Sistemas Operativos
Scheduler I

Transición User-Kernel en xv6

Conceptos clave:

- Cambio de contexto
- Memoria virtual
- Trampoline
- Trapframe

Estructura de un proceso en xv6

En la clase anterior se vio que un proceso tiene asociado un contexto. El contexto informalmente se puede pensar como el estado completo del proceso en ejecución conformado por:

- Espacio de direcciones de memoria
- Registros del procesador
- Estructuras del Kernel

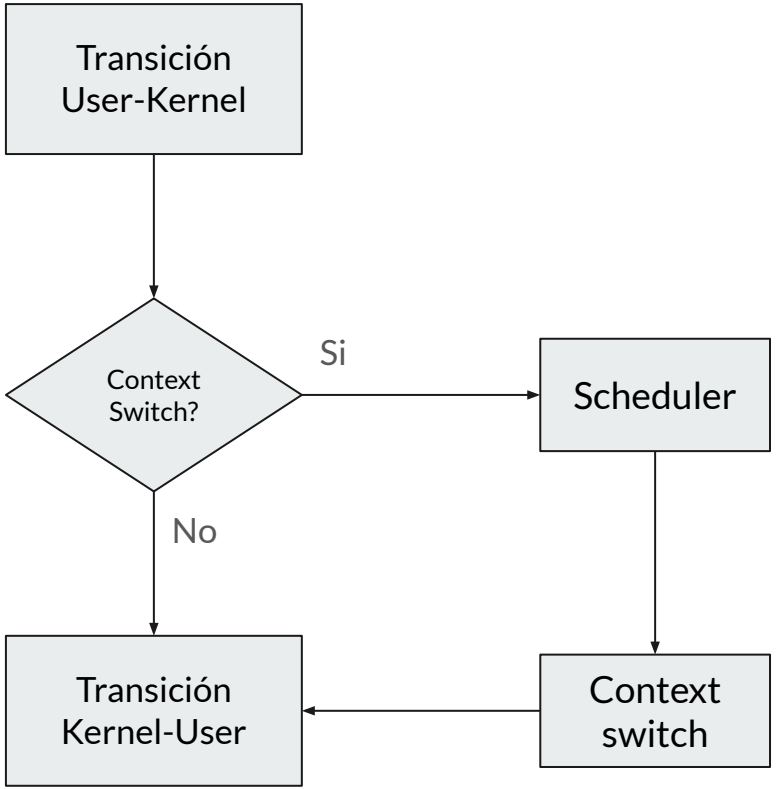
Cuando el sistema operativo interrumpe momentáneamente la ejecución de un proceso y restaura la ejecución de uno suspendido se produce un cambio de contexto.

Diagrama de cambio de contexto

- System Call
- I/O Interrupt
- Timer Interrupt
- Exception

Hacer el context switch cuando
- Timer
- I/O Bloqueante
- Proceso terminado

Si no, hacer una transición al kernel, normal.



El scheduler elige el proximo proceso…

… y se ejecuta el context switch a dicho proceso

Cambio de contexto: Resumiendo

1. Comienza con una transición a Kernel-Space. Los procesos no pueden cambiar desde user space.
   a. Esto puede ocurrir tanto por interrupciones, excepciones, como cuando el usuario llama a una system call.
   b. En todos los casos se puede o no producir un cambio de contexto

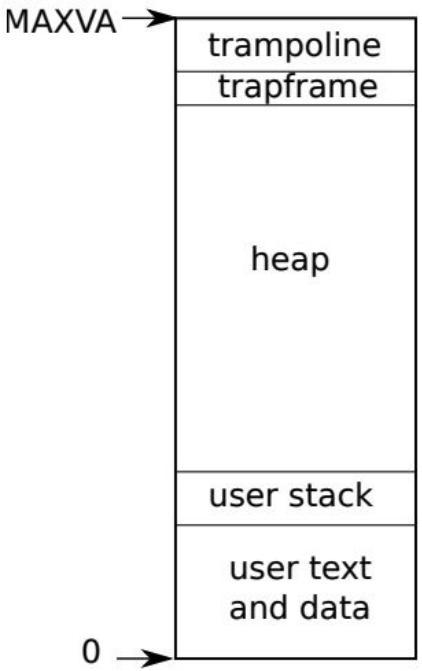
2. Si se decide que hay que realizar un cambio de contexto, se invoca al scheduler.

3. El scheduler elije el siguiente proceso a ejecutar y llama a switch.

4. Switch le saca una foto al estado del kernel-space y del user-space (en general este ultimo ya estaba guardado desde 1) y restaura el proceso candidato (que queda en Kernel-Space)

5. Regresa al User-Space del nuevo proceso

Comparativa xv6 y Linux pre-KPTI



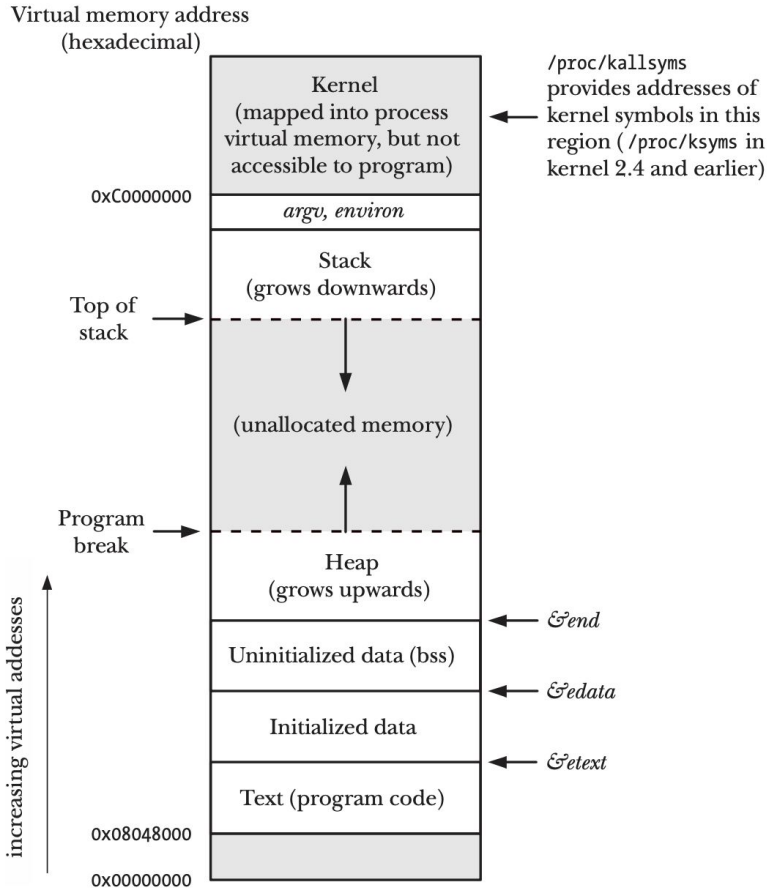


Figure 6-1: Typical memory layout of a process on Linux/x86-32

Estructura de un proceso en xv6

Es una estructura estándar de un proceso en C con dos particularidades:

- Trampoline
- Trapframe

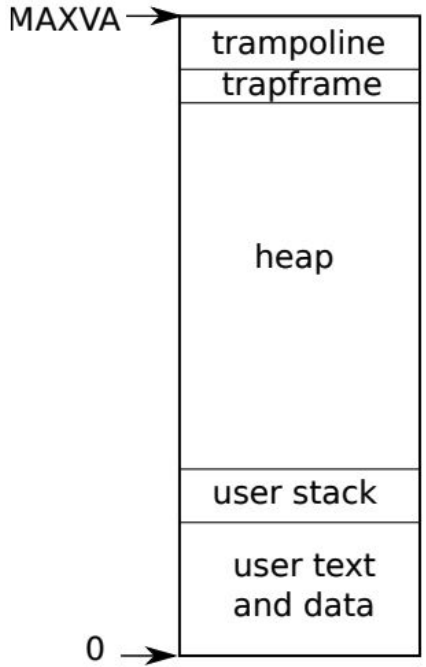


Figure 2.3: Layout of a process's virtual address space

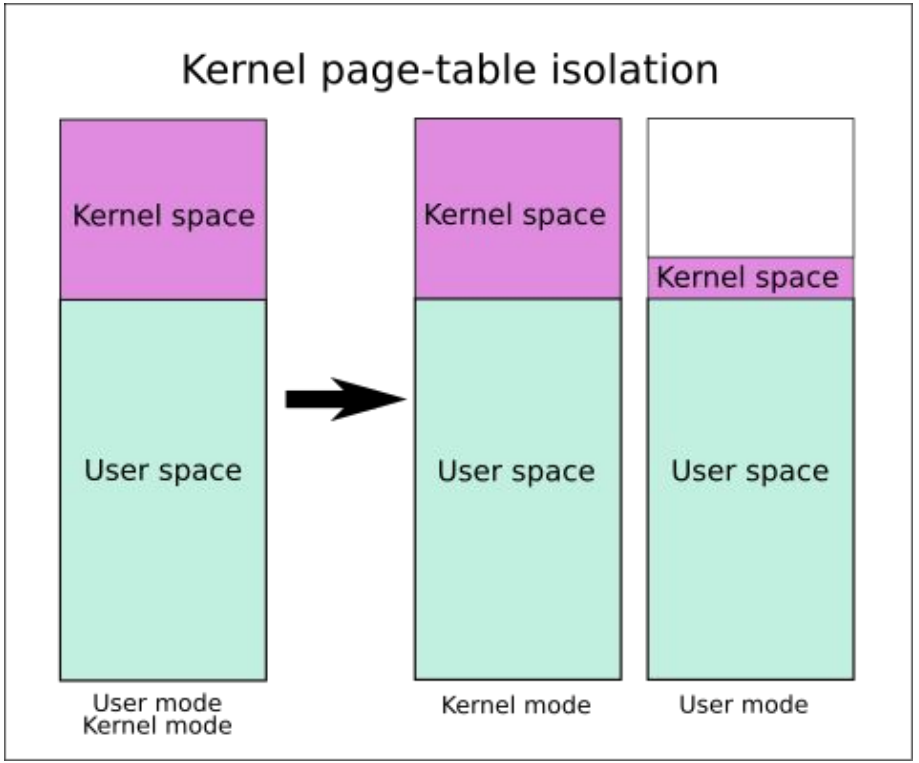
Trampoline

- **Código assembler para transicionar de modo usuario a modo kernel**
- Conceptualmente es la única página del kernel que está todo el tiempo cargada en el user-space del proceso. Es la única página compartida por el address space del usuario y el address space del kernel.
- El resto del kernel se cargará cuando sea necesario.
- Esto se diferencia de Linux previo a 4.15 en donde el kernel reside completo en el address space del proceso.
- Linux modernos usan **Kernel page-table isolation (KPTI)**, que es una técnica similar a la de xv6

Trapframe

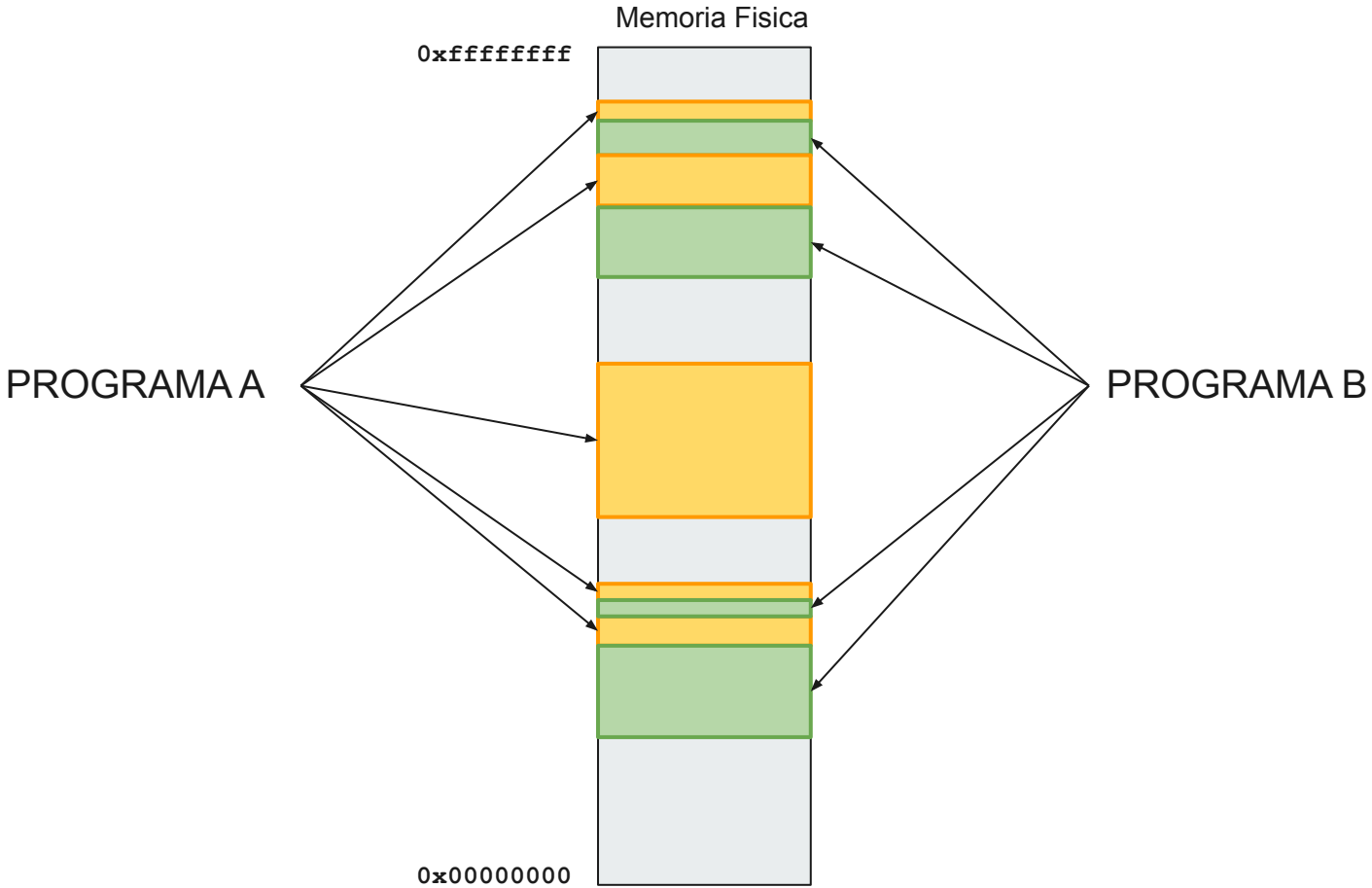
- Es el lugar donde el kernel guarda “la foto de los registros”, para poder restaurarlos luego.
- Es un espacio donde el Kernel va a guardar el estado de todos los registros del procesador, previo a pasar al modo kernel, en el cual no hay garantías de que estos registros se hubieran conservado.
- Reciprocamente, el Kernel cuando restaure un proceso va a leer todos los registros guardados en el Trapframe y los colocara nuevamente en el procesador.

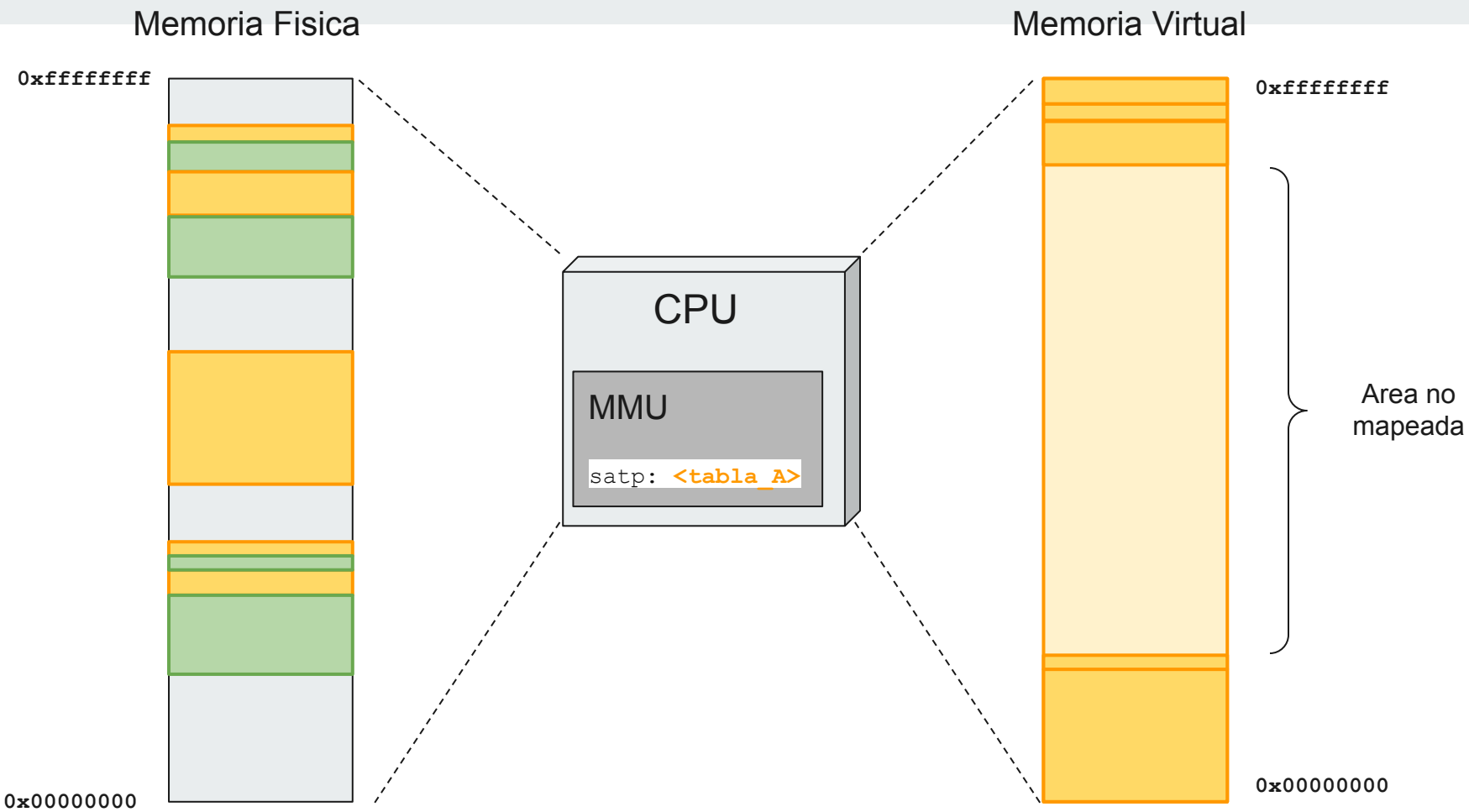
Vulnerabilidad Meltdown y su mitigación

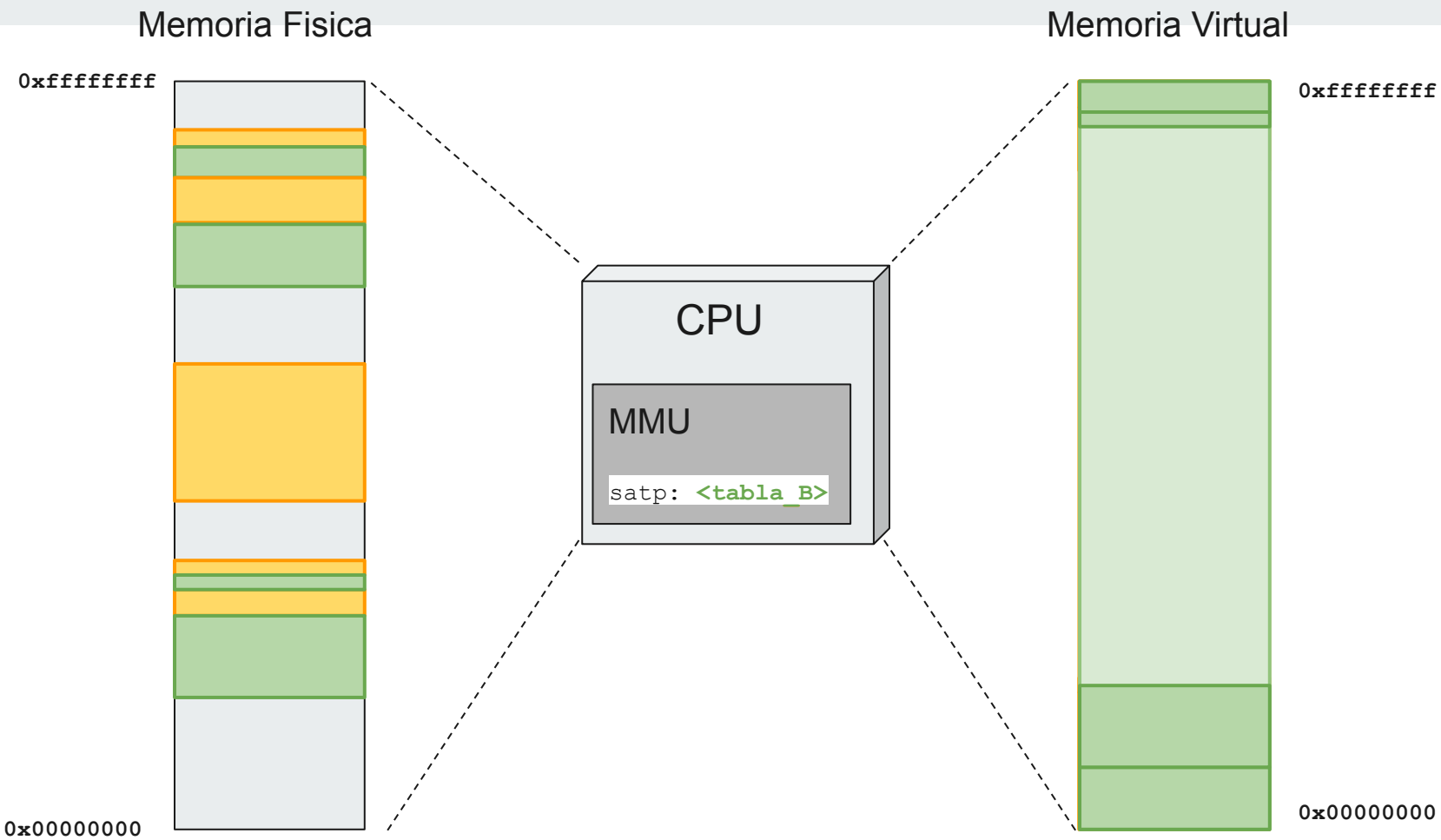


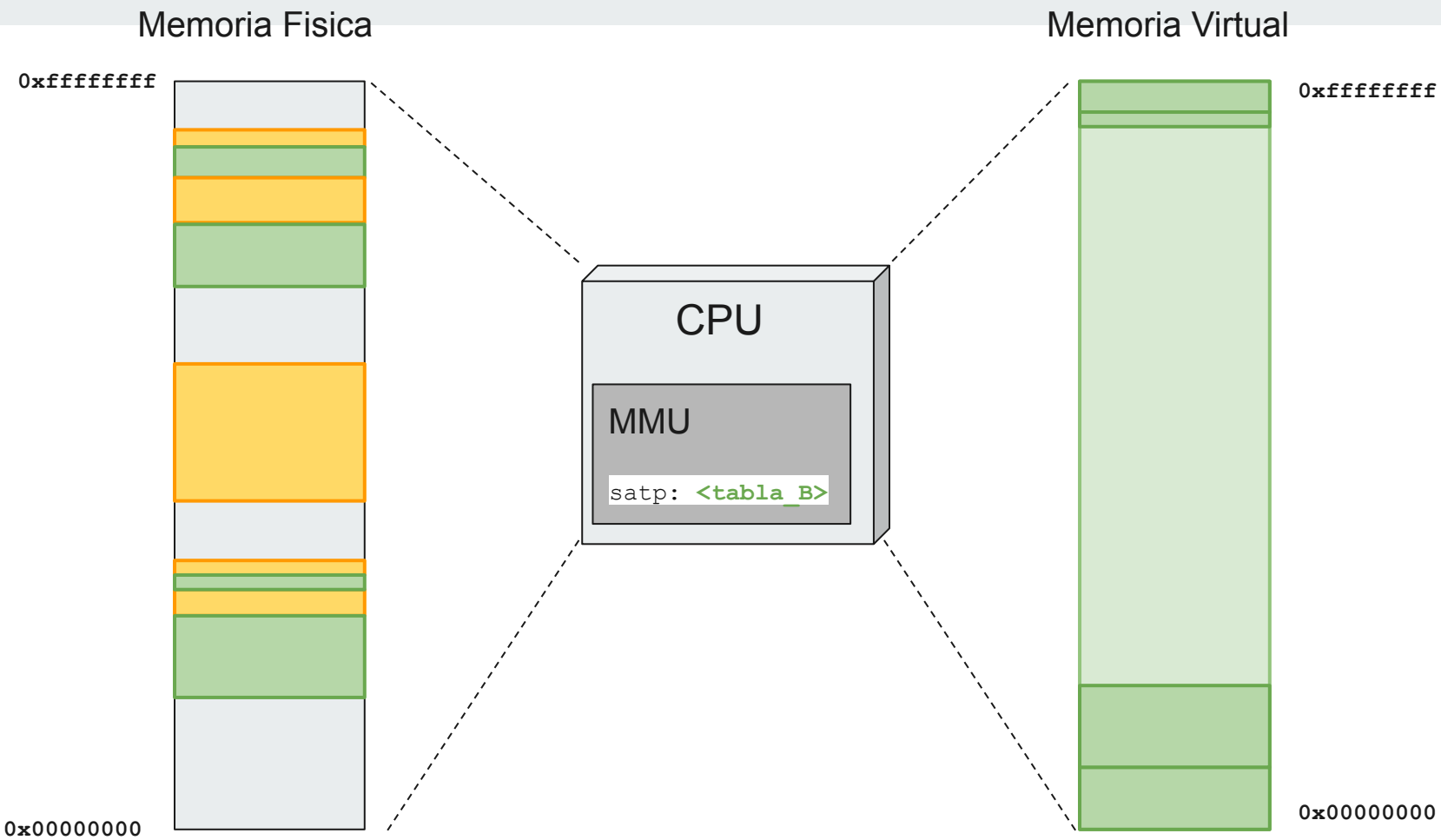
Meltdown: https://medium.com/@mattklein123/meltdown-spectre-explained-6bc8634cc0c2
KPTS: https://en.wikipedia.org/wiki/Kernel_page-table_isolation

Repaso Memoria Virtual




Memoria Fisica
0xffffffff
0x00000000
CPU
MMU
satp: <tabla A>
Memoria Virtual
0xffffffff
0x00000000
Area no mapeada


Memoria Fisica
0xffffffff
0x00000000
CPU
MMU
satp: <tabla_B>
Memoria Virtual
0xffffffff
0x00000000


Memoria Fisica
0xffffffff
0x00000000
CPU
MMU
satp: <tabla_B>
Memoria Virtual
0xffffffff
0x00000000

Invocación de Sistem Calls

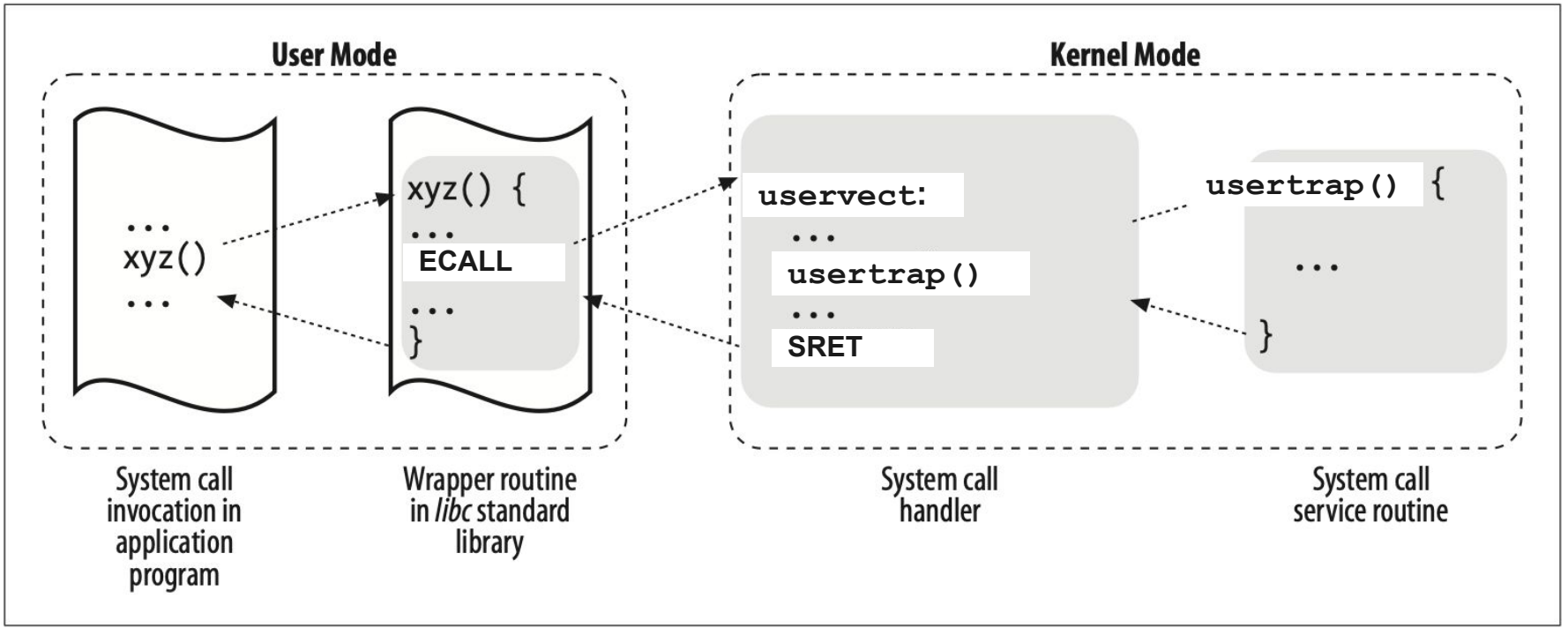
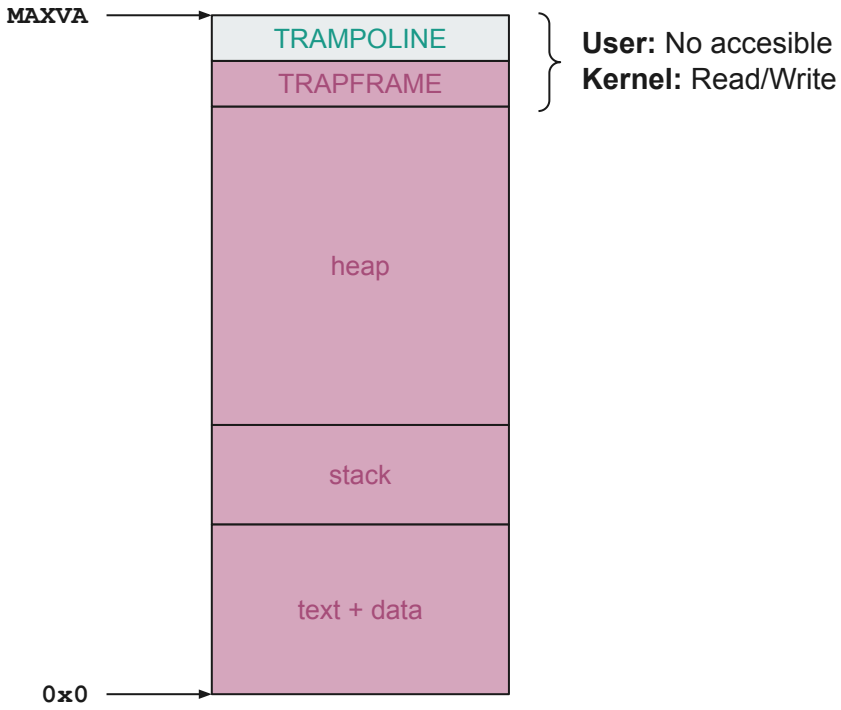


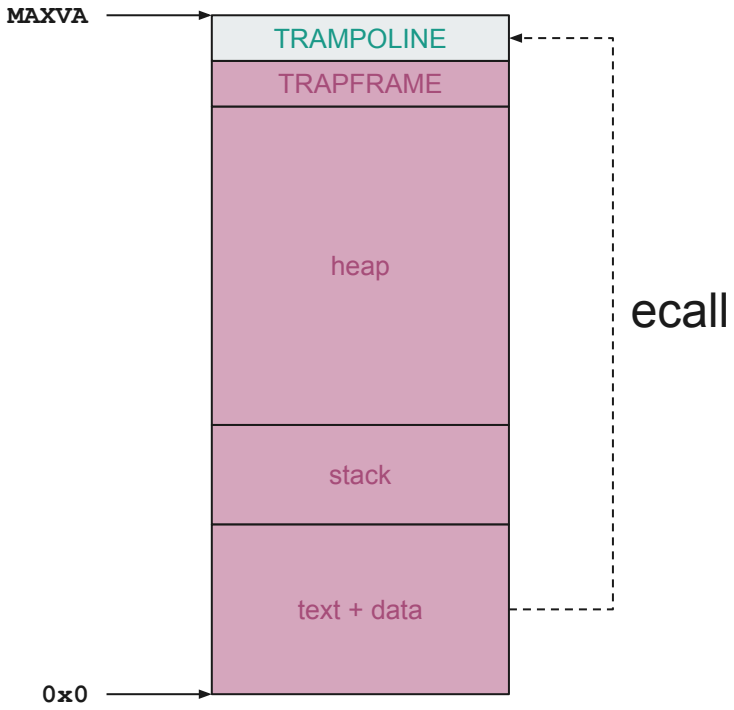
Figure 10-1. Invoking a system call

Transición User-Kernel

Memoria virtual en user space



Transición User-Kernel

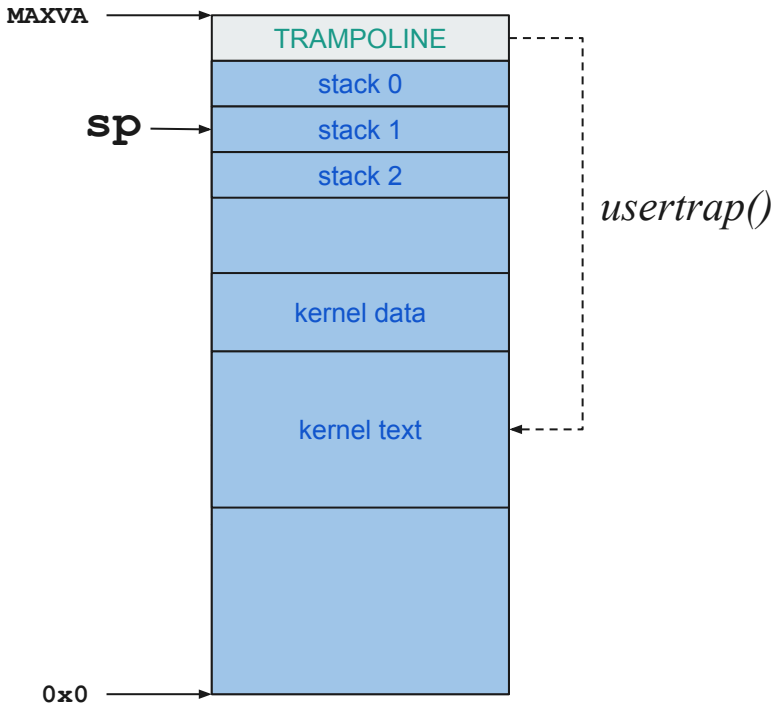


La transición inicia cuando se genera una interrupción por software mediante ecall

El procesador “salta” al **trampoline** y cambia automáticamente a Kernel Mode

Transición User-Kernel

Memoria virtual en kernel space



Código del Trampoline en este orden:

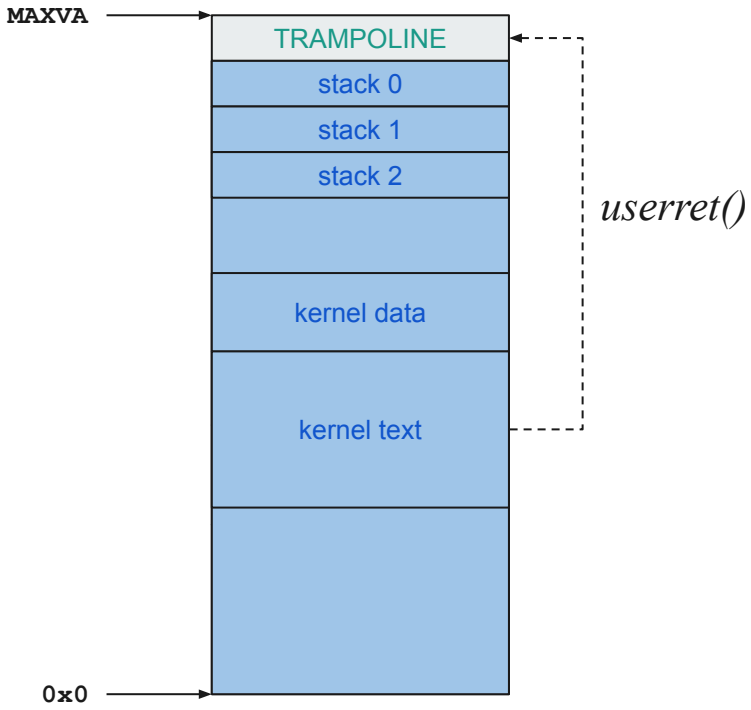
1 - Guarda todos los registros del procesador en el **trapframe**

2 - Cambia el registro **satp**, cambiando la tabla de páginas y efectivamente, el address space.

3 - Indica en el registro sp apuntando al kstack del proceso actual

4 - Invoca la funcion usertrap()

Transición User-Kernel



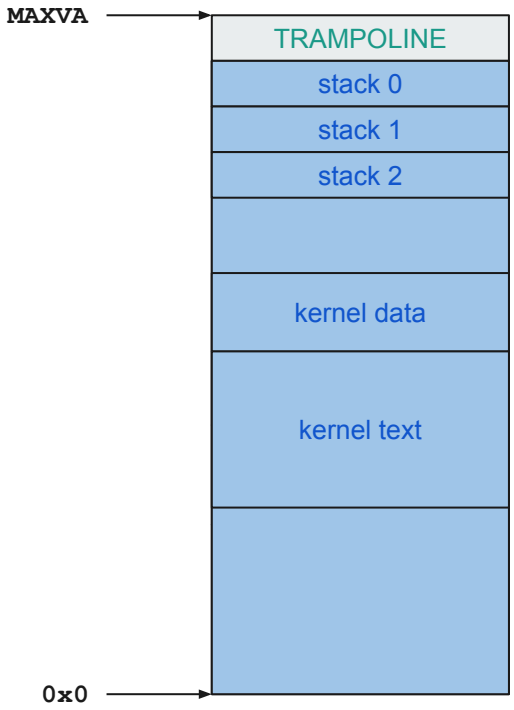
1 - Invoca la funcion userret() en el trampoline

2 - Cambia el registro **satp**, cambiando la tabla de páginas y efectivamente, el address space.

3 - Restaura todos los registros guardados en el **trapframe**

4 - Invoca sret para volver al punto original

Transición User-Kernel



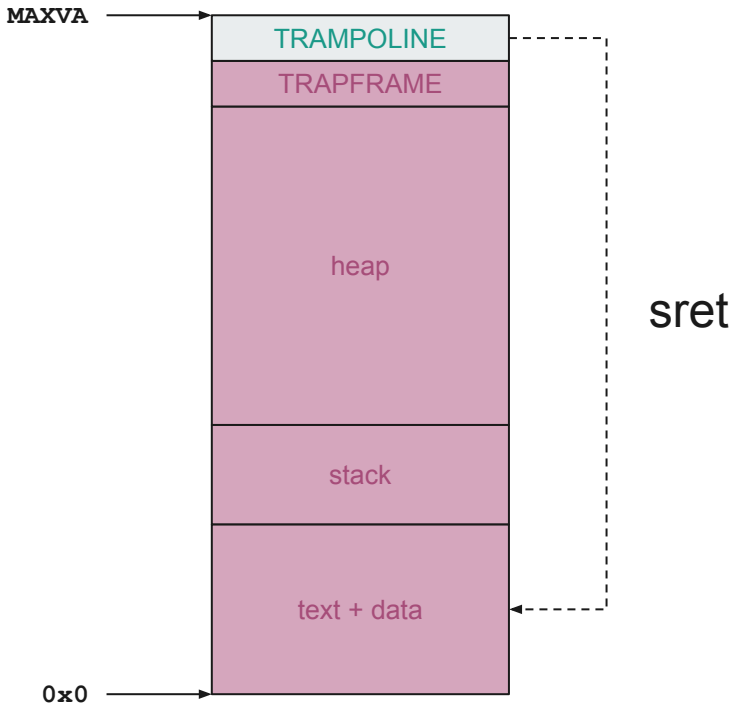
1 - Invoca la funcion userret() (parte del trampoline)

2 - Cambia el registro **satp**, cambiando la tabla de páginas y efectivamente, el address space.

3 - Restaura todos los registros guardados en el **trapframe**

4 - Invoca sret para volver al punto original

Transición User-Kernel



1 - Invoca la funcion userret() en el trampoline

2 - Cambia el registro **satp**, cambiando la tabla de páginas y efectivamente, el address space.

3 - Restaura todos los registros guardados en el **trapframe**

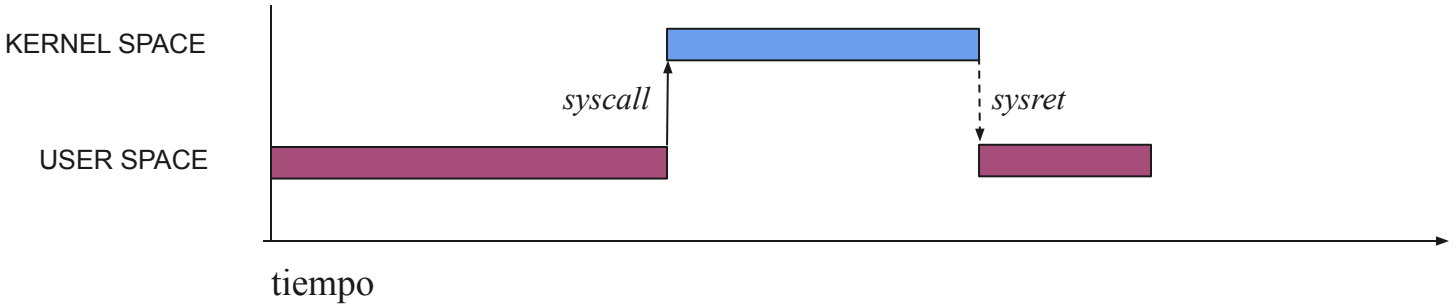
4 - Invoca sret para volver al punto original

Cambio de contexto en xv6

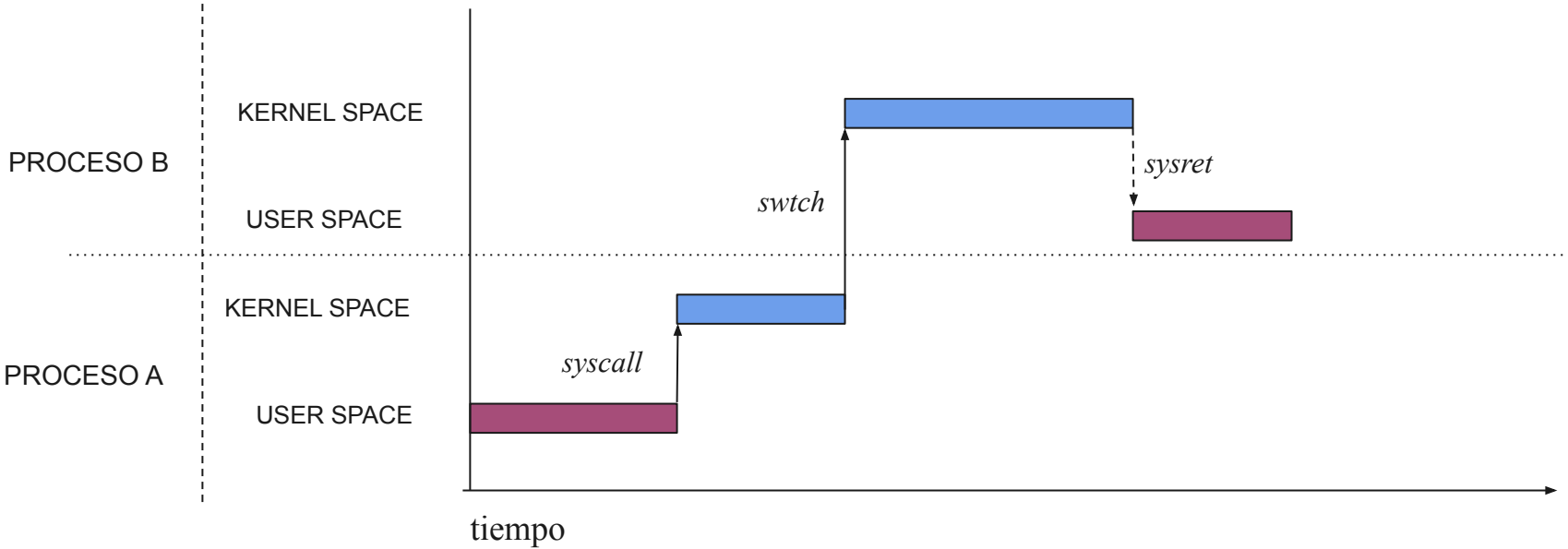
Conceptos clave:

- Scheduler thread
- Guardado de contexto del Kernel

Transición User-Kernel



Context Switch: en general

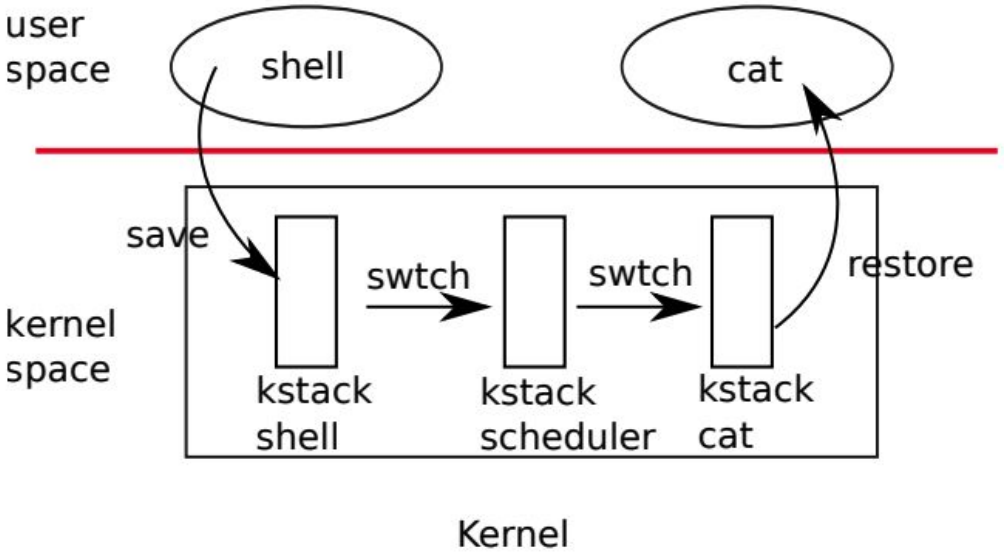


Context Switch: en xv6

Cada CPU tiene un proceso de kernel (sin user-space) que es el scheduler

Un context switch en xv6 son 2 context switch:

- Proceso A -> Scheduler
- Scheduler -> Proceso B



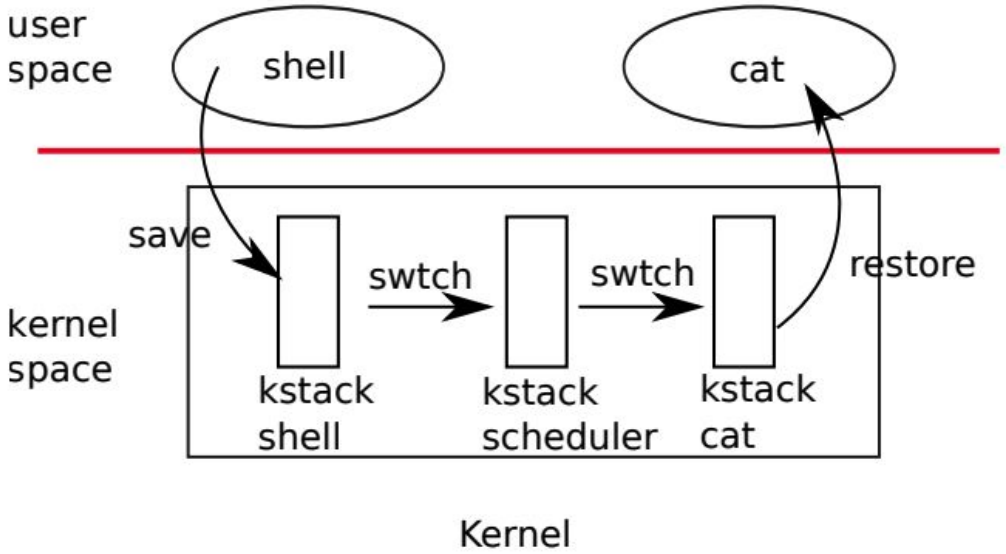
Context Switch: en xv6

Switch guarda el estado (registros) del CPU (seguimos en el kernel) en:

```
(struct proc) p->context
```

Switch carga el estado del scheduler. Como hay un scheduler por CPU, este contexto se guarda en el struct cpu

```
(struct cpu) c->context
```



Context Switch: en xv6

Estado del CPU	**Donde se guarda**	
User-Space	`(struct proc) p->trapframe`	● Mapeado tambien en el user address space ● Uno por proceso
Kernel en contexto del proceso	`(struct proc) p->context`	● Uno por proceso
Kernel en contexto del scheduler	`(struct cpu) c->context`	● Uno por CPU

Scheduling

Conceptos clave:

- Metricas de planificación
- First In, First Out (FIFO)
- Shortest Job First (SJF)
- Shortest Time-to-Completion (STCF)
- Round Robin (RR)

Multiprogramación

Más de un proceso estaba preparado para ser ejecutado en algún determinado momento, y el sistema operativo intercalaba dicha ejecución según la circunstancia. Haciendo esto se mejoró efectivamente el uso de la CPU, tal mejora en la eficiencia fue particularmente decisiva en esos días en la cual una computadora costaba cientos de miles o tal vez millones de dólares.

En esta era, múltiples procesos están listos para ser ejecutados un determinado tiempo según el S.O. lo decidiese en base a ciertas políticas de planificación o scheduling.

Time Sharing

Tiempo compartido se refiere a compartir de forma concurrente un recurso computacional (tiempo de ejecución en la CPU, uso de la memoria, etc.) entre muchos usuarios por medio de las tecnologías de multiprogramación y la inclusión de interrupciones de reloj por parte del sistema operativo, permitiendo a este último acotar el tiempo de respuesta del computador y limitar el uso de la CPU por parte de un proceso dado.

Time Sharing

A medida que los tiempos de respuesta entre procesos se fueron haciendo cada vez más pequeños, más procesos podían ser cargados en memoria para su ejecución.

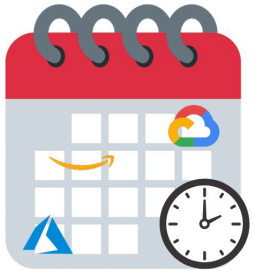
Una variante de la técnica de multiprogramación consistió en asignar una terminal a cada usuario en línea.

Time Sharing

Teniendo en cuenta que los seres humanos tienen un tiempo de respuesta lento (0.25 seg para estímulos visuales) en comparación a una computadora (operación en nanosegundos). Debido a esta diferencia de tiempos y a que no todos los usuarios necesitan de la cpu al mismo tiempo, este sistema daba la sensación de asignar toda la computadora a un usuario determinado Corbató y otros. Este concepto fue popularizado por MULTICS.

Planificación

Cuando un Sistema Operativo se dice que realiza multi-programación de varios procesos debe existir una entidad que se encargue de coordinar la forma en que estos se ejecutan, el momento en que estos se ejecutan y el momento en el cual paran de ejecutarse. En un sistema operativo esta tarea es realizada por el **Planificador** o S**cheduler** que forma parte del Kernel del Sistema Operativo.



Scheduling o Planificación de Procesos

Cuando hay múltiples cosas qué hacer ¿Cómo se elige cuál de ellas hacer primero?

Debe existir algún mecanismo que permita determinar cuanto tiempo de CPU le toca a cada proceso. Ese período de tiempo que el kernel le otorga a un proceso se denomina **time slice o time quantum**.

El Workload

El Workload es carga de trabajo de un proceso corriendo en el sistema.

Determinar cómo se calcula el workload es fundamental para determinar partes de las políticas de planificación. Cuanto mejor es el cálculo, mejor es la política. Las suposiciones que se harán para el cálculo del workload son más que irreales.

Una clasificacion de schedulers

Tipo de sistema

- **Interactivo**: necesita tiempos de respuesta bajos. Los procesos deben recibir atención frecuentemente. Ejemplo: Round Robin.

  Eg. Un procesador de texto

- **Por lotes (batch)**: los procesos pueden tardar más en completarse. Se optimiza el tiempo total de ejecución más que la respuesta inmediata. Ejemplo: FIFO, SJF.

  Eg. Calcular una liquidación de sueldos

Una clasificacion de schedulers

Preemptivo vs No Preemptivo

- **Preemptivo:** el sistema puede interrumpir un proceso en ejecución para darle tiempo a otro. Esto permite más equidad y respuesta rápida (como en RR o STCF).

- **No Preemptivo:** una vez que un proceso empieza, no se interrumpe hasta que termina (como FIFO y SJF).

Supuestos y simplificaciones iniciales

Con fines didácticos, asumimos las siguientes simplificaciones sobre los procesos ejecutándose en nuestro sistema para definir las primeras políticas. Estos no son supuestos realistas.

- Cada proceso se ejecuta la misma cantidad de tiempo.
- Todos los jobs llegan al mismo tiempo para ser ejecutados.
- Una vez que empieza un job sigue hasta completarse.
- Todos los jobs usan únicamente cpu (no realizan I/O).
- El tiempo de ejecución (run-time) de cada job es conocido.

Métricas de Planificación

Para poder hacer algún tipo de análisis se debe tener algún tipo de métrica estandarizada para comparar las distintas políticas de planificación o scheduling. Bajo estas premisas, por ahora, para que todo sea simple se utilizará una única métrica llamada ***turnaround time.*** Que se define como ***el tiempo en el cual el proceso se completa menos el tiempo de arribo al sistema***:

$$T_{turnaround} = T_{completion} - T_{arrival}$$

Debido a 2 (todos los jobs llegan al mismo tiempo) el $T_{arrival}=0$

El **turnaround** time es una métrica que mide **performance**.

Políticas de Scheduling

- **Batch**
  - First In, First Out (FIFO)
  - Shortest Job First (SJF)
  - Shortest Time-to-Completion (STCF)
- **Interactiva**
  - Round Robin (RR)

First In, First Out (FIFO)

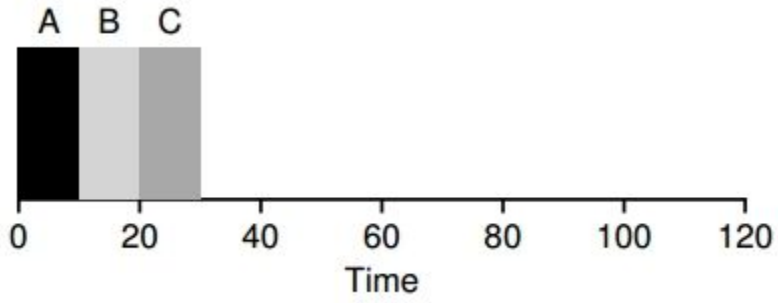
El algoritmo más básico para implementar como política de planificaciones es el First In First Out o Fist Come, First Served.

Ventajas:

- Es simple.
- Es fácil de implementar.
- Funciona bárbaro para las suposiciones iniciales.

First In, First Out (FIFO)

Por ejemplo se tiene tres procesos A, B y C con Tarrival=0.



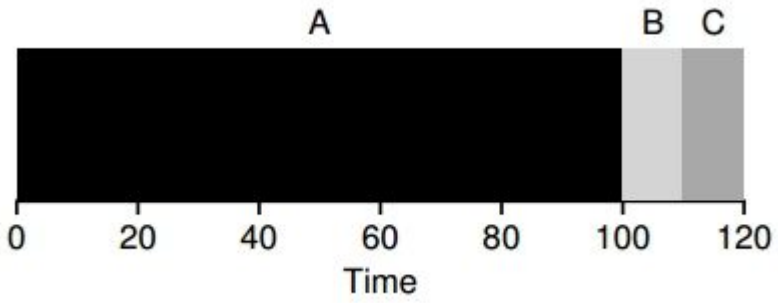
Si bien llegan todos al mismo tiempo llegaron con un insignificante retraso de forma tal que llegó A, B y C. S i se asume que todos tardan 10 segundos en ejecutarse…

¿cuánto es el Taround?

(10+20+30)/3= 20

First In, First Out (FIFO)

Ahora relajemos la suposición 1 y no se asume que todas las tareas duran el mismo tiempo. Ahora A dura 100 segundos.



¿Cuánto es el Turnaround?

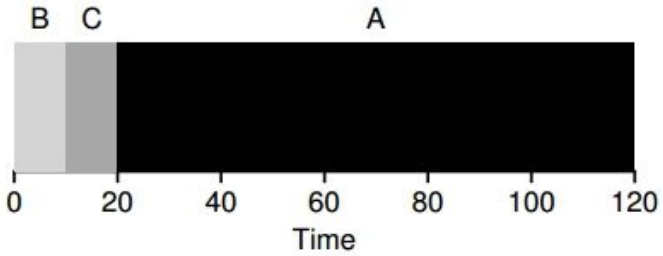
(100+110+120)/3=110

First In, First Out (FIFO)

Convoy effect

Shortest Job First (SJF)

Para resolver el problema que se presenta en la política FIFO, se modifica la política para que se ejecute el proceso de duración mínima, una vez finalizado esto se ejecuta el proceso de duración mínima y así sucesivamente.

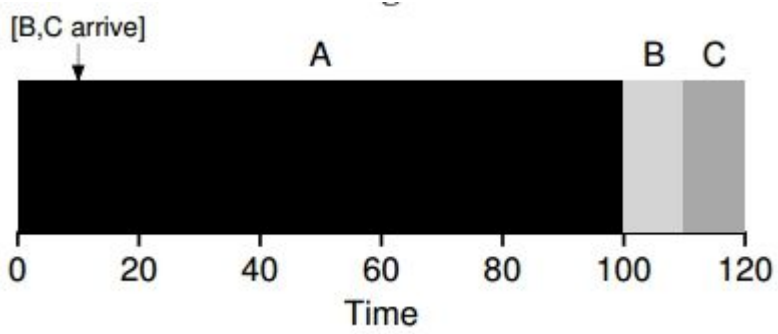


En el mismo caso de arriba, se mejora el turnaround time con el sencillo hecho de ejecutar B, C y A en ese orden:

(10+20+120)/3=50

Shortest Job First (SJF)

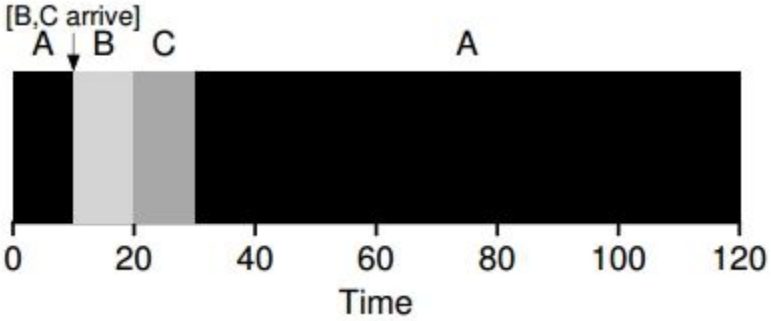
Utilizando SJF se obtuvo una significativa mejora… pero con las suposiciones iniciales que son muy poco realistas. Si se relaja la suposición 2, en la cual no todos los procesos llegan al mismo tiempo, por ejemplo llega el proceso A y a los 10 segundos llegan el proceso B y el proceso C. ¿Cómo sería el cálculo, ahora? _t_=10 seg



(100+110−10+120−10)/3=103.33

Shortest Time-to-Completion (STCF)

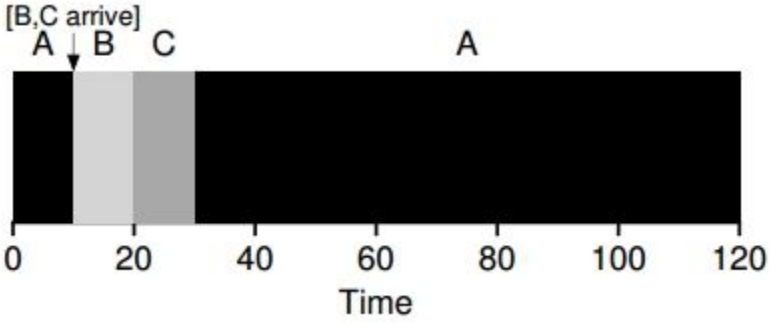
Para poder solucionar este problema se necesita relajar la suposición 3 (los procesos se tienen que ejecutarse hasta el final). La idea es que el planificador o scheduler pueda adelantarse y determinar qué proceso debe ser ejecutado. Entonces cuando los procesos B y C llegan se puede **desalojar (preempt)** al proceso A y decidir que otro proceso se ejecute y luego retomar la ejecución del proceso A.



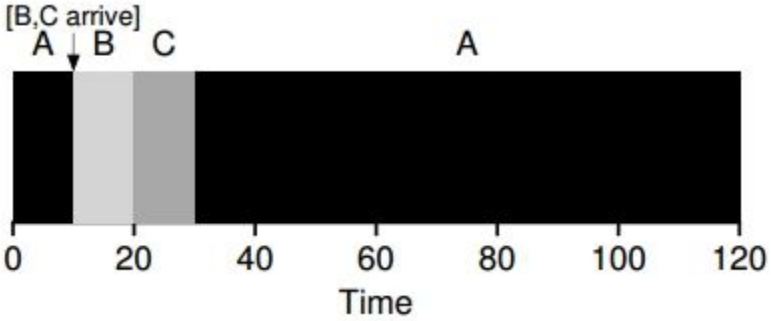
El caso anterior el de SFJ es una política non-preemptive

Shortest Time-to-Completion (STCF)

Cuando un nuevo job llega al sistema, el scheduler determina cuál de los jobs en el sistema (incluyendo al recientemente arribado) **le falta menos tiempo para terminar**, y elige a dicho proceso, **desalojando al proceso actual**.



Shortest Time-to-Completion (STCF)



El cálculo para el turnaround time sería

$$(120-0+20-10+30-10)/3=50$$

Una nueva métrica: Tiempo de Respuesta

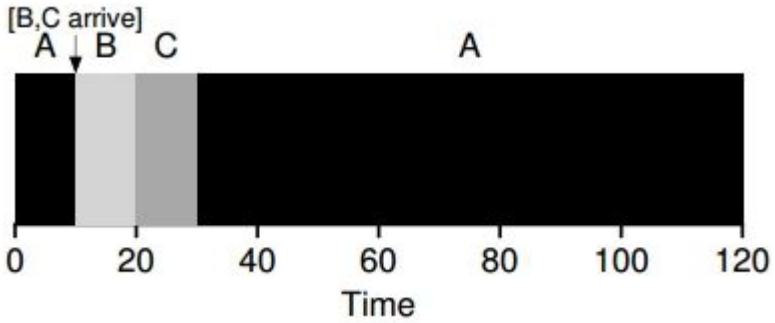
El tiempo de respuesta (o response time) surge con el advenimiento del time-sharing ya que los usuarios se sientan en una terminal de una computadora y pretenden una interacción con rapidez. Por eso nace el response time como métrica:

$$T_{response} = T_{firstrun} - T_{arrival}$$

Una nueva métrica: Tiempo de Respuesta

El $T_{response}$ del proceso A es 0.
El $T_{response}$ del proceso B es… 0… llega en 10 pero tarda 10 (10-10)
El $T_{response}$ del proceso C es… 10… llega en 10 pero termina en 20 (20-10)

En promedio el $T_{response}$ es de 3.33 seg. Entonces

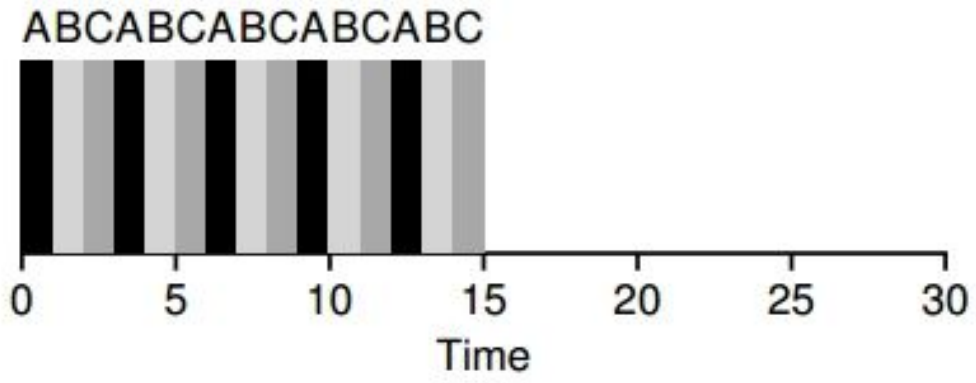


¿Cómo escribir un planificador que tenga noción del tiempo de respuesta?

Round Robin

La idea del algoritmo es bastante simple, se ejecuta un proceso por un período determinado de tiempo (**time slice**) y transcurrido el período se pasa a otro proceso, y así sucesivamente cambiando de proceso en la cola de ejecución.

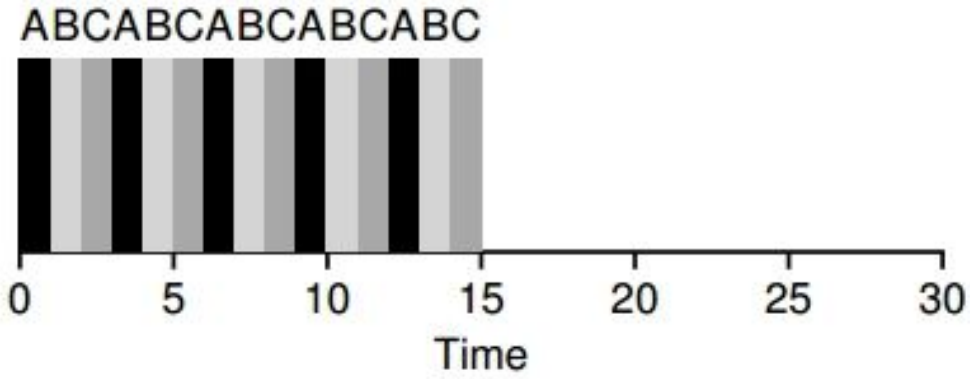
Tambien se llama **time-slicing**.



Round Robin

El time-slice es un multiplo el timer interrupt.

Ejemplo. Si el timer interrupt es 10 ms, el time slice podrá ser 10ms, 20ms y cualquier n*10ms



Response time:

$$(0 + 1 + 2) / 3 = 1$$

Round Robin

Lo importante de RR es la elección de un buen time slice, se dice que el time slice tiene que amortizar el cambio de contexto sin hacer que esto resulte en que el sistema no responda más.

Por ejemplo, si el tiempo de cambio de contexto está seteado en 1 ms y el time slice está seteado en 10 ms, el 10% del tiempo se estará utilizando para cambio de contexto.

Sin embargo, si el time slice se setea en 100 ms, solo el 1% del tiempo será dedicado al cambio de contexto.

Pregunta: ¿Qué pasa si se trae a colación a la métrica del turnaround time ?

Algoritmo	Tipo de sistema	¿Preemptivo?	Tiempo de respuesta	Turnaround Time
FIFO	Batch	❌ No	❌ Malo (puede ser alto)	⚠️ Variable (puede ser malo)
SJF	Batch	❌ No	❌ Malo (espera inicial larga)	✅ Excelente (mínimo promedio)
STCF	Batch	✅ Sí	✅ Bueno (procesos cortos responden rápido)	✅ Muy bueno (casi óptimo)
RR	Interactivo	✅ Sí	✅ Excelente (respuesta rápida inicial)	❌ Regular (turnaround más alto)

Round-Robin en xv6

```
c->proc = 0;

for(;;){

  int found = 0;

  for(p = proc; p < &proc[NPROC]; p++) {

    if(p->state == RUNNABLE) {

      p->state = RUNNING;

      c->proc = p;

      swtch(&c->context, &p->context);

      c->proc = 0;

      found = 1;

    }

  }

}
```

Considerando el I/O

Un proceso (un solo thread) puede estar haciendo dos cosas:

- Usando el CPU
- Esperando por un dispositivo de I/O

Objetivo: queremos que siempre haya alguien usando el CPU.
Si no, estamos desperidicando hardware!

Considerando el I/O

Mal uso del CPU; desperdiciamos recursos y perdemos tiempo de multitasking!

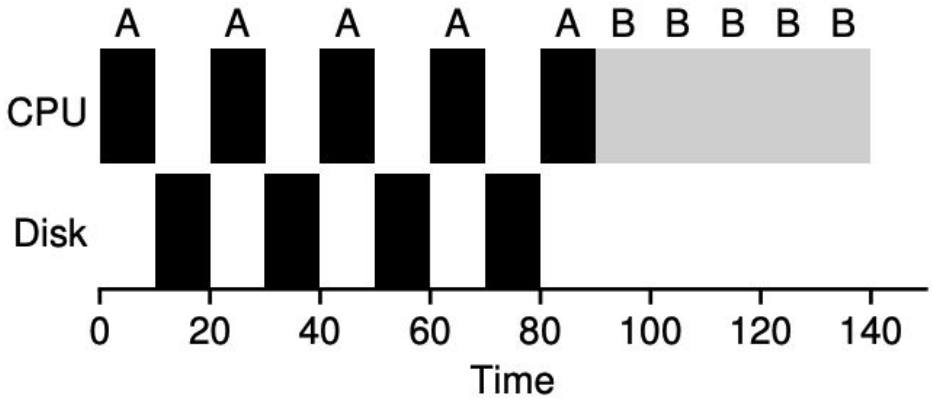


Figure 7.8: **Poor Use Of Resources**

Considerando el I/O

Una estrategia mejor

Superponer (overlap): mientras se ejecuta I/O, ejecutar otros procesos

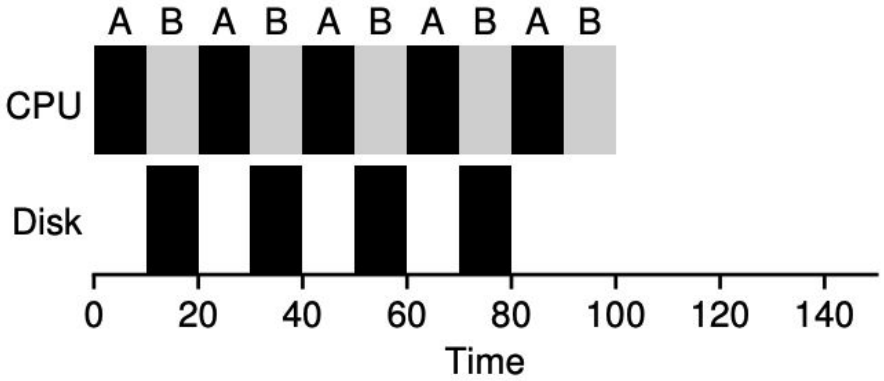


Figure 7.9: **Overlap Allows Better Use Of Resources**

Utilización de la CPU

Se puede pensar en dos clases de programas:

- **CPU intensive:** Usan mucho las capacidades de cálculo del CPU. Eg. calculo de matrices, aplicaciones gráficas, entrenamiento de redes neuronales.

- **I/O intensive:** Usan mucho la entrada y salida, y el CPU se dedica principalmente a gestionar estos procesos. Eg. un file server, un reproductor de video que usa aceleración de video en una GPU.

Utilización de la CPU

I/O intensive:

- Si no tenemos cuidado, un proceso I/O intensive va a estar desperdiciando el CPU, esperando a los dispositivos de entrada y salida.

- **Solucion**. Ejecutar muchos procesos I/O intensive simultáneamente. Ocupar el CPU al máximo!

- Notar que esta solución no funciona para el caso de muchos procesos CPU intensive. Si tenemos un proceso que usa el 100% del CPU, agregar más procesos simplemente hará que el CPU siga al 100%.

Utilización de la CPU

CPU intensive:

- No hay escapatoria, el proceso necesita el CPU.
- Todos estos procesos van a competir por el CPU, siendo desalojados por el timer de vez en cuando.
- En la actualidad, casi todas las aplicaciones CPU intensive tienen alguna forma de aceleración por hardware, esto es, mediante el uso de co-procesadores (eg. GPU)
- Al delegar el cálculo al coprocesador, la aplicación se transforma en I/O intensive (o mas bien GPU intensive)

Utilización de la CPU

Los GPUs inicialmente usados para aplicaciones graficas se generalizaron para implementar algoritmos facilmente paralelizables.
A esto se lo llama **General-purpose computing on graphics processing units (GPGPU).**

En la actualidad, las GPU son centrales en el entrenamiento de redes neuronales.

El GPU se transforma en un co-procesador del CPU.

Libera al CPU de los cálculos, y **transforma al proceso en un proceso I/O intensive.**

Utilización de la CPU

Estrategias para procesos I/O Intensive

Asumiendo una **calendarización perfecta**: si el 20% del tiempo de ejecución de un programa es sólo cómputo y el 80% son operaciones de entrada y salida, con tener 5 procesos en memoria se pueden acomodar los procesos para utilizar el 100% de la CPU.

El I/O es mucho más lento que las instrucciones. Siendo un poco más realista se supone que las operaciones de E/S son bloqueantes (una operación de lectura a disco tarda 10 miliseg y una instrucción registro registro tarda 1-2 nanoseg), es decir, paran el procesamiento hasta que se haya realizado la operación de E/S.

Utilización de la CPU

Cálculo probabilístico de utilización del CPU

Si se supone que un proceso permanece una fracción p bloqueado a la espera de E/S entonces la probabilidad de que en un instante determinado ese proceso este bloqueado es p. Ejemplo. Si el proceso se la pasa el 80% del tiempo bloqueado por E/S, la probabilidad de que en un instante determinado el proceso está bloqueado será de 0.8.

Si tenemos n procesos idénticos, la probabilidad de que todos ellos estén bloqueados simultáneamente es p^n

Reciprocamente, la probabilidad de que exista al menos un proceso no bloqueado, y por lo tanto ejecutable es $1-p^n$
Esta fórmula es conocida como **utilización de CPU**:

Utilización del CPU: $1-p^n$

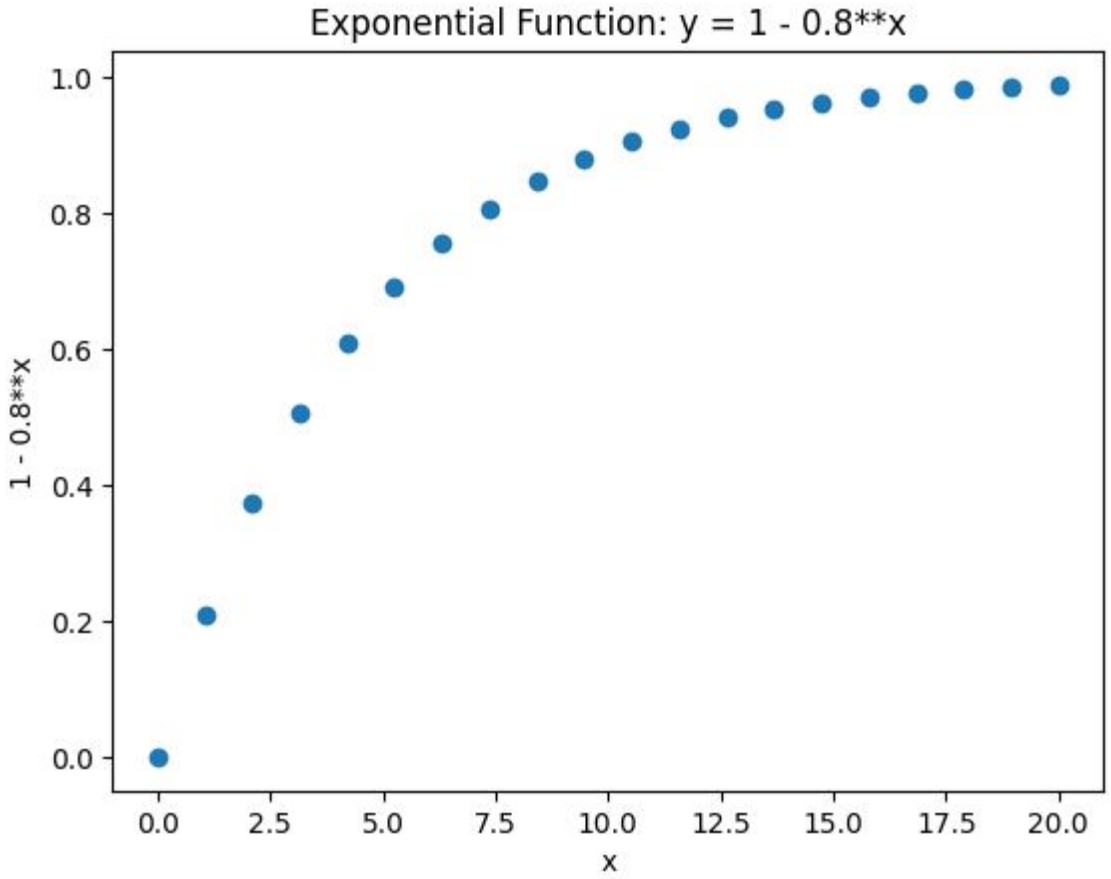
Utilización de la CPU

Por ejemplo: si se tiene un solo proceso en memoria y este invierte un 80% del tiempo en operaciones de E/S el tiempo de utilización de CPU es 1 - 0.8 = 0.2 que es el **20%**.

Ahora bien, si se tienen 3 procesos con la misma propiedad, el grado de utilización de la cpu es $1-0.8^3= 0.488$ es decir el **49%** de ocupación de la cpu.

Si se supone que se tienen 10 procesos, entonces la fórmula cambia a $1-0.8^{10}=0.89$ el **89%** de utilización, aquí es donde se ve la IMPORTANCIA de la Multiprogramación.

Utilización de la CPU



Multi-Level Feedback Queue

Conceptos clave:

- Problemas que resuelve
- Algoritmo MLFQ

Planificación en la Vida Real

¿Qué debería proporcionar un marco de trabajo básico que permita pensar en políticas de planificaciones ?

¿Cuáles deberían ser las suposiciones a tener en cuenta?

¿Cuáles son las métricas importantes?

Multi Level Feedback Queue

Esta técnica llamada Multi-Level Feedbak Queue de planificación fue descripta inicialmente en los años 60 en un sistema conocido como Compatible Time Sharing System CTSS. Este trabajo en conjunto con el realizado sobre MULTICS llevó a que su creador ganara el Turing Award.

Este planificador ha sido refinado con el paso del tiempo hasta llegar a las implementaciones que se encuentran hoy en un sistema moderno.

Multi Level Feedback Queue

MLQF intenta atacar principalmente 2 problemas:

- Intenta optimizar el **turnaround time**, lo cual se logra ejecutando primero los trabajos más cortos; desafortunadamente, el sistema operativo generalmente no sabe cuánto tiempo va a durar un trabajo, precisamente el conocimiento que requieren algoritmos como SJF (Primero el Trabajo Más Corto) o STCF (Primero el Trabajo Más Corto con Expiración).

- MLFQ intenta que el sistema sea responsivo a los usuarios interactivos, minimizando el **response time**; sin embargo, algoritmos como Round Robin reducen el response time, pero son terribles para el turnaround time.

Multi Level Feedback Queue

Entonces:

- ¿Cómo se hace para que un planificador pueda lograr estos dos objetivos si generalmente no se sabe nada sobre el proceso a priori?.

- ¿Cómo se planifica sin tener un conocimiento acabado?

- ¿Cómo se construye un planificador que minimice el tiempo de respuesta para las tareas interactivas y también minimice el **timearound time** sin un conocimiento a priori de cuanto dura la tarea?

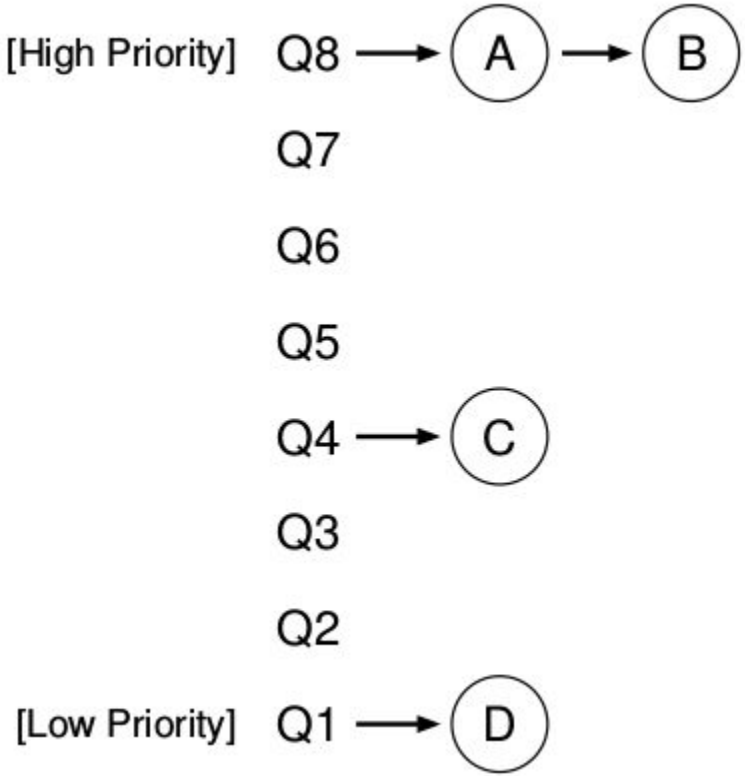
MLQF: Las reglas básicas

MLFQ tiene un conjunto de distintas colas, cada una de estas colas tiene asignado un nivel de prioridad.

En un determinado tiempo, una tarea que está lista para ser ejecutada está en una única cola. MLFQ usa las prioridades para decidir cual tarea debería correr en un determinado tiempo t0: la tarea con mayor prioridad o la tarea en la cola mas alta sera elegida para ser ejecutada.

Dado el caso que existan más de una tarea con la misma prioridad entonces se utilizará el algoritmo de Round Robin para planificar estas tareas entre ellas.

MLQF: Las reglas básicas



MLQF: Las reglas básicas

Las 2 reglas básicas de MLFQ:

REGLA 1: si la prioridad (A) es mayor que la prioridad de (B), (A) se ejecuta y (B) no.

REGLA 2: si la prioridad de (A) es igual a la prioridad de (B), (A) y (B)se ejecutan en Round-Robin.

La clave para la planificación MLFQ subyace entonces en cómo el planificador setea las prioridades. En vez de dar una prioridad fija a cada tarea, **MLFQ varía la prioridad de la tarea basándose en su comportamiento observado.**

MLQF: Las reglas básicas

- Por ejemplo, si una determinada tarea repetidamente no utiliza la CPU mientras espera que un dato sea ingresado por el teclado, MLFQ va a mantener su prioridad alta, así es como un proceso interactivo debería comportarse.
- Si por lo contrario, una tarea usa intensivamente por largos periodos de tiempo la CPU, MLFQ reducirá su prioridad. De esta forma MLFQ va a aprender mientras los procesos se ejecutan y entonces va a usar la historia de esa tarea para predecir su futuro comportamiento

Primer intento: ¿Cómo cambiar la prioridad ?

Se debe decidir cómo MLFQ va a cambiar el nivel de prioridad a una tarea durante toda la vida de la misma (por ende en que cola esta va a residir).

Para hacer esto hay que tener en cuenta nuestra carga de trabajo (workload): *una mezcla de tareas interactivas que tienen un corto tiempo de ejecución y que pueden renunciar a la utilización de la CPU y algunas tareas de larga ejecución basadas en la CPU que necesitan tiempo de CPU , pero poco tiempo de respuesta*.

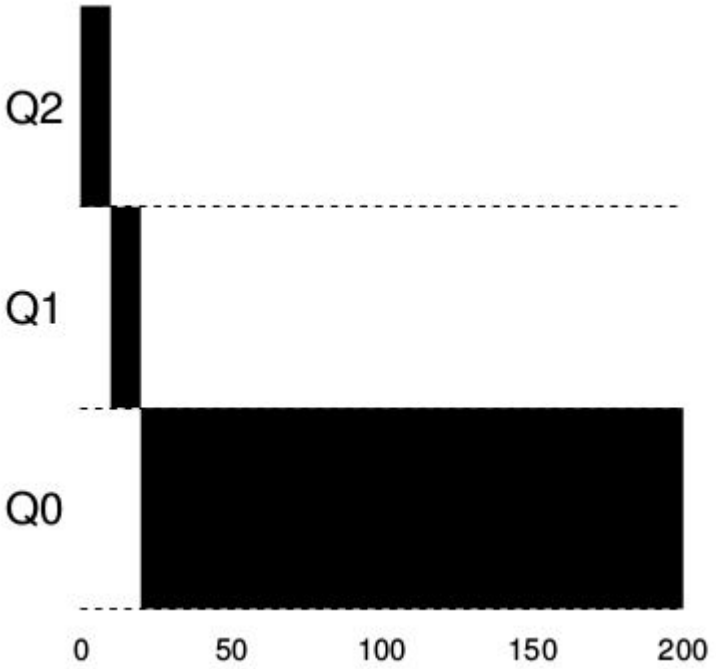
Primer intento: ¿Cómo cambiar la prioridad ?

A continuación se muestra un primer intento de algoritmo de ajuste de prioridades:

> **REGLA 3**: Cuando llega una tarea al sistema se la coloca en la cola de más alta prioridad.
>
> **REGLA 4a**: Si durante una ejecución una tarea usa su time slice completo entonces su prioridad se reduce en una unidad (baja a la cola directamente inferior)
>
> **REGLA 4b**: Si una tarea renuncia al uso de la CPU antes de completar un time slice se queda en el mismo nivel de prioridad.

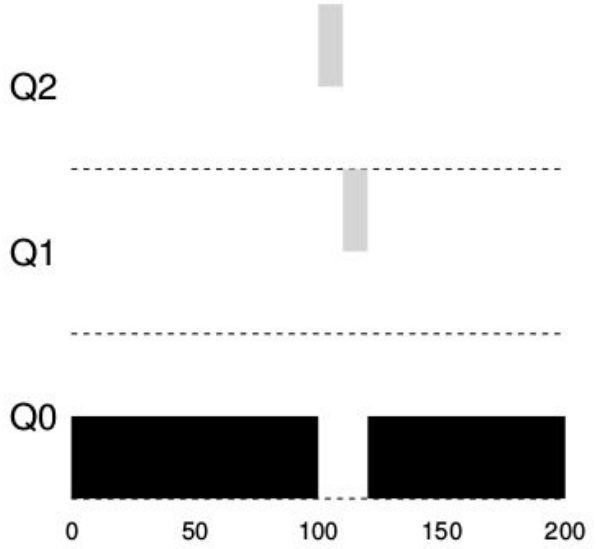
Primer intento: ¿Cómo cambiar la prioridad ?

Ejemplo 1: Una única tarea con ejecución larga.



Primer intento: ¿Cómo cambiar la prioridad ?

Ejemplo 1: Llega una tarea corta.



Primer intento: ¿Cómo cambiar la prioridad ?

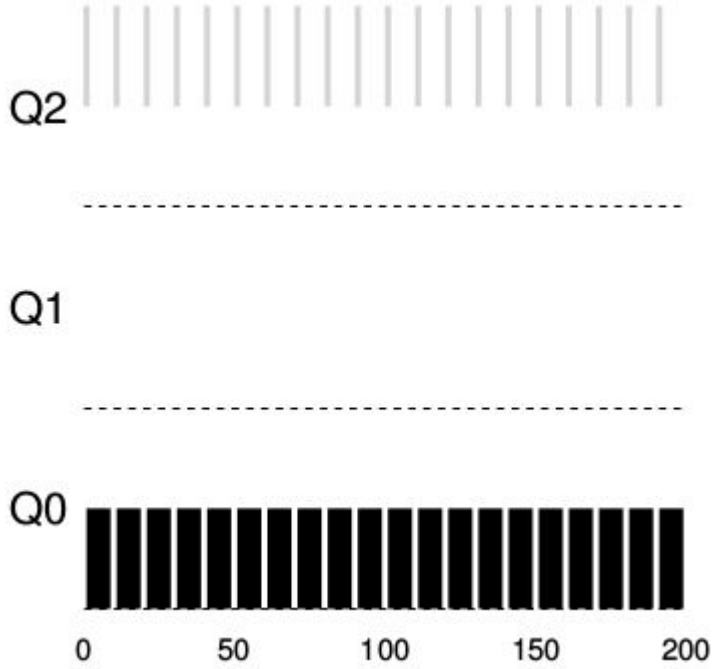
Existen 2 tareas, una de larga ejecución de CPU, A y B con una ejecución corta e interactiva. B tarda 20 milisegundos en ejecutarse.

De este ejemplo se puede ver una de las metas del algoritmo dado que no sabe si la tarea va a ser de corta o larga duración de ejecución, inicialmente asume que va a ser corta, entonces le da la mayor prioridad.

Si realmente es una tarea corta se va a ejecutar rápidamente y va a terminar, si no lo es se moverá lentamente hacia abajo en las colas de prioridad haciéndose que se parezca más a un proceso BATCH .

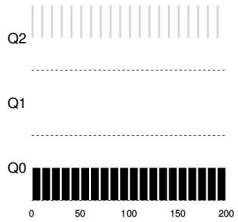
Primer intento: ¿Cómo cambiar la prioridad ?

Ejemplo 1: Qué pasa con la entrada y salida.



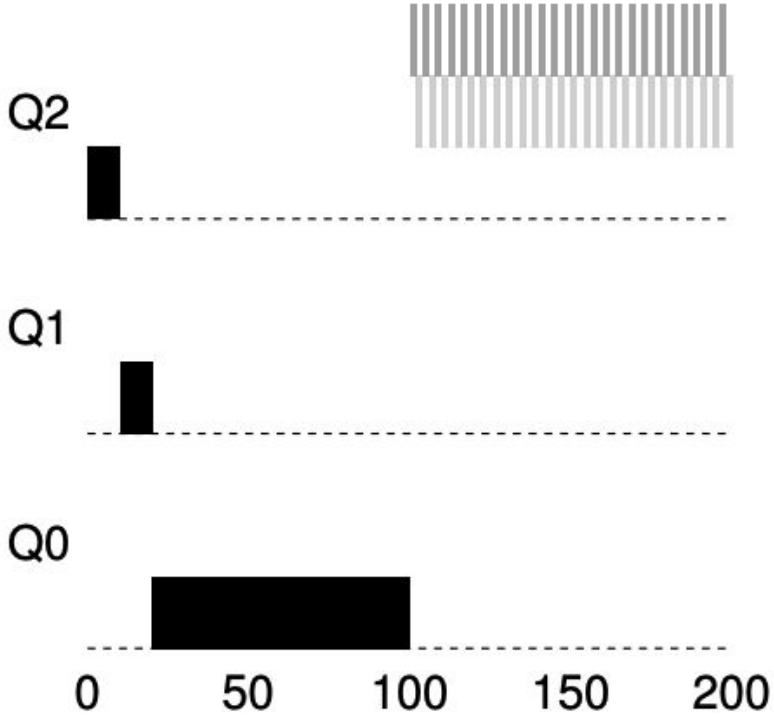
Primer intento: ¿Cómo cambiar la prioridad ?

Como se considera en la regla 4 si la tarea renuncia al uso del procesador antes de un time slice se mantiene en el mismo nivel de prioridad. El objetivo de esta regla es simple: si una tarea es interactiva por ejemplo entrada de datos por teclado o movimiento del mouse esta no va a requerir uso de CPU antes de que su time slice se complete en ese caso no será penalizada y mantendrá su mismo nivel de prioridad.



PROBLEMA con este Approach de MLFQ

Starvation (inhanición): Si hay demasiadas tareas interactivas en el sistema se van a combinar para consumir todo el tiempo del CPU y las tareas de larga duración nunca se van a ejecutar.



PROBLEMA con este Approach de MLFQ

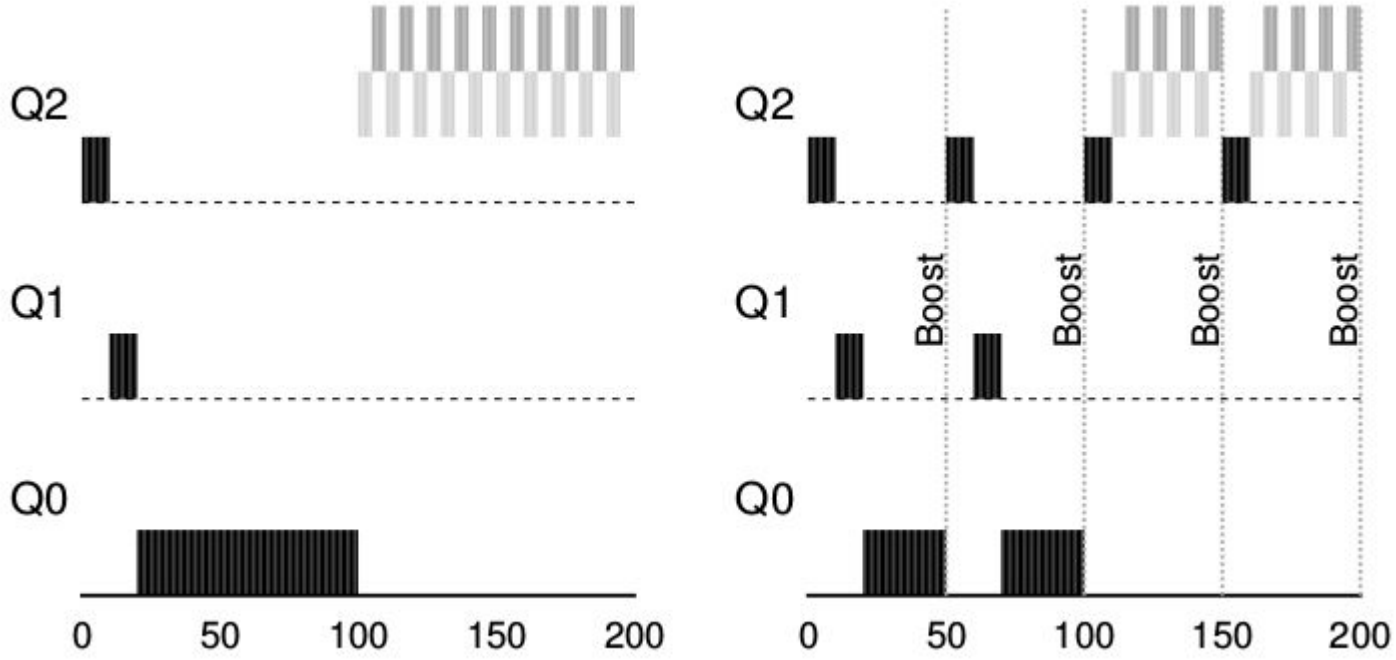
Un usuario inteligente podría “hackear” el scheduler; reescribir sus programas para obtener más tiempo de CPU. Por ejemplo: Antes de que termine el time slice se realiza una operación de entrada y salida entonces se va a relegar el uso de CPU haciendo esto se va a mantener la tarea en la misma cola de prioridad. Entonces la tarea puede monopolizar toda el tiempo de CPU.

Segundo Approach

¿ Cómo mejorar la prioridad? Para cambiar el problema del starvation y permitir que las tareas con larga utilización de CPU puedan progresar lo que se hace es aplicar una idea simple, se mejora la prioridad de todas las tareas en el sistema. Se agrega una nueva regla:

Regla 5: Después de cierto periodo de tiempo S, se mueven las tareas a la cola con más prioridad.

Segundo Approach: Boost



Segundo Approach

Haciendo esto se matan 2 pájaros de 1 tiro:

Se garantiza que los procesos no se van a starve: Al ubicarse en la cola tope con las otras tareas de alta prioridad estos se van a ejecutar utilizando round-robin y por ende en algún momento recibirá atención.

si un proceso que consume CPU se transforma en interactivo el planificador lo tratara como tal una vez que haya recibido el boost de prioridad.

Segundo Approach

Obviamente el agregado del periodo de tiempo **S** va a desembocar en la pregunta obvia: Cuánto debería ser el valor del tiempo **S**. Algunos investigadores suelen llamar a este tipo de valores dentro de un sistema **VOO-DOO CONSTANTs** porque parece que requieren cierta magia negra para ser determinados correctamente.

Este es el caso de **S**, si el valor es demasiado alto,los procesos que requieren mucha ejecución van a caer en starvation; si se setea a **S** con valores muy pequeños las tareas interactivas no van a poder compartir adecuadamente la CPU.

Segundo Approach

Se debe solucionar otro problema: Cómo prevenir que ventajeen (gaming) al planificador.

La solución es llevar una mejor contabilidad del tiempo de uso de la CPU en todos los niveles del MLFQ.

En lugar de que el planificador se **olvide** de cuanto time slice un determinado proceso utiliza en un determinado nivel el planificador debe hacer un seguimiento desde que un proceso ha sido asignado a una cola hasta que es trasladado a una cola de distinta prioridad.

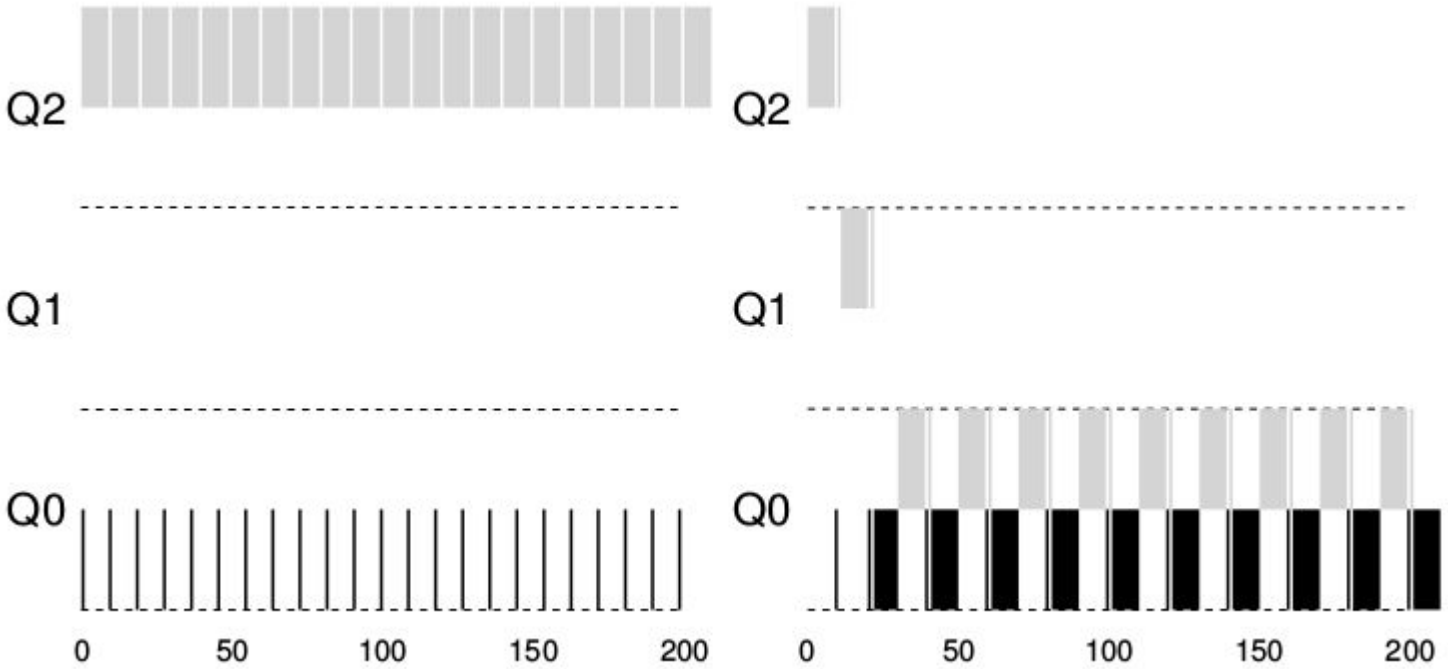
Cada proceso tiene un contador, que trackea cuando time slice van consumiendo en total, independientemente de las interrupciones.

Segundo Approach

Ya sea si usa su time slice de una o en pequeños trocitos, esto no importa por ende se reescriben las reglas 4a y 4b en una única regla:

> **Regla 4**: Una vez que una tarea consume su time-slice en un nivel dado (independientemente de cuántas veces haya renunciado al uso de la CPU) su prioridad se reduce; baja un nivel en la cola de prioridad.

Segundo Approach



Segundo Approach

Se vio la técnica de planificación conocida como multi-level feed back queue (MLFQ). Se puede ver porque es llamado así, tiene un conjunto de colas de multiniveles y utiliza feed back para determinar la prioridad de una tarea dada.

Se guía por la historia de cada proceso: observa como las tareas se comportan a través del tiempo y opera con ellas de acuerdo a ello.

Recapitulando

REGLA 1: si la prioridad(A) > prioridad(B), (A) se ejecuta y (B) no.

REGLA 2: si la prioridad(A) = prioridad(B), (A) y (B)se ejecutan en Round-Robin.

REGLA 3: Cuando una tarea llega al sistema se la coloca en la más alta prioridad.

REGLA 4: Una vez que una tarea consume su time-slice en un nivel dado (independientemente de cuántas veces haya renunciado al uso de la CPU) su prioridad se reduce; baja un nivel en la cola de prioridad.

REGLA 5: Después de un cierto periodo de tiempo S, se mueven todas las tareas a la cola con mas prioridad.

Bibliografía

Bibliografía

- **Operating Systems: Three Easy Pieces** - Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau - Arpaci-Dusseau Books - November, 2023 (Version 1.10)
  - Capítulo 7 - CPU Scheduling,
  - Capítulo 8 - Multi-level Feedback

- **xv6: a simple, Unix-like teaching operating system**- Russ Cox Frans Kaashoek Robert Morris - September 5, 2022
  - Capítulo 4 - Traps and system calls
  - Capítulo 7 - Scheduling