación, las condiciones para evitar **dead locks**.

Condiciones para evitar dead locks

Condiciones para evitar dead locks

- ***Condición de Espera denegada***: Cada proceso debe solicitar todos sus recursos de una sola vez y no puede continuar hasta que no le hayan sido asignado todos.

Condiciones para evitar dead locks

- ***Condición de Espera denegada***: Cada proceso debe solicitar todos sus recursos de una sola vez y no puede continuar hasta que no le hayan sido asignado todos.
- ***Condición de No Preferencia denegada***: Si un proceso tiene ya tomados algunos recursos y se le deniega el acceso a un recurso adicional, deberá liberar todos los recursos que tiene tomados y requerirlos nuevamente cuando le sea asignado el recurso que se le denegó.

Condiciones para evitar dead locks

- ***Condición de Espera denegada***: Cada proceso debe solicitar todos sus recursos de una sola vez y no puede continuar hasta que no le hayan sido asignado todos.
- ***Condición de No Preferencia denegada***: Si un proceso tiene ya tomados algunos recursos y se le deniega el acceso a un recurso adicional, deberá liberar todos los recursos que tiene tomados y requerirlos nuevamente cuando le sea asignado el recurso que se le denegó.
- ***Ordenamiento lineal de recursos en todas las tareas***: Si a una tarea se le han asignado recursos del tipo r_i, a posteriori solo puede solicitar recursos de tipo posterior a r_i en orden. Si hay un solo tipo de recursos el orden es por recurso.

Condiciones para evitar dead locks

La primer alternativa es muy costosa: Un recurso puede estar tomado por un tiempo muy prolongado. La segunda opción solo es factible en el caso de recursos anticipables (preemptables), cuyo estado puede ser fácilmente salvado y recuperado mas tarde (tal el caso de un procesador). El mas factible es el tercero excepto cuando hay varias tareas iniciadas por uno o mas procesos del tipo job-step.

Sincronización de alto nivel

Sincronización de alto nivel

- Existen mecanismos mas sofisticados muchos de los cuales ya son conocidos.

Sincronización de alto nivel

- Existen mecanismos mas sofisticados muchos de los cuales ya son conocidos.
- En general tienen como características distintivas respecto de los semáforos, que en general no inducen a dead locks.

Sincronización de alto nivel

- Existen mecanismos mas sofisticados muchos de los cuales ya son conocidos.
- En general tienen como características distintivas respecto de los semáforos, que en general no inducen a dead locks.
- Lo enumeramos a continuación

Sincronización de alto nivel

- Existen mecanismos mas sofisticados muchos de los cuales ya son conocidos.
- En general tienen como características distintivas respecto de los semáforos, que en general no inducen a dead locks.
- Lo enumeramos a continuación

✓ Variables de condición

Sincronización de alto nivel

- Existen mecanismos mas sofisticados muchos de los cuales ya son conocidos.
- En general tienen como características distintivas respecto de los semáforos, que en general no inducen a dead locks.
- Lo enumeramos a continuación

✓ Variables de condición

✓ Shared Memory

Sincronización de alto nivel

- Existen mecanismos mas sofisticados muchos de los cuales ya son conocidos.
- En general tienen como características distintivas respecto de los semáforos, que en general no inducen a dead locks.
- Lo enumeramos a continuación

✓ Variables de condición

✓ Shared Memory

✓ Colas de mensajes (Sincrónicas y Asincrónicas)

Sincronización de alto nivel

- Existen mecanismos mas sofisticados muchos de los cuales ya son conocidos.
- En general tienen como características distintivas respecto de los semáforos, que en general no inducen a dead locks.
- Lo enumeramos a continuación

✓ Variables de condición

✓ Shared Memory

✓ Colas de mensajes (Sincrónicas y Asincrónicas)

✓ Pipes

Sincronización de alto nivel

- Existen mecanismos mas sofisticados muchos de los cuales ya son conocidos.
- En general tienen como características distintivas respecto de los semáforos, que en general no inducen a dead locks.
- Lo enumeramos a continuación

✓ Variables de condición

✓ Shared Memory

✓ Colas de mensajes (Sincrónicas y Asincrónicas)

✓ Pipes

✓ Señales

Temario

Memoria: El otro recurso crítico de un SO

Memoria: El otro recurso crítico de un SO

- En el arranque un procesador ARM ejecuta el Handler de Reset en Modo Supervisor (SVC)

Memoria: El otro recurso crítico de un SO

- En el arranque un procesador ARM ejecuta el Handler de Reset en Modo Supervisor (SVC)
- Copia la tabla de vectores de interrupción en la dirección 0, habilita stacks de diferente nivel de privilegio para las tareas y el kernel, y habilita las interrupciones.

Memoria: El otro recurso crítico de un SO

- En el arranque un procesador ARM ejecuta el Handler de Reset en Modo Supervisor (SVC)
- Copia la tabla de vectores de interrupción en la dirección 0, habilita stacks de diferente nivel de privilegio para las tareas y el kernel, y habilita las interrupciones.
- En el modelo estático las tareas ejecutan en el mismo espacio de memoria del kernel.

Memoria: El otro recurso crítico de un SO

- En el arranque un procesador ARM ejecuta el Handler de Reset en Modo Supervisor (SVC)
- Copia la tabla de vectores de interrupción en la dirección 0, habilita stacks de diferente nivel de privilegio para las tareas y el kernel, y habilita las interrupciones.
- En el modelo estático las tareas ejecutan en el mismo espacio de memoria del kernel.
- Esto genera lacks de eficiencia en la protección de memoria, ya que al compartir el espacio de direcciones en principio las tareas pueden escribir en zonas de memoria que interfieran con las de otras tareas.

Memoria: El otro recurso crítico de un SO

- En el arranque un procesador ARM ejecuta el Handler de Reset en Modo Supervisor (SVC)
- Copia la tabla de vectores de interrupción en la dirección 0, habilita stacks de diferente nivel de privilegio para las tareas y el kernel, y habilita las interrupciones.
- En el modelo estático las tareas ejecutan en el mismo espacio de memoria del kernel.
- Esto genera lacks de eficiencia en la protección de memoria, ya que al compartir el espacio de direcciones en principio las tareas pueden escribir en zonas de memoria que interfieran con las de otras tareas.
- Si además todo el conjunto kernel tareas se mantiene en Modo Supervisor las tareas pueden interferir las áreas de memoria utilizadas por el kernel

Protección de Memoria

Protección de Memoria

- Si se avanza en pasar las tareas en Modo User la Memoria del kernel tiene protección, por hardware. Las excepciones ***Prefetch Abort*** o ***Data Abort*** son las que señalarán los accesos indebidos a memoria según el caso.

Protección de Memoria

- Si se avanza en pasar las tareas en Modo User la Memoria del kernel tiene protección, por hardware. Las excepciones ***Prefetch Abort*** o ***Data Abort*** son las que señalarán los accesos indebidos a memoria según el caso.
- Para lograrlo al crear la tarea simplemente se inicializa el campo de bits correspondiente al Modo en el registro `CPSR` con el patrón correspondiente a User Mode: `10000`.

Protección de Memoria

- Si se avanza en pasar las tareas en Modo User la Memoria del kernel tiene protección, por hardware. Las excepciones ***Prefetch Abort*** o ***Data Abort*** son las que señalarán los accesos indebidos a memoria según el caso.
- Para lograrlo al crear la tarea simplemente se inicializa el campo de bits correspondiente al Modo en el registro `CPSR` con el patrón correspondiente a User Mode: `10000`.
- Lo que no es trivial en Modo User es pasar a Modo Supervisor. Para esto solo hay tres mecanismos:

Protección de Memoria

- Si se avanza en pasar las tareas en Modo User la Memoria del kernel tiene protección, por hardware. Las excepciones ***Prefetch Abort*** o ***Data Abort*** son las que señalarán los accesos indebidos a memoria según el caso.
- Para lograrlo al crear la tarea simplemente se inicializa el campo de bits correspondiente al Modo en el registro **`CPSR`** con el patrón correspondiente a User Mode: **`10000`**.
- Lo que no es trivial en Modo User es pasar a Modo Supervisor. Para esto solo hay tres mecanismos:

  Excepciones Cuando el programa de Modo User efectúa por ejemplo un acceso a un área privilegiada de memoria, o ejecuta una instrucción indebida o de un coprocesador no habilitado.

Protección de Memoria

- Si se avanza en pasar las tareas en Modo User la Memoria del kernel tiene protección, por hardware. Las excepciones ***Prefetch Abort*** o ***Data Abort*** son las que señalarán los accesos indebidos a memoria según el caso.
- Para lograrlo al crear la tarea simplemente se inicializa el campo de bits correspondiente al Modo en el registro **`CPSR`** con el patrón correspondiente a User Mode: **`10000`**.
- Lo que no es trivial en Modo User es pasar a Modo Supervisor. Para esto solo hay tres mecanismos:

  Excepciones Cuando el programa de Modo User efectúa por ejemplo un acceso a un área privilegiada de memoria, o ejecuta una instrucción indebida o de un coprocesador no habilitado.

  Interrupciones Respuesta a requerimientos desde la E/S. El Procesador pasa a modo **`FIQ`** o **`IRQ`**.

Protección de Memoria

- Si se avanza en pasar las tareas en Modo User la Memoria del kernel tiene protección, por hardware. Las excepciones ***Prefetch Abort*** o ***Data Abort*** son las que señalarán los accesos indebidos a memoria según el caso.
- Para lograrlo al crear la tarea simplemente se inicializa el campo de bits correspondiente al Modo en el registro **`CPSR`** con el patrón correspondiente a User Mode: **`10000`**.
- Lo que no es trivial en Modo User es pasar a Modo Supervisor. Para esto solo hay tres mecanismos:

  Excepciones Cuando el programa de Modo User efectúa por ejemplo un acceso a un área privilegiada de memoria, o ejecuta una instrucción indebida o de un coprocesador no habilitado.

  Interrupciones Respuesta a requerimientos desde la E/S. El Procesador pasa a modo **`FIQ`** o **`IRQ`**.

  Por Software Ejecuta la instrucción **`SVC`** o la legacy **`SWI`**.

Protección de Memoria

Protección de Memoria

- El siguiente nivel en protección de memoria, consiste en asignar a cada tarea su propio y exclusivo espacio de direcciones de memoria.

Protección de Memoria

- El siguiente nivel en protección de memoria, consiste en asignar a cada tarea su propio y exclusivo espacio de direcciones de memoria.
- Esto garantiza que cada tarea no puede salir de ese espacio y por lo tanto no puede interferir ya no solo con el espacio de memoria del kernel, sino tampoco con el de las demás tareas.

Protección de Memoria

- El siguiente nivel en protección de memoria, consiste en asignar a cada tarea su propio y exclusivo espacio de direcciones de memoria.
- Esto garantiza que cada tarea no puede salir de ese espacio y por lo tanto no puede interferir ya no solo con el espacio de memoria del kernel, sino tampoco con el de las demás tareas.
- Para dar este paso necesitamos del Coprocesador `cp15`.

Protección de Memoria

- El siguiente nivel en protección de memoria, consiste en asignar a cada tarea su propio y exclusivo espacio de direcciones de memoria.
- Esto garantiza que cada tarea no puede salir de ese espacio y por lo tanto no puede interferir ya no solo con el espacio de memoria del kernel, sino tampoco con el de las demás tareas.
- Para dar este paso necesitamos del Coprocesador **`cp15`**.
- En este Coprocesador se habilitan dos modos posibles de gestión de memoria: **`VMSA`** (CORTEX-A) y **`PMSA`** (Cortex-R).

Protección de Memoria

- El siguiente nivel en protección de memoria, consiste en asignar a cada tarea su propio y exclusivo espacio de direcciones de memoria.
- Esto garantiza que cada tarea no puede salir de ese espacio y por lo tanto no puede interferir ya no solo con el espacio de memoria del kernel, sino tampoco con el de las demás tareas.
- Para dar este paso necesitamos del Coprocesador **`cp15`**.
- En este Coprocesador se habilitan dos modos posibles de gestión de memoria: **`VMSA`** (CORTEX-A) y **`PMSA`** (Cortex-R).
- **`VMSA`**:Permite gestión dinámica de la memoria, por mapeo del espacio virtual (el que maneja el software) al espacio físico (la dirección de memoria en el hardware externo). Todo un mundo

Protección de Memoria

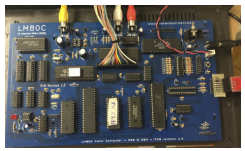
- El siguiente nivel en protección de memoria, consiste en asignar a cada tarea su propio y exclusivo espacio de direcciones de memoria.
- Esto garantiza que cada tarea no puede salir de ese espacio y por lo tanto no puede interferir ya no solo con el espacio de memoria del kernel, sino tampoco con el de las demás tareas.
- Para dar este paso necesitamos del Coprocesador **`cp15`**.
- En este Coprocesador se habilitan dos modos posibles de gestión de memoria: **`VMSA`** (CORTEX-A) y **`PMSA`** (Cortex-R).
- **`VMSA`**:Permite gestión dinámica de la memoria, por mapeo del espacio virtual (el que maneja el software) al espacio físico (la dirección de memoria en el hardware externo). Todo un mundo
- **`PMSA`**: Es algo mas limitado por lo que es apto para gestión estática de memoria. Es decir sistemas en donde el espacio de cada tarea se crea desde el inicio y no se modifica.

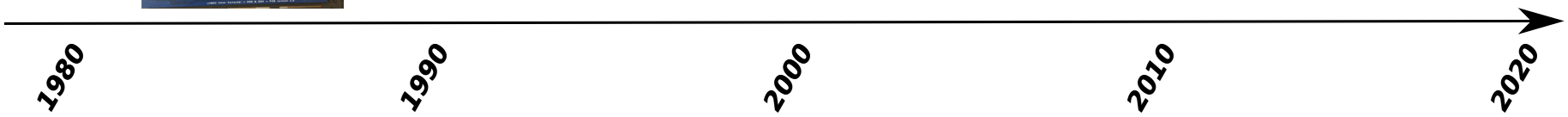
Temario

Definición

Un **GPOS** (por **G**eneral **P**urpose **O**perating **S**ystem, es un sistema operativo que implementa gestión de procesos, gestión de memoria, gestión de la Entrada/Salida, soporte a File System(s) como parte de la gestión de sistemas de storage, y posee una interfaz de usuario que le permite interactuar a demanda con el computador. Además cada proceso ejecuta en un espacio propio protegido del resto, por medio del hardware de manejo de memoria.
Los procesos ejecutan y finalizan en forma dinámica la cantidad de veces que el usuario defina o cada vez que los ejecute a demanda. Solicitan recursos y cuando finalizan liberan todos los recursos que pueden ser reasignados a otro proceso.

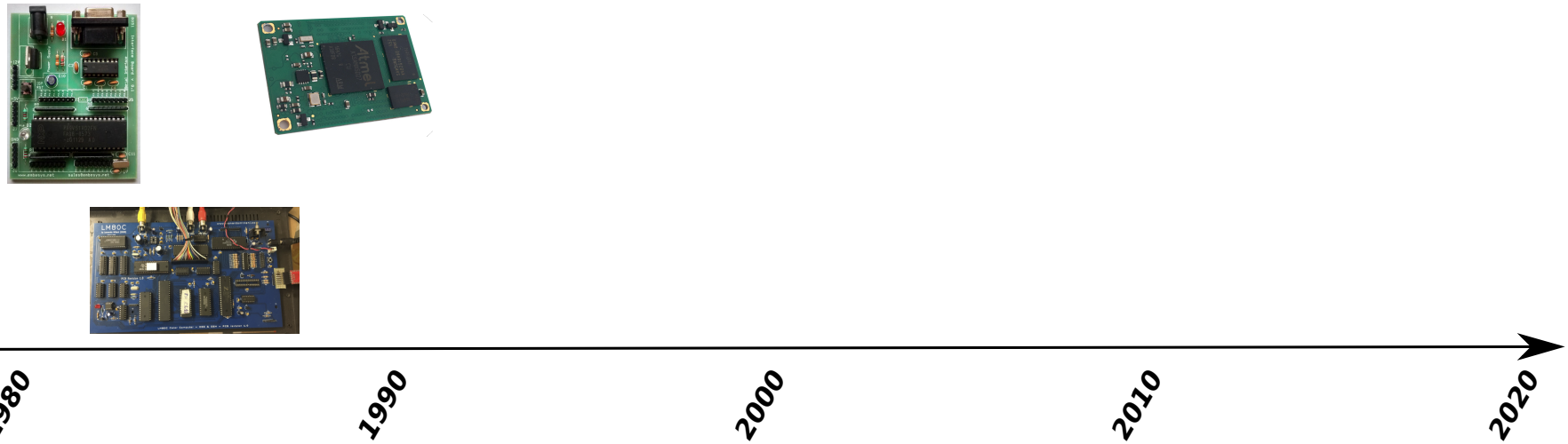
Sistemas Operativos Embedded de Propósito General





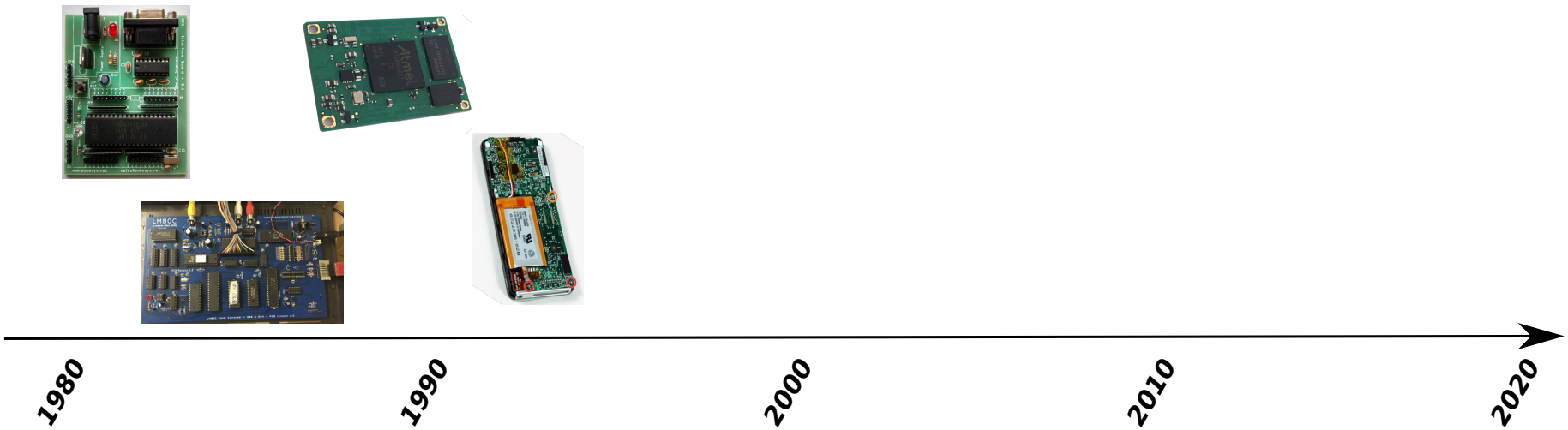
Al inicio los Sistemas Embedded eran parte de un sistema mucho mayor, y se basaban en procesadores muy pequeños de propósito dedicado y con una cantidad de hardware mas que moderada (cantidad de memoria, dispositivos de E/S muy sencillos. Rango de frecuencia alrededor de 10 MHz.

Sistemas Operativos Embedded de Propósito General



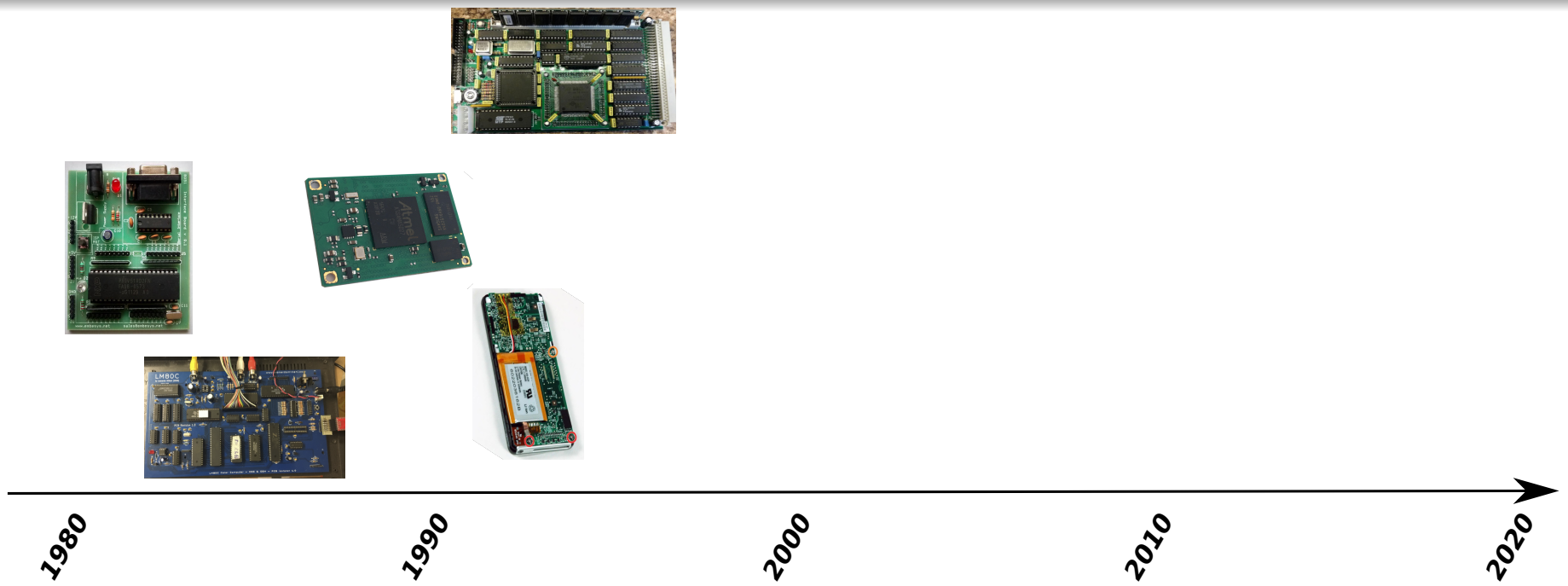
A medida que nos acercamos a los 90 comenzaron a ampliarse, tanto en capacidades de las CPUs como en el resto de los recursos, y se empiezan a explorar mayores frecuencias de clock.

Sistemas Operativos Embedded de Propósito General



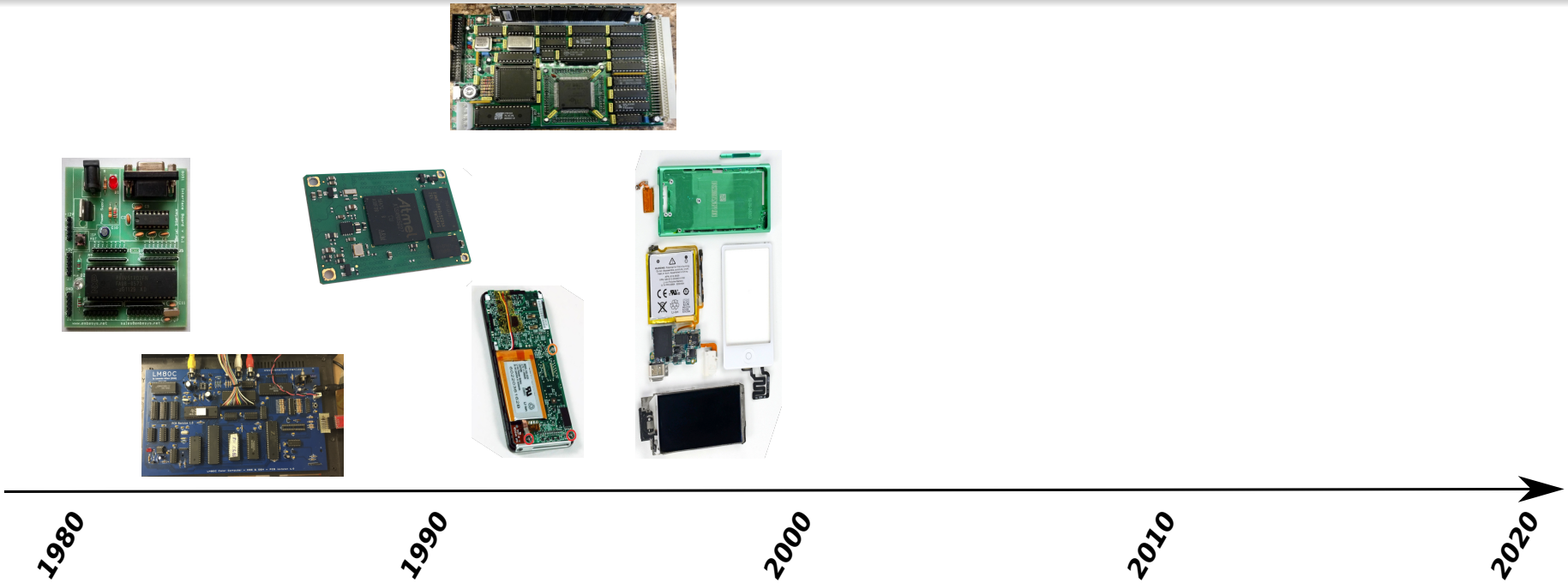
En los años 90 se continúan desarrollando cada vez mas variedad de sistemas con mejores capacidades, aparecen los primeros modelos de ARM en este mercado, comienzan a planearse dispositivos portátiles bastante rústicos, pero se empieza a avizorar una tendencia.

Sistemas Operativos Embedded de Propósito General



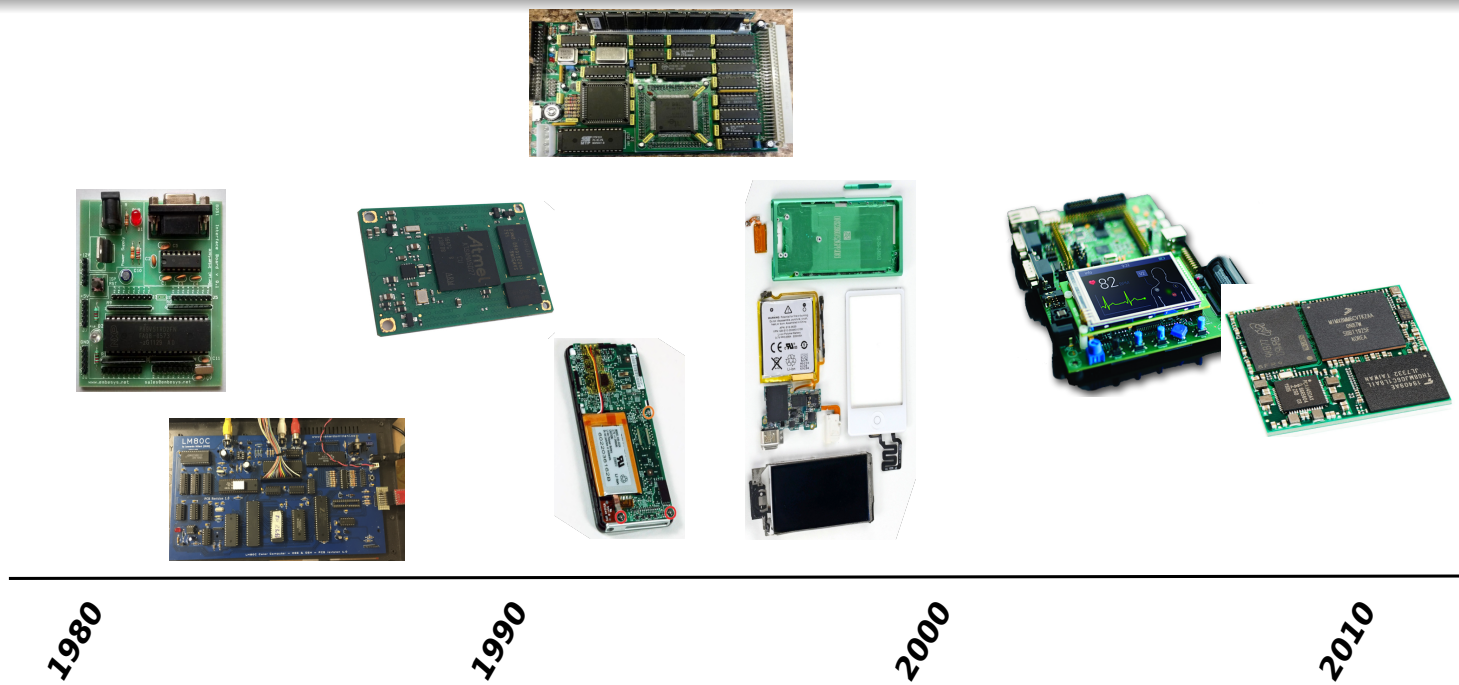
En 1994 Intel lanza el 80386EX, un sistema embedded 100 % compatible con una PC pero con la E/S incluida en el chip, basado en su procesador 80386 (original de 1984). Con él se equipó el telescopio Hubble de la NASA que orbitó la tierra 20 años aproximadamente, y mas tarde fue la base de los primeros sistemas GPS Garmin. Recién en la segunda mitad de esta década empieza a tener sentido hablar de GPOS para este rango de sistemas.

Sistemas Operativos Embedded de Propósito General



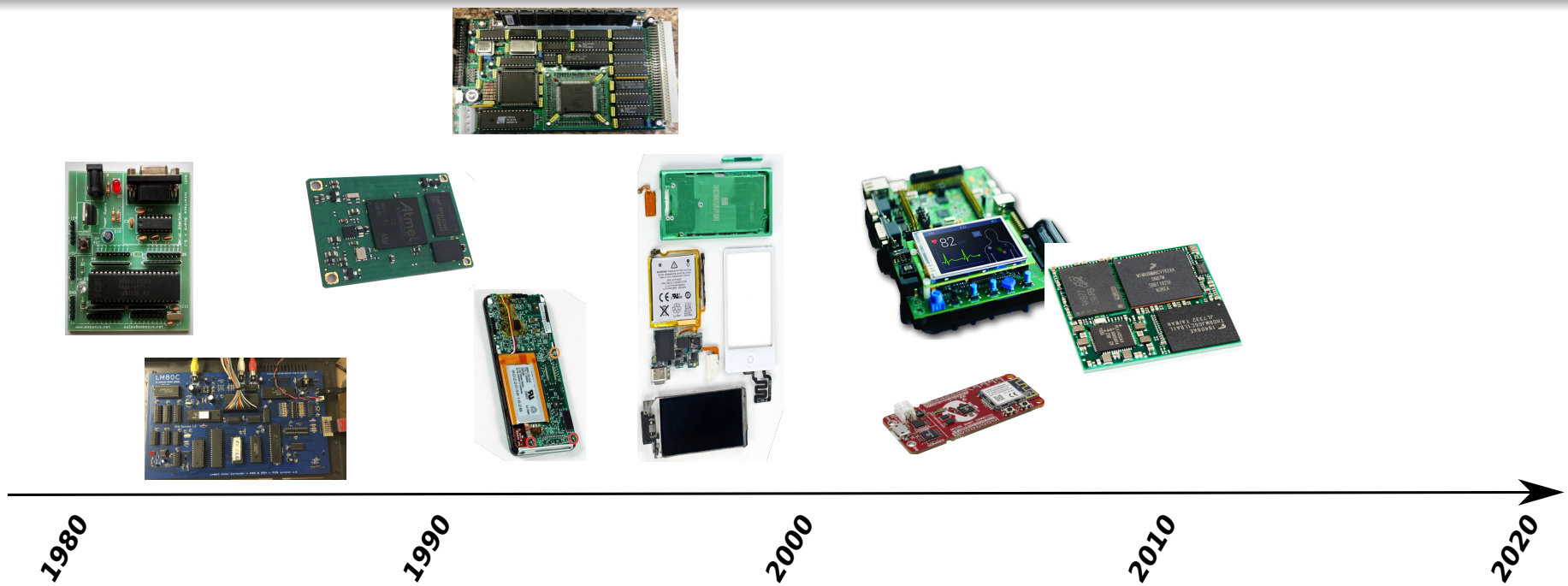
Sobre el final de los 90 Apple trabaja en el iPod que será presentado en 2001. El consumo de energía eléctrica directamente relacionado con la duración de la batería comenzará a ser muy significativo.

Sistemas Operativos Embedded de Propósito General



Luego del 2000 comenzaría a extenderse Linux, muy fuertemente en el mercado de servidores, pero además se empieza a disponer de varios incipientes intentos de sistema Linux muy compactos que resuelven soluciones embedded de alta prestación. Web server integrados, Dispositivos de aplicaciones médicas, entre otros. Comienza a mirarse a los procesadores ARM mas potentes para este segmento. Problema: La no estandarización del hardware de E/S

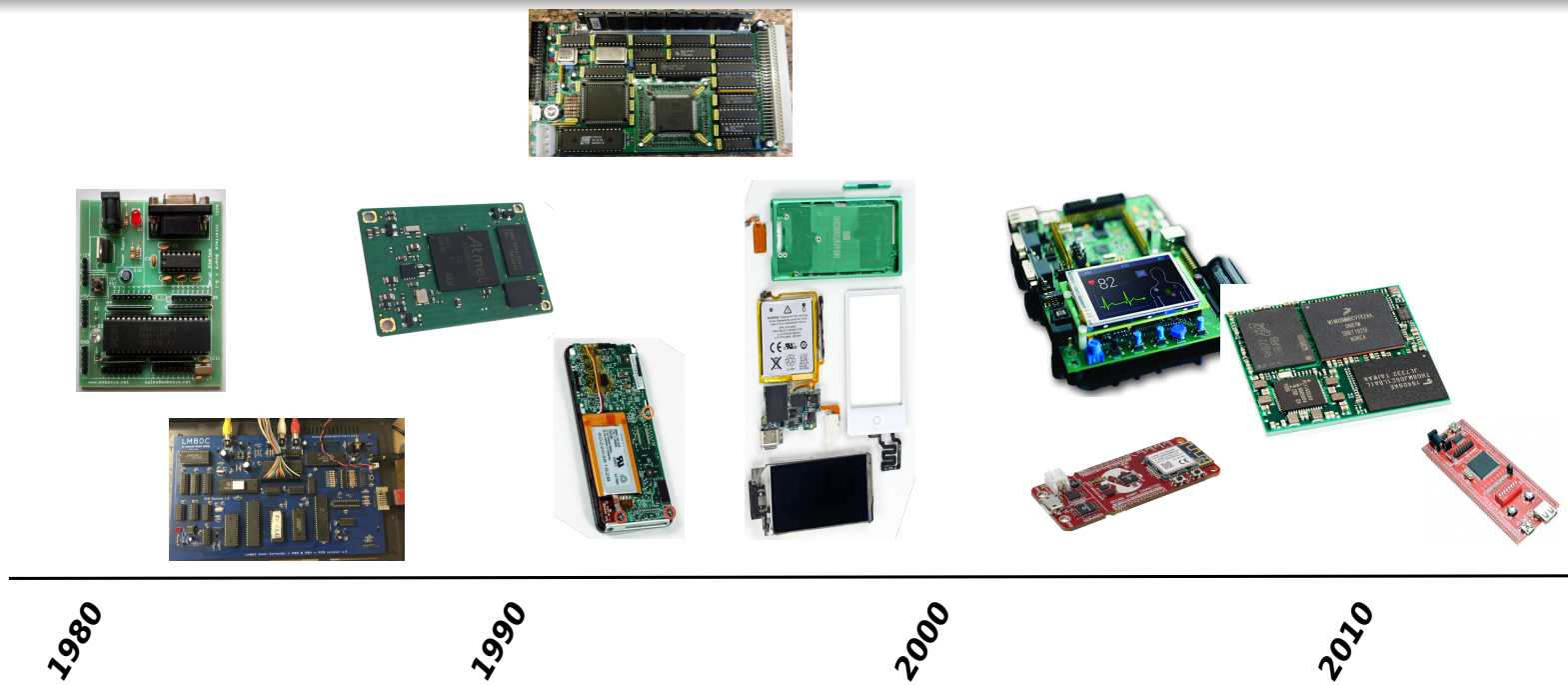
Sistemas Operativos Embedded de Propósito General



No obstante el desarrollo de dispositivos en el segmento inferior es masivo. Aparecen los sticks como nuevas versiones económicas de kits de desarrollo, todos son interfaces JTAG para debug directo en el hardware.

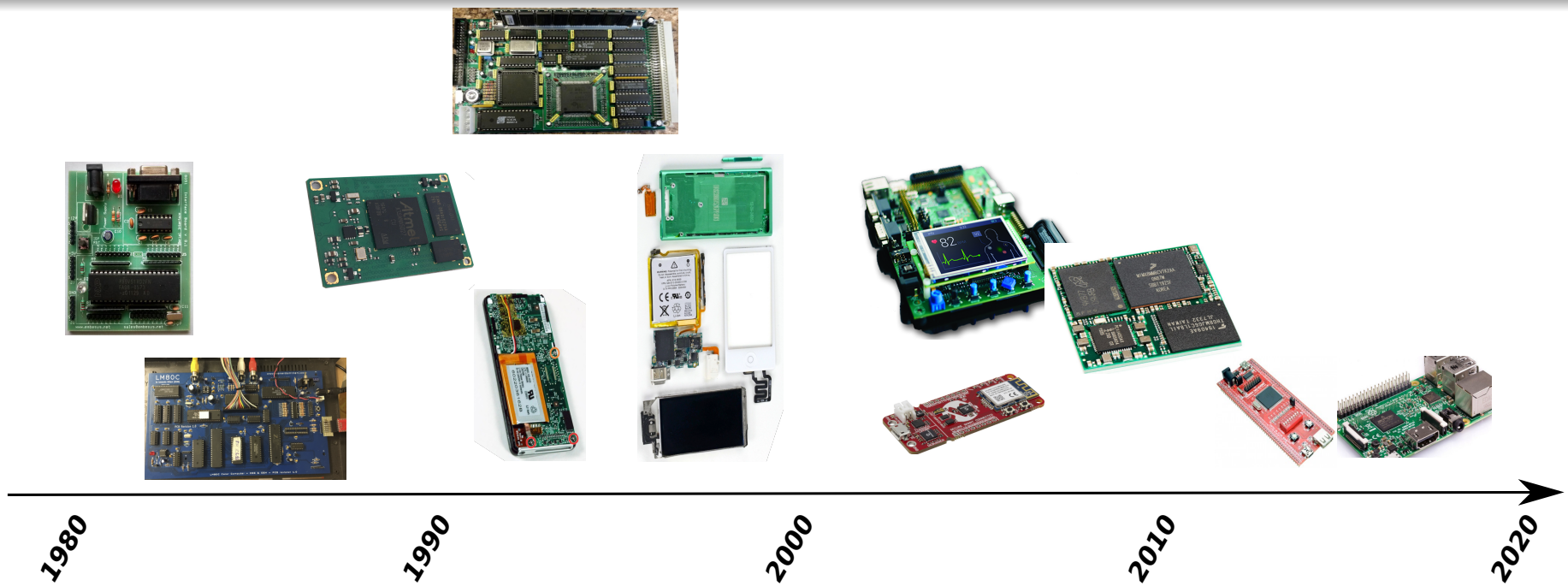
Comienza la diversidad de sistemas operativos para sistemas embedded. Real Time? Linux? Bare metal? Discusión sin fin, ni sentido... En el medio de esto... en 2007 el iphone... ¿Y no es un embedded System?

Sistemas Operativos Embedded de Propósito General



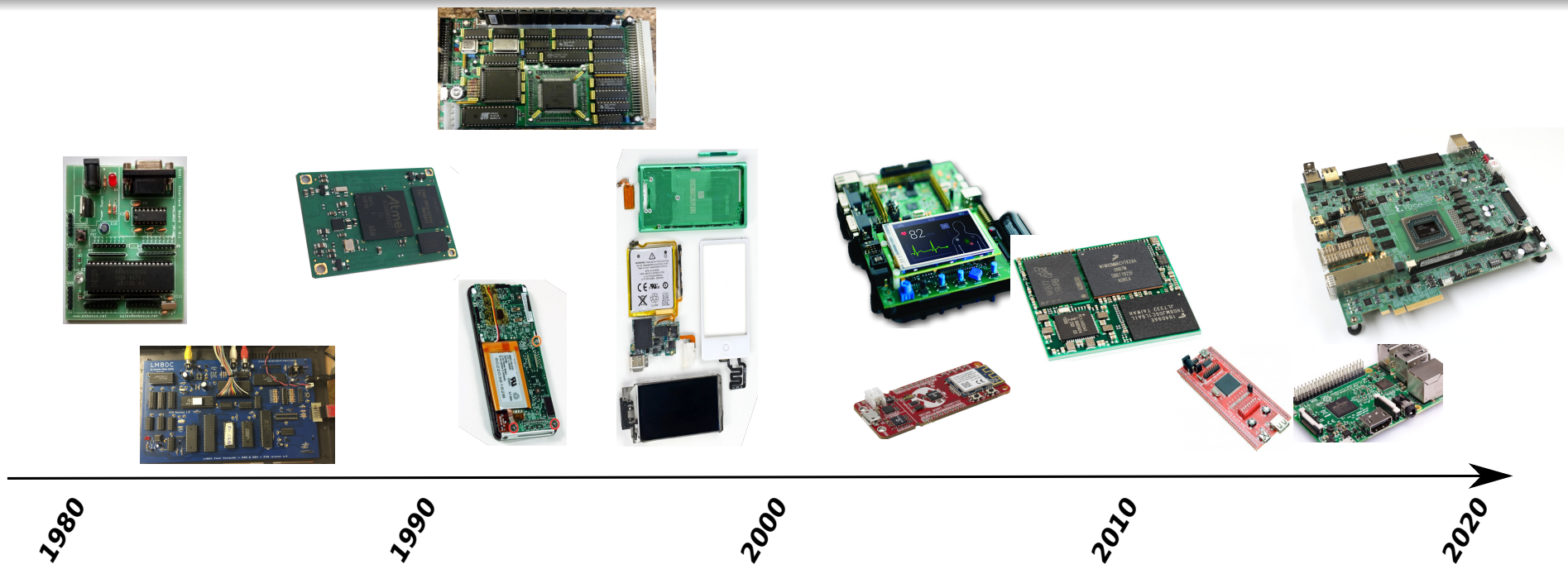
Iniciada esta última década, se continua desplegando el segmento de microcontrollers con mayores capacidades de hardware y se afirman varios RTOS como referencias dentro de ese segmento.
Los fabricantes de FPGA comienzan a proveer cores para diseño de SoC. Aparecen los primeros SoC que combinan Cores con E/S, y recursos FPGA. Todos soportan Linux.

Sistemas Operativos Embedded de Propósito General



Sin embargo en el segmento alto las soluciones de bajo costo comienzan a ganar terreno y se afianza Linux Embedded como la alternativa mas empleada en el rango de embedded systems.
En el segmento alto los smartphones y tablet catapultan a ARM como líder en el mercado embedded. En 2012 Raspberry Pi, y en 2013 BeagleBone presentan sistemas de muy bajo costo y basados en Linux Embedded sobre ARM

Sistemas Operativos Embedded de Propósito General



Y llegamos a nuestros días con la aparición en estos últimos años de SoCs que combinan varios procesadores de propósito general, real time, con un sistema embebido de E/S, y amplios recursos FPGA con celdas y LUTs estándar, DSP Y celdas de Inteligencia artificial en un chip.

Portando GPOS existentes a sistemas embebidos

Portando GPOS existentes a sistemas embebidos

- Lo mas eficiente que se encontró fue portar Linux, FreeBSD, NetBSD, y alguna otra implementación Linux Like a los embedded systems.

Portando GPOS existentes a sistemas embebidos

- Lo mas eficiente que se encontró fue portar Linux, FreeBSD, NetBSD, y alguna otra implementación Linux Like a los embedded systems.
- Mas específicamente a ARM.

Portando GPOS existentes a sistemas embebidos

- Lo mas eficiente que se encontró fue portar Linux, FreeBSD, NetBSD, y alguna otra implementación Linux Like a los embedded systems.
- Mas específicamente a ARM.
- Intel abandono este segmento de modo que portar a embedded equivale a portar a ARM. Si bien hay otras arquitecturas, ARM es abrumadoramente mayoritaria, al momento.

Portando GPOS existentes a sistemas embebidos

- Lo mas eficiente que se encontró fue portar Linux, FreeBSD, NetBSD, y alguna otra implementación Linux Like a los embedded systems.
- Mas específicamente a ARM.
- Intel abandono este segmento de modo que portar a embedded equivale a portar a ARM. Si bien hay otras arquitecturas, ARM es abrumadoramente mayoritaria, al momento.
- Cada sistema ARM tiene su propio hardware, diferente del resto. Linux nació en una PC 80386, donde todo el hardware de E/S es tan compatible como el del procesador.

Portando GPOS existentes a sistemas embebidos

- Lo mas eficiente que se encontró fue portar Linux, FreeBSD, NetBSD, y alguna otra implementación Linux Like a los embedded systems.
- Mas específicamente a ARM.
- Intel abandono este segmento de modo que portar a embedded equivale a portar a ARM. Si bien hay otras arquitecturas, ARM es abrumadoramente mayoritaria, al momento.
- Cada sistema ARM tiene su propio hardware, diferente del resto. Linux nació en una PC 80386, donde todo el hardware de E/S es tan compatible como el del procesador.
- Esto generó por sí solo un proyecto gigante para el port de Linux al mundo ARM.

Portando GPOS existentes a sistemas embebidos

- Lo mas eficiente que se encontró fue portar Linux, FreeBSD, NetBSD, y alguna otra implementación Linux Like a los embedded systems.
- Mas específicamente a ARM.
- Intel abandono este segmento de modo que portar a embedded equivale a portar a ARM. Si bien hay otras arquitecturas, ARM es abrumadoramente mayoritaria, al momento.
- Cada sistema ARM tiene su propio hardware, diferente del resto. Linux nació en una PC 80386, donde todo el hardware de E/S es tan compatible como el del procesador.
- Esto generó por sí solo un proyecto gigante para el port de Linux al mundo ARM.
- ***Raspbian*** (Debian para Raspberry Pi), ***Android*** (GPOS para smartphones), son todos ports de Linux hacia ARM. Además si se quiere optimizar el kernel hay que profundizar el re planteo de muchos módulos del kernel para adaptarlos a la arquitectura de ARM.

GPOS Lo que sabemos y lo que veremos

Un curso de Embedded Linux es el siguiente nivel. No obstante ya sabemos (o deberíamos saber), como crear procesos, manejar señales e intercomunicar procesos.

Disclaimer:

Si hay problemas con estos conocimientos, tenemos material adicional que no forma parte de este curso sino que deberían traer incorporados, de modo que lo que subamos reviste caracter de material accesorio de repaso o refuerzo.

Temario

Definición de RTOS

Real Time Operating System

Sistema operativo destinado a aplicaciones que generalmente tienen requisitos de tiempo muy estrictos. Primero, un RTOS debe poder responder a eventos rápidamente, es decir dentro de un lapso breve, conocido como *latencia de la interrupción*. En segundo lugar, debe completar cada pedido dentro de un límite de tiempo pre establecido, conocido como *tiempo límite de la tarea*.

Definición de RTOS

Real Time Operating System

Sistema operativo destinado a aplicaciones que generalmente tienen requisitos de tiempo muy estrictos. Primero, un RTOS debe poder responder a eventos rápidamente, es decir dentro de un lapso breve, conocido como *latencia de la interrupción*. En segundo lugar, debe completar cada pedido dentro de un límite de tiempo pre establecido, conocido como *tiempo límite de la tarea*.

- Si un RTOS siempre puede cumplir con estos requisitos críticos de tiempo, hablamos de un hard Real Time System.

Definición de RTOS

Real Time Operating System

Sistema operativo destinado a aplicaciones que generalmente tienen requisitos de tiempo muy estrictos. Primero, un RTOS debe poder responder a eventos rápidamente, es decir dentro de un lapso breve, conocido como *latencia de la interrupción*. En segundo lugar, debe completar cada pedido dentro de un límite de tiempo pre establecido, conocido como *tiempo límite de la tarea*.

- Si un RTOS siempre puede cumplir con estos requisitos críticos de tiempo, hablamos de un hard Real Time System.
- Si solo puede cumplir los requisitos la mayor parte del tiempo pero no siempre, hablamos de un soft Real Time System.

Capacidades de un RTOS

Para cumplir con requisitos de tiempo estrictos, los RTOS suelen ser diseñados con las siguientes capacidades:

Capacidades de un RTOS

Para cumplir con requisitos de tiempo estrictos, los RTOS suelen ser diseñados con las siguientes capacidades:

- **Interrupt Latency Mínima**

Capacidades de un RTOS

Para cumplir con requisitos de tiempo estrictos, los RTOS suelen ser diseñados con las siguientes capacidades:

- **Interrupt Latency Mínima**
- **Regiones Críticas Cortas**

Capacidades de un RTOS

Para cumplir con requisitos de tiempo estrictos, los RTOS suelen ser diseñados con las siguientes capacidades:

- **Interrupt Latency Mínima**
- **Regiones Críticas Cortas**
- **Preemptive Scheduling**

Capacidades de un RTOS

Para cumplir con requisitos de tiempo estrictos, los RTOS suelen ser diseñados con las siguientes capacidades:

- **Interrupt Latency Mínima**
- **Regiones Críticas Cortas**
- **Preemptive Scheduling**
- **Algoritmos de Scheduling Avanzados**

Interrupt Latency

Es el tiempo transcurrido desde que la CPU recibe la interrupción hasta que comienza a ejecutar la primer instrucción del interrupt handler.
A fin de minimizar el Interrupt Latency, un RTOS no debe deshabilitar interrupciones durante períodos de tiempo prolongados.
Esto significa soportar anidamiento de interrupciones de alta prioridad respecto de interrupciones de baja prioridad, de modo de asegurar que las de menor prioridad no demoren a las mas prioritarias.

Regiones Críticas Cortas

Todos los kernels poseen regiones de código críticas en las que manipulan objetos compartidos, para por ejemplo sincronizar procesos.
Esto se logra deshabilitando las interrupciones al inicio de esa sección de código y rehabilitando las interrupciones al final.
Se las suele llamar operaciones atómicas (ya que en la práctica se comportan como un bloque indivisible).
Sin perjuicio de su existencia, debe procurarse que estas regiones de código involucren la menor cantidad de instrucciones posible, es decir sean lo mas cortas posible.

Preemptive Scheduling

Preemption, como ya hemos dicho, significa expropiar la CPU a una tarea de menor prioridad a expensas de una tarea de mayor prioridad, y en cualquier momento.
Para cumplir con los plazos temporales de todas las tareas, un RTOS debe tener un scheduler preemptive.
Además (no menor) el tiempo de conmutación de tarea debe ser lo mas corto posible, ya que suma dramáticamente en ocasiones a la demora en atender una tarea sujeta a un evento por ejemplo.

Algoritmos de Scheduling Avanzados

El Scheduler Preemptive es condición necesaria pero no suficiente, para diseñar un RTOS.
Si el Scheduler no es Preemptive, una tarea de muy alta prioridad puede ser demorada por una de prioridad menor. Inaceptable. Es imposible cumplir requerimientos temporales estrictos.
Sin embargo, aún siendo preemptive, no siempre puede garantizarse que una tarea pueda cumplir sus deadlines.
Además de ser Preemptive, el algoritmo de scheduling debe además incluir políticas (en ocasiones sofisticadas) que permitan asegurar requerimientos muy estrictos.

Temario

Introducción

Diferencias entre GPOS y RTOS

En un GPOS el scheduling de tareas considera generalmente el mejor balance de rendimiento general posible, considerando de manera global los requerimientos de las tareas que se ejecutan. Típico caso el *fear scheduling* de Linux.

En un RTOS, en cambio, se busca el menor tiempo de respuesta a eventos y el cumplimiento estricto de los deadlines temporales de cada tarea. El algoritmo de scheduling debe privilegiar a aquellas tareas con mayores restricciones temporales. Dicho de otro modo se mapean restricciones temporales en prioridades.

Un scheduler para un RTOS es un Preemptive Scheduler basado en prioridades.

Algoritmos de Scheduling

Para diseñar un scheduler existen las siguientes dos líneas principales (y sus variantes).

Algoritmos de Scheduling

Para diseñar un scheduler existen las siguientes dos líneas principales (y sus variantes).

- Rate Monotonic Scheduler (RMS).

Algoritmos de Scheduling

Para diseñar un scheduler existen las siguientes dos líneas principales (y sus variantes).

- Rate Monotonic Scheduler (RMS).
- Earliest-Deadline-First (EDF).

Rate Monotonic Scheduler (RMS)

Rate Monotonic Scheduler (RMS)

- Basado en el trabajo *"Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment"*, C.L.Liu (MIT) y James W. Layland (CALTECH), Journal of the Association for Computing Machinery (ACM), Vol 20, Nro. 1. 1973

Rate Monotonic Scheduler (RMS)

- Basado en el trabajo *"Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment"*, C.L.Liu (MIT) y James W. Layland (CALTECH), Journal of the Association for Computing Machinery (ACM), Vol 20, Nro. 1. 1973
- Este trabajo muestra que para un set de n tareas periódicas con períodos únicos es posible despacharlas a todas si el uso de CPU se mantiene por debajo de cierto límite el cual depende de la cantidad de tareas.

Rate Monotonic Scheduler (RMS)

- Basado en el trabajo *"Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment"*, C.L.Liu (MIT) y James W. Layland (CALTECH), Journal of the Association for Computing Machinery (ACM), Vol 20, Nro. 1. 1973
- Este trabajo muestra que para un set de n tareas periódicas con períodos únicos es posible despacharlas a todas si el uso de CPU se mantiene por debajo de cierto límite el cual depende de la cantidad de tareas.
- Numéricamente:

$$u = \sum_{i=1}^{n} \frac{C_i}{T_i} \leqslant n.(2^{\frac{1}{n}} - 1)$$

Rate Monotonic Scheduler (RMS)

- Basado en el trabajo *"Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment"*, C.L.Liu (MIT) y James W. Layland (CALTECH), Journal of the Association for Computing Machinery (ACM), Vol 20, Nro. 1. 1973
- Este trabajo muestra que para un set de n tareas periódicas con períodos únicos es posible despacharlas a todas si el uso de CPU se mantiene por debajo de cierto límite el cual depende de la cantidad de tareas.
- Numéricamente:

$$u = \sum_{i=1}^{n} \frac{C_i}{T_i} \leqslant n.(2^{\frac{1}{n}} - 1)$$

- Donde U es la factibilidad de scheduling, n es la cantidad de tareas, C_i es el tiempo de cómputo de la tarea en cada ciclo de activación y T_i es el período de activación de la tarea.

Rate Monotonic Scheduler (RMS)

Rate Monotonic Scheduler (RMS)

- De este modo si se tiene dos tareas, el límite es $U \leqslant 2.(\sqrt{2} - 1) \simeq 0,828427$, es decir que es factible realizar el scheduling de dos procesos si la carga de CPU no supera el 82,8427 %.

Rate Monotonic Scheduler (RMS)

- De este modo si se tiene dos tareas, el límite es $U \leqslant 2.(\sqrt{2} - 1) \simeq 0,828427$, es decir que es factible realizar el scheduling de dos procesos si la carga de CPU no supera el 82,8427 %.
- Cuando la cantidad de tareas crece, se es mas estricto en el uso de CPU. Así para una cantidad suficientemente grande de tareas no se debe superar la carga dada por:

$$\lim_{n \to \infty} n.(2^{\frac{1}{n}} - 1) = ln(2) = 0,693147$$

Rate Monotonic Scheduler (RMS)

- De este modo si se tiene dos tareas, el límite es $U \leqslant 2.(\sqrt{2} - 1) \simeq 0,828427$, es decir que es factible realizar el scheduling de dos procesos si la carga de CPU no supera el 82,8427 %.
- Cuando la cantidad de tareas crece, se es mas estricto en el uso de CPU. Así para una cantidad suficientemente grande de tareas no se debe superar la carga dada por:

$$\lim_{n\to\infty} n.(2^{\frac{1}{n}} - 1) = ln(2) = 0,693147$$

- Significa que si el consumo de CPU requerido por una cantidad suficientemente grande de tareas no supera el 69,31 %, este algoritmo las puede “schedular”a todas cumpliendo los requerimientos temporales de c/u.

Rate Monotonic Scheduler (RMS)

Se asume además que las tareas cumplen las siguientes propiedades:

- ✓ Tareas periódicas. Es decir tareas que requieran ejecutarse a intervalos estrictamente regulares.

Rate Monotonic Scheduler (RMS)

Se asume además que las tareas cumplen las siguientes propiedades:

- ✓ Tareas periódicas. Es decir tareas que requieran ejecutarse a intervalos estrictamente regulares.
- ✓ Sistema de prioridades estático. Tareas con períodos de ejecución mas cortos tienen la prioridad mas alta.

Rate Monotonic Scheduler (RMS)

Se asume además que las tareas cumplen las siguientes propiedades:

- ✓ Tareas periódicas. Es decir tareas que requieran ejecutarse a intervalos estrictamente regulares.
- ✓ Sistema de prioridades estático. Tareas con períodos de ejecución mas cortos tienen la prioridad mas alta.
- ✓ Preemption. El sistema ejecutará siempre las tareas de mayor prioridad. Si al momento de su activación se encuentra ejecutando una de menor prioridad esta será inmediatamente interrumpida (preempted), para darle la CPU a la tarea mas prioritaria.

Rate Monotonic Scheduler (RMS)

Se asume además que las tareas cumplen las siguientes propiedades:

- ✓ Tareas periódicas. Es decir tareas que requieran ejecutarse a intervalos estrictamente regulares.
- ✓ Sistema de prioridades estático. Tareas con períodos de ejecución mas cortos tienen la prioridad mas alta.
- ✓ Preemption. El sistema ejecutará siempre las tareas de mayor prioridad. Si al momento de su activación se encuentra ejecutando una de menor prioridad esta será inmediatamente interrumpida (preempted), para darle la CPU a la tarea mas prioritaria.
- ✓ Las tareas no comparten recursos de modo que nada pueda bloquearlas o ponerlas en espera.

Rate Monotonic Scheduler (RMS)

Se asume además que las tareas cumplen las siguientes propiedades:

- ✓ Tareas periódicas. Es decir tareas que requieran ejecutarse a intervalos estrictamente regulares.
- ✓ Sistema de prioridades estático. Tareas con períodos de ejecución mas cortos tienen la prioridad mas alta.
- ✓ Preemption. El sistema ejecutará siempre las tareas de mayor prioridad. Si al momento de su activación se encuentra ejecutando una de menor prioridad esta será inmediatamente interrumpida (preempted), para darle la CPU a la tarea mas prioritaria.
- ✓ Las tareas no comparten recursos de modo que nada pueda bloquearlas o ponerlas en espera.
- ✓ Tiempo de cambio de contexto cero, lo mismo que otras operaciones de tareas como por ejemplo startup, o release.

Rate Monotonic Scheduler (RMS)

Tarea1 $C_1=2$ $T_1=4$

Tarea2 $C_2=4$ $T_2=8$

Rate Monotonic Scheduler (RMS)

Tarea1 C_1=2 T_1=4

Tarea2 C_2=4 T_2=8

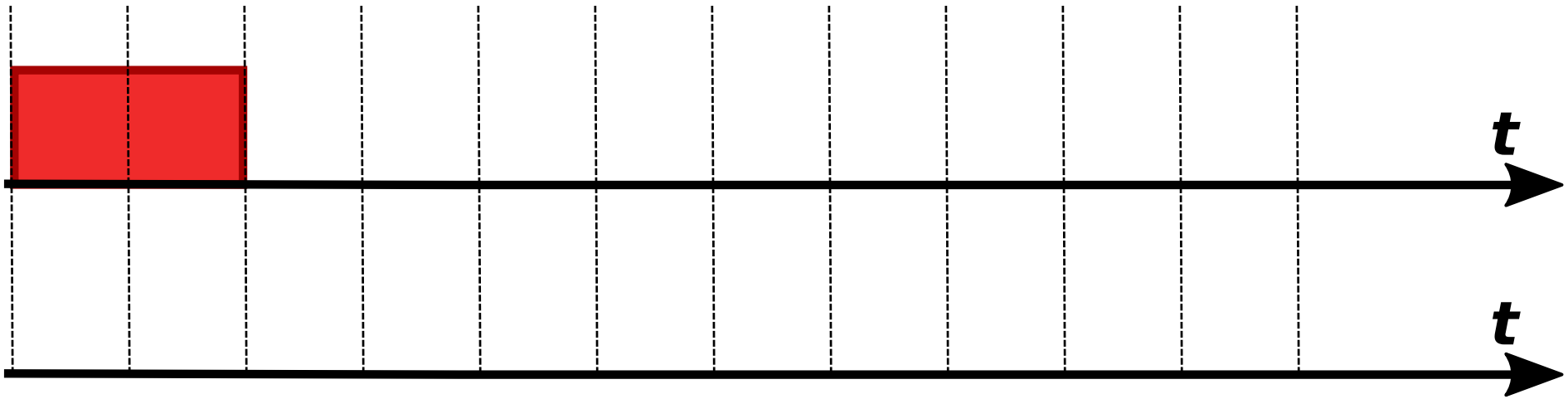
$T_1 < T_2$ => Prioridad Tarea 1 > Prioridad Tarea 2

Rate Monotonic Scheduler (RMS)

Tarea1 C_1=2 T_1=4

Tarea2 C_2=4 T_2=8

$T_1 < T_2$ => Prioridad Tarea 1 > Prioridad Tarea 2

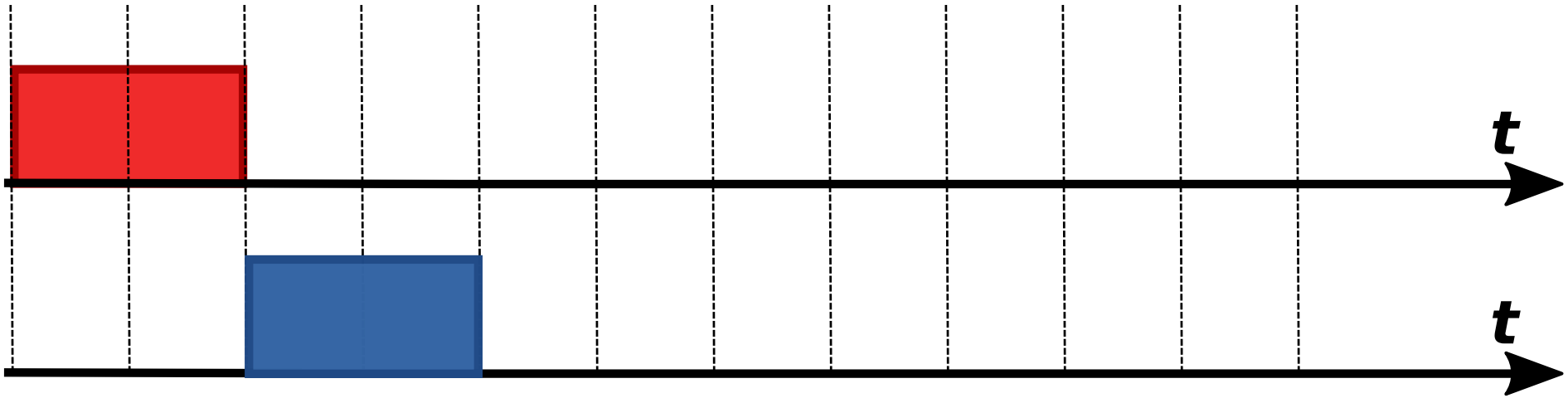


Rate Monotonic Scheduler (RMS)

Tarea1 $C_1=2$ $T_1=4$

Tarea2 $C_2=4$ $T_2=8$

$T_1 < T_2 \Rightarrow$ Prioridad Tarea 1 $>$ Prioridad Tarea 2



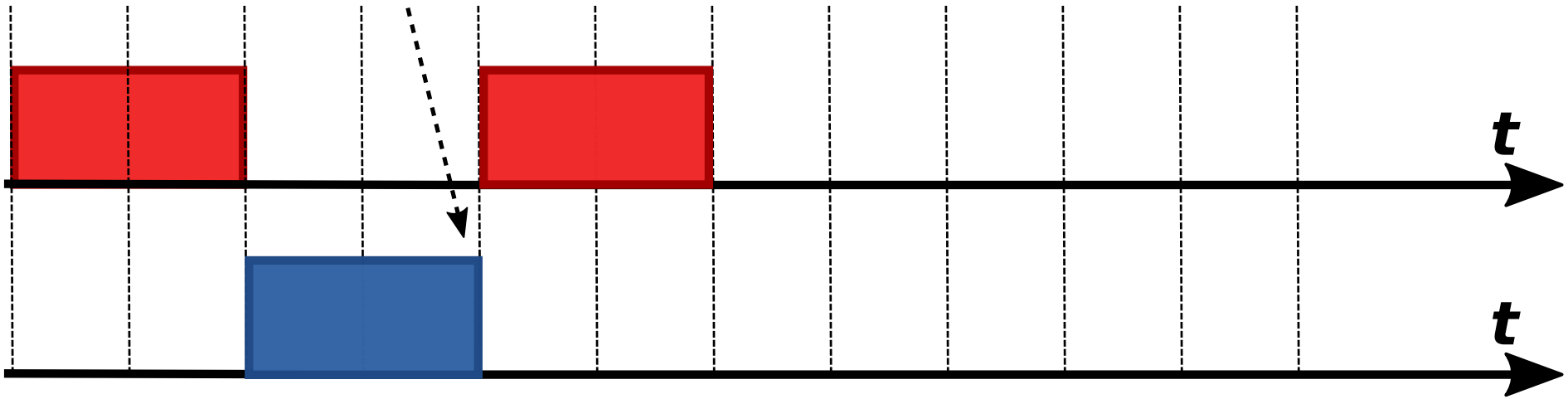
Rate Monotonic Scheduler (RMS)

Tarea1 C_1=2 T_1=4

Tarea2 C_2=4 T_2=8

$T_1 < T_2$ => Prioridad Tarea 1 > Prioridad Tarea 2

Tarea 1 Interrumpe a Tarea2

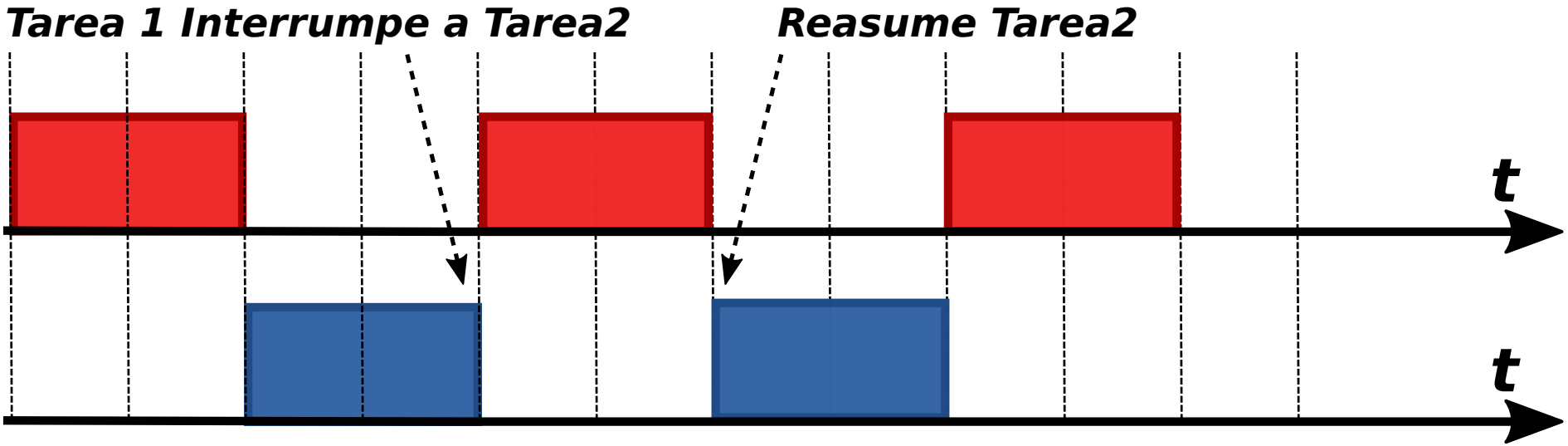


Rate Monotonic Scheduler (RMS)

Tarea1 C_1=2 T_1=4

Tarea2 C_2=4 T_2=8

$T_1 < T_2 \Rightarrow$ Prioridad Tarea 1 $>$ Prioridad Tarea 2



$$U = \frac{2}{4} + \frac{4}{8} = 1 \nleq 2 * (\sqrt{2} - 1)$$

Rate Monotonic Scheduler (RMS)

Rate Monotonic Scheduler (RMS)

- Conclusiones

Rate Monotonic Scheduler (RMS)

- Conclusiones

✓ RMS establece condición suficiente, pero no necesaria. El ejemplo anterior no cumple pero igualmente es factible cumplir las demandas temporales de ambas tareas. Lógicamente se consume entre ambas toda la CPU. No queda margen para otra actividad por parte de la CPU.

Rate Monotonic Scheduler (RMS)

- Conclusiones

✓ RMS establece condición suficiente, pero no necesaria. El ejemplo anterior no cumple pero igualmente es factible cumplir las demandas temporales de ambas tareas. Lógicamente se consume entre ambas toda la CPU. No queda margen para otra actividad por parte de la CPU.

✓ El % de CPU sobrante, en caso de haberlo, puede dedicarse a la ejecución de otras tareas no periódicas (algún proceso interactivo por ejemplo).

Rate Monotonic Scheduler (RMS)

- Conclusiones

✓ RMS establece condición suficiente, pero no necesaria. El ejemplo anterior no cumple pero igualmente es factible cumplir las demandas temporales de ambas tareas. Lógicamente se consume entre ambas toda la CPU. No queda margen para otra actividad por parte de la CPU.

✓ El % de CPU sobrante, en caso de haberlo, puede dedicarse a la ejecución de otras tareas no periódicas (algún proceso interactivo por ejemplo).

✓ Ideal para Sistemas Monoprocesador

Rate Monotonic Scheduler (RMS)

- Conclusiones

✓ RMS establece condición suficiente, pero no necesaria. El ejemplo anterior no cumple pero igualmente es factible cumplir las demandas temporales de ambas tareas. Lógicamente se consume entre ambas toda la CPU. No queda margen para otra actividad por parte de la CPU.

✓ El % de CPU sobrante, en caso de haberlo, puede dedicarse a la ejecución de otras tareas no periódicas (algún proceso interactivo por ejemplo).

✓ Ideal para Sistemas Monoprocesador

✓ Las tareas guardan una relación armónica: El período de cada tarea puede expresarse en términos del período de una tarea mas prioritaria

Earliest-Deadline-First (EDF)

Earliest-Deadline-First (EDF)

- RMS es eficiente pero plantea un esquema algo rígido, y en ocasiones no da resultados óptimos.

Earliest-Deadline-First (EDF)

- RMS es eficiente pero plantea un esquema algo rígido, y en ocasiones no da resultados óptimos.
- Leung y Merrill en 1980 publicaron un trabajo cuyo título es “A note on preemptive scheduling of periodic, real-time tasks”en el que plantean un sistema de scheduling para una cantidad *m* de CPU’s.

Earliest-Deadline-First (EDF)

- RMS es eficiente pero plantea un esquema algo rígido, y en ocasiones no da resultados óptimos.
- Leung y Merrill en 1980 publicaron un trabajo cuyo título es “A note on preemptive scheduling of periodic, real-time tasks”en el que plantean un sistema de scheduling para una cantidad *m* de CPU’s.
- Sugiere la necesidad de un algoritmo para incluir un proceso en una cola de prioridades y moverlo dentro de la misma dinámicamente a medida que su tiempo de ejecución se acerca (aumenta su posibilidad de preemption)

Earliest-Deadline-First (EDF)

Earliest-Deadline-First (EDF)

- A diferencia de RMS acepta tareas periódicas y no periódicas.

Earliest-Deadline-First (EDF)

- A diferencia de RMS acepta tareas periódicas y no periódicas.
- La tarea mas próxima en el tiempo es la que gana mayor prioridad.

Earliest-Deadline-First (EDF)

- A diferencia de RMS acepta tareas periódicas y no periódicas.
- La tarea mas próxima en el tiempo es la que gana mayor prioridad.
- La condición de para poder despachar todas las tareas es:

$$u = \sum_{i=1}^{n} \frac{C_i}{T_i} \leqslant 1$$

Earliest-Deadline-First (EDF)

- A diferencia de RMS acepta tareas periódicas y no periódicas.
- La tarea mas próxima en el tiempo es la que gana mayor prioridad.
- La condición de para poder despachar todas las tareas es:

$$u = \sum_{i=1}^{n} \frac{C_i}{T_i} \leqslant 1$$

- Dos tareas idénticas con idéntico período pueden tomar el 100 % de la CPU, a diferencia de RMS que salvaba parte del uso de CPU permitiendo “schedular”alguna tarea adicional.

Earliest-Deadline-First (EDF)

- A diferencia de RMS acepta tareas periódicas y no periódicas.
- La tarea mas próxima en el tiempo es la que gana mayor prioridad.
- La condición de para poder despachar todas las tareas es:

$$u = \sum_{i=1}^{n} \frac{C_i}{T_i} \leqslant 1$$

- Dos tareas idénticas con idéntico período pueden tomar el 100 % de la CPU, a diferencia de RMS que salvaba parte del uso de CPU permitiendo “schedular”alguna tarea adicional.
- La lista de prioridades es dinámica (otra diferencia de fondo con RMS).

Earliest-Deadline-First (EDF)

Earliest-Deadline-First (EDF)

- La lista de ejecución dinámica, lo hace mas complejo de implementar ya que se debe estar chequeando los deadlines de cada tarea y en función de estos, actualizar el orden de las mismas en la cola de ejecución.

Earliest-Deadline-First (EDF)

- La lista de ejecución dinámica, lo hace mas complejo de implementar ya que se debe estar chequeando los deadlines de cada tarea y en función de estos, actualizar el orden de las mismas en la cola de ejecución.
- En caso de sobrecarga de la CPU ($U > 1$), las tareas que se activan menos frecuentemente son las que se penalizan siempre en consecución de la CPU, ya que siempre prioriza los deadlines mas urgentes.

Earliest-Deadline-First (EDF)

- La lista de ejecución dinámica, lo hace mas complejo de implementar ya que se debe estar chequeando los deadlines de cada tarea y en función de estos, actualizar el orden de las mismas en la cola de ejecución.
- En caso de sobrecarga de la CPU ($U > 1$), las tareas que se activan menos frecuentemente son las que se penalizan siempre en consecución de la CPU, ya que siempre prioriza los deadlines mas urgentes.
- Parece no ser predecible además el tiempo de starving al que se verá sometida a tarea que perdió la CPU en las condiciones del ítem anterior.

Earliest-Deadline-First (EDF)

Earliest-Deadline-First (EDF)

- En 2005 el trabajo *“Rate Monotonic vs. EDF: Judgment Day”*, publicado por G.C.Butazzo, alumbra un poco este panorama adverso para EDF.

Earliest-Deadline-First (EDF)

- En 2005 el trabajo *“Rate Monotonic vs. EDF: Judgment Day”*, publicado por G.C.Butazzo, alumbra un poco este panorama adverso para EDF.

- No obstante a la hora de elegir se impone RMS, debido a que al ser estático, es determinístico. Musica para los oídos de la mayor parte de diseñadores.

Deadline-Monotonic Scheduling (DMS)

Deadline-Monotonic Scheduling (DMS)

- Presentado en *Deadline Monotonic Scheduling*, de Neil C. Audsley, en 1990, es una forma general de RMS.

Deadline-Monotonic Scheduling (DMS)

- Presentado en *Deadline Monotonic Scheduling*, de Neil C. Audsley, en 1990, es una forma general de RMS.
- En lugar de manejarse con el período mas cercano en el tiempo, trabaja con el deadline mas cercano en el tiempo como disparador del aumento de prioriad.

Deadline-Monotonic Scheduling (DMS)

- Presentado en *Deadline Monotonic Scheduling*, de Neil C. Audsley, en 1990, es una forma general de RMS.
- En lugar de manejarse con el período mas cercano en el tiempo, trabaja con el deadline mas cercano en el tiempo como disparador del aumento de prioriad.
- En este caso se pueden considerar deadlines dentro de los períodos, es decir que la distancia entre dos deadlines consecutivos dentro de una tarea puede ser menor que el período de activación de ésta.

Deadline-Monotonic Scheduling (DMS)

- Presentado en *Deadline Monotonic Scheduling*, de Neil C. Audsley, en 1990, es una forma general de RMS.
- En lugar de manejarse con el período mas cercano en el tiempo, trabaja con el deadline mas cercano en el tiempo como disparador del aumento de prioriad.
- En este caso se pueden considerar deadlines dentro de los períodos, es decir que la distancia entre dos deadlines consecutivos dentro de una tarea puede ser menor que el período de activación de ésta.
- Cuando los deadlines y los periodos de las tareas coinciden DMS se reduce a RMS.

Task Scheduling: Conclusiones

Task Scheduling: Conclusiones

- RMS y EDF son lineamientos para construir un scheduler reales.

Task Scheduling: Conclusiones

- RMS y EDF son lineamientos para construir un scheduler reales.
- Entre las fallas mas destacables en ambos modelos se destaca la imposibilidad de que las tareas bloqueen o esperen en una cola por un evento por ejemplo.

Task Scheduling: Conclusiones

- RMS y EDF son lineamientos para construir un scheduler reales.
- Entre las fallas mas destacables en ambos modelos se destaca la imposibilidad de que las tareas bloqueen o esperen en una cola por un evento por ejemplo.
- Considerar nulo el tiempo overhead que introduce un task switch, por otra parte, no es realista.

Task Scheduling: Conclusiones

- RMS y EDF son lineamientos para construir un scheduler reales.
- Entre las fallas mas destacables en ambos modelos se destaca la imposibilidad de que las tareas bloqueen o esperen en una cola por un evento por ejemplo.
- Considerar nulo el tiempo overhead que introduce un task switch, por otra parte, no es realista.
- Si se trata de extender RMS y EDF a esquemas mas realistas aparecen nuevas variables a considerar, que son difíciles de cuantificar (deadlines en lugar de períodos es un claro ejemplo)

Task Scheduling: Conclusiones

- RMS y EDF son lineamientos para construir un scheduler reales.
- Entre las fallas mas destacables en ambos modelos se destaca la imposibilidad de que las tareas bloqueen o esperen en una cola por un evento por ejemplo.
- Considerar nulo el tiempo overhead que introduce un task switch, por otra parte, no es realista.
- Si se trata de extender RMS y EDF a esquemas mas realistas aparecen nuevas variables a considerar, que son difíciles de cuantificar (deadlines en lugar de períodos es un claro ejemplo)
- Se termina recurriendo a escenarios del tipo “considerar el peor caso”.

Task Scheduling: Conclusiones

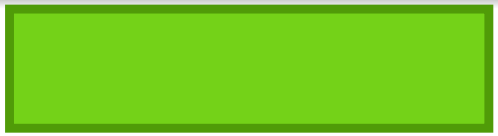
- RMS y EDF son lineamientos para construir un scheduler reales.
- Entre las fallas mas destacables en ambos modelos se destaca la imposibilidad de que las tareas bloqueen o esperen en una cola por un evento por ejemplo.
- Considerar nulo el tiempo overhead que introduce un task switch, por otra parte, no es realista.
- Si se trata de extender RMS y EDF a esquemas mas realistas aparecen nuevas variables a considerar, que son difíciles de cuantificar (deadlines en lugar de períodos es un claro ejemplo)
- Se termina recurriendo a escenarios del tipo “considerar el peor caso”.
- Si se va a compartir recursos, necesariamente se cae en consideraciones de inversión de deadlines o de prioridades.

Temario

Necesitamos bloquear tareas!

En el código de cualquier sistema operativo es necesario utilizar Regiones Críticas (**CR** por **C**ritical **R**egions) a fín de proteger recursos compartidos. Un RTOS no puede escapar de esto .Es inevitable.
Se utilizan ECB's (Event Control Blocks), semáforos, mutexes, colas de mensaje, etc. Todos estos mecanismos se basan en mecanismos de bloqueo (locking protocolos), a través de los cuales una tarea se bloquea si no puede acceder a una **CR**.
Además de los problemas ya estudiados, como dealocks y starvation, si se permite preemption en medio de una **CR**, se arriba a un nuevo problema conocido como Inversión de prioridad.

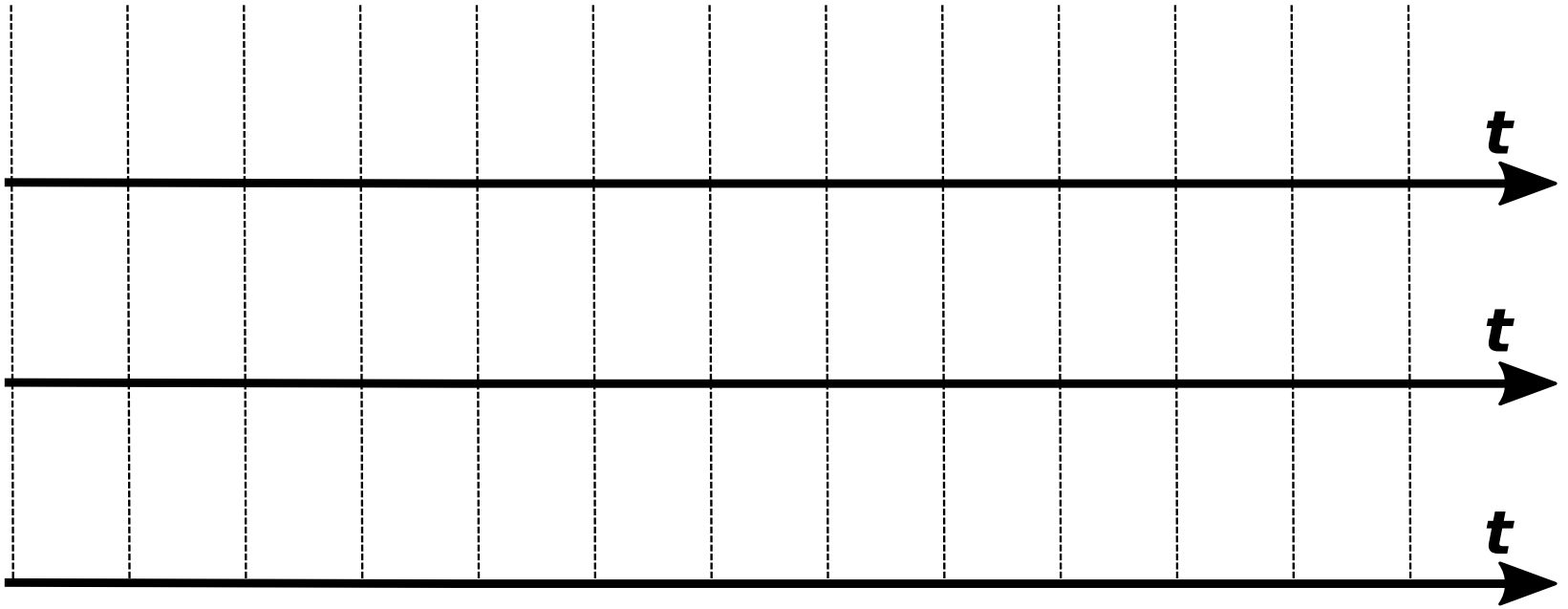
¿Que es una Inversión de Prioridades?



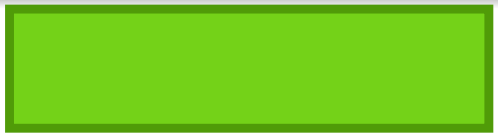
$T_0 : C_0=4, T_0=6, P_0 = Alta$

$T_1 : C_1=3, T_1=8, P_1 = Media$

$T_2 : C_2=4, T_2=12, P_2 = Baja$



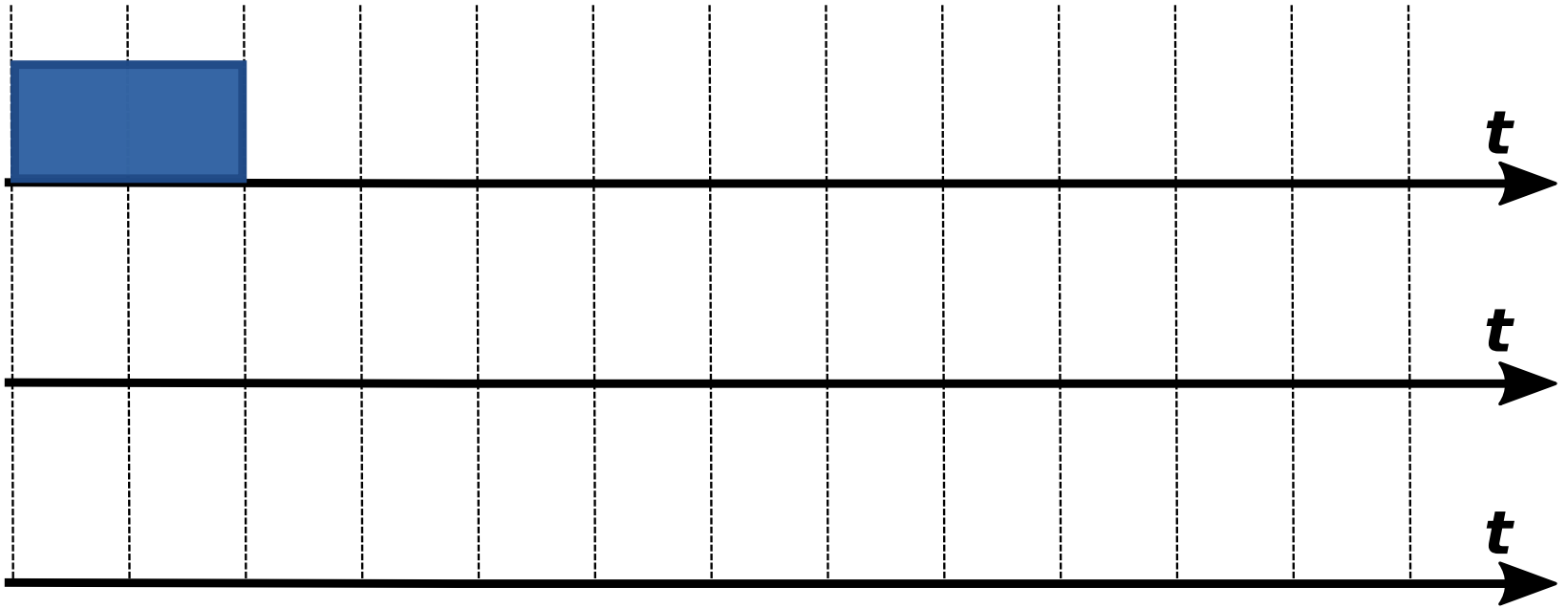
¿Que es una Inversión de Prioridades?



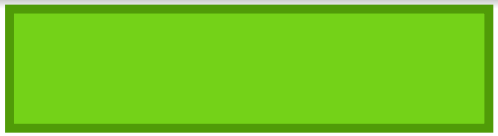
$T_0 : C_0=4, T_0=6, P_0 = Alta$

$T_1 : C_1=3, T_1=8, P_1 = Media$

$T_2 : C_2=4, T_2=12, P_2 = Baja$



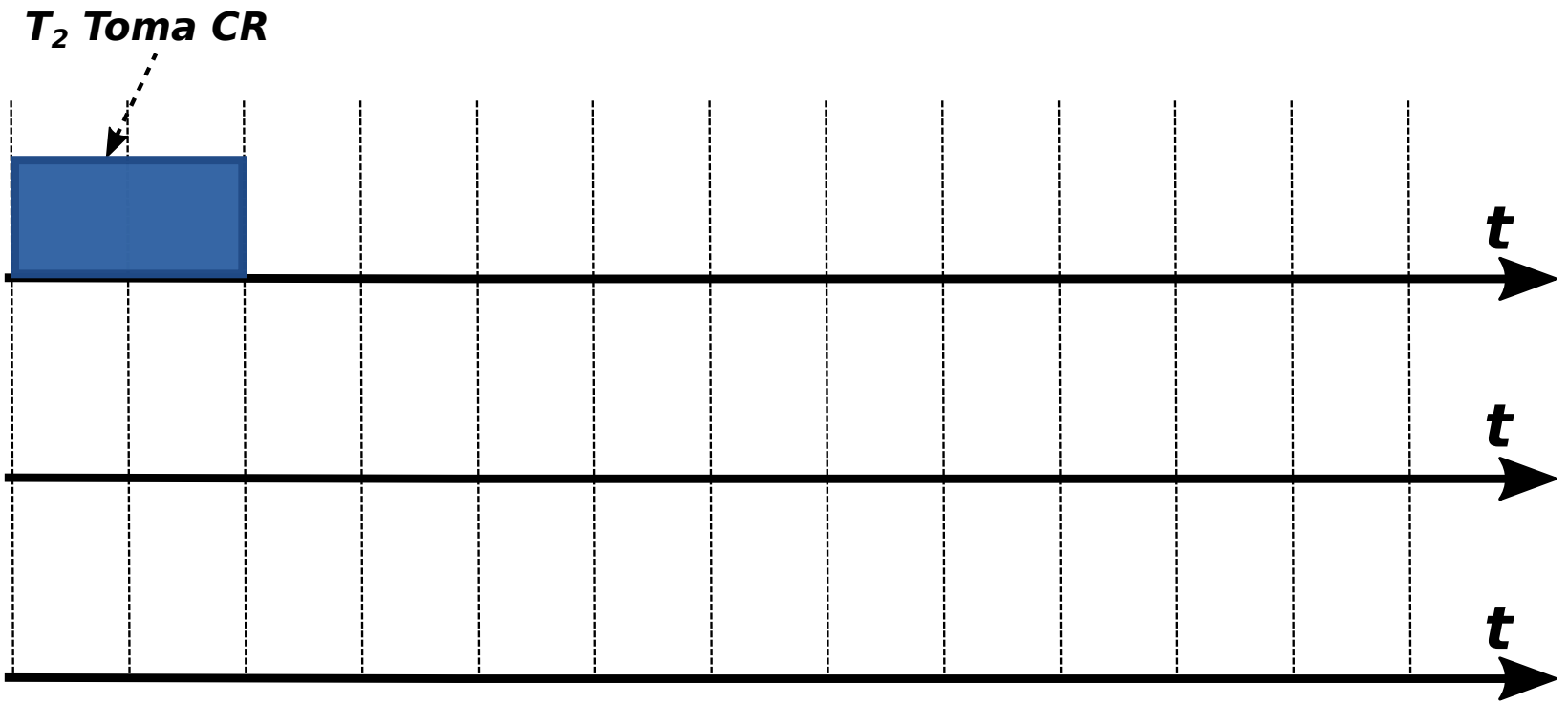
¿Que es una Inversión de Prioridades?



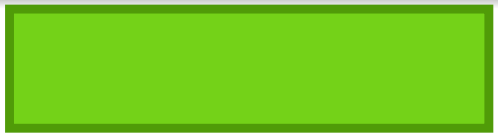
$T_0 : C_0=4, T_0=6, P_0 = Alta$

$T_1 : C_1=3, T_1=8, P_1 = Media$

$T_2 : C_2=4, T_2=12, P_2 = Baja$



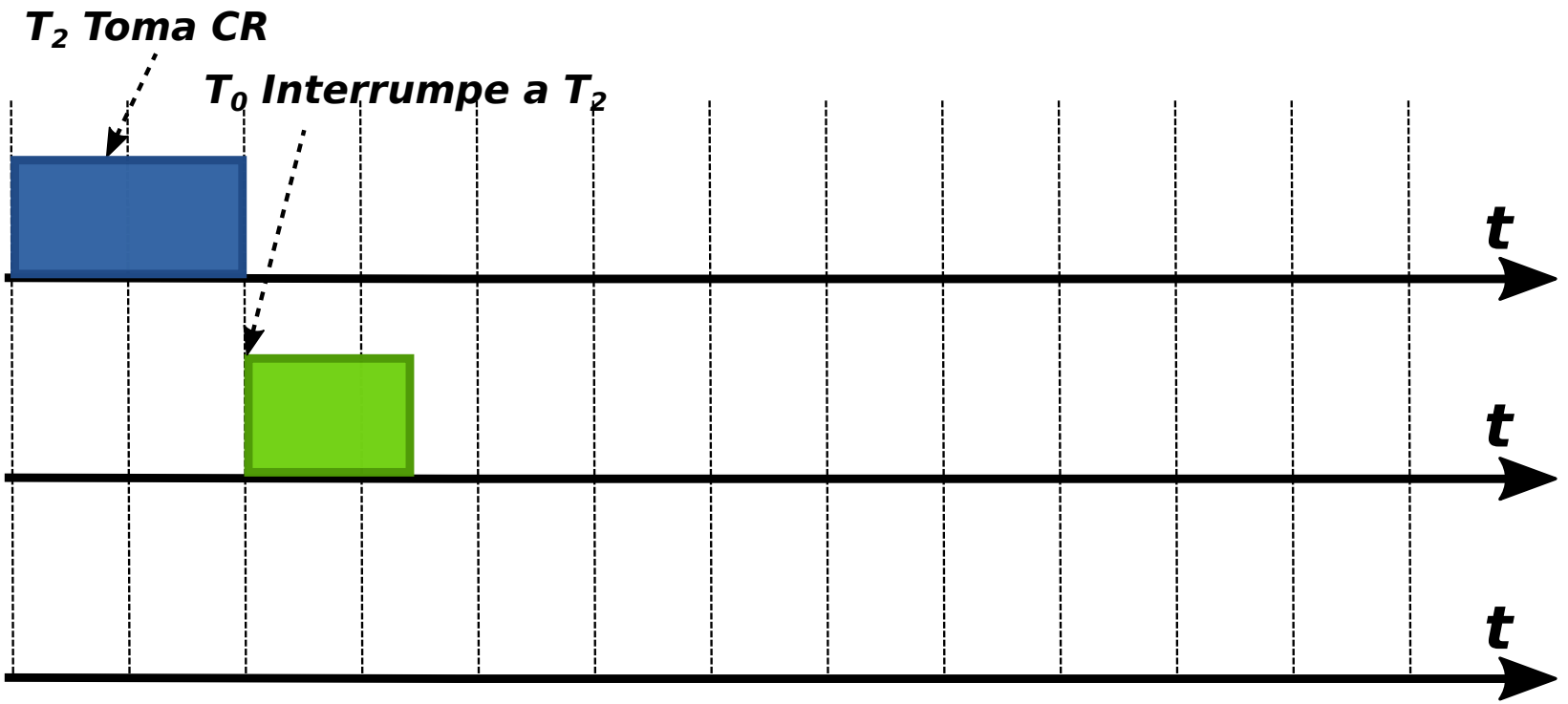
¿Que es una Inversión de Prioridades?



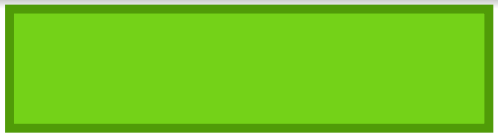
$T_0 : C_0=4, T_0=6, P_0 = Alta$

$T_1 : C_1=3, T_1=8, P_1 = Media$

$T_2 : C_2=4, T_2=12, P_2 = Baja$



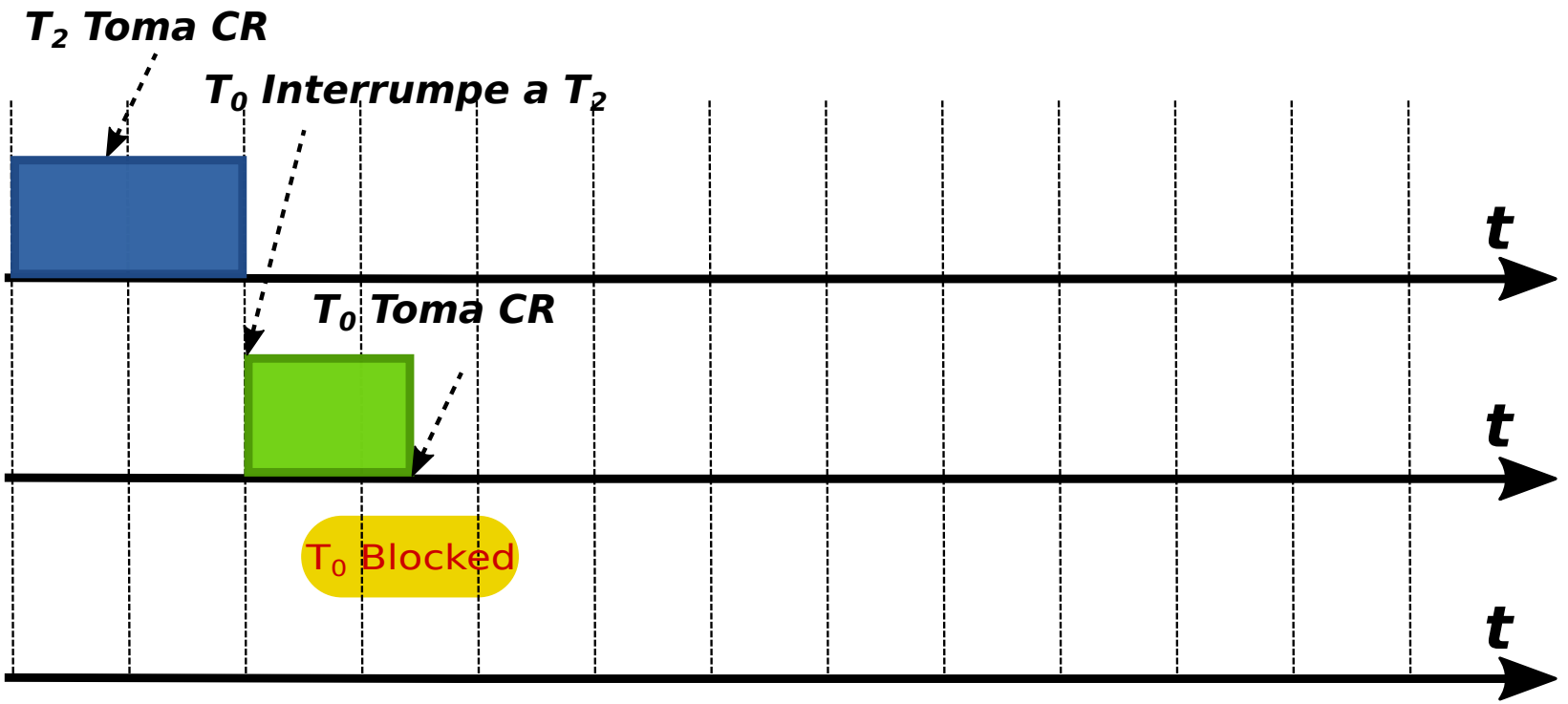
¿Que es una Inversión de Prioridades?



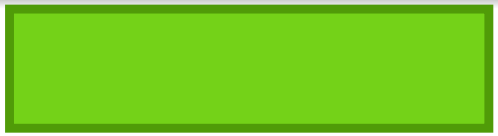
$T_0 : C_0=4, T_0=6, P_0 = Alta$

$T_1 : C_1=3, T_1=8, P_1 = Media$

$T_2 : C_2=4, T_2=12, P_2 = Baja$



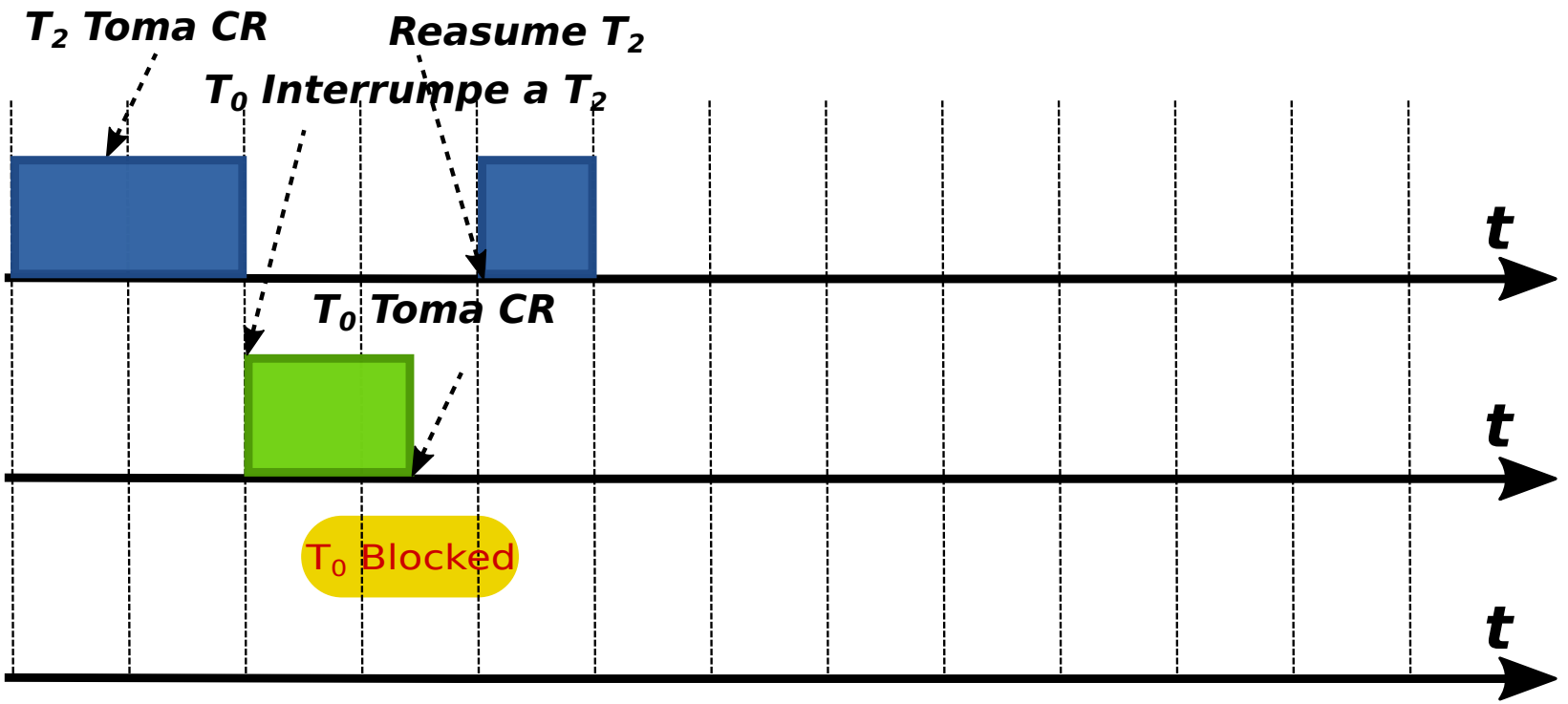
¿Que es una Inversión de Prioridades?



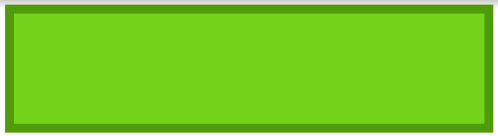
$T_0 : C_0=4, T_0=6, P_0 = Alta$

$T_1 : C_1=3, T_1=8, P_1 = Media$

$T_2 : C_2=4, T_2=12, P_2 = Baja$



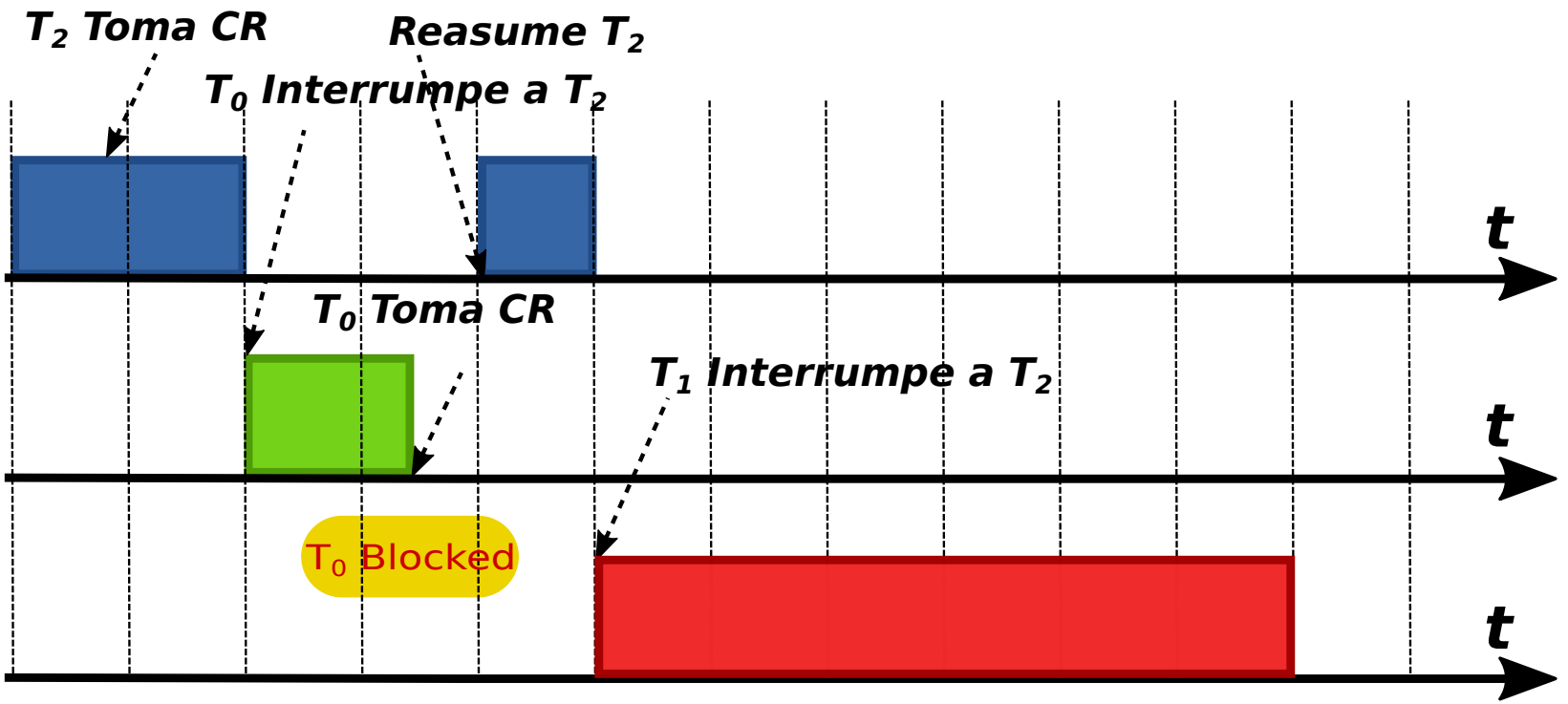
¿Que es una Inversión de Prioridades?



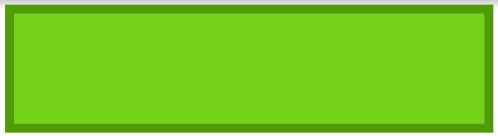
$T_0 : C_0=4, T_0=6, P_0 = Alta$

$T_1 : C_1=3, T_1=8, P_1 = Media$

$T_2 : C_2=4, T_2=12, P_2 = Baja$



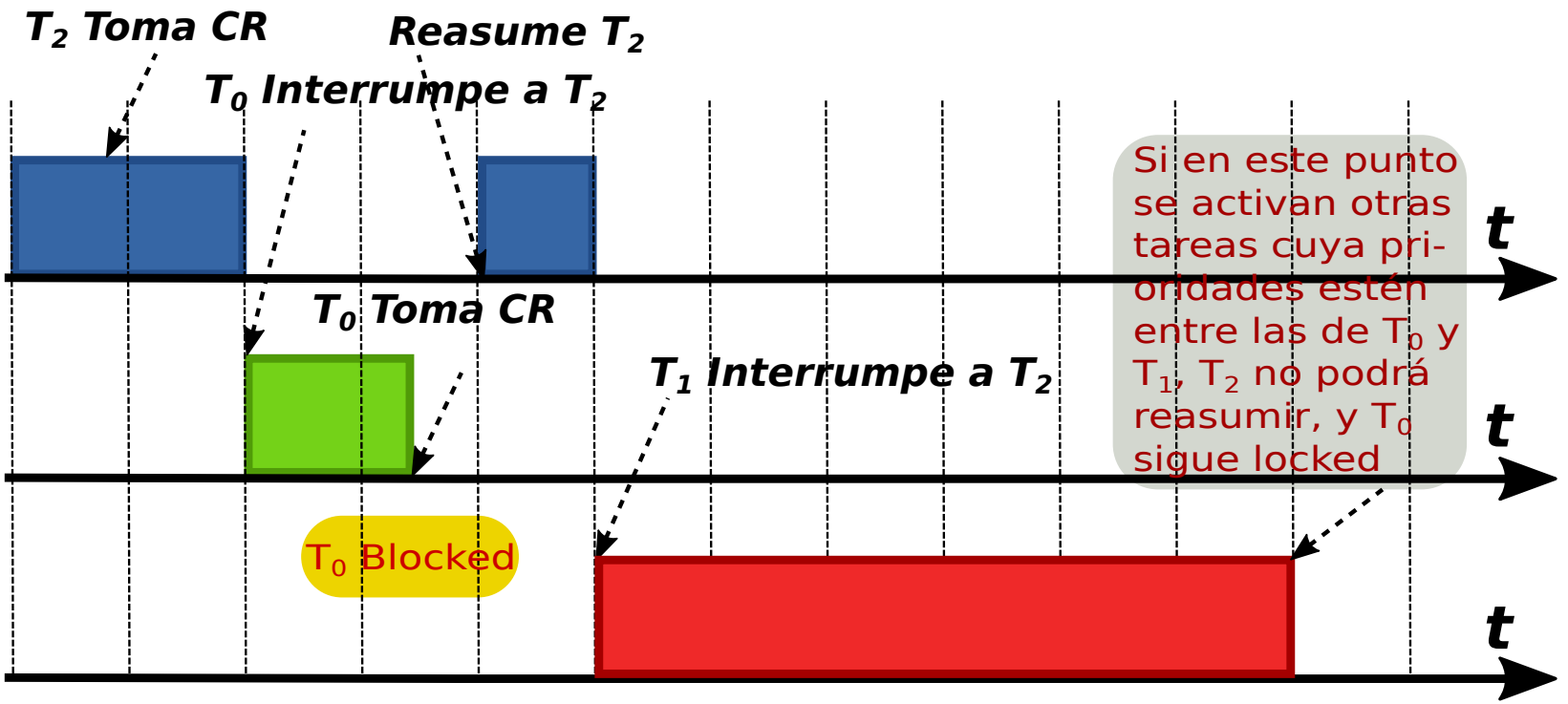
¿Que es una Inversión de Prioridades?



$T_0 : C_0=4, T_0=6, P_0 = Alta$

$T_1 : C_1=3, T_1=8, P_1 = Media$

$T_2 : C_2=4, T_2=12, P_2 = Baja$



Inversión de Prioridades y Deadlines

- El escenario descripto se conoce como ***unbounded priority inversion***, ya que no puede determinarse cuanto tardará T_2 en volver a ganar el control como para desbloquear la **CR** y que T_0.

Inversión de Prioridades y Deadlines

- El escenario descripto se conoce como ***unbounded priority inversion***, ya que no puede determinarse cuanto tardará T_2 en volver a ganar el control como para desbloquear la **CR** y que T_0.
- En el caso de un sistema derivado de EDF, se habla de ***deadline inversion***.

Inversión de Prioridades y Deadlines

- El escenario descripto se conoce como ***unbounded priority inversion***, ya que no puede determinarse cuanto tardará T_2 en volver a ganar el control como para desbloquear la **CR** y que T_0.
- En el caso de un sistema derivado de EDF, se habla de ***deadline inversion***.
- En el kernel de un GPOS éste escenario se puede producir y ni siquiera notarse ni generarse ninguna falla.

Inversión de Prioridades y Deadlines

- El escenario descripto se conoce como ***unbounded priority inversion***, ya que no puede determinarse cuanto tardará T_2 en volver a ganar el control como para desbloquear la **CR** y que T_0.
- En el caso de un sistema derivado de EDF, se habla de ***deadline inversion***.
- En el kernel de un GPOS éste escenario se puede producir y ni siquiera notarse ni generarse ninguna falla.
- En un kernel de un RTOS una inversión de prioridad en una tarea de alta prioridad implica que ésta no cumpla un deadline y es muy probable que se registre una falla en el funcionamiento.

Inversión de Prioridades y Deadlines

- El escenario descripto se conoce como ***unbounded priority inversion***, ya que no puede determinarse cuanto tardará T_2 en volver a ganar el control como para desbloquear la **CR** y que T_0.
- En el caso de un sistema derivado de EDF, se habla de ***deadline inversion***.
- En el kernel de un GPOS éste escenario se puede producir y ni siquiera notarse ni generarse ninguna falla.
- En un kernel de un RTOS una inversión de prioridad en una tarea de alta prioridad implica que ésta no cumpla un deadline y es muy probable que se registre una falla en el funcionamiento.
- En ocasiones puede ser mas o menos crítica. Sino lean "What really happened on Mars?". Imaginen el Mars Pathfinder colgado por un Priority Inversion ... en Marte!!

Prevención de Priority Inversion

Prevención de Priority Inversion

- Impracticables

Prevención de Priority Inversion

- Impracticables
- Impedir que las tareas bloqueen (Modelos RMS o EDF estrictos)

Prevención de Priority Inversion

- Impracticables
- Impedir que las tareas bloqueen (Modelos RMS o EDF estrictos)
- Asignar a todas las tareas la misma prioridad

Prevención de Priority Inversion

- Impracticables
- Impedir que las tareas bloqueen (Modelos RMS o EDF estrictos)
- Asignar a todas las tareas la misma prioridad
- Soluciones Factibles

Prevención de Priority Inversion

- Impracticables
- ⊘ Impedir que las tareas bloqueen (Modelos RMS o EDF estrictos)
- ⊘ Asignar a todas las tareas la misma prioridad
- Soluciones Factibles
- ✓ Prioridad Techo

Prevención de Priority Inversion

- Impracticables
- ⊘ Impedir que las tareas bloqueen (Modelos RMS o EDF estrictos)
- ⊘ Asignar a todas las tareas la misma prioridad
- Soluciones Factibles
- ✓ Prioridad Techo
- ✓ Herencia de Prioridades

Prioridad Techo

Consiste en asignar a cada **CR** una prioridad mayor que la mas alta prioridad de todas las tareas. Cada tarea que toma el **CR** gana la prioridad del **CR** de modo que no puede ser interrumpida hasta que termine la ejecución del **CR**. Una vez que libera el **CR** recupera su prioridad original.
Es muy simple de implementar.
Sin embargo una tarea de baja prioridad mientras ejecuta una **CR** puede evitar que una tarea de alta prioridad se ejecute ya que no tiene mas prioridad que la **CR**, aunque esa tarea de mayor prioridad que la que tomó la **CR** No necesite ejecutar esa **CR**.

Herencia de Prioridad

Consiste en mantener la prioridad de una tarea que adquiere una **CR** sin modificación hasta que otra tarea de mayor prioridad bloquea en esa **CR**. En ese momento la prioridad de la tarea que ejecuta la **CR** se iguala a la de la que está bloqueada. Si la que bloqueó es de menor prioridad la tarea que ejecuta la **CR** no se modifica. Cuando la tarea libera la **CR** recupera su prioridad original, si esta había cambiado.

Su implementación tiene cierto overhead respecto de la anterior, pero es mas flexible. Mientras ejecuta la **CR** la prioridad de la tarea puede cambiar dinámicamente.

Temario

Free-RTOS

Free-RTOS

- Es un kernel open source con las prestaciones mínimas necesarias, orientado a sistemas basados en Microcontroladores pequeños.

Free-RTOS

- Es un kernel open source con las prestaciones mínimas necesarias, orientado a sistemas basados en Microcontroladores pequeños.
- **Tareas**: Unidades de ejecución, basadas en una estructura de control llamado **T**ask **C**ontrol **B**lock (**TCB**). Servicios para crear, detener, reasumir, cambar prioridad, y borrar tareas. Usa una variante con menos necesidad de stack llamada co-rutina.

Free-RTOS

- Es un kernel open source con las prestaciones mínimas necesarias, orientado a sistemas basados en Microcontroladores pequeños.
- **Tareas**: Unidades de ejecución, basadas en una estructura de control llamado **T**ask **C**ontrol **B**lock (**TCB**). Servicios para crear, detener, reasumir, cambar prioridad, y borrar tareas. Usa una variante con menos necesidad de stack llamada co-rutina.
- **Scheduling**: Preemptive en base a prioridad. Para tareas de igual prioridad usa round robin. No soporta task switch cuando procesa interrupciones anidadas. Puede deshabilitar el scheduler si se ingresa en una **CR** larga para evitar task switch en ese contexto.

Free-RTOS

- Es un kernel open source con las prestaciones mínimas necesarias, orientado a sistemas basados en Microcontroladores pequeños.
- **Tareas**: Unidades de ejecución, basadas en una estructura de control llamado **T**ask **C**ontrol **B**lock (**TCB**). Servicios para crear, detener, reasumir, cambar prioridad, y borrar tareas. Usa una variante con menos necesidad de stack llamada co-rutina.
- **Scheduling**: Preemptive en base a prioridad. Para tareas de igual prioridad usa round robin. No soporta task switch cuando procesa interrupciones anidadas. Puede deshabilitar el scheduler si se ingresa en una **CR** larga para evitar task switch en ese contexto.
- **Sincronización de Tareas**: Fundamentalmente se maneja a través de colas. También utiliza semáforos incrementales y binarios, así como mutexes recursivos con herencia de prioridad para proteger **CR**. Los mutexes recursivos tienen 256 niveles de profundidad para que el owner los tome/libere.

Free-RTOS

- **Protección de Memoria**: No tiene en forma nativa. Para algunos microcontroladores de ARM que empezaron a incluir MPU (Cortex-M3 en adelante), hay una versión FreeRTOS-MPU.

Free-RTOS

- **Protección de Memoria**: No tiene en forma nativa. Para algunos microcontroladores de ARM que empezaron a incluir MPU (Cortex-M3 en adelante), hay una versión FreeRTOS-MPU.
- **Servicios de Temporización**: El kernel de FReeRTOS soporta timer tick y temporizadores de software.

Free-RTOS

- **Protección de Memoria**: No tiene en forma nativa. Para algunos microcontroladores de ARM que empezaron a incluir MPU (Cortex-M3 en adelante), hay una versión FreeRTOS-MPU.
- **Servicios de Temporización**: El kernel de FReeRTOS soporta timer tick y temporizadores de software.
- **Extensiones**: En forma de Bibliotecas de código, le permiten soportar TCP/IP, por ejemplo.

Free-RTOS

- **Protección de Memoria**: No tiene en forma nativa. Para algunos microcontroladores de ARM que empezaron a incluir MPU (Cortex-M3 en adelante), hay una versión FreeRTOS-MPU.
- **Servicios de Temporización**: El kernel de FReeRTOS soporta timer tick y temporizadores de software.
- **Extensiones**: En forma de Bibliotecas de código, le permiten soportar TCP/IP, por ejemplo.
- **Portabilidad**: Es portable. Es un kernel muy sencillo (pocos archivos fuente).

Free-RTOS

- **Protección de Memoria**: No tiene en forma nativa. Para algunos microcontroladores de ARM que empezaron a incluir MPU (Cortex-M3 en adelante), hay una versión FreeRTOS-MPU.
- **Servicios de Temporización**: El kernel de FReeRTOS soporta timer tick y temporizadores de software.
- **Extensiones**: En forma de Bibliotecas de código, le permiten soportar TCP/IP, por ejemplo.
- **Portabilidad**: Es portable. Es un kernel muy sencillo (pocos archivos fuente).
- **Estandarización**: No se apega a ningún standard. Desde hace pocos años existe una versión FreeRTOS+POSIX que solo implementa un subset del API de POSIX-threads.

MicroC/OS (μC/OS)

MicroC/OS (μC/OS)

- **Generalidades**: Kernel Full Preemptive. Amplio rango de CPUs: Microcontroladores, Microprocesadores, y DSP's. Versión actual μC/OS-III. Mantenido por Micrium.

MicroC/OS (μC/OS)

- **Generalidades**: Kernel Full Preemptive. Amplio rango de CPUs: Microcontroladores, Microprocesadores, y DSP's. Versión actual μC/OS-III. Mantenido por Micrium.
- **Tareas**: Son esencialmente threads que ejecutan en el mismo espacio de ejecución del kernel. Soporte para crear, suspender, reasumir, y borrar tareas. API para obtener estadísticas de tareas. μC/OS-II limita la máxima cantidad de tareas a 256. μC/OS-III a la memoria disponible.

MicroC/OS (μC/OS)

- **Generalidades**: Kernel Full Preemptive. Amplio rango de CPUs: Microcontroladores, Microprocesadores, y DSP's. Versión actual μC/OS-III. Mantenido por Micrium.
- **Tareas**: Son esencialmente threads que ejecutan en el mismo espacio de ejecución del kernel. Soporte para crear, suspender, reasumir, y borrar tareas. API para obtener estadísticas de tareas. μC/OS-II limita la máxima cantidad de tareas a 256. μC/OS-III a la memoria disponible.
- **Scheduling**: Preemptive en base a prioridad estática. Asignadas por el usuario. Todo indica que utiliza un esquema RMS. μC/OS-II no acepta tareas con igual prioriad. μC/OS-III sí, e implementa round-robin entre tareas de nivel de prioridad equivalente.

MicroC/OS (μC/OS)

- **Generalidades**: Kernel Full Preemptive. Amplio rango de CPUs: Microcontroladores, Microprocesadores, y DSP's. Versión actual μC/OS-III. Mantenido por Micrium.
- **Tareas**: Son esencialmente threads que ejecutan en el mismo espacio de ejecución del kernel. Soporte para crear, suspender, reasumir, y borrar tareas. API para obtener estadísticas de tareas. μC/OS-II limita la máxima cantidad de tareas a 256. μC/OS-III a la memoria disponible.
- **Scheduling**: Preemptive en base a prioridad estática. Asignadas por el usuario. Todo indica que utiliza un esquema RMS. μC/OS-II no acepta tareas con igual prioriad. μC/OS-III sí, e implementa round-robin entre tareas de nivel de prioridad equivalente.
- **Gestión de Memoria**: No proporciona protección de memoria. Permite al usuario definir bloques de memoria de tamaño fijo que serán asignados a las tareas.

MicroC/OS (μC/OS)

- **Sincronización**: habilitar/deshabilitar interrupciones para proteger **CR** cortas, y habilitar/deshabilitar el scheduler para proteger **CR** largas. Además se agregan Event Control Blocks (ECB's), semáforos, colas de mensajes, y mailboxes. μC/OS-III agrega mutexes con herencia de prioridades para evitar Inversión de Prioridad.

MicroC/OS (μC/OS)

- **Sincronización**: habilitar/deshabilitar interrupciones para proteger **CR** cortas, y habilitar/deshabilitar el scheduler para proteger **CR** largas. Además se agregan Event Control Blocks (ECB's), semáforos, colas de mensajes, y mailboxes. μC/OS-III agrega mutexes con herencia de prioridades para evitar Inversión de Prioridad.
- **Interrupciones**: Soporta anidamiento de interrupciones. Las interrupciones se resuelven en el handler. El scheduler se deshabilita durante el procesamiento anidado de interrupciones, y una vez finalizado el procesamiento de una o mas interrupciones anidadas se reasume la conmutación de tareas.

MicroC/OS (μC/OS)

- **Sincronización**: habilitar/deshabilitar interrupciones para proteger **CR** cortas, y habilitar/deshabilitar el scheduler para proteger **CR** largas. Además se agregan Event Control Blocks (ECB's), semáforos, colas de mensajes, y mailboxes. μC/OS-III agrega mutexes con herencia de prioridades para evitar Inversión de Prioridad.
- **Interrupciones**: Soporta anidamiento de interrupciones. Las interrupciones se resuelven en el handler. El scheduler se deshabilita durante el procesamiento anidado de interrupciones, y una vez finalizado el procesamiento de una o mas interrupciones anidadas se reasume la conmutación de tareas.
- **Temporizadores**: Base de tiempo (timer tick), para implementar timeouts que permiten a una tarea suspender su ejecución mediante un servicio del kernel, y delays que permiten reasumir una tarea luego de transcurrida una cantidad de ticks. Una tarea puede ser suspendida antes de su timeout por preemption.

MicroC/OS (μC/OS)

- **Estandarización**: La versión x86 utiliza una nomenclatura de nombres de archivos, y un entorno de desarrollo compatible con Microsoft IDE.

MicroC/OS (μC/OS)

- **Estandarización**: La versión x86 utiliza una nomenclatura de nombres de archivos, y un entorno de desarrollo compatible con Microsoft IDE.
- **Portabilidad**: Está escrito mayormente en ANSI-C. Hay versiones para NEOS-II de **Altera**, Analog Devices (Blackfin, y ADSP-CM4xxx), **ARM** (ARM7, ARM9, ARM 11, Cortex A8, A9, A15, A53, A57, R4, R5, R7, M0, M1, M3, M4, M4(F), y M7), **Cypress** (PSoC 4, y PSoC 5 (Cortex-M)), **Infineon** XMC, **Microchip** (AVR, AVR 32, eSi-RISC, PIC24, PIC32, SAM 32 bit, SmartFusion2), **NXP** (ColdFire, HCS12, i.MX, Kinetis, LPC, Power Architecture, Vfxxx), **RISC-V**, **Renesas** (RL78, RX, RZ, R-IN32, SuperH-2A, V850E/2/S), Geckos de **Silicon Labs**, **ST Microelectronics** (STM32F, STM32H, STM32L, STR9), **Texas Instruments** (C28x, MSP430, MSP432, Hercules RM, Hercules TMS570), y **Xilinx** (MicroBlaze, Zynq-7000 (Cortex-A), Zynq Ultrascale+ MPSoC (Cortex-A y R))

NuttX

NuttX

- **Generalidades**: Su principal énfasis es la estandarización. Es POSIX 1003 y ANSI compatible. Además se adapta a otros estándares menores de RTOS. Ejecuta en un amplio rango de CPUs .

NuttX

- **Generalidades**: Su principal énfasis es la estandarización. Es POSIX 1003 y ANSI compatible. Además se adapta a otros estándares menores de RTOS. Ejecuta en un amplio rango de CPUs .
- **Tareas**: El kernel utiliza un conjunto de funciones para crear y gestionar tareas. Implementa además todo el esquema de POSIX, que establece la relación de parentesco (parent-child) entre procesos, y el API de creación de procesos de POSIX. Los procesos transfieren sus recursos a los child cuando los crean, y deben esperar su finalización.

NuttX

- **Generalidades**: Su principal énfasis es la estandarización. Es POSIX 1003 y ANSI compatible. Además se adapta a otros estándares menores de RTOS. Ejecuta en un amplio rango de CPUs .
- **Tareas**: El kernel utiliza un conjunto de funciones para crear y gestionar tareas. Implementa además todo el esquema de POSIX, que establece la relación de parentesco (parent-child) entre procesos, y el API de creación de procesos de POSIX. Los procesos transfieren sus recursos a los child cuando los crean, y deben esperar su finalización.
- **Scheduling**: Es un kernel preemptive basado en prioridades. Cada tarea puede configurar su política de scheduling ya sea en una cola FIFO o round-robin dentro de un time slice prescripto. Además las tareas pueden cambiar de prioridad y ceder la CPU a otra tarea de la misma prioridad.

NuttX

- **Sincronización**: Utiliza semáforos incrementales con herencia de prioridad, disponible como opción de configuración. Maneja colas de mensaje POSIX con nombre, que pueden ser utilizadas incluso desde los handlers de interrupción. Los mensajes pueden incluir un timeout para evitar bloqueos permanentes. El timeout se implementa por señales POSIX idénticas a las de UNIX.

NuttX

- **Sincronización**: Utiliza semáforos incrementales con herencia de prioridad, disponible como opción de configuración. Maneja colas de mensaje POSIX con nombre, que pueden ser utilizadas incluso desde los handlers de interrupción. Los mensajes pueden incluir un timeout para evitar bloqueos permanentes. El timeout se implementa por señales POSIX idénticas a las de UNIX.
- **Señales**: Maneja el clásico mecanismo de UNIX en el que la señal se comporta como una interrupción de software. La única diferencia es que no existen handlers predefinidos para las señales.

NuttX

- **Sincronización**: Utiliza semáforos incrementales con herencia de prioridad, disponible como opción de configuración. Maneja colas de mensaje POSIX con nombre, que pueden ser utilizadas incluso desde los handlers de interrupción. Los mensajes pueden incluir un timeout para evitar bloqueos permanentes. El timeout se implementa por señales POSIX idénticas a las de UNIX.
- **Señales**: Maneja el clásico mecanismo de UNIX en el que la señal se comporta como una interrupción de software. La única diferencia es que no existen handlers predefinidos para las señales.
- **Servicios de Reloj y Timer**: Soporta las funciones de temporización y temporización de intervalos de POSIX. Cada tarea puede crear un reloj propio que le genere ticks. Cuando expira un timeout, la tarea recibe una señal timeout. En el handler de la señal se debe definir el tratamiento del timeout.

NuttX

- **File System y Redes**: NuttX puede ejecutar sin un File System. En caso de tener un Fie system habilitado arranca con un pseudo root file system mapeado en memoria. Sobre éste se montarán los demás File Systems “reales ”. El API de File System es un subset del de POSIX: open, close, read, write, etc. Implementa un susbset del API de sockets.

NuttX

- **File System y Redes**: NuttX puede ejecutar sin un File System. En caso de tener un Fie system habilitado arranca con un pseudo root file system mapeado en memoria. Sobre éste se montarán los demás File Systems “reales ”. El API de File System es un subset del de POSIX: open, close, read, write, etc. Implementa un susbset del API de sockets.
- **Portabilidad**: Corre en múltiples plataformas: ARM, AVR, hc, Mips, misoc, or1k, renesas, RISC-V, Sim, Intel x86, Intel x86_64, xtensa, z16, y hasta en el z80.

NuttX

- **File System y Redes**: NuttX puede ejecutar sin un File System. En caso de tener un Fie system habilitado arranca con un pseudo root file system mapeado en memoria. Sobre éste se montarán los demás File Systems “reales ”. El API de File System es un subset del de POSIX: open, close, read, write, etc. Implementa un susbset del API de sockets.
- **Portabilidad**: Corre en múltiples plataformas: ARM, AVR, hc, Mips, misoc, or1k, renesas, RISC-V, Sim, Intel x86, Intel x86_64, xtensa, z16, y hasta en el z80.
- **Estandarización**: Respeta POSIX en buena medida (en algunas funciones cumple solo un subset). Open source y bien documentado. Recomiendo visitar su *wiki de documentación*. Sus fuentes están disponibles en *github*

QNX

QNX

- **Generalidades**: Arrancó como un RTOS propietario destinado al mercado embedded. Evoluciona a un RTOS POSIX con arquitectura MicroKernel como característica distintiva. Desde entonces la implementación se llama Neutrino.

QNX

- **Generalidades**: Arrancó como un RTOS propietario destinado al mercado embedded. Evoluciona a un RTOS POSIX con arquitectura MicroKernel como característica distintiva. Desde entonces la implementación se llama Neutrino.
- **Microkernel**: Contiene el scheduler, intercomunicación de procesos, Redirección de Interrupciones, y Temporizadores. El resto de los servicios del Sistema Operativo, ejecutan en la forma de procesos, en espacio de kernel y con privilegios de kernel. Pero procesos.

QNX

- **Generalidades**: Arrancó como un RTOS propietario destinado al mercado embedded. Evoluciona a un RTOS POSIX con arquitectura MicroKernel como característica distintiva. Desde entonces la implementación se llama Neutrino.
- **Microkernel**: Contiene el scheduler, intercomunicación de procesos, Redirección de Interrupciones, y Temporizadores. El resto de los servicios del Sistema Operativo, ejecutan en la forma de procesos, en espacio de kernel y con privilegios de kernel. Pero procesos.
- **Threads**: Es la mínima unidad de ejecución en QNX. Cada procesos se compone de threads que ejecutan en el mismo espacio del proceso. Soporta el API pthreads del standard POSIX.

QNX

- **Generalidades**: Arrancó como un RTOS propietario destinado al mercado embedded. Evoluciona a un RTOS POSIX con arquitectura MicroKernel como característica distintiva. Desde entonces la implementación se llama Neutrino.
- **Microkernel**: Contiene el scheduler, intercomunicación de procesos, Redirección de Interrupciones, y Temporizadores. El resto de los servicios del Sistema Operativo, ejecutan en la forma de procesos, en espacio de kernel y con privilegios de kernel. Pero procesos.
- **Threads**: Es la mínima unidad de ejecución en QNX. Cada procesos se compone de threads que ejecutan en el mismo espacio del proceso. Soporta el API pthreads del standard POSIX.
- **Scheduling**: Esquema preemptive por prioridad. Además soporta **APS** (**A**daptive **P**artition **S**cheduling) que le permite garantizar porcentajes mínimos de CPU para grupos seleccionados de subprocesos, a pesar de que otros pueden tener mayor prioridad.

QNX

- **Procesos**: Cada uno ejecuta en su propio espacio de memoria, lo que lo hace sumamente robusto desde el punto de vista de la seguridad.

QNX

- **Procesos**: Cada uno ejecuta en su propio espacio de memoria, lo que lo hace sumamente robusto desde el punto de vista de la seguridad.
- **Comunicación entre procesos**: Se realiza a través del Microkernel. La implementación considera la problemática de restricciones temporales de los RTOS. Cuando un proceso envía un mensaje a otro, el mismo se transfiere desde el espacio de memoria del emisor al del receptor en una sola operación. *Es decir, que el receptor no necesita bloquear para esperar el mensaje*.
  Así resuelve la mensajería de manera sumamente ágil y sin afectar los requerimientos temporales de las tareas. Una de las claves en el port a la versión POSIX del microkernel (Neutrino).
  Operaciones de E/S, Networking, y Acceso a archivos, se manejan con el microkernel a través de mensajes. Los mensajes se priorizan de acuerdo con la prioridad de los threads receptores. E/S se lleva la prioridad máxima.

Real Time Linux

- Linux no fue diseñado originalmente como un RTOS, sino como un GPOS.

Real Time Linux

- Linux no fue diseñado originalmente como un RTOS, sino como un GPOS.
- No obstante su diseño es eficiente y muy adecuado para resolver requerimientos temporales del orden del miliseg. sin inconvenientes apreciables, de modo que entra sin problemas en el mundo embeded.

Real Time Linux

- Linux no fue diseñado originalmente como un RTOS, sino como un GPOS.
- No obstante su diseño es eficiente y muy adecuado para resolver requerimientos temporales del orden del miliseg. sin inconvenientes apreciables, de modo que entra sin problemas en el mundo embeded.
- Por debajo de ese límite temporal no siempre es adecuado en virtud de no ser un kernel full preemptive.

Real Time Linux

- Linux no fue diseñado originalmente como un RTOS, sino como un GPOS.
- No obstante su diseño es eficiente y muy adecuado para resolver requerimientos temporales del orden del miliseg. sin inconvenientes apreciables, de modo que entra sin problemas en el mundo embeded.
- Por debajo de ese límite temporal no siempre es adecuado en virtud de no ser un kernel full preemptive.
- En el kernel de linux que utilizamos en nuestra computadora puede interrupir (preempt) una tarea bajo las siguientes condiciones:

Real Time Linux

- Linux no fue diseñado originalmente como un RTOS, sino como un GPOS.
- No obstante su diseño es eficiente y muy adecuado para resolver requerimientos temporales del orden del miliseg. sin inconvenientes apreciables, de modo que entra sin problemas en el mundo embeded.
- Por debajo de ese límite temporal no siempre es adecuado en virtud de no ser un kernel full preemptive.
- En el kernel de linux que utilizamos en nuestra computadora puede interrupir (preempt) una tarea bajo las siguientes condiciones:
  - Cuando la tarea ejecuta en User Mode.

Real Time Linux

- Linux no fue diseñado originalmente como un RTOS, sino como un GPOS.
- No obstante su diseño es eficiente y muy adecuado para resolver requerimientos temporales del orden del miliseg. sin inconvenientes apreciables, de modo que entra sin problemas en el mundo embeded.
- Por debajo de ese límite temporal no siempre es adecuado en virtud de no ser un kernel full preemptive.
- En el kernel de linux que utilizamos en nuestra computadora puede interrupir (preempt) una tarea bajo las siguientes condiciones:
  - Cuando la tarea ejecuta en User Mode.
  - Cuando una tarea está retornando a User Mode ya sea desde una Syscall o desde una Interrupción.

Real Time Linux

- Linux no fue diseñado originalmente como un RTOS, sino como un GPOS.
- No obstante su diseño es eficiente y muy adecuado para resolver requerimientos temporales del orden del miliseg. sin inconvenientes apreciables, de modo que entra sin problemas en el mundo embeded.
- Por debajo de ese límite temporal no siempre es adecuado en virtud de no ser un kernel full preemptive.
- En el kernel de linux que utilizamos en nuestra computadora puede interrupir (preempt) una tarea bajo las siguientes condiciones:
  - Cuando la tarea ejecuta en User Mode.
  - Cuando una tarea está retornando a User Mode ya sea desde una Syscall o desde una Interrupción.
  - Cuando una tarea bloquea en Kernel Mode y cede el control deliberadamente a otra tarea.

Real Time Linux

La limitación de Linux es cuando la tarea a ser interrumpida ejecuta en modo kernel. El Kernel no es preemptive. Por lo tanto si en ese estado de ejecución se produce una interrupción en respuesta a un evento que requiere reactivar una tarea de mayor prioridad de la que está ejecutando, hasta que la tarea de menor prioridad no retorne a modo usuario la de mayor prioridad quedará demorada. Esto podría incumplir su condición de temporización.

Real Time Linux

A partir de la versión 2.6.22 el kernel de Linux introduce lo que se conoce como Patch Real Time. Consiste en opciones de configuración que se aplican durante la compilación de la imagen del kernel.

Real Time Linux

A partir de la versión 2.6.22 el kernel de Linux introduce lo que se conoce como Patch Real Time. Consiste en opciones de configuración que se aplican durante la compilación de la imagen del kernel.

- **`CONFIG_PREEMPT_VOLUNTARY`**: Introduce en el kernel el chequeo de las principales causas de “long latencies”cuando una tarea ejecuta en Kernel Mode. Si las detecta, el propio kernel cede el control voluntariamente a una tarea de mayor prioridad que demanda ejecución. La ventaja es que es simple de implementar y no tiene efecto notorio en el rendimiento global del sistema (throughput).

Real Time Linux

A partir de la versión 2.6.22 el kernel de Linux introduce lo que se conoce como Patch Real Time. Consiste en opciones de configuración que se aplican durante la compilación de la imagen del kernel.

- **`CONFIG_PREEMPT_VOLUNTARY`**: Introduce en el kernel el chequeo de las principales causas de “long latencies”cuando una tarea ejecuta en Kernel Mode. Si las detecta, el propio kernel cede el control voluntariamente a una tarea de mayor prioridad que demanda ejecución. La ventaja es que es simple de implementar y no tiene efecto notorio en el rendimiento global del sistema (throughput).
- **`CONFIG_PREEMPT`**: Todas las regiones del kernel se tratan como preemptive kernel, excepto **CR** protegidas por spinlocks, y los handlers de interrupción. Con esta opción, el peor caso de latency se ubica en el orden del miliseg.

Real Time Linux

A partir de la versión 2.6.22 el kernel de Linux introduce lo que se conoce como Patch Real Time. Consiste en opciones de configuración que se aplican durante la compilación de la imagen del kernel.

- **CONFIG_PREEMPT_VOLUNTARY**: Introduce en el kernel el chequeo de las principales causas de “long latencies”cuando una tarea ejecuta en Kernel Mode. Si las detecta, el propio kernel cede el control voluntariamente a una tarea de mayor prioridad que demanda ejecución. La ventaja es que es simple de implementar y no tiene efecto notorio en el rendimiento global del sistema (throughput).
- **CONFIG_PREEMPT**: Todas las regiones del kernel se tratan como preemptive kernel, excepto **CR** protegidas por spinlocks, y los handlers de interrupción. Con esta opción, el peor caso de latency se ubica en el orden del miliseg.
- **CONFIG_PREEMPT_RT**: El kernel es full preemptive. Se tienen resoluciones por debajo del miliseg.

Real Time Linux - Full Preemptive

La opción **`CONFIG_PREEMPT_RT`** genera cambios profundos en la imagen del kernel, cambiando bloques completos de código para adaptarlos a los requerimiento de un kernel RT.

Real Time Linux - Full Preemptive

La opción **`CONFIG_PREEMPT_RT`** genera cambios profundos en la imagen del kernel, cambiando bloques completos de código para adaptarlos a los requerimiento de un kernel RT.

- Implementa preemption en todas las primitivas de locking (*spinlocks*), re implementándolos mediante mutexes con herencia de prioridad.

Real Time Linux - Full Preemptive

La opción **`CONFIG_PREEMPT_RT`** genera cambios profundos en la imagen del kernel, cambiando bloques completos de código para adaptarlos a los requerimiento de un kernel RT.

- Implementa preemption en todas las primitivas de locking (*spinlocks*), re implementándolos mediante mutexes con herencia de prioridad.
- El proceso owner del mutex toma la mayor prioridad de los procesos encolados en el mutex. Los llaman rt-mutexes.

Real Time Linux - Full Preemptive

La opción **`CONFIG_PREEMPT_RT`** genera cambios profundos en la imagen del kernel, cambiando bloques completos de código para adaptarlos a los requerimiento de un kernel RT.

- Implementa preemption en todas las primitivas de locking (*spinlocks*), re implementándolos mediante mutexes con herencia de prioridad.
- El proceso owner del mutex toma la mayor prioridad de los procesos encolados en el mutex. Los llaman rt-mutexes.
- En un encadenamiento de rt-mutexes al tomar otro mutex el proceso propaga la prioridad aumentada del mutex anterior.

Real Time Linux - Full Preemptive

La opción **`CONFIG_PREEMPT_RT`** genera cambios profundos en la imagen del kernel, cambiando bloques completos de código para adaptarlos a los requerimiento de un kernel RT.

- Implementa preemption en todas las primitivas de locking (*spinlocks*), re implementándolos mediante mutexes con herencia de prioridad.
- El proceso owner del mutex toma la mayor prioridad de los procesos encolados en el mutex. Los llaman rt-mutexes.
- En un encadenamiento de rt-mutexes al tomar otro mutex el proceso propaga la prioridad aumentada del mutex anterior.
- Cuando el proceso libera el primer mutex la prioridad desciende a su valor original, o a la mayor de los procesos encolados en los restantes mutexes.

Real Time Linux - Full Preemptive

La opción **`CONFIG_PREEMPT_RT`** genera cambios profundos en la imagen del kernel, cambiando bloques completos de código para adaptarlos a los requerimiento de un kernel RT.

- Implementa preemption en todas las primitivas de locking (*spinlocks*), re implementándolos mediante mutexes con herencia de prioridad.
- El proceso owner del mutex toma la mayor prioridad de los procesos encolados en el mutex. Los llaman rt-mutexes.
- En un encadenamiento de rt-mutexes al tomar otro mutex el proceso propaga la prioridad aumentada del mutex anterior.
- Cuando el proceso libera el primer mutex la prioridad desciende a su valor original, o a la mayor de los procesos encolados en los restantes mutexes.
- En procesadores que poseen instrucciones del tipo Compare & Exchange, o Lock/Unlock esta función es mas sencilla de implementar.

Real Time Linux - Full Preemptive

Real Time Linux - Full Preemptive

- Los handlers de interrupción se convierten en preemptive kernel threads.

Real Time Linux - Full Preemptive

- Los handlers de interrupción se convierten en preemptive kernel threads.
- El kernel con el Patch RT trata a los handlers de interrupción como threads de muy alta prioridad. Pero pueden ser interrumpidos (preempted) por otro kernel thread de mayor prioridad.

Real Time Linux - Full Preemptive

- Los handlers de interrupción se convierten en preemptive kernel threads.
- El kernel con el Patch RT trata a los handlers de interrupción como threads de muy alta prioridad. Pero pueden ser interrumpidos (preempted) por otro kernel thread de mayor prioridad.
- Esto le permite implementar anidamiento de interrupciones en base a prioridades, de manera muy eficiente.

Real Time Linux - Full Preemptive

- Los handlers de interrupción se convierten en preemptive kernel threads.
- El kernel con el Patch RT trata a los handlers de interrupción como threads de muy alta prioridad. Pero pueden ser interrumpidos (preempted) por otro kernel thread de mayor prioridad.
- Esto le permite implementar anidamiento de interrupciones en base a prioridades, de manera muy eficiente.
- Implementa timers de alta resolución, y funciones básicas de watch dog para manejo de timeouts.

Real Time Linux - Full Preemptive

- Los handlers de interrupción se convierten en preemptive kernel threads.
- El kernel con el Patch RT trata a los handlers de interrupción como threads de muy alta prioridad. Pero pueden ser interrumpidos (preempted) por otro kernel thread de mayor prioridad.
- Esto le permite implementar anidamiento de interrupciones en base a prioridades, de manera muy eficiente.
- Implementa timers de alta resolución, y funciones básicas de watch dog para manejo de timeouts.
- Reemplaza el API tradicional de Linux por nuevas funciones de temporización POSIX-compatibles de alta resolución, en User Mode.

Real Time Linux - Full Preemptive

- Los handlers de interrupción se convierten en preemptive kernel threads.
- El kernel con el Patch RT trata a los handlers de interrupción como threads de muy alta prioridad. Pero pueden ser interrumpidos (preempted) por otro kernel thread de mayor prioridad.
- Esto le permite implementar anidamiento de interrupciones en base a prioridades, de manera muy eficiente.
- Implementa timers de alta resolución, y funciones básicas de watch dog para manejo de timeouts.
- Reemplaza el API tradicional de Linux por nuevas funciones de temporización POSIX-compatibles de alta resolución, en User Mode.
- En 2005 se publicaron resultados de rendimiento en el Linux Journal. *"Real-Time and Performance Improvements in the 2.6 Linux Kernel "por William von Hagen*

Temario

Conclusiones

Del análisis de los casos de estudio anteriores debemos extraer conclusiones que nos guíen en el hipotético diseño de nuestro propio sistema si llegamos a esta situación en nuestra vida profesional.

Conclusiones

Del análisis de los casos de estudio anteriores debemos extraer conclusiones que nos guíen en el hipotético diseño de nuestro propio sistema si llegamos a esta situación en nuestra vida profesional. ¿Improbable?

Conclusiones

Del análisis de los casos de estudio anteriores debemos extraer conclusiones que nos guíen en el hipotético diseño de nuestro propio sistema si llegamos a esta situación en nuestra vida profesional. ¿Improbable?. . . por las dudas nunca digas nunca. . .

Tareas

Tareas

- Una cuestión básica a definir es la ejecución en dos modos de privilegio: User y Kernel (o Supervisor).

Tareas

- Una cuestión básica a definir es la ejecución en dos modos de privilegio: User y Kernel (o Supervisor).
- Si implementamos tareas en forma de procesos debemos considerar que requieren su propio espacio de ejecución en User Mode, a diferencia de los threads que trabajan en el mismo espacio de ejecución. En Kernel Mode comparten siempre el espacio de direccionamiento del kernel.

Tareas

- Una cuestión básica a definir es la ejecución en dos modos de privilegio: User y Kernel (o Supervisor).
- Si implementamos tareas en forma de procesos debemos considerar que requieren su propio espacio de ejecución en User Mode, a diferencia de los threads que trabajan en el mismo espacio de ejecución. En Kernel Mode comparten siempre el espacio de direccionamiento del kernel.
- En Sistemas Embedded pequeños con un modelo de threads generalmente resulta suficiente.

Tareas

- Una cuestión básica a definir es la ejecución en dos modos de privilegio: User y Kernel (o Supervisor).
- Si implementamos tareas en forma de procesos debemos considerar que requieren su propio espacio de ejecución en User Mode, a diferencia de los threads que trabajan en el mismo espacio de ejecución. En Kernel Mode comparten siempre el espacio de direccionamiento del kernel.
- En Sistemas Embedded pequeños con un modelo de threads generalmente resulta suficiente.
- A medida que los requisitos del sistema escalan, conviene robustecer el modelo con mejor hardware y pasar a procesos.

Tareas

- Una cuestión básica a definir es la ejecución en dos modos de privilegio: User y Kernel (o Supervisor).
- Si implementamos tareas en forma de procesos debemos considerar que requieren su propio espacio de ejecución en User Mode, a diferencia de los threads que trabajan en el mismo espacio de ejecución. En Kernel Mode comparten siempre el espacio de direccionamiento del kernel.
- En Sistemas Embedded pequeños con un modelo de threads generalmente resulta suficiente.
- A medida que los requisitos del sistema escalan, conviene robustecer el modelo con mejor hardware y pasar a procesos.
- Si se pretende un modelo POSIX, se debe implementar el modelo fork-exec-wait para los procesos.

Memoria

Memoria

- **Real Address Space**: Tareas y kernel en el mismo espacio de ejecución. Apto para sistemas pequeños y sencillos. Evita el overhead de mapear memoria, y la intercomunicación entre tareas es lo mas directa y rápida posible. Desventaja: cero protección. Cualquier error de programación puede eventualmente ocasionar que las tareas se interfieran.

Memoria

- **Real Address Space**: Tareas y kernel en el mismo espacio de ejecución. Apto para sistemas pequeños y sencillos. Evita el overhead de mapear memoria, y la intercomunicación entre tareas es lo mas directa y rápida posible. Desventaja: cero protección. Cualquier error de programación puede eventualmente ocasionar que las tareas se interfieran.
- **Virtual Address Space**: Es el escenario de procesos en diferentes espacios de ejecución. Se diferencia la dirección virtual que trabajan los registros con la dirección física en la que se mapea por hardware en la MMU. Aparecen nuevos conceptos de hardware a considerar.

Memoria

- **Alojamiento dinámico de memoria**: Habilitar a las tareas para solicitar áreas de memoria dinámicamente a un RTOS a medida que ejecutan, requiere un diseño cuidadoso del allocator. Un RTOS debe tener un funcionamiento predecible y determinístico. Si `malloc()` y `free()` trabajan con listas enlazadas buscando secuencialmente nodos libres para asignar, el tiempo de ejecución dependerá de estado de la memoria. No solamente es variable sino que la variabilidad tiene un caracter aleatorio. Claramente este abordaje para `malloc()` no es una buena idea.

Memoria

- **Chequeo de stack overflow**: Se necesita estimar un tamaño adecuado de stack en función del anidamiento de interrupciones y llamadas a funciones que se preve, mas un margen de seguridad. El stack se crea junto con la tarea y por lo general no se modifica. En el caso de implementar memory mapping se puede generar un stack del tamaño de la página de memoria y el hardware se encargará de interrumpir con una excepción si se desborda.

Scheduling, Sincronización e interrupciones

- Procurar establecer un sistema de prioridades fijo si el sistema tiene fuertes restricciones temporales.

Scheduling, Sincronización e interrupciones

- Procurar establecer un sistema de prioridades fijo si el sistema tiene fuertes restricciones temporales.
- En lo posible tratar de aproximar a EDF ya que es mas realista en materia de adaptabilidad a los deadlines de las tareas.

Scheduling, Sincronización e interrupciones

- Procurar establecer un sistema de prioridades fijo si el sistema tiene fuertes restricciones temporales.
- En lo posible tratar de aproximar a EDF ya que es mas realista en materia de adaptabilidad a los deadlines de las tareas.
- Aplicar herencia de prioridades en los mecanismos de sincronización y tratar de manejar **CR** lo mas cortas posible.

Scheduling, Sincronización e interrupciones

- Procurar establecer un sistema de prioridades fijo si el sistema tiene fuertes restricciones temporales.
- En lo posible tratar de aproximar a EDF ya que es mas realista en materia de adaptabilidad a los deadlines de las tareas.
- Aplicar herencia de prioridades en los mecanismos de sincronización y tratar de manejar **CR** lo mas cortas posible.
- La mayoría de los RTOS asignan máxima prioridad a las interrupciones y permiten anidamiento basado en prioridades.

Scheduling, Sincronización e interrupciones

- Procurar establecer un sistema de prioridades fijo si el sistema tiene fuertes restricciones temporales.
- En lo posible tratar de aproximar a EDF ya que es mas realista en materia de adaptabilidad a los deadlines de las tareas.
- Aplicar herencia de prioridades en los mecanismos de sincronización y tratar de manejar **CR** lo mas cortas posible.
- La mayoría de los RTOS asignan máxima prioridad a las interrupciones y permiten anidamiento basado en prioridades.
- En general se deshabilita el Task switch cuando se ejecutan handlers de interrupción anidados.