

Gestión de procesos

Yolanda Becerra Fontal
Juan José Costa Prats

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
BarcelonaTech
2019-2020 QP

Índice

- Conceptos previos
 - Procesos
- Visión de usuario
- Estructuras de datos
- Operaciones
 - Identificación
 - Cambio contexto
 - Creación
 - Destrucción
 - Planificación
 - Flujos (Threads)
 - Concurrencia y paralelismo
 - Sincronización entre procesos

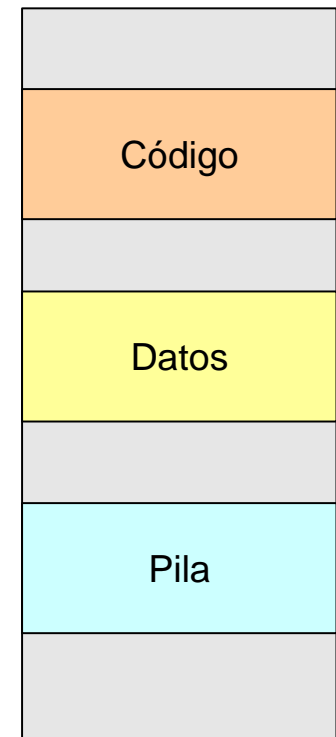
Concepto de proceso

- Un fichero ejecutable es algo estático, código almacenado en disco
- Cuando se carga en memoria y se empieza a ejecutar es un programa en ejecución o PROCESO.
- **Unidad de actividad** caracterizada por:
 - la ejecución de una secuencia de instrucciones,
 - un estado actual
 - y un conjunto asociado de recursos del sistema
- La definición de proceso engloba todo lo necesario para que un programa se ejecute.

Características de un proceso

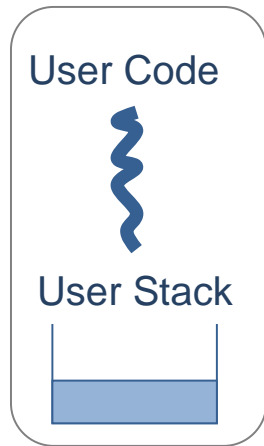
- Espacio lógico de @
 - Imagen en memoria del proceso
 - Código
 - Datos
 - Pila
 - Pila de kernel
- Contexto
 - Hardware
 - Pentium:
 - Valor de los registros
 - » Hay un conjunto único de registros para todos
 - TSS (Task Segment Selector)
 - Software
 - Info planificación, info dispositivos...
 - Linux: prioridad, quantum, canales....

Memoria

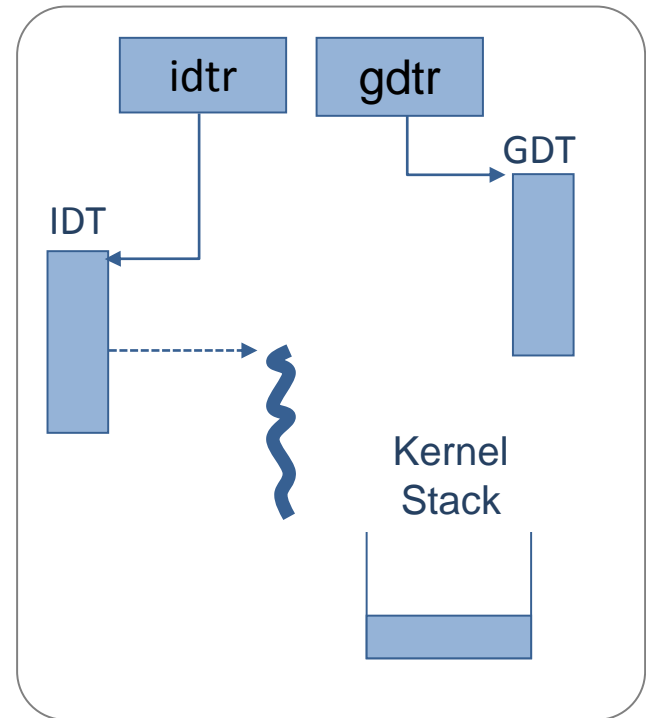


¿Donde estamos?

Procedure for entering the system

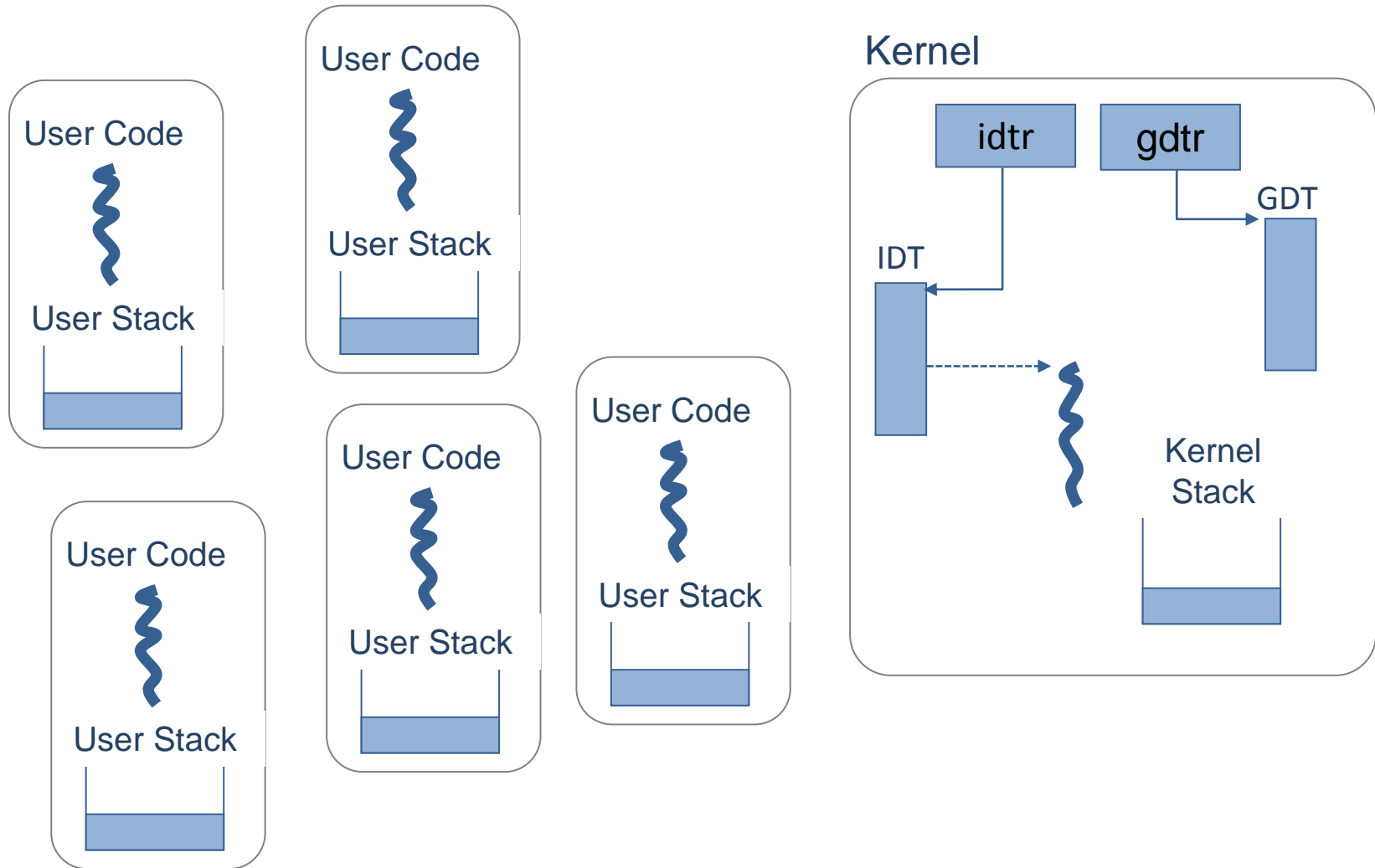


Kernel



¿Qué queremos?

Procedure for entering the system



Índice

- Conceptos previos
 - Procesos
- **Visión de usuario**
- Estructuras de datos
- Operaciones
 - Identificación
 - Cambio contexto
 - Creación
 - Destrucción
 - Planificación
 - Flujos (Threads)
 - Concurrencia y paralelismo
 - Sincronización entre procesos

Visión de usuario

- Creación
 - `int fork ()`
- Identificación
 - `int getpid ()`
- Destrucción
 - `void exit ()`

```
int pid;  
...  
pid = fork ();  
if (pid < 0) { error(); exit (); }  
if (pid == 0) { //hijo... }  
if (pid > 0) { //padre... }  
...
```


Índice

- Conceptos previos
 - Procesos
- Visión de usuario
- **Estructuras de datos**
- Operaciones
 - Identificación
 - Cambio contexto
 - Creación
 - Destrucción
 - Planificación
 - Flujos (Threads)
 - Concurrencia y paralelismo
 - Sincronización entre procesos

Representación de un proceso (sistema)

- Cada proceso tiene un Process Control Block (PCB)
 - Identificador del proceso
 - Estado del proceso
 - Recursos del proceso (páginas de memoria, ficheros abiertos, ...)
 - Estadísticas del proceso (cpu consumida, total memoria ocupada, ...)
 - Información de planificación (prioridad, quantum, ...)
 - Contexto de ejecución
 - Dentro del PCB
 - En la pila del proceso y en el PCB solo la @
 - ...
- Pila de sistema

Estructuras internas del sistema

- ¿Cómo implementamos esta información en el sistema?
- ¿Dónde lo guardamos?

Zeos: task_struct y task_union

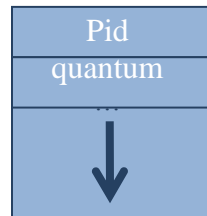
```
union task_union {  
    struct task_struct task;  
    unsigned long stack[1024];  
};
```

Kernel stack



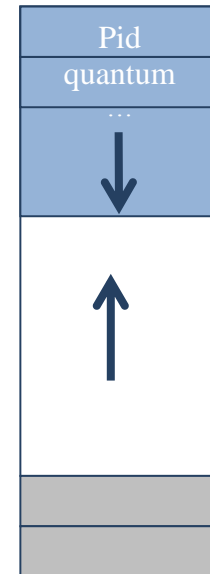
+

Task_struct



=

Task_union



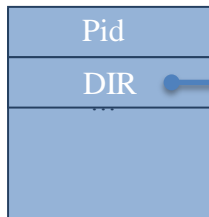
} task_struct

} kernel stack

Zeos: Espacio de direcciones

- Cada proceso tiene su propio espacio de @s
 - Restringimos espacio a 1 única tabla de paginas

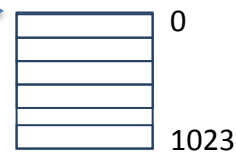
Task_struct



Directory



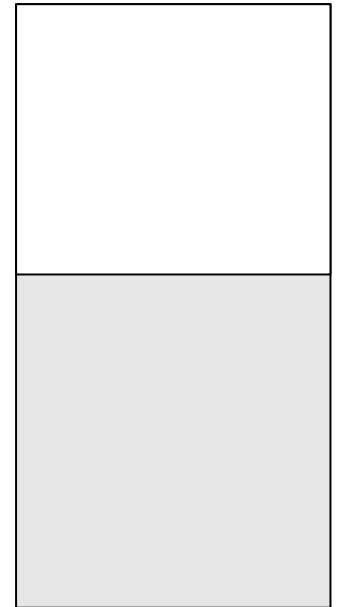
Page Table



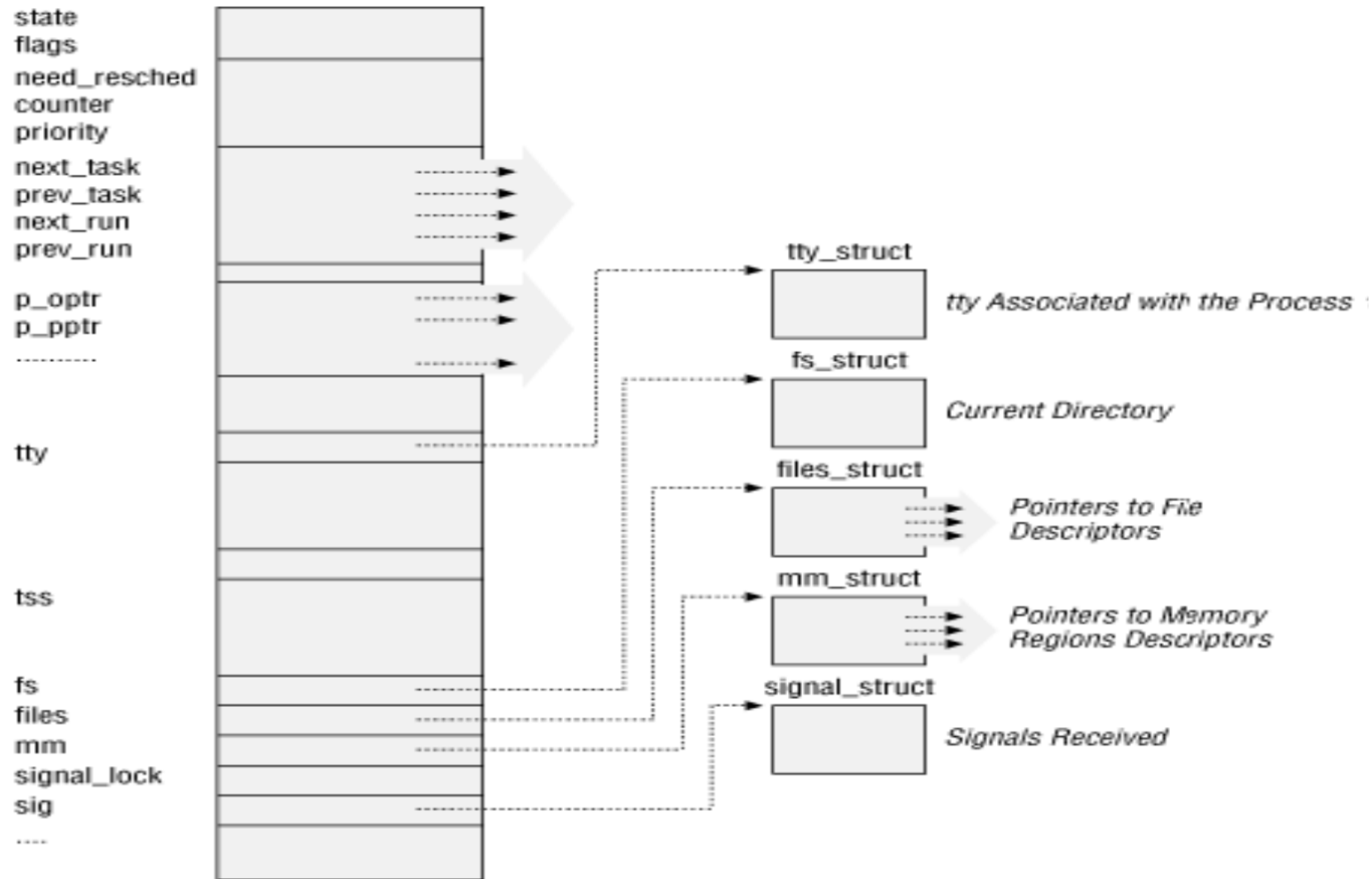
0x0

0x400000

0xFFFFFFFF



linux task_struct

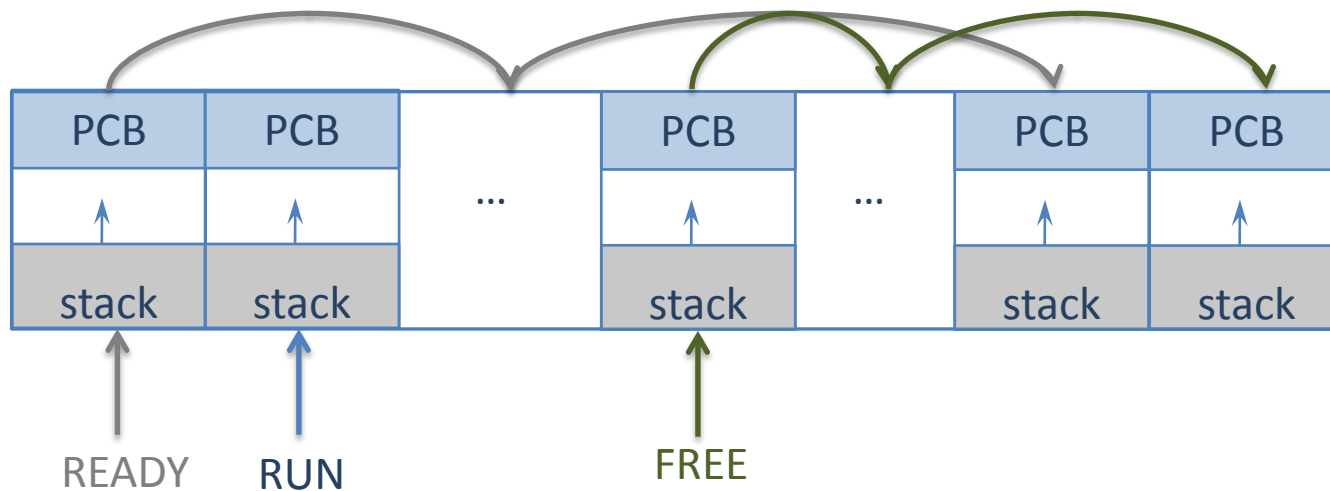


Recursos limitados: Listas

- Listas de procesos *READY*
 - Procesos que podrían ejecutarse pero que no tienen ninguna CPU disponible
 - Útil para planificación
- Listas de PCB's disponibles (*FREE*)
 - Para acelerar creación de procesos
- Listas de procesos esperando recursos
 - Dispositivos, sincronizaciones, ...

Particularidades de ZeOS

- No hay memoria dinámica ☹
 - Tabla con todos los posibles pcb's: task



Índice

- Conceptos previos
 - Procesos
- Visión de usuario
- Estructuras de datos
- **Operaciones**
 - Identificación
 - Cambio contexto
 - Creación
 - Destrucción
 - Planificación
 - Flujos (Threads)
 - Concurrencia y paralelismo
 - Sincronización entre procesos

Operaciones

- Identificación
- Creación de procesos
- Planificación
 - Cambio de contexto
- Destrucción de procesos

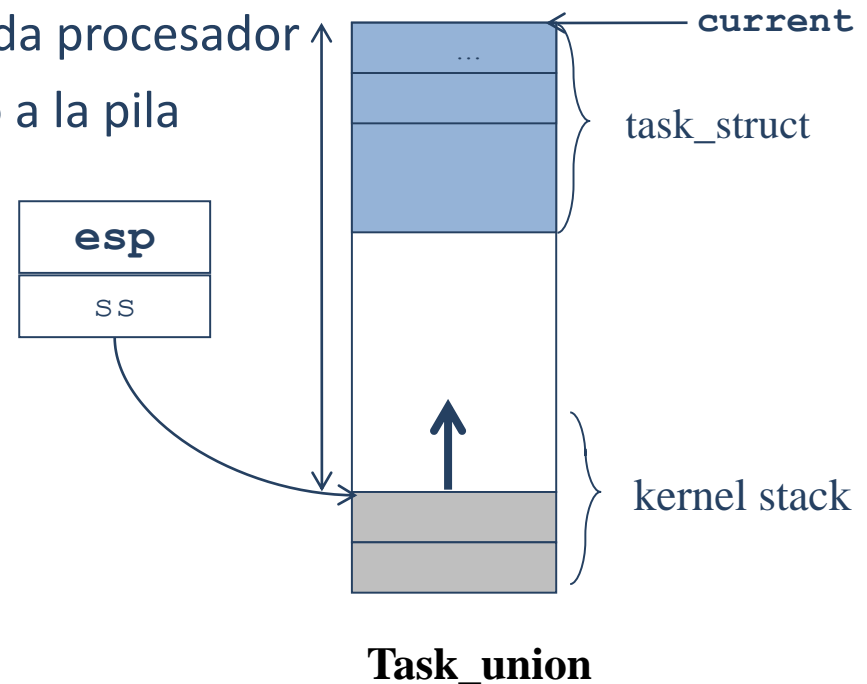
Identificación

- Como sabe el sistema que proceso está ejecutando?

Identificación

- Como sabe el sistema que proceso está ejecutando?

- Windows: puntero al actual por cada procesador
- Linux: se calcula usando el puntero a la pila
 - Pila de kernel y PCB comparten la misma página de memoria (4Kb)
 - Conocemos el puntero a la pila del proceso actual (esp)
 - Aplicando una mascara al esp podemos obtener la dirección del task_struct del proceso actual (current)



Índice

- Conceptos previos
 - Procesos
- Visión de usuario
- Estructuras de datos
- Operaciones
 - Identificación
 - Cambio contexto
 - Creación
 - Destrucción
 - Planificación
 - Flujos (Threads)
 - Concurrencia y paralelismo
 - Sincronización entre procesos

Cambio de contexto

- Suspender la ejecución del proceso actual y continuar la ejecución de otro proceso “previamente” suspendido
- Pasos
 - Guardar contexto ejecución proceso actual
 - Para poder restaurarlo más tarde
 - Restaurar contexto ejecución proceso suspendido
 - Espacio de direcciones
 - TSS
 - Pila de kernel
 - Contexto hardware

Guardar contexto proceso actual

- Qué hay que guardar y dónde?
 - Espacio de direcciones modo usuario
 - El contenido es privado y cada proceso tiene el suyo
 - En el PCB ya tenemos siempre la información sobre donde se encuentra
 - No hace falta guardarlo
 - Espacio de direcciones modo kernel
 - Pila de kernel
 - El contenido es privado y cada proceso tiene la suya
 - » No hace falta guardarlo
 - TSS (@ de la pila de kernel)
 - La TSS es compartida por todos los procesos, se sobrescribe en cada cambio de contexto
 - La @ de la pila se calcula a partir de la @ del PCB
 - » No hace falta guardarlo.
 - Contexto hardware
 - El hw es compartido por todos los procesos, se sobrescribe en cada cambio de contexto
 - **Hay que guardar el contexto hw y cómo acceder a él**
 - Linux
 - » Guarda el contexto en la pila de kernel
 - » Guarda en el pcb la posición en la pila para acceder a él

Restaurar contexto proceso suspendido

- Qué hay que restaurar y dónde?
 - Espacio de direcciones modo usuario
 - Actualizar las estructuras de la MMU
 - Dar acceso a la espacio de direcciones físico del proceso (código, pila, datos, ...)
 - Espacio de direcciones modo kernel: TSS
 - Tiene que apuntar a la base de la pila de sistema del proceso (esp0)
 - Contexto Hardware
 - Acceder al contexto guardado y sobrescribir el hw con esos valores
 - Linux
 - » Hay que cambiar el esp para que apunte al contexto guardado del proceso
 - » Restaurar todo el contexto
 - Pasar a ejecutar el código del nuevo proceso
 - Al restaurar el contexto se debe cargar en el PC la dirección del código a ejecutar

Operaciones: cambio de contexto

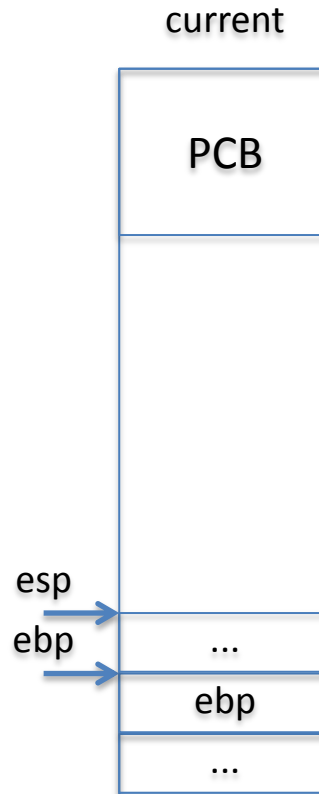
Implementación task_switch

- Como una rutina de C normal
 - *void task_switch (task_struct * new)*
- Usa el enlace dinámico para cambiar el contexto
 1. Restaurar espacio direcciones del nuevo proceso
 2. Cambiar la TSS para que apunte a la pila de *new*
 3. Guardar un puntero al enlace dinámico (EBP) en el PCB
 4. Cambiar a la pila del nuevo proceso
 - A la dirección guardada en el PCB
 - O sea donde empieza el enlace dinámico
 5. Deshacer el enlace dinámico
 6. Y retornar (en el contexto del nuevo proceso)

Operaciones: cambio de contexto

Cambio de contexto

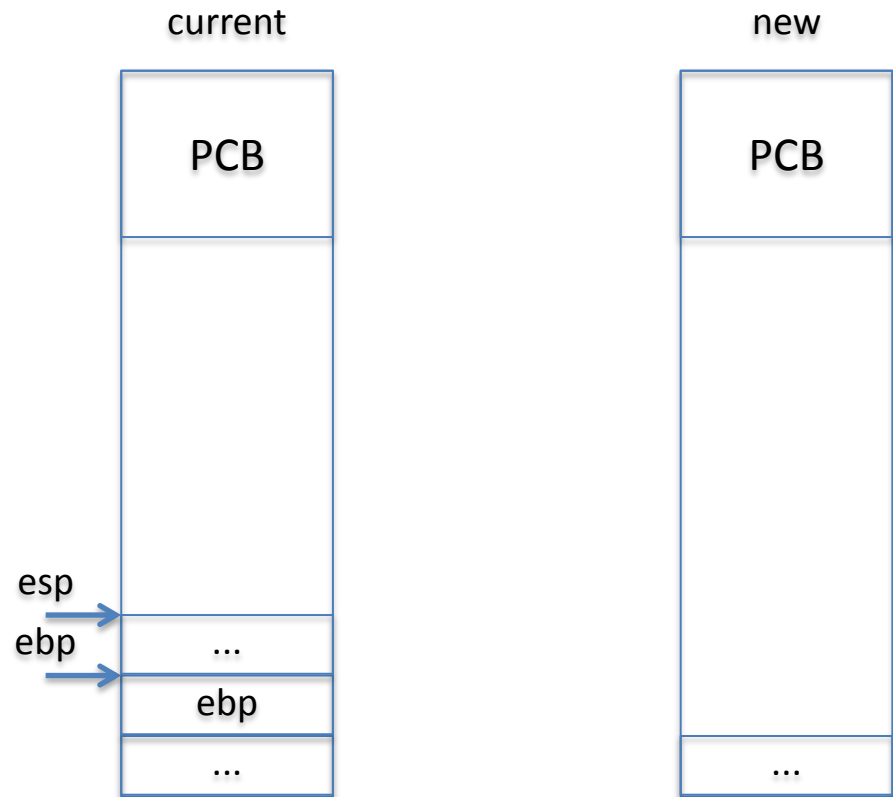
- Proceso ejecutando código sistema



Operaciones: cambio de contexto

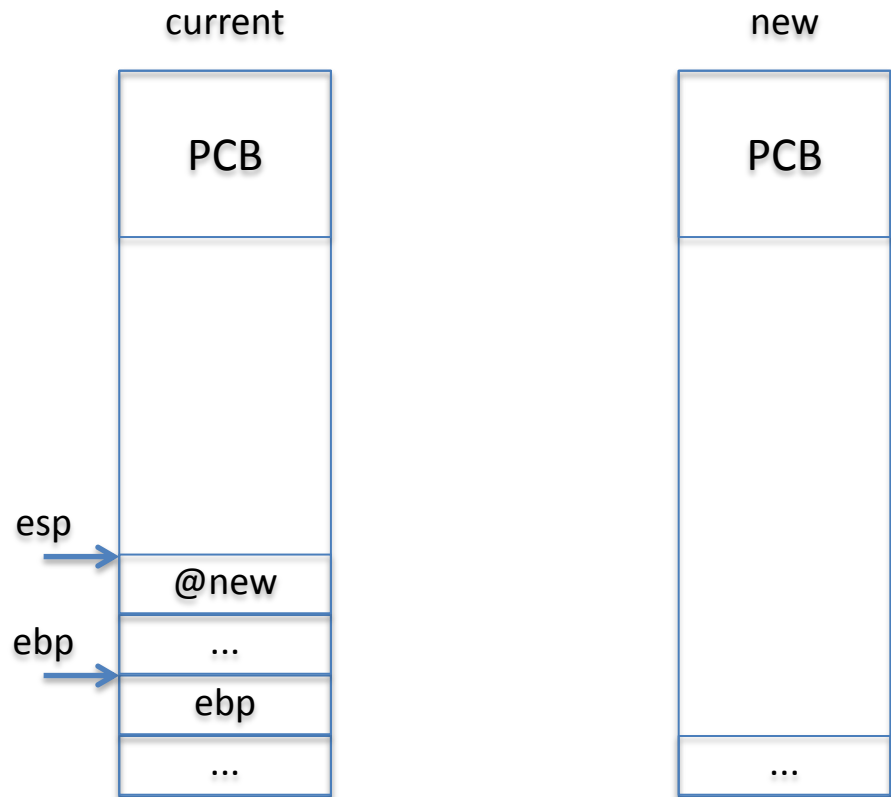
Cambio de contexto

- `task_switch(new)`



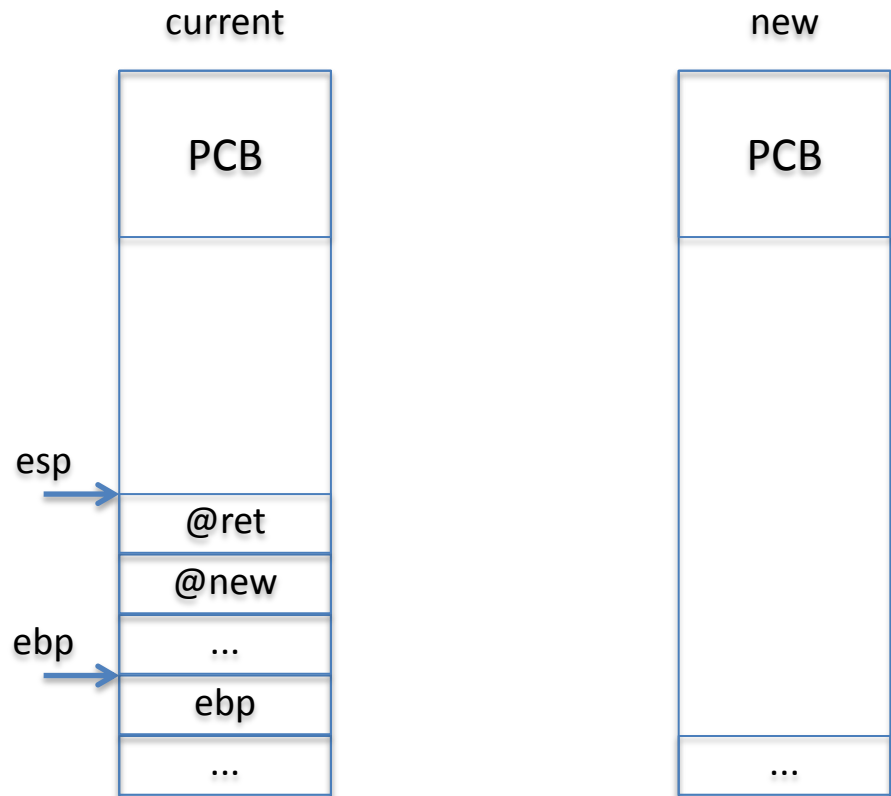
Cambio de contexto

- `task_switch(new)`
 - `pushl @new`



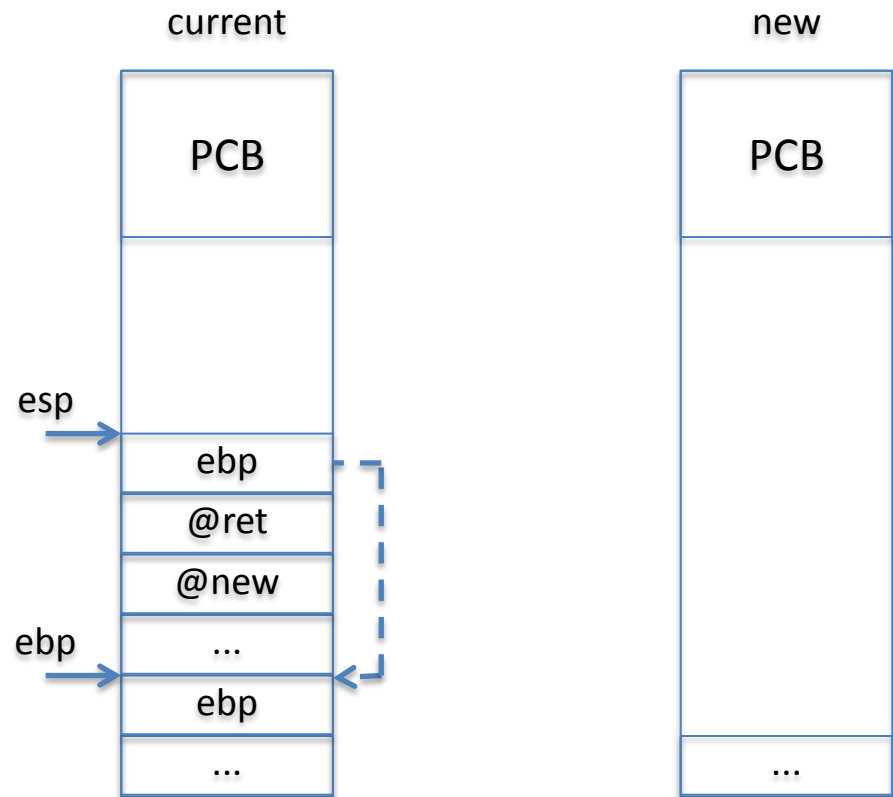
Cambio de contexto

- `task_switch(new)`
 - `pushl @new`
 - `call task_switch`



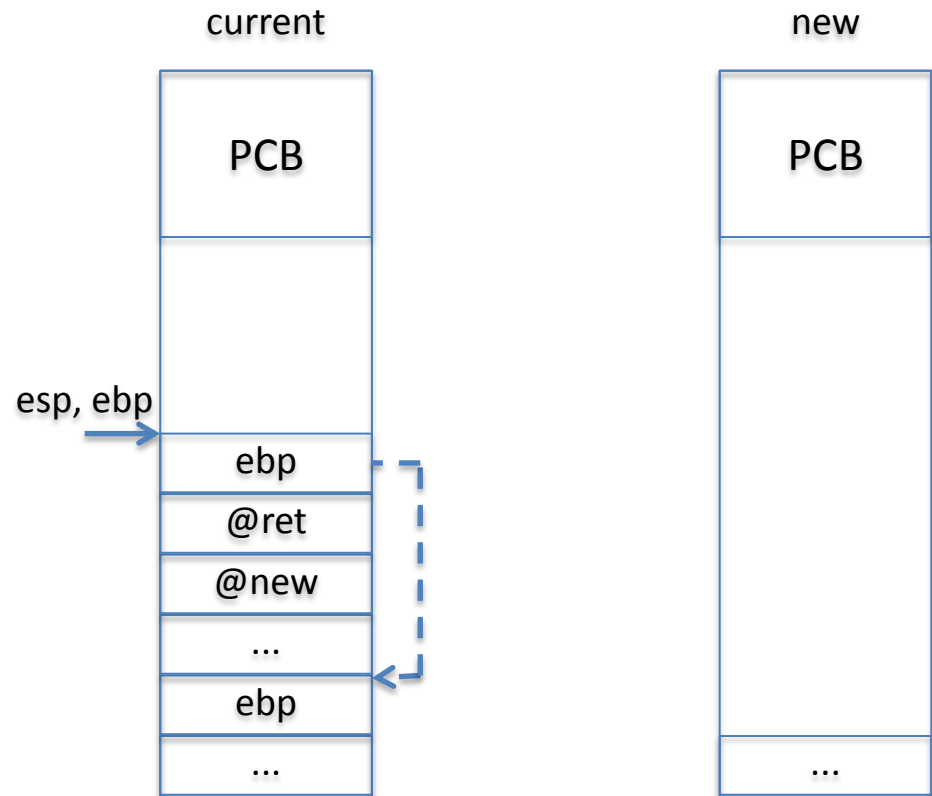
Cambio de contexto

- dynamic link
 - pushl ebp



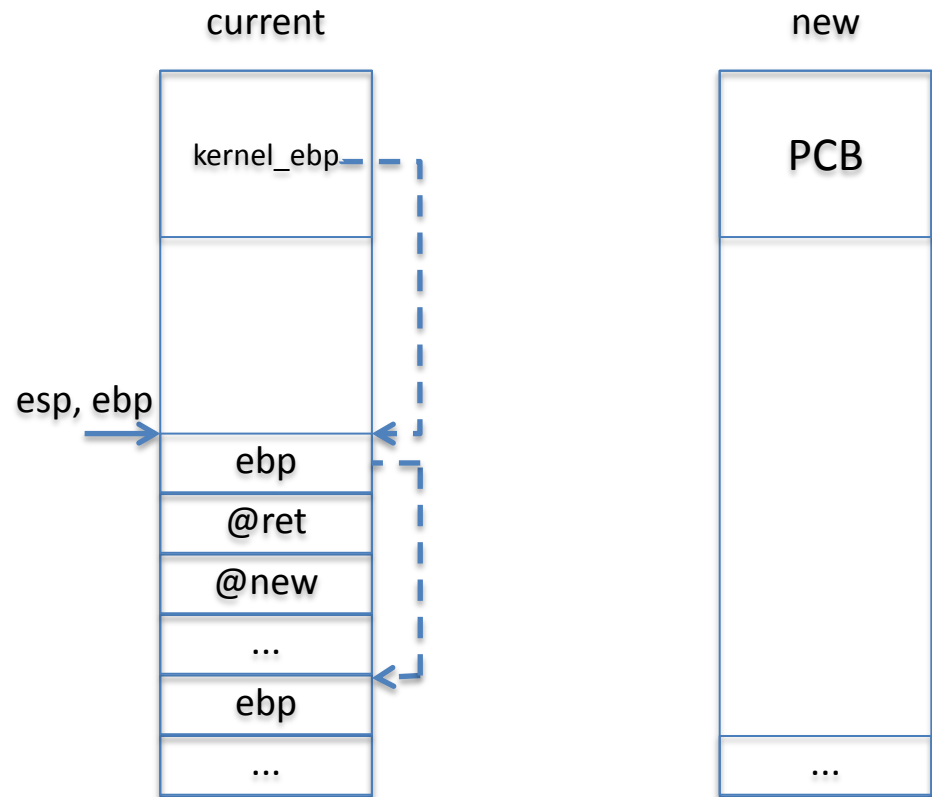
Cambio de contexto

- dynamic link
 - pushl ebp
 - $ebp \leftarrow esp$



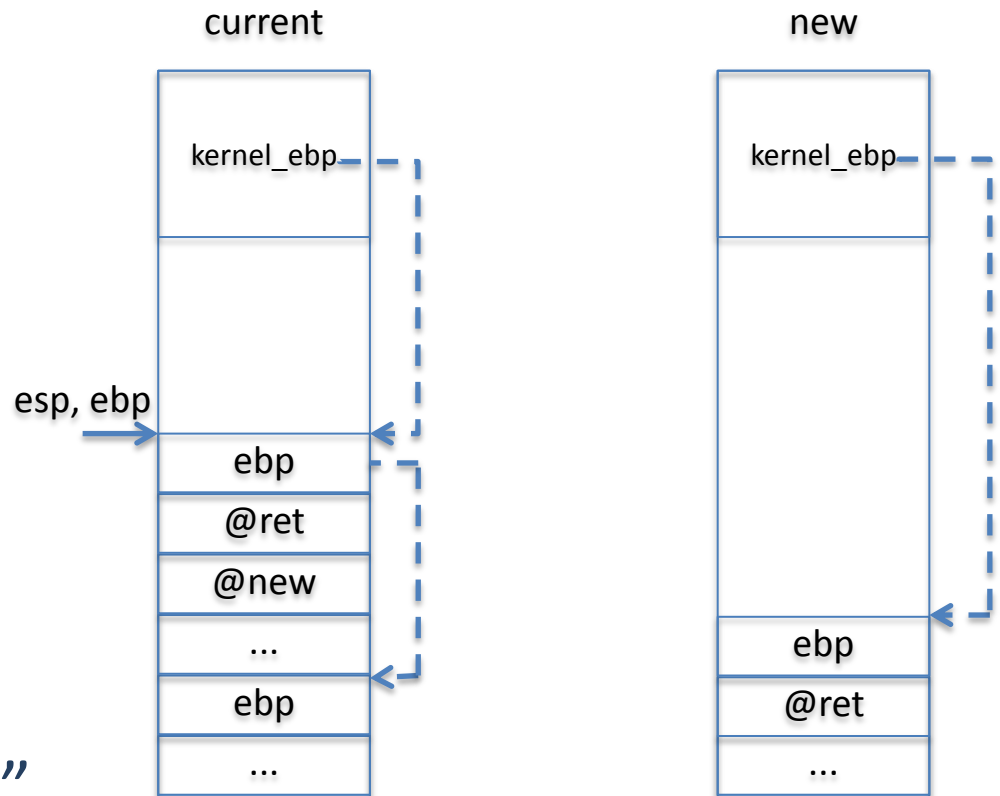
Cambio de contexto

- dynamic link
 - pushl ebp
 - ebp <- esp
 - kernel_ebp <- ebp



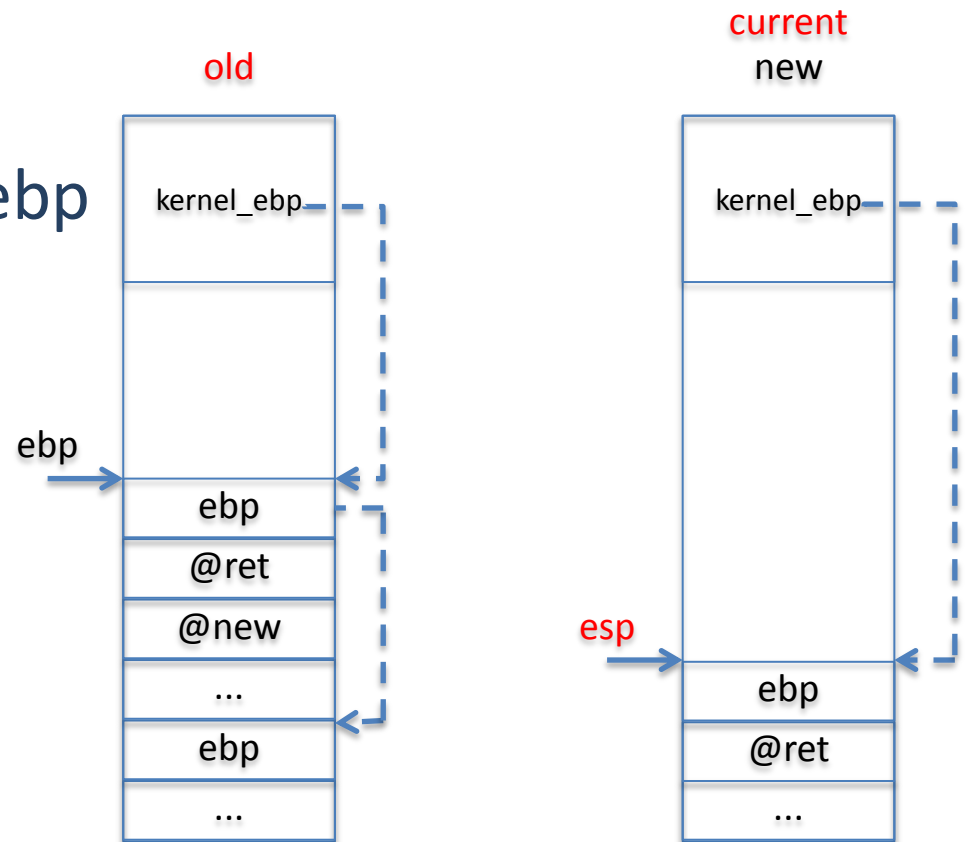
Cambio de contexto

- dynamic link
 - pushl ebp
 - ebp <- esp
 - kernel_ebp <- ebp
- *New* tiene una pila equivalente
 - previamente “ha hecho *task_switch*”



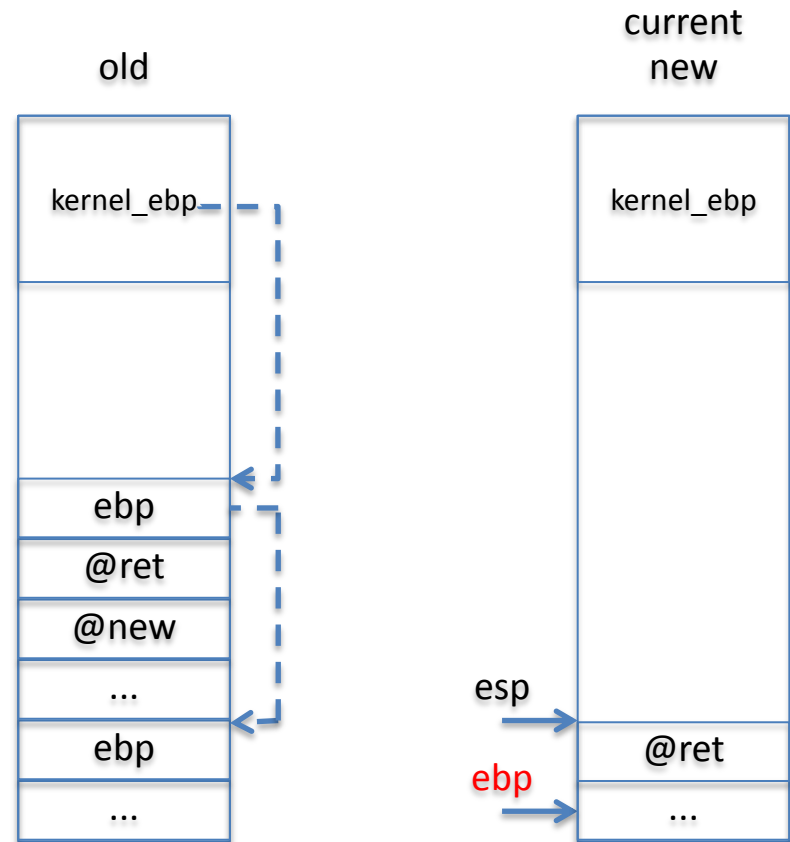
Cambio de contexto

- Cambio pila
 - $esp \leftarrow new.kernel_ebp$



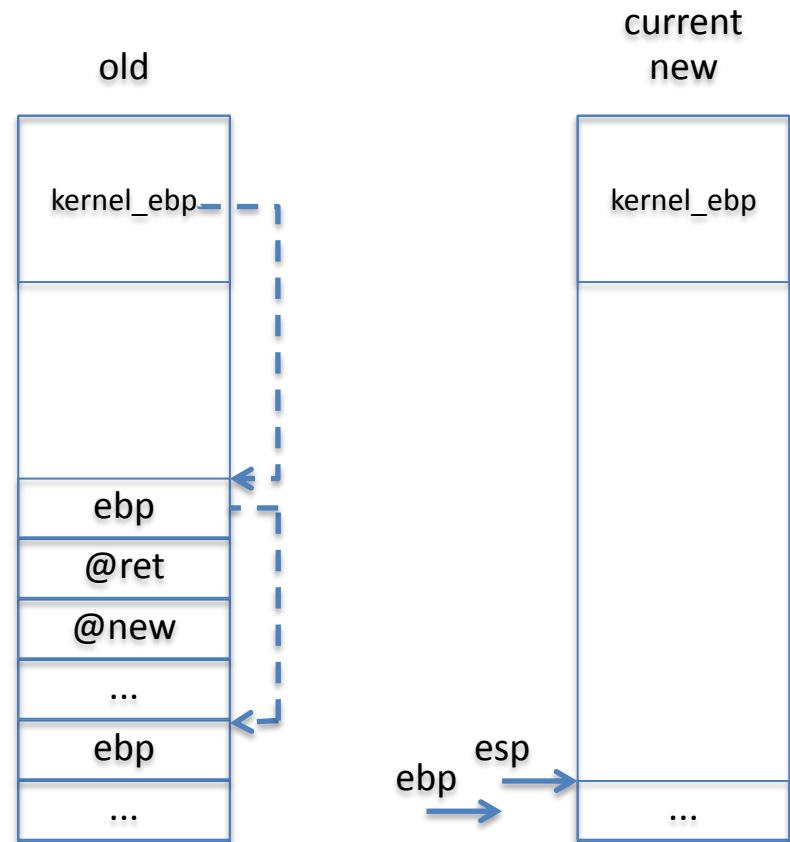
Cambio de contexto

- Cambio pila
 - `esp <- new.kernel_ebp`
- Deshacer enlace din.
 - `popl ebp`



Cambio de contexto

- Cambio pila
 - $esp \leftarrow new.kernel_ebp$
- Deshacer enlace din.
 - `popl ebp`
- Salir de la función (continuando en el código de new)
 - `ret: eip` ← top pila



Zeos: Cambio de contexto

- Detalle implementación
- Compilador guarda automáticamente registros ESI, EDI, y EBX si se modifican en una función
 - En este código no se modifican...
 - ➔ ...por lo tanto no los guarda!
 - Solución: *Wrapper* que los guarde/restaure manualmente:
 - task_switch (new)
 - save ESI, EDI, EBX
 - call inner_task_switch (new) ← "antiguo"
 - restore ESI, EDI, EBX

Índice

- Conceptos previos
 - Procesos
- Visión de usuario
- Estructuras de datos
- Operaciones
 - Identificación
 - Cambio contexto
 - Creación
 - Destrucción
 - Planificación
 - Flujos (Threads)
 - Concurrencia y paralelismo
 - Sincronización entre procesos

Creación de procesos

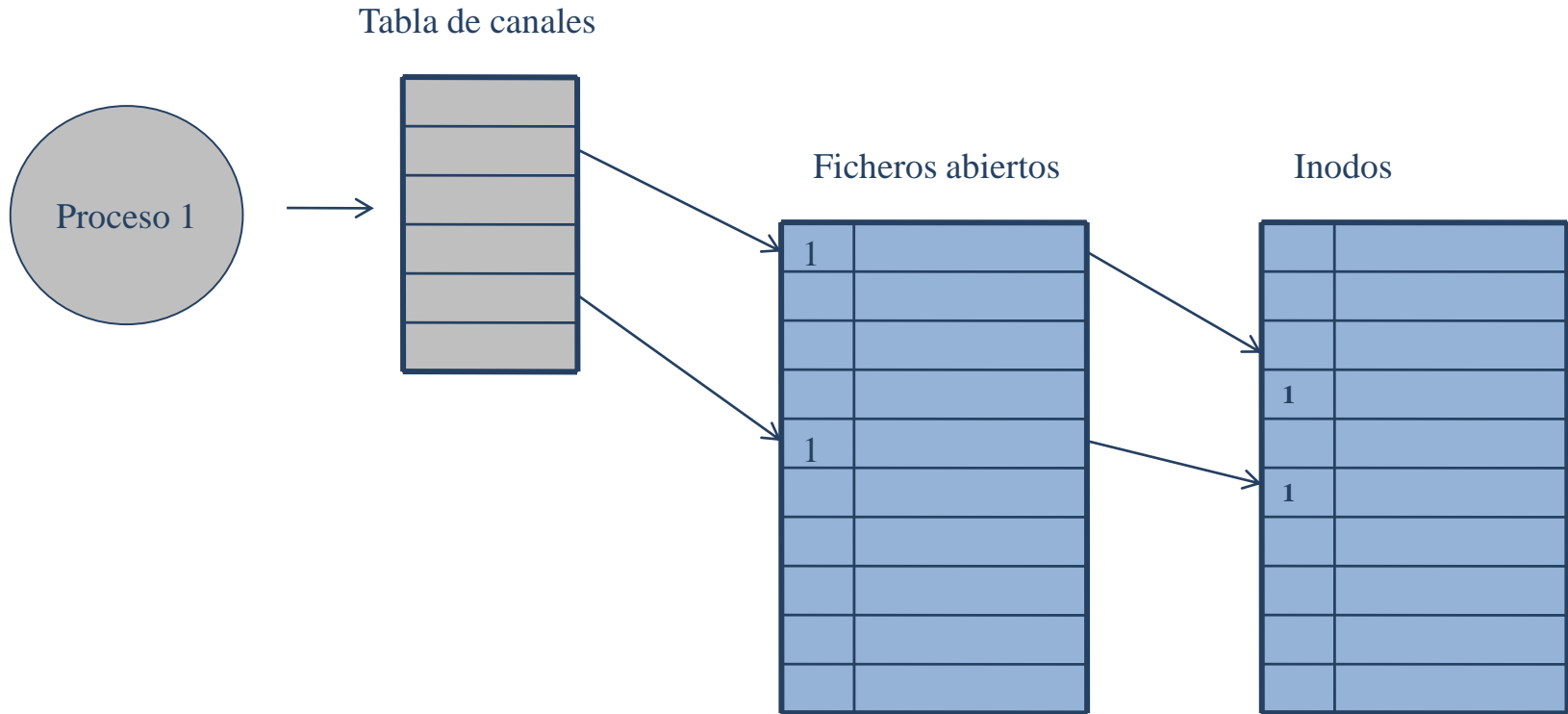
- Buscar PCB libre
- Inicializar PCB
- Inicializar espacio de direcciones
- Encolar PCB (planificación)

Inicializar PCB

- Asignar PID
- Actualizar campos del proceso (2 opciones)
 1. Inicialización de nuevos recursos, o
 2. Heredado del padre (UNIX)
 - Tabla de canales, programación de signals
 - La replicación de estructuras se tiene que hacer de forma “segura”
 - Asegurar la integridad del sistema

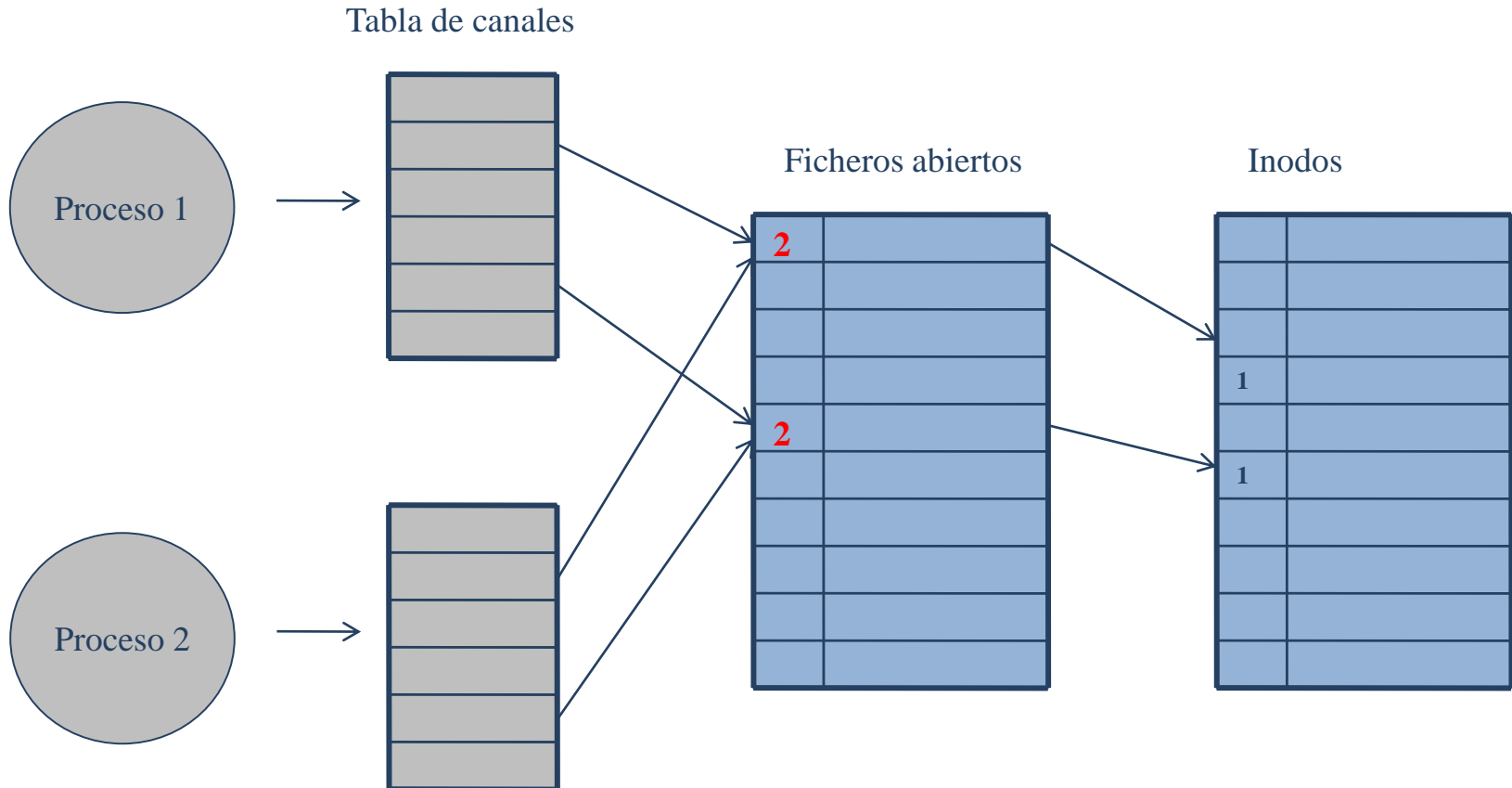
Duplicar tabla de canales

Operaciones: creación de procesos



Duplicar tabla de canales

Operaciones: creación de procesos



Inicializar espacio direcciones

- Cargar un ejecutable en memoria (loader), o
- Heredado del padre (UNIX)

Carga de un ejecutable

- Poner en memoria física la información del fichero ejecutable y dar control a la primera instrucción del programa
 - Reservar memoria física
 - Copiar código
 - Inicializar datos
 - Expandir pila y datos no inicializados

Heredado del padre (UNIX)

- 2 métodos:
 - Copia de toda la memoria del proceso padre
 - Implica reservar memoria física
 - Compartición del espacio de direcciones con el proceso padre
 - Flujos (clone en linux)

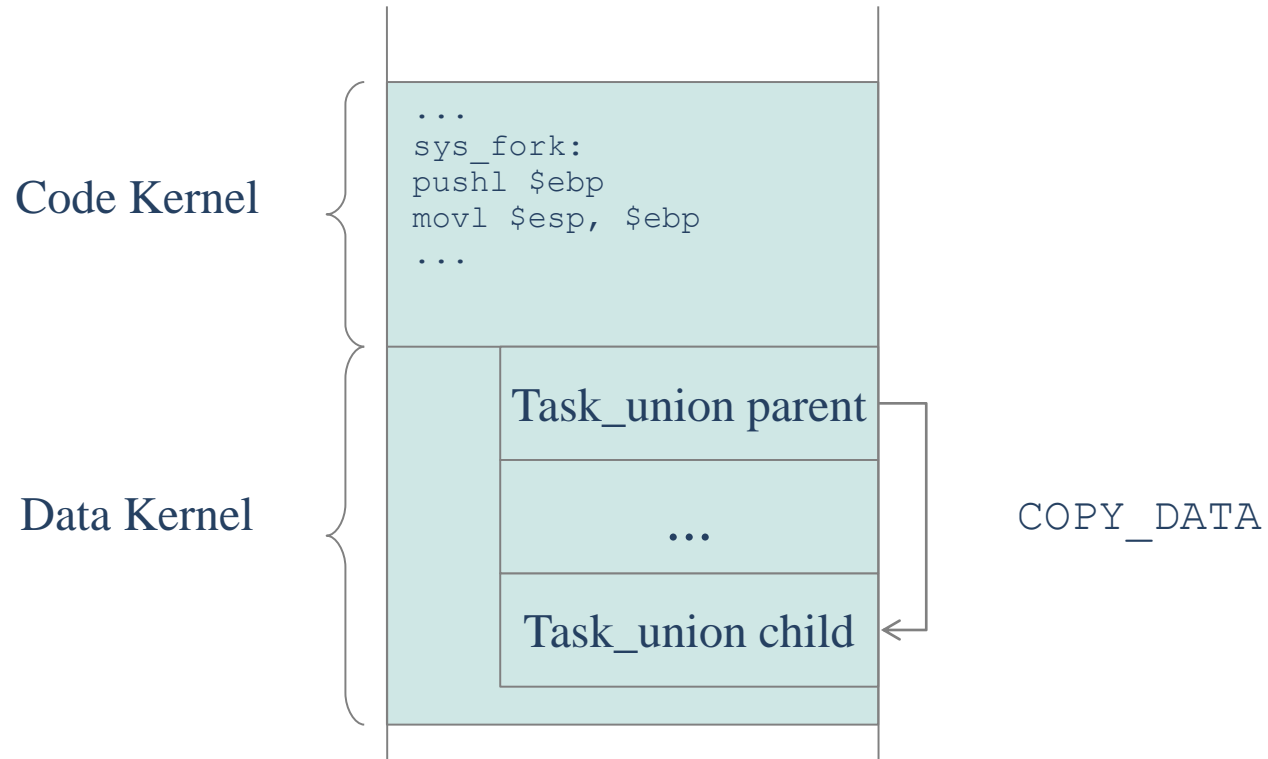
ZeOS: Fork

- Obtener PCB libre (lista FREE)
- Asignar un nuevo PID
- Heredar los datos de sistema (PCB i pila)
- Asignar un nuevo espacio de @: directorio
- Heredar los datos de usuario
- Actualizar task_union hijo
- Insertar proceso en la lista de procesos READY
- Devolver el pid del nuevo proceso creado
 - 0 al hijo

Fork: Herencia datos sistema

- Código y datos del sistema son siempre accesibles desde modo sistema
- Copiar el task_union (stack + task_struct) desde el padre al hijo
 - Heredamos información del PCB (task_struct)
 - Pero también el contexto del proceso padre (en la pila)
 - Para poder restaurarlo a posteriori

Fork: Herencia datos sistema

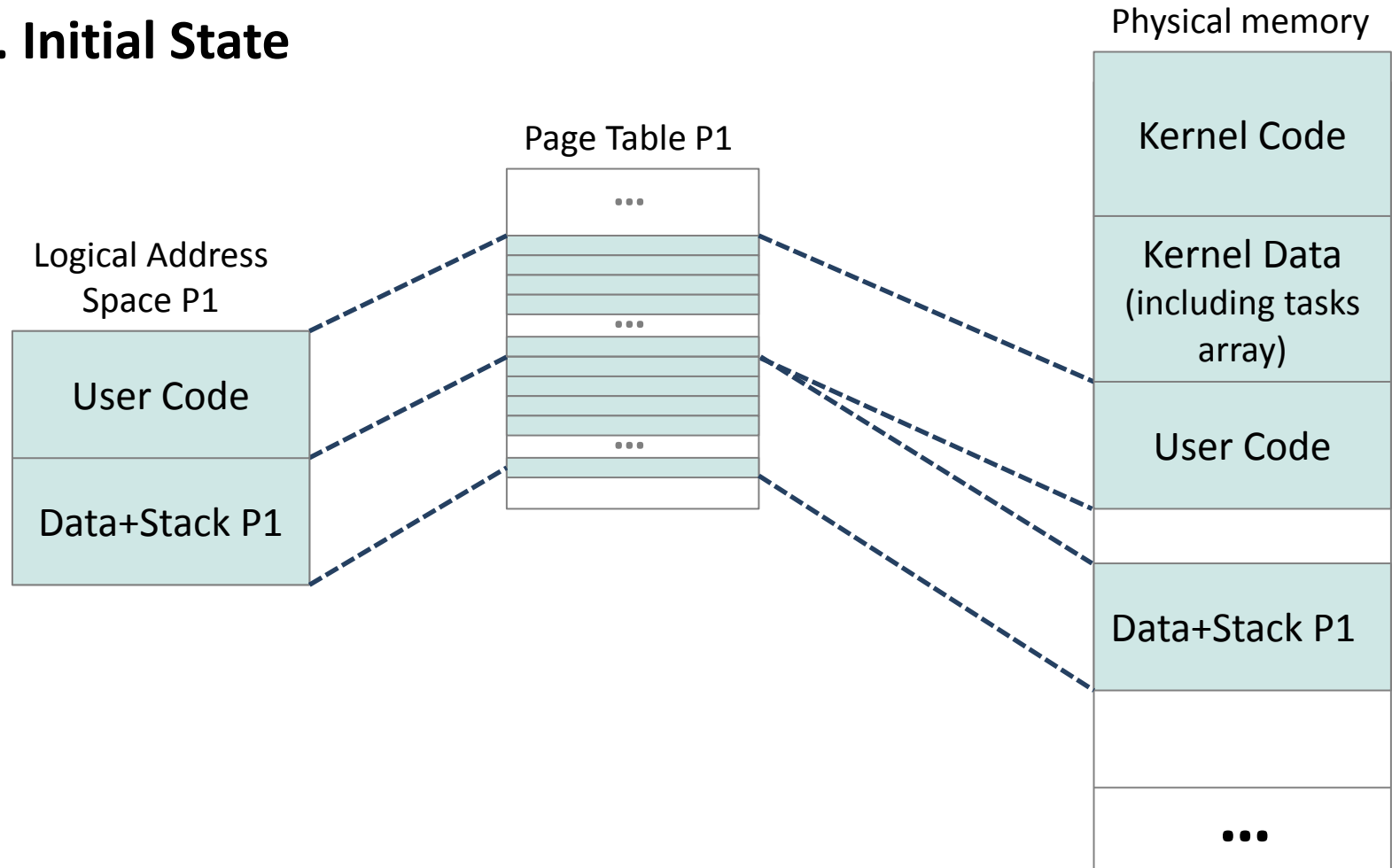


Fork: Herencia datos de usuario

- Código usuario: Compartido entre el padre y el hijo
 - Actualizar tabla de páginas del hijo
- Datos y pila de usuario: Privados
 - Hay que crear una zona de memoria nueva para el hijo
 - Pero por defecto sólo podemos acceder a los del padre...
- Pasos para copiar los datos del padre al hijo
 - Buscar frames libres para guardar las páginas de datos y pila del hijo y actualizar la tabla de páginas del hijo
 - Permitir al padre acceso a los frames del hijo
 - HAY QUE MODIFICAR LA TABLA DE PAGINAS
 - Copiar los datos y pila del padre al hijo
 - Quitar el acceso al padre a los frames del hijo

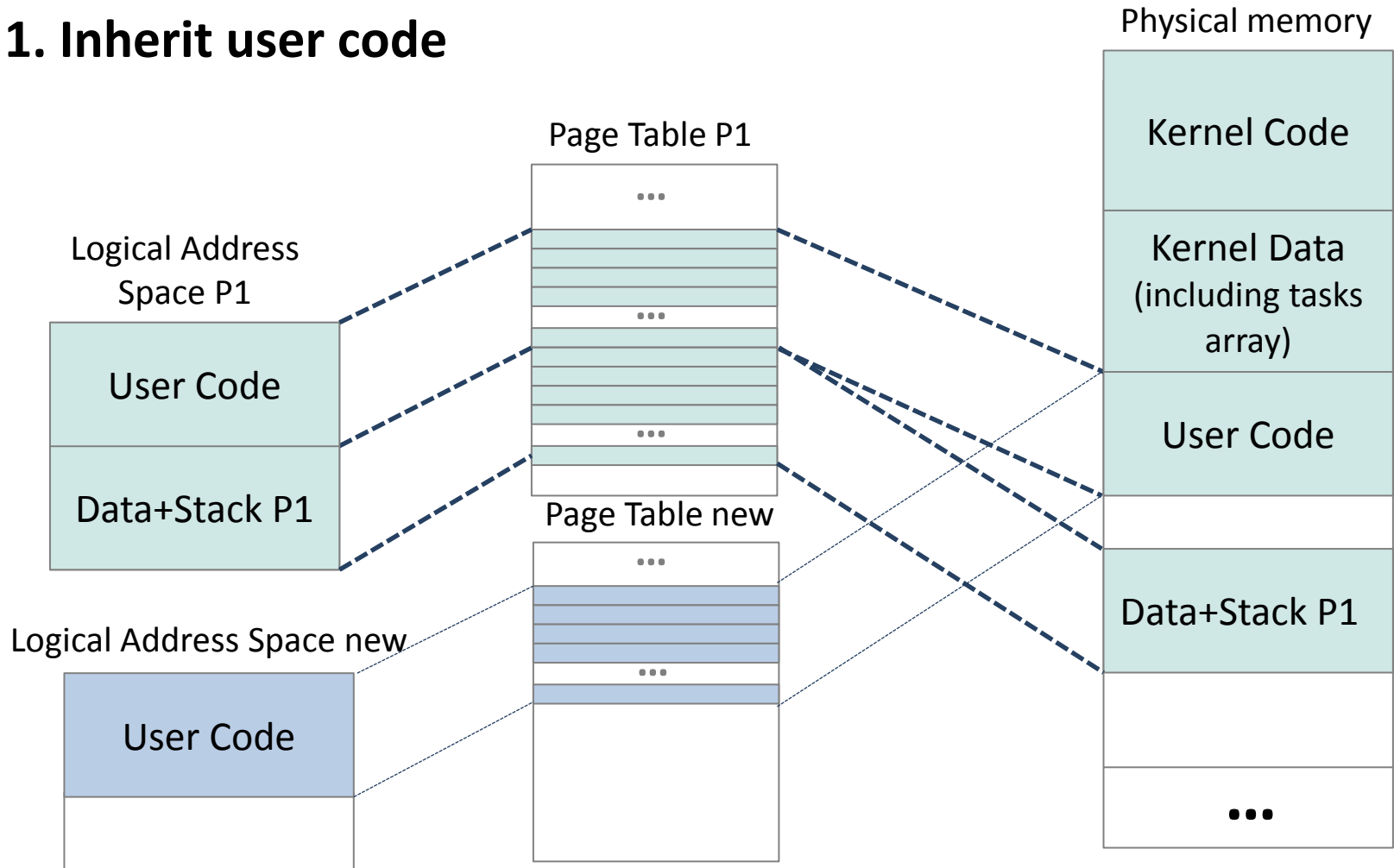
Fork: Herencia datos de usuario

0. Initial State



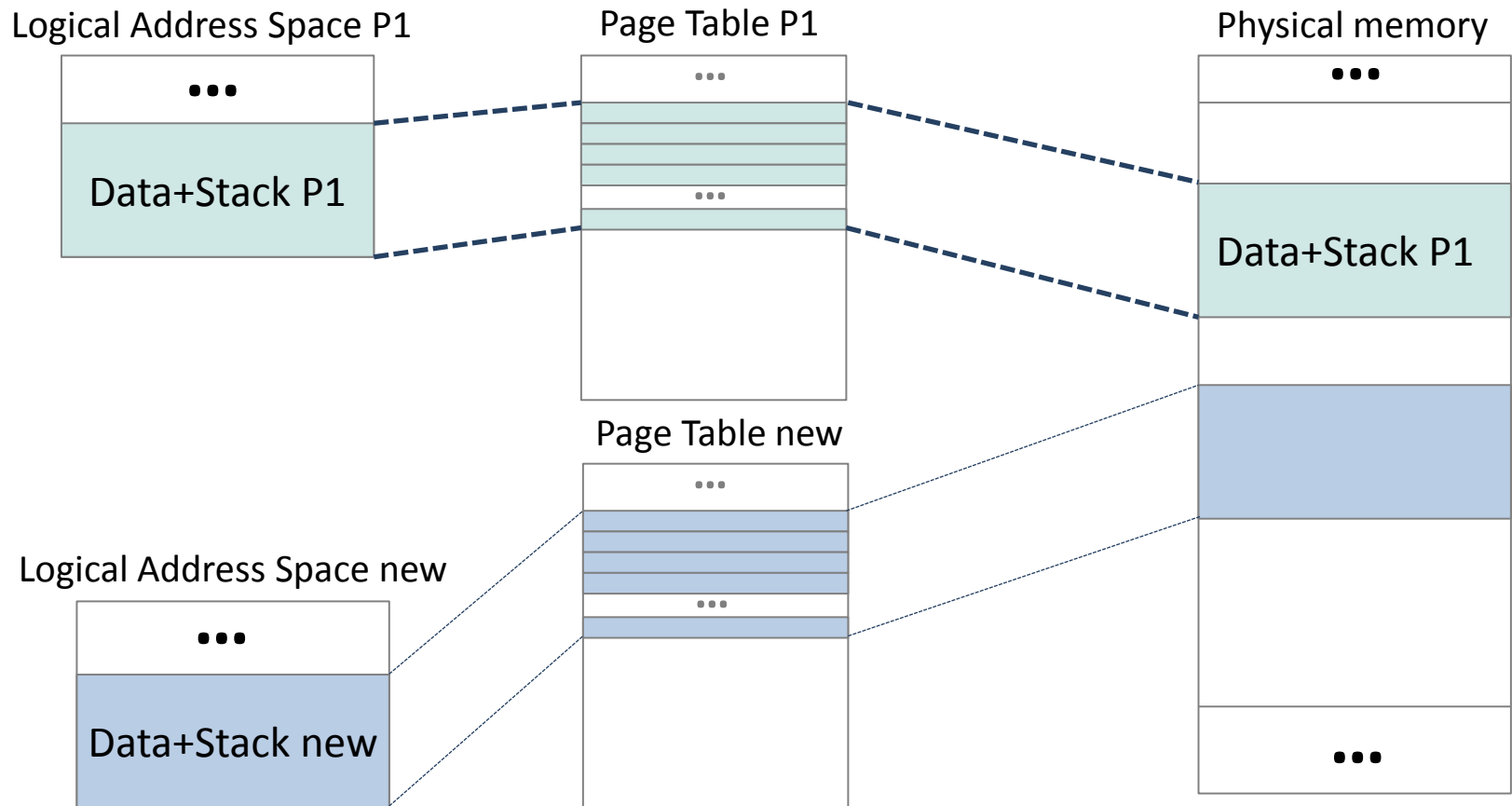
Fork: Herencia datos de usuario

1. Inherit user code



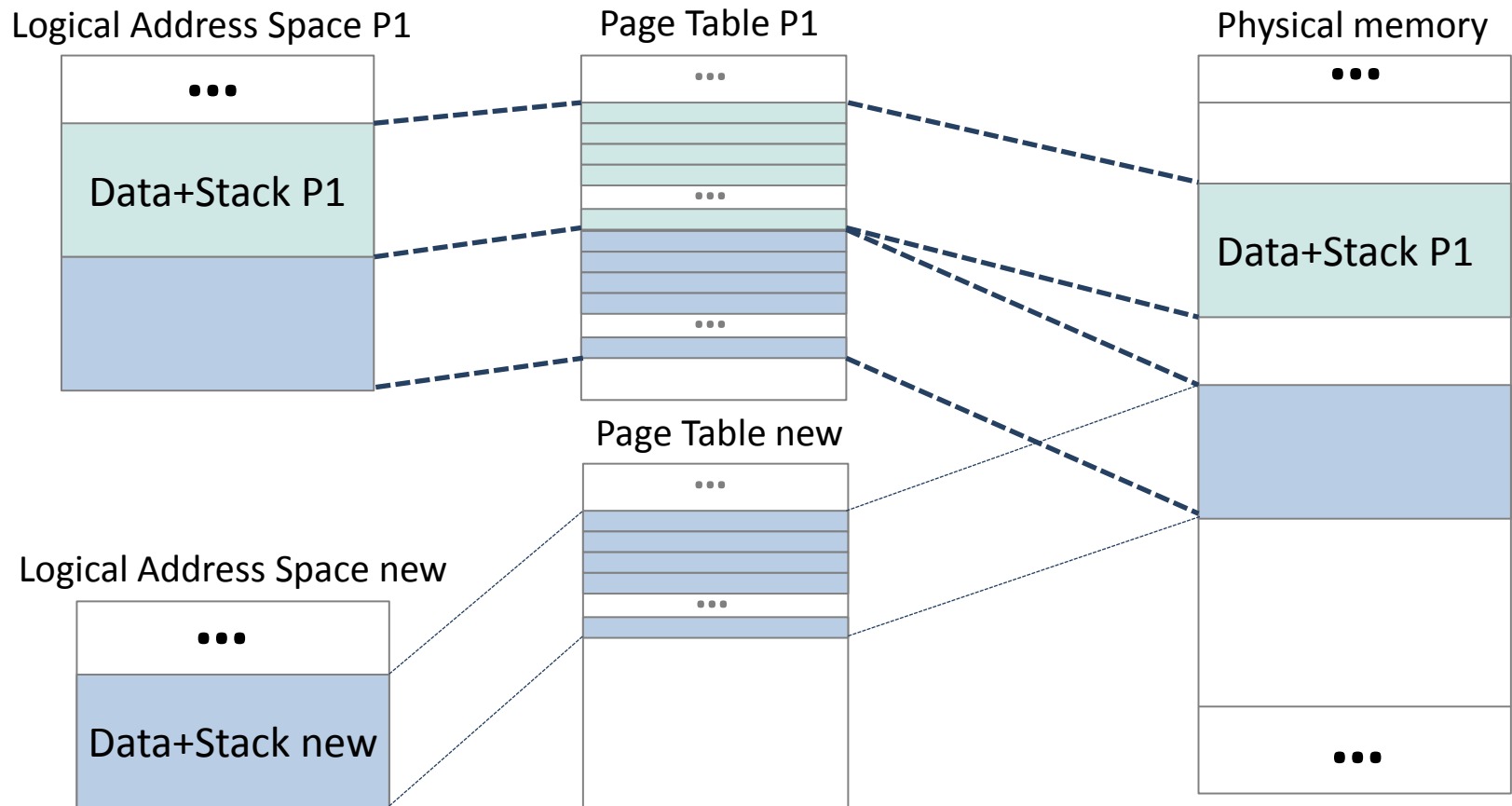
Fork: Herencia datos de usuario

2. Allocate frames and update page table of the child process



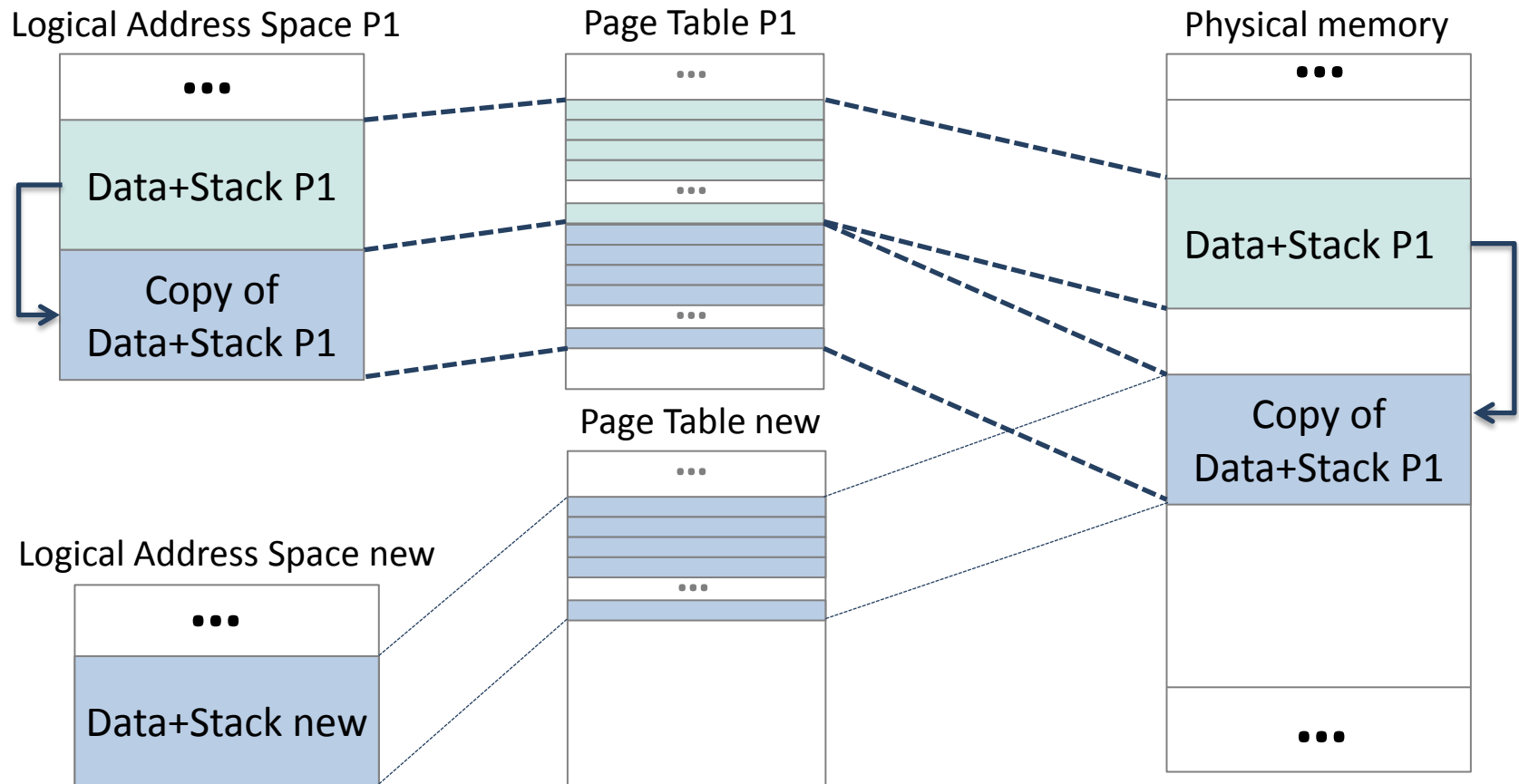
Fork: Herencia datos de usuario

3. Grant temporary access to parent process



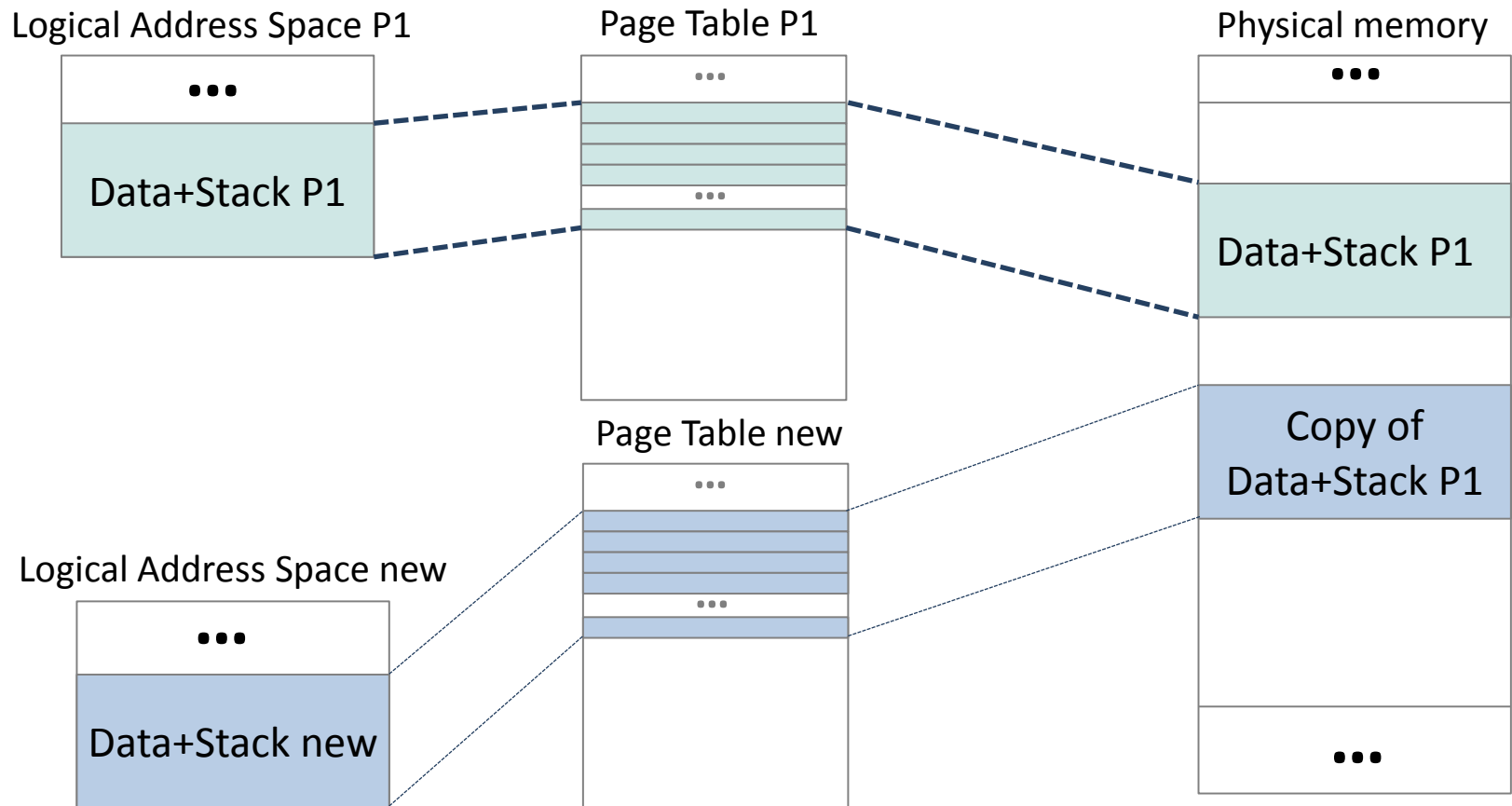
Fork: Herencia datos de usuario

4. Copy parent data+stack to child



Fork: Herencia datos de usuario

5. Deny Access to parent (and think about tlb!!)

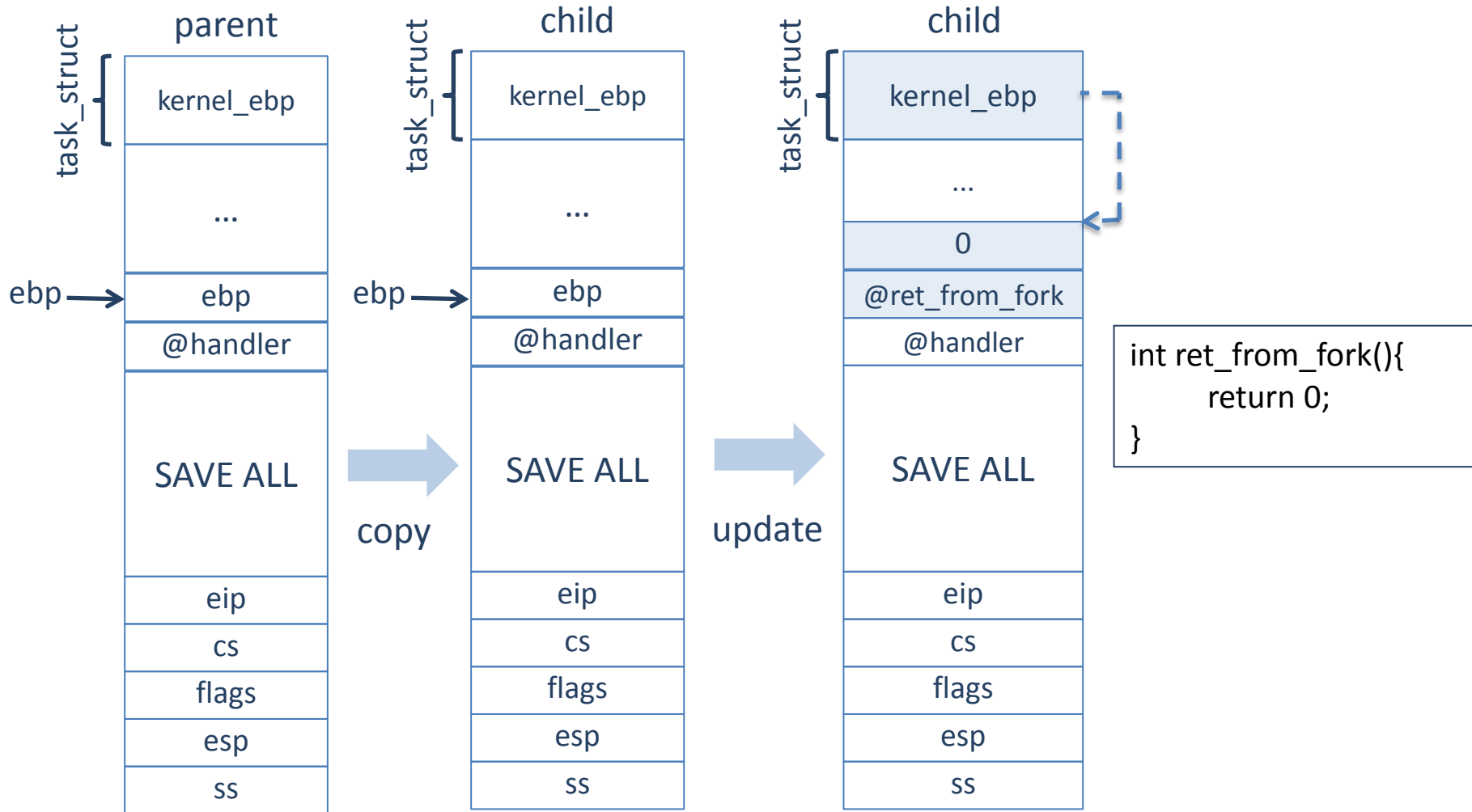


Fork: Actualizar task_struct

- Campos diferentes
 - pid, ...
- Preparación para el cambio de contexto:
 - El hijo tiene que guardar:
 - Posición inicial de su contexto
 - Posición de código a ejecutar: ret_from_fork
 - Código para gestionar la devolución de resultado del hijo

Fork: Actualizar task_struct

Operaciones: creación de procesos



Zeos: Creación de procesos iniciales

- Durante la inicialización de ZeOS se crean dos procesos iniciales
 - Init: primer proceso de usuario
 - Idle: proceso que sólo ocupa la cpu si no hay ningún otro candidato

Proceso *init*

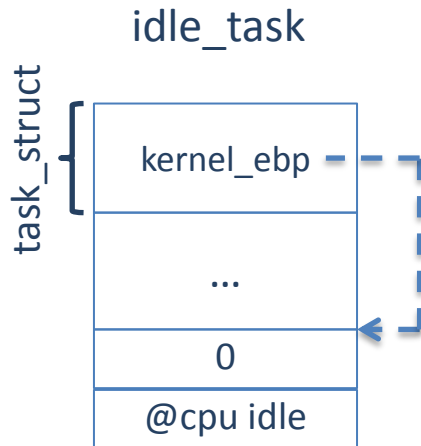
- Al completar la inicialización del sistema, se ejecuta el código del programa principal del usuario
 - No se usa FORK! ➔ Tratamiento especial
- Proceso encargado de gestionar este primer programa
- Inicialización
 - Reservar PCB
 - Asignar PID
 - Inicializar espacio de direcciones
 - Asignar Directorio de páginas
 - Mapear espacio lógico con espacio físico
 - Hacer que pase a ser proceso actual
 - Actualizar TSS con la dirección base de su pila de kernel
 - Actualizar registro CR3 con la dirección de su directorio

Proceso *idle*

- Proceso que sólo se ejecuta en modo sistema
- Ejecuta la función de kernel *cpu_idle*
- No forma parte de ninguna cola de procesos
 - Variable global *idle_task*
- Inicialización
 - Reservar PCB e inicializar variable *idle_task*
 - Asignar PID
 - Inicializar espacio de direcciones
 - Asignar Directorio de páginas
 - Preparar su contexto para cuando la política de planificación lo ponga en ejecución
 - La primera vez no viene de ningún cambio de contexto
 - Similar al caso de los procesos hijo creados con fork

Proceso *idle*: inicialización contexto

- Inicialización de pila
- Inicialización de campo `kernel_ebp`



Índice

- Conceptos previos
 - Procesos
- Visión de usuario
- Estructuras de datos
- Operaciones
 - Identificación
 - Cambio contexto
 - Creación
 - Destrucción
 - Planificación
 - Flujos (Threads)
 - Concurrencia y paralelismo
 - Sincronización entre procesos

Destrucción de procesos

- Exit: llamada a sistema para destruir un proceso
- Hay que liberar todos los recursos asignados
 - Liberar espacio de direcciones
 - Liberar PCB
 - Unix/Linux pueden retrasar la liberación del PCB
 - Todo lo que sea necesario: Entrada/Salida, semaforos, ...
- Borrar proceso de la lista de procesos en ejecución
- Ejecutar planificación:
 - Seleccionar nuevo proceso y restaurar su ejecución

Destrucción de procesos en Unix

- Proceso padre se puede sincronizar con el fin de sus hijos: llamada waitpid
 - Recoge causa muerte hijo
 - Si exit o si signal
 - Parámetro del exit
- Se guarda esta información en el PCB del hijo
 - Hasta que el padre no hace waitpid no se libera Estado zombie
- Si el padre muere sin hacer waitpid, el proceso init “adopta” a sus hijos y se encarga de hacer waitpid

Destrucción de procesos en ZeOS

- Simplificación: *exit* no recibe parámetros
- No implementamos sincronización con padre (no hay equivalente a *waitpid*)
- Al hacer *exit* se liberan todos los recursos del hijo incluido el PCB

Índice

- Conceptos previos
 - Procesos
- Visión de usuario
- Estructuras de datos
- Operaciones
 - Identificación
 - Cambio contexto
 - Creación
 - Destrucción
 - Planificación
 - Flujos (Threads)
 - Concurrencia y paralelismo
 - Sincronización entre procesos

Planificación de procesos

- Ciclo de vida de los procesos en este punto?

Planificación de procesos

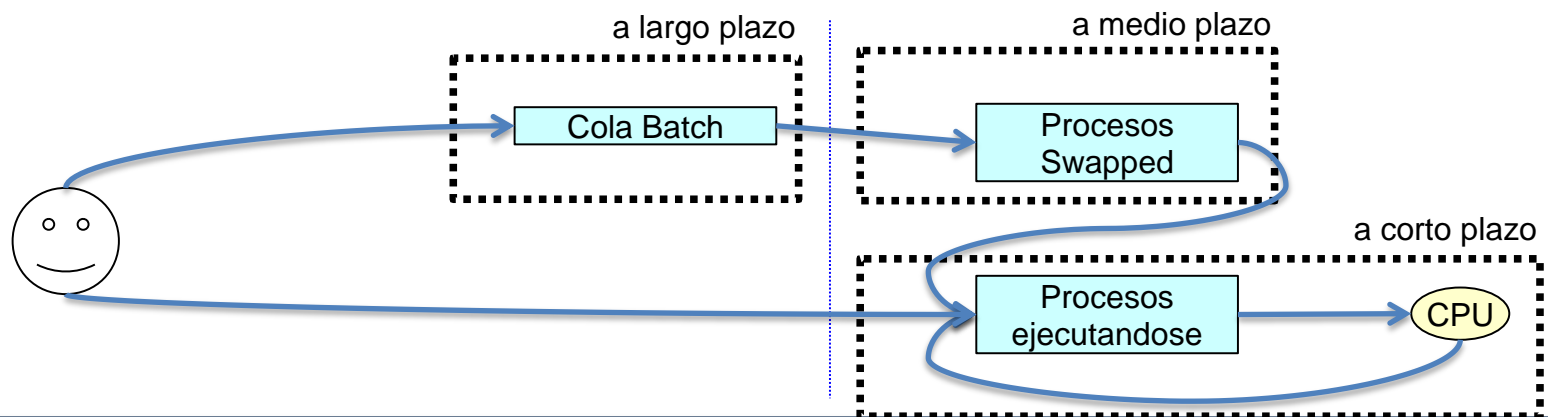
- Estados del proceso
 - Ready, run,
- El proceso pasa de Ready a Run según una política de planificación
- Existe un planificador
 - Se encarga de decidir el siguiente proceso a ejecutar
 - Y hasta cuándo se ejecutará
- Si el proceso que abandona la cpu no ha acabado la ejecución hay que guardar su contexto de ejecución

Planificación de procesos

- Algorithme basic:
 updateSchedulingData()
 if (mustChangeProcess()) {
 queueProcess(current())
 next = selectNext()
 task_switch(next)
 }

Planificadores del sistema (I)

- Normalmente hay 3 planificadores en el sistema en función del tiempo para planificar el proceso
 - Corto plazo: procesos pendientes del Procesador
 - Medio plazo: procesos swapped
 - Largo plazo: en colas de batch



Planificadores del sistema (II)

- Planificador a largo plazo (batch)
 - Controla el grado de multiprogramación en el sistema
 - Se ejecuta cuando empieza/acaba un proceso
 - Opcional en sistemas de tiempo compartido
- Planificador a medio plazo
 - Encargado de suspender y restaurar posteriormente procesos (swap out y swap in)
 - Se ejecuta cuando hay escasez de recursos
 - p.ej. Muchos procesos ejecutándose
 - Corrige errores del planificador a largo plazo

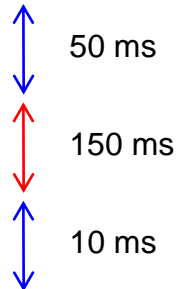
Planificador a corto plazo

- Selecciona el siguiente proceso a ejecutar
- Se ejecuta frecuentemente
 - Cuando se crea/acaba un proceso
 - Cada cierto tiempo (dependiente de la planificación)
 - Cuando un proceso inicia/finaliza la E/S
 - Ha de ser eficiente
- Veremos diferentes políticas de planificación
 - FCFS
 - Prioridades
 - Round Robin
 - ...

Caracterización de los procesos

- Ráfaga de CPU
 - Intervalo de tiempo consecutivo que un proceso está ejecutándose en la CPU
- Ráfaga de E/S
 - Intervalo de tiempo consecutivo que un proceso está realizando una E/S

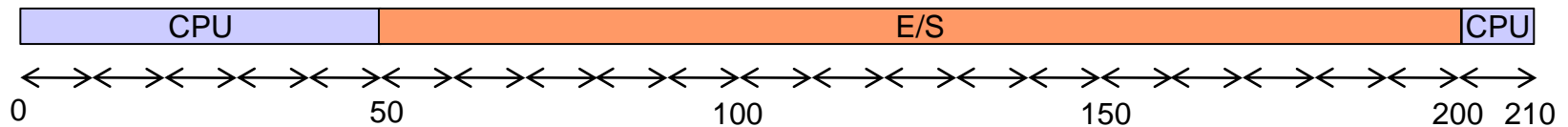
```
for ( i = 0 ; i < 5 ; i ++ )  
s += sqrt( a[i] );  
  
printf("Resultado:%f\n",s);  
  
for ( i = 0 ; i < 5 ; i ++ )  
a[i] = 0;
```



- 2 ráfagas de CPU de 50 y 10 ms respectivamente
- 1 ráfaga de E/S de 150 ms

Diagrama de gantt

- Diagrama horizontal que muestra para cada instante que valor toma un cierto parámetro
 - En nuestro caso, el estado de un proceso
- P. ej. diagrama de gantt de las ráfagas del proceso anterior:

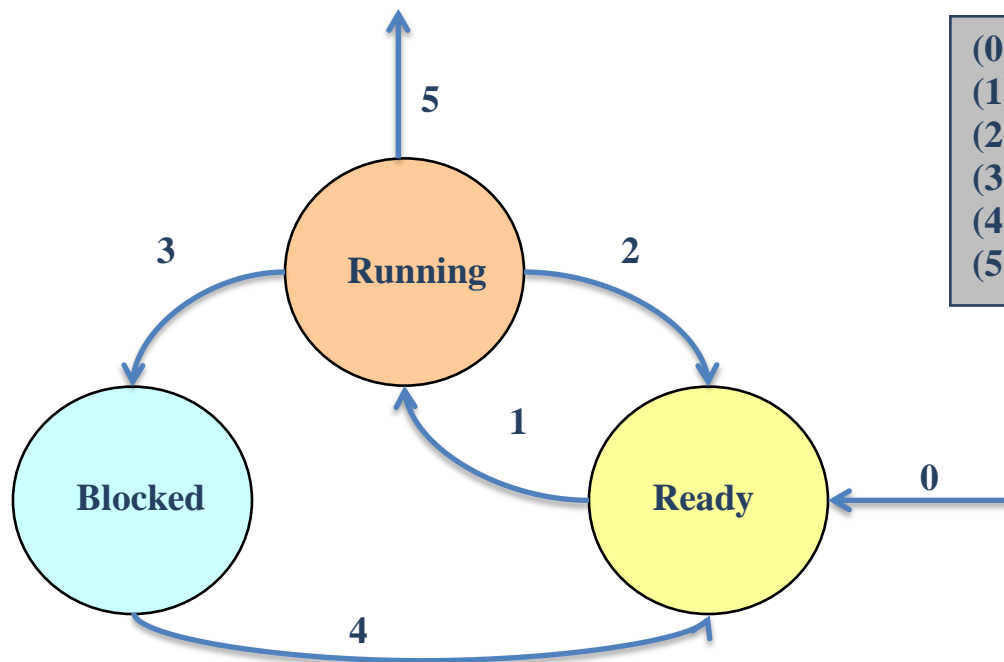


Estado blocked

- Un proceso pasa de Run a Blocked cuando espera un evento (fin E/S, operación sincronización, etc) que le impide avanzar con la ejecución
 - OS define qué operaciones son bloqueantes
- Para ello tiene que ejecutar una llamada al sistema
- El PCB del proceso se encola en la cola que hace referencia a la operación
- Dependiendo de la política de planificación, cuando se acaba la E/S se pasa a:
 - Ready: apropiación diferida
 - Run: apropiación inmediata

Ciclo de vida de un proceso

- Grafo simplificado de estados

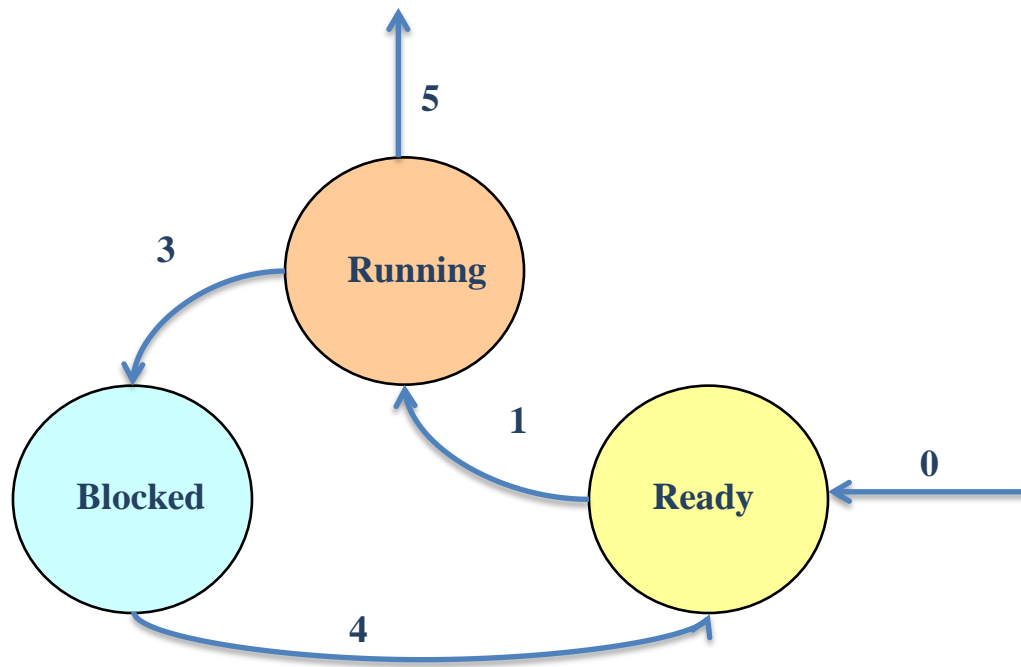


- (0) Creación del proceso
- (1) Planificador escoge este proceso
- (2) Planificador coge otro proceso
- (3) Proceso bloqueado por E/S
- (4) E/S disponible
- (5) Fin del proceso

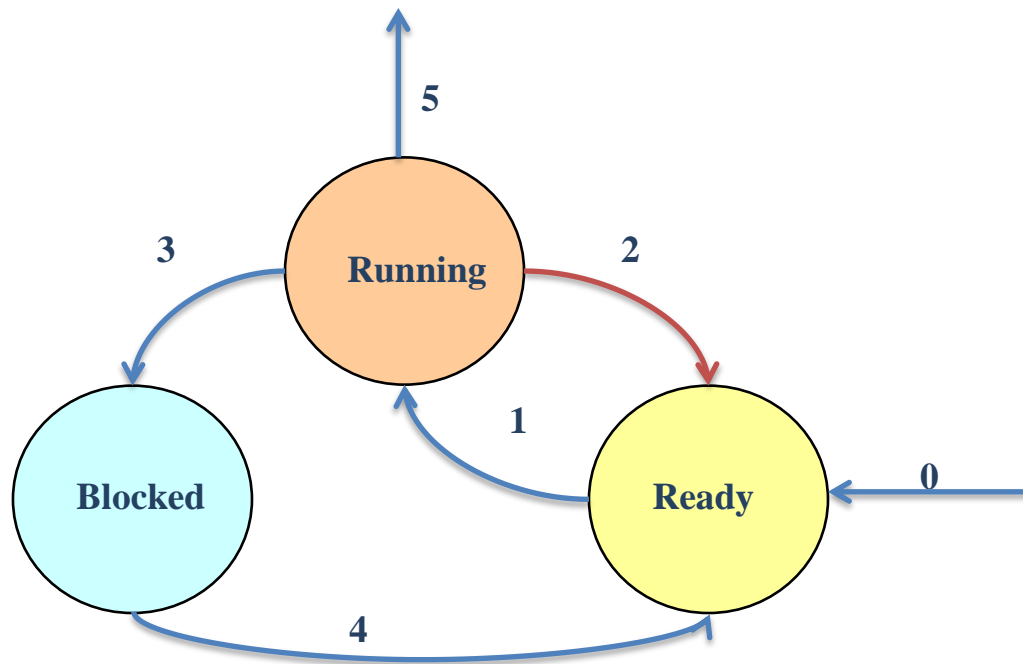
Tipos de políticas de planificación

- No Apropiativas
 - El Sistema Operativo no expulsa nunca al proceso de la CPU
 - El proceso abandona voluntariamente la CPU
 - p.ej.: mediante el inicio de una E/S
- Apropiativas
 - El Sistema Operativo puede decidir expulsar a un proceso del procesador y dárselo a otro (apropiación)
 - La apropiación puede ser:
 - Diferida
 - El Sistema Operativo hace efectiva la planificación cada cierto tiempo
 - Inmediata
 - El Sistema Operativo hace efectiva la planificación tan pronto como hay un cambio

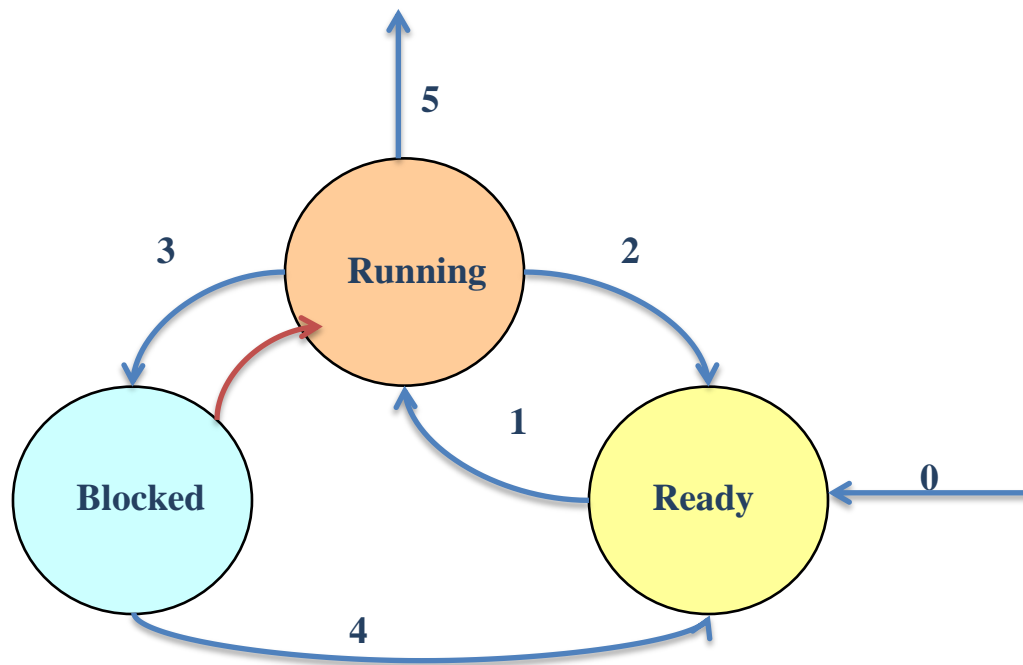
No Apropiativa



Apropiativa diferida

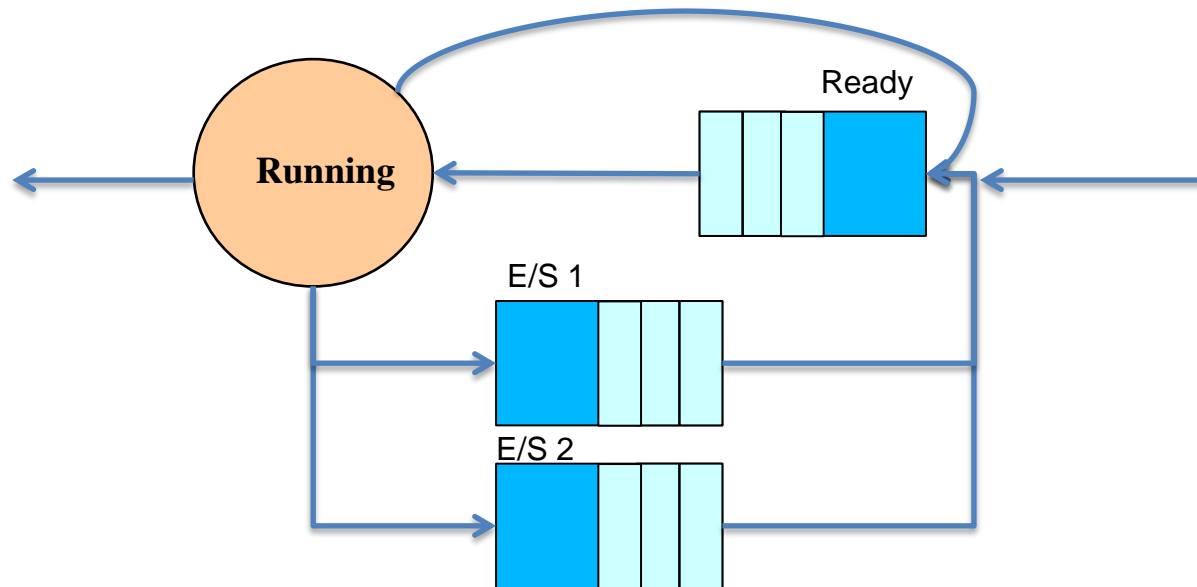


Apropiativa Inmediata



Estructuras de datos

- El sistema utiliza diferentes colas para gestionar los estados
- Cada cola puede tener una política diferente



Algoritmos de planificación

- FCFS
- Prioridades
- Round Robin
- Colas Multinivel

First Come, First Served (FCFS)

- El primer proceso en llegar a Ready es el primero en ser planificado
- No apropiativo
- Tiempo de espera elevado
 - No es apropiado para sistemas de tiempo compartido
- Provoca un efecto convoy con los procesos de E/S

Prioridades

- Cada proceso tiene asignada una prioridad
 - estática
 - dinámica
 - estática + dinámica
- El proceso más prioritario es planificado para ejecutarse
- Apropiativas o no apropiativas
- Entre procesos de igual prioridad: fifo

Prioridades

- Puede provocar inanición (starvation)
 - No se planifica NUNCA un proceso por no tener suficiente prioridad
 - Solución: envejecimiento (aging). Se aumenta la prioridad del proceso cada unidad de tiempo
- Ejemplos de prioridades dinámicas:
 - Shortest Job First
 - No apropiativo
 - La prioridad del proceso es el tiempo de ráfaga de CPU (predicción)
 - Shortest Remaining Time
 - Apropiativo
 - La prioridad del procesos es el tiempo restante de ráfaga

Round Robin

- El SO asigna un tiempo de CPU a cada proceso llamado **quantum**
 - puede ser constante o calcularse dinámicamente
- Un proceso abandona la CPU por dos motivos:
 - Su quantum ha finalizado y es apropiado
 - La abandona voluntariamente para hacer E/S
- El proceso a planificar se elige según FCFS
 - El Sistema asigna un nuevo quantum a ese proceso
- La elección del quantum es muy importante
 - Pequeño: demasiados cambios de contexto
 - Grande: se aproxima a FCFS

Round Robin con prioridades

- Se aplican prioridades normalmente
 - no apropiativas
 - apropiativas (inmediatas o diferidas)
- En caso de empate: fifo

Propiedades de los algoritmos

- ¿ Cómo sabemos qué algoritmo es mejor ?
 - Cada algoritmo maximiza diferentes criterios o propiedades
 - Dependiendo de nuestros objetivos el mejor será uno u otro

Propiedades de los algoritmos

- Justicia
 - Un algoritmo es justo si garantiza que todo proceso recibirá CPU en algún momento
- Eficiencia
 - Maximizar el % del tiempo que está la CPU ocupada
- Productividad (*Throughput*)
 - Maximizar el número de trabajos por unidad de tiempo
- Tiempo de espera
 - Minimizar el tiempo en la cola de ready

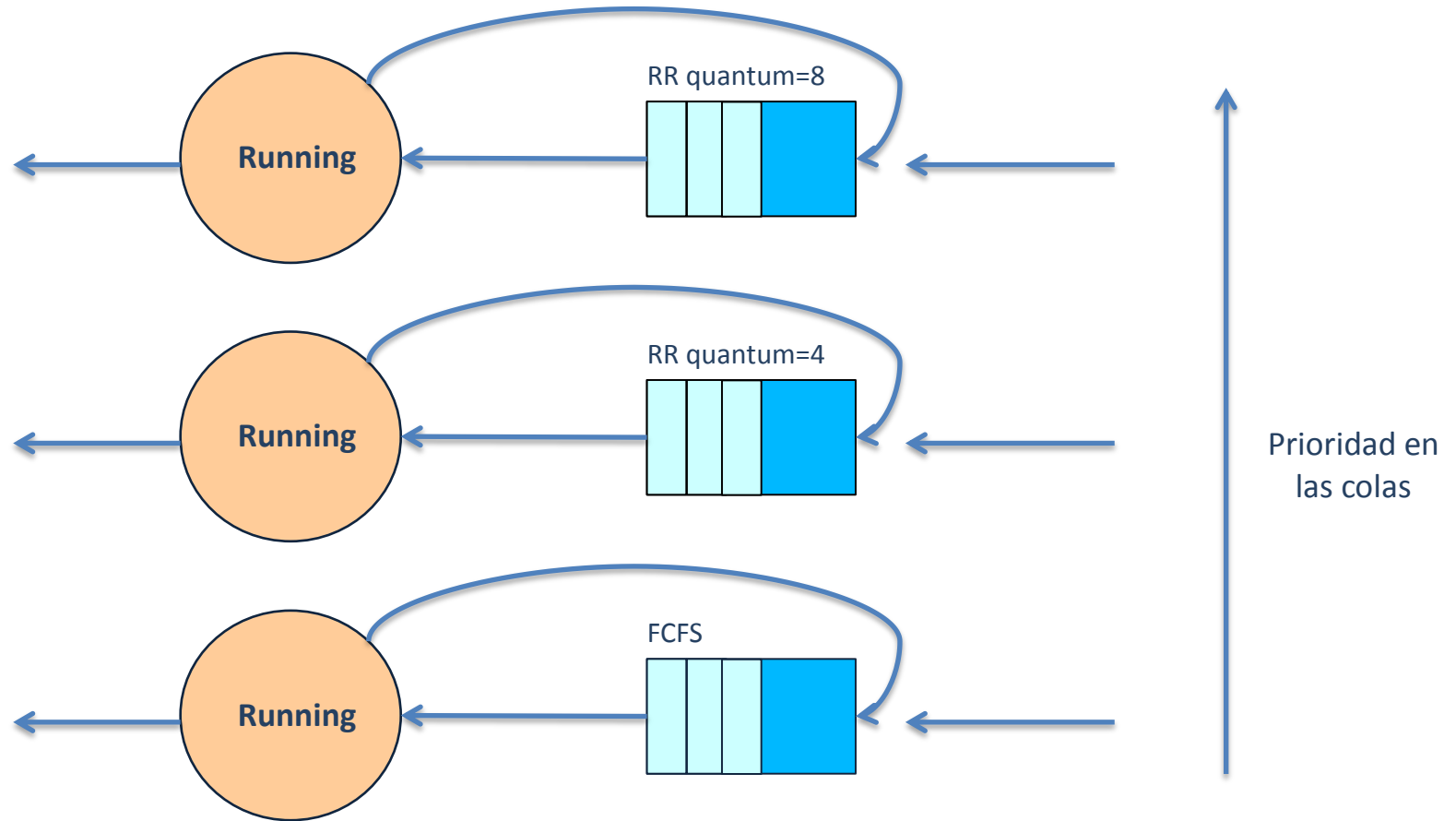
Propiedades de los algoritmos

- Tiempo de respuesta
 - Minimizar el tiempo que tarda un proceso en obtener su primer resultado
- Tiempo de retorno
 - Minimizar el tiempo total que tarda en ejecutarse 1 proceso

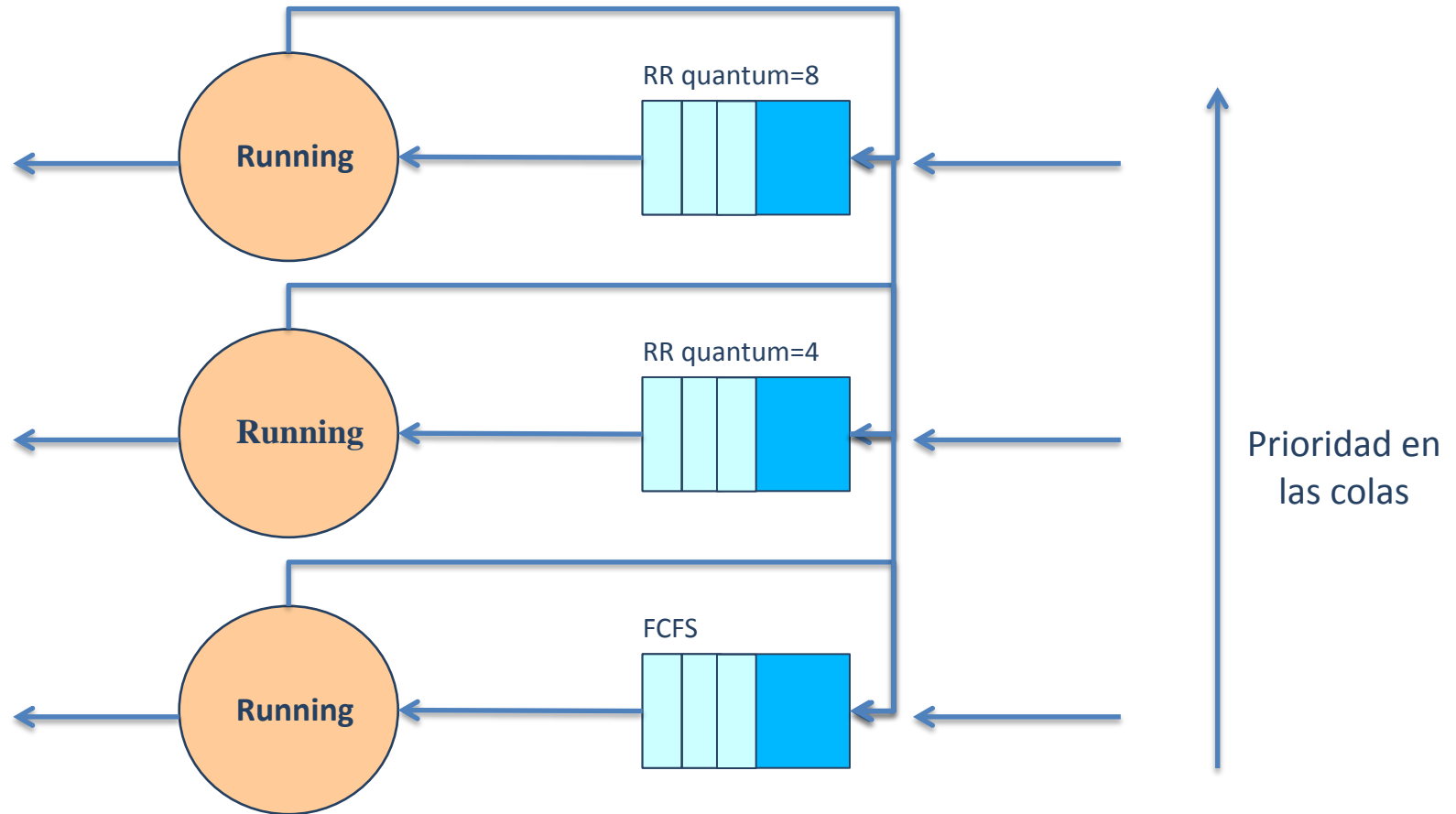
Colas Multinivel

- El sistema tiene diferentes colas de Ready
 - Cada cola tiene una prioridad
 - Se elige el proceso de la cola más prioritaria no vacía
 - Dentro de cada cola se aplica una política diferente
 - Diferentes tipos de procesos van a diferentes colas

Colas Multinivel



Colas Multinivel Realimentadas



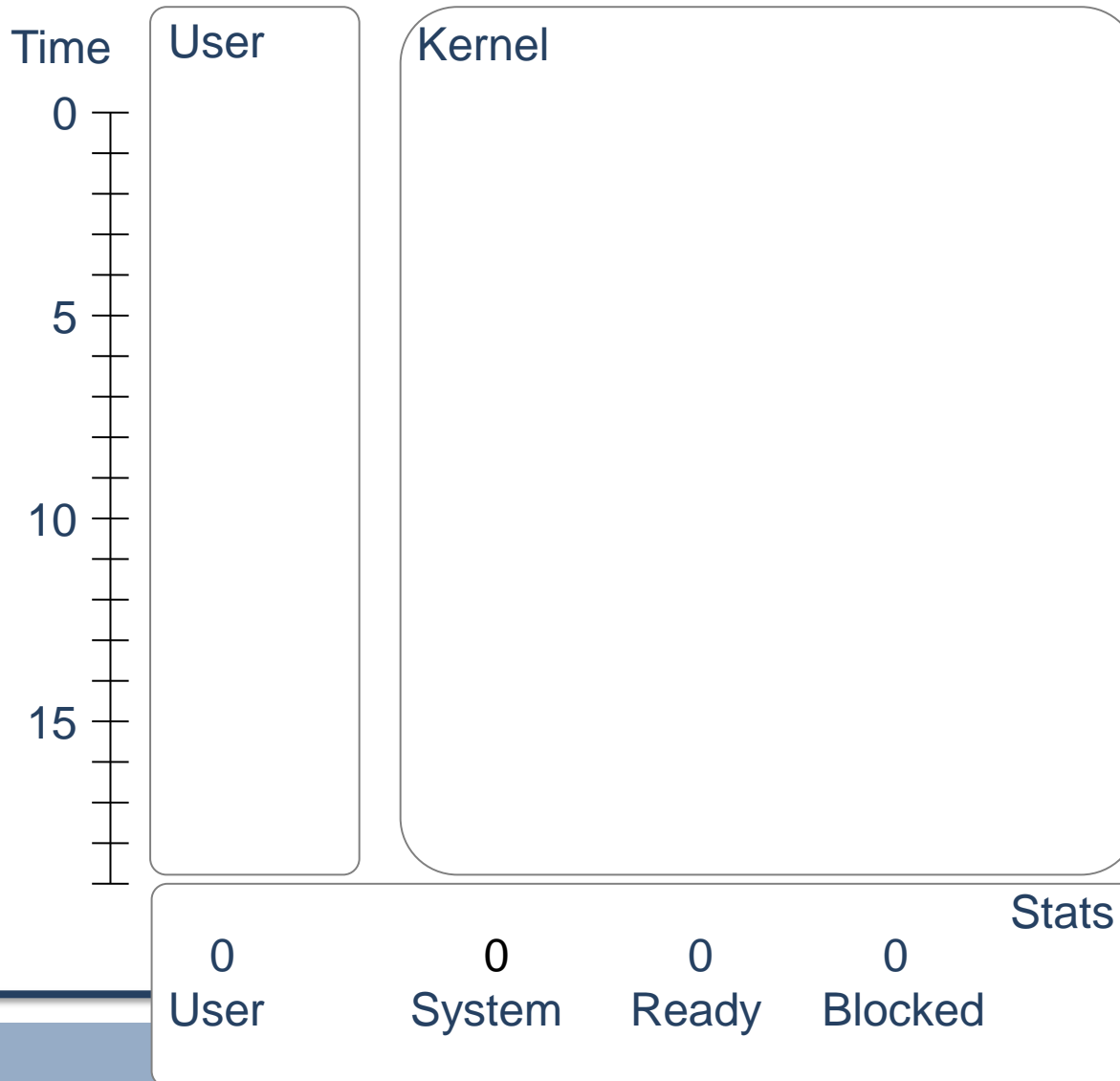
Zeos: Planificación

- Implementa Round Robin sin prioridad
 - Quantum es una característica de cada proceso
- Estados:
 - Ready (lista de procesos)
 - Run (No esta en ninguna cola)
 - Blocked
- Política de planificación apropiativa diferida
- Proceso idle se ejecuta si no hay nadie mas
- Lista de procesos libres (free)

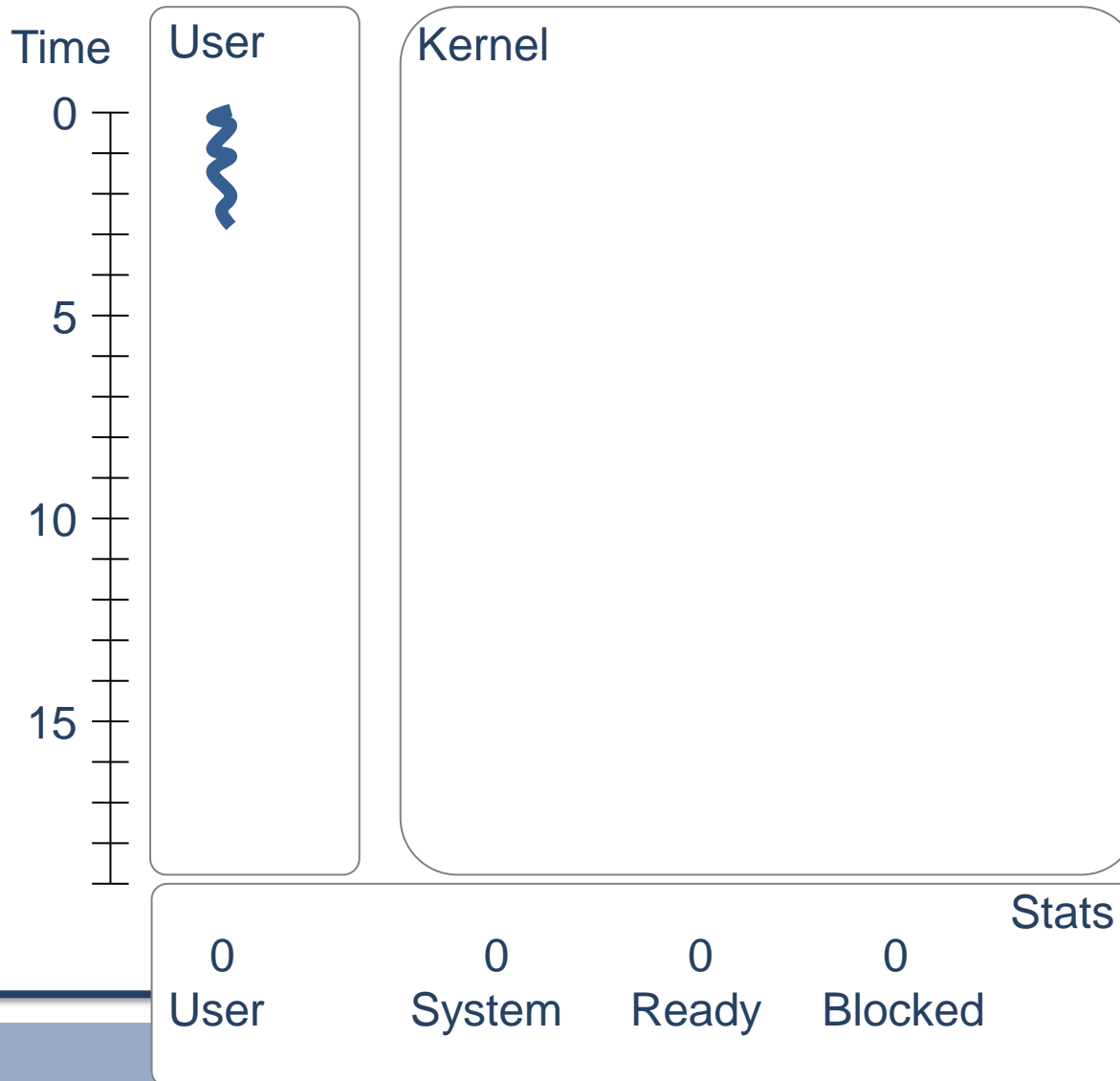
Zeos: Estadísticas

- ¿Cuánto tiempo está un proceso en ejecución?
 - Y de este... ¿Cuánto en usuario? ¿Y en sistema?
 - Otros
- En Zeos
 - Usuario, Sistema, Ready, Blocked

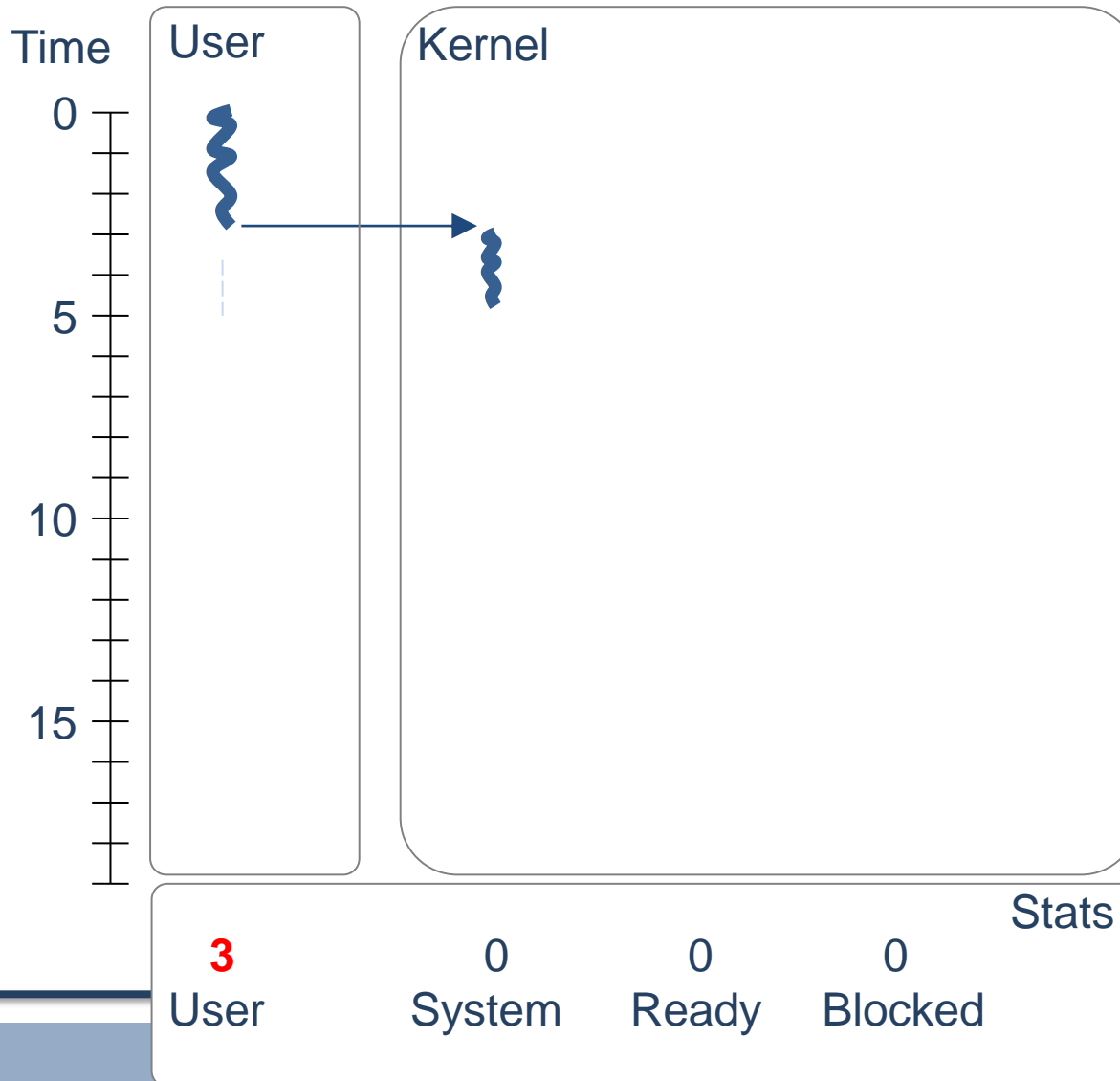
Zeos: Statistics



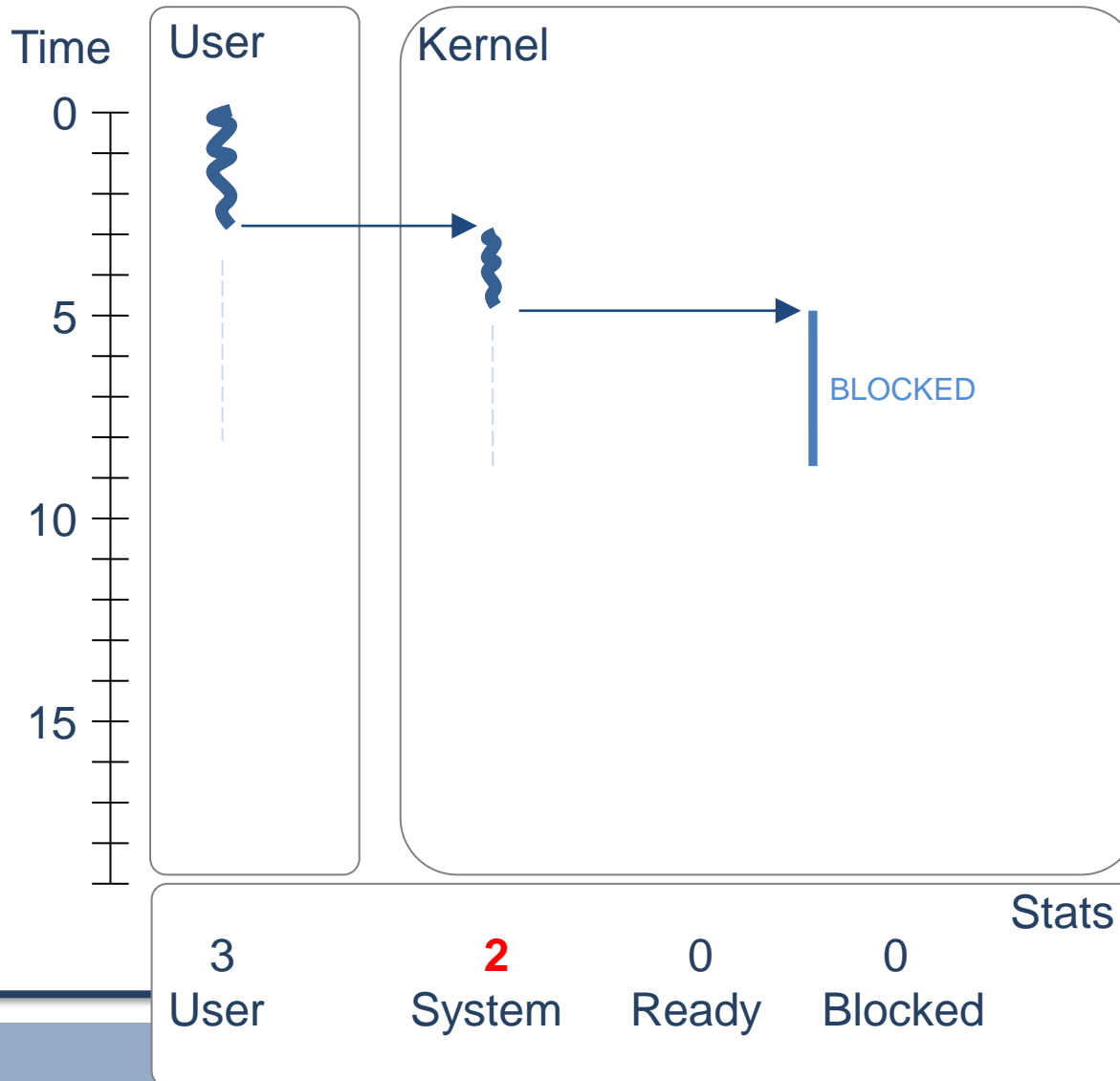
Zeos: Statistics



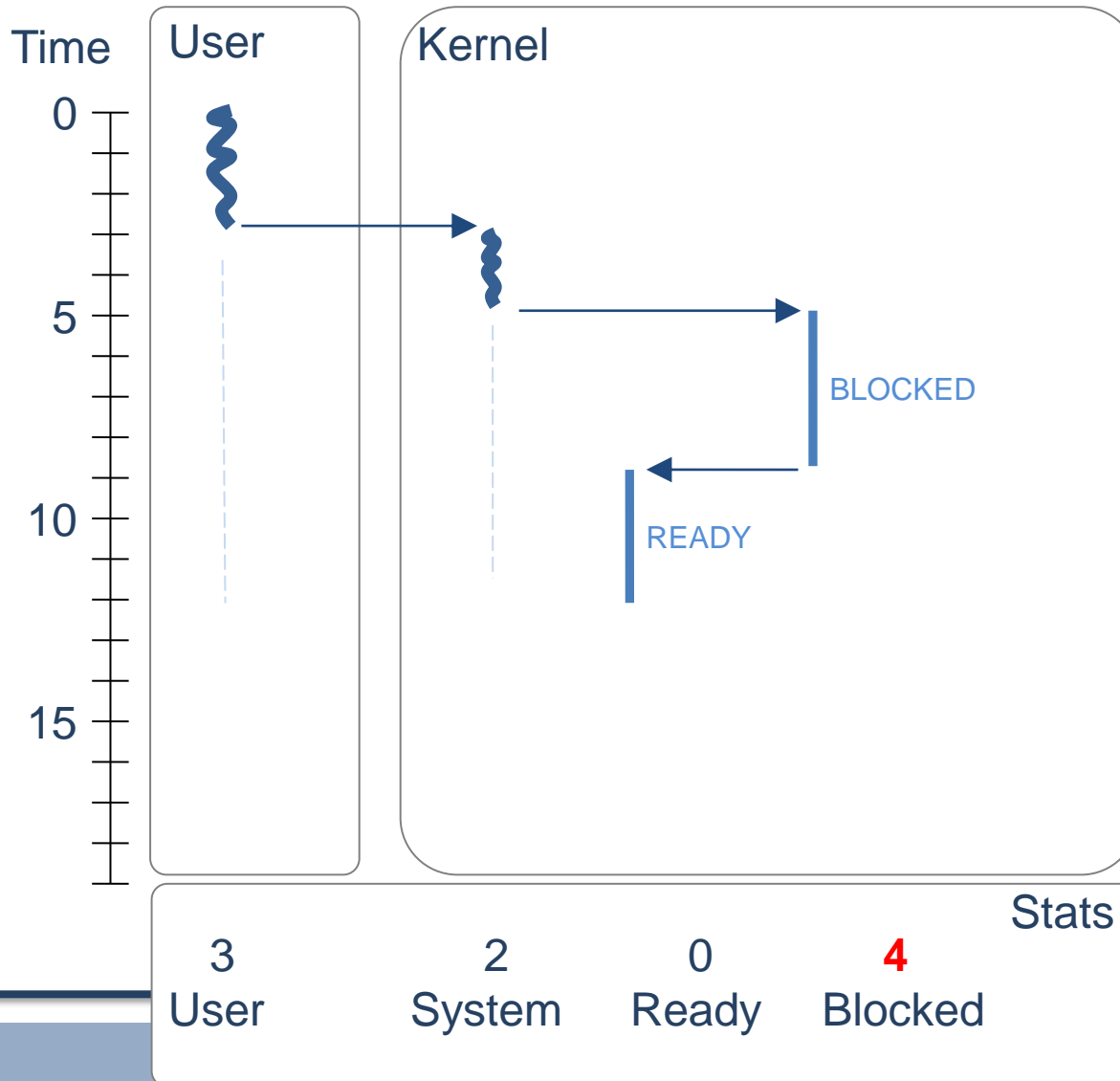
Zeos: Statistics



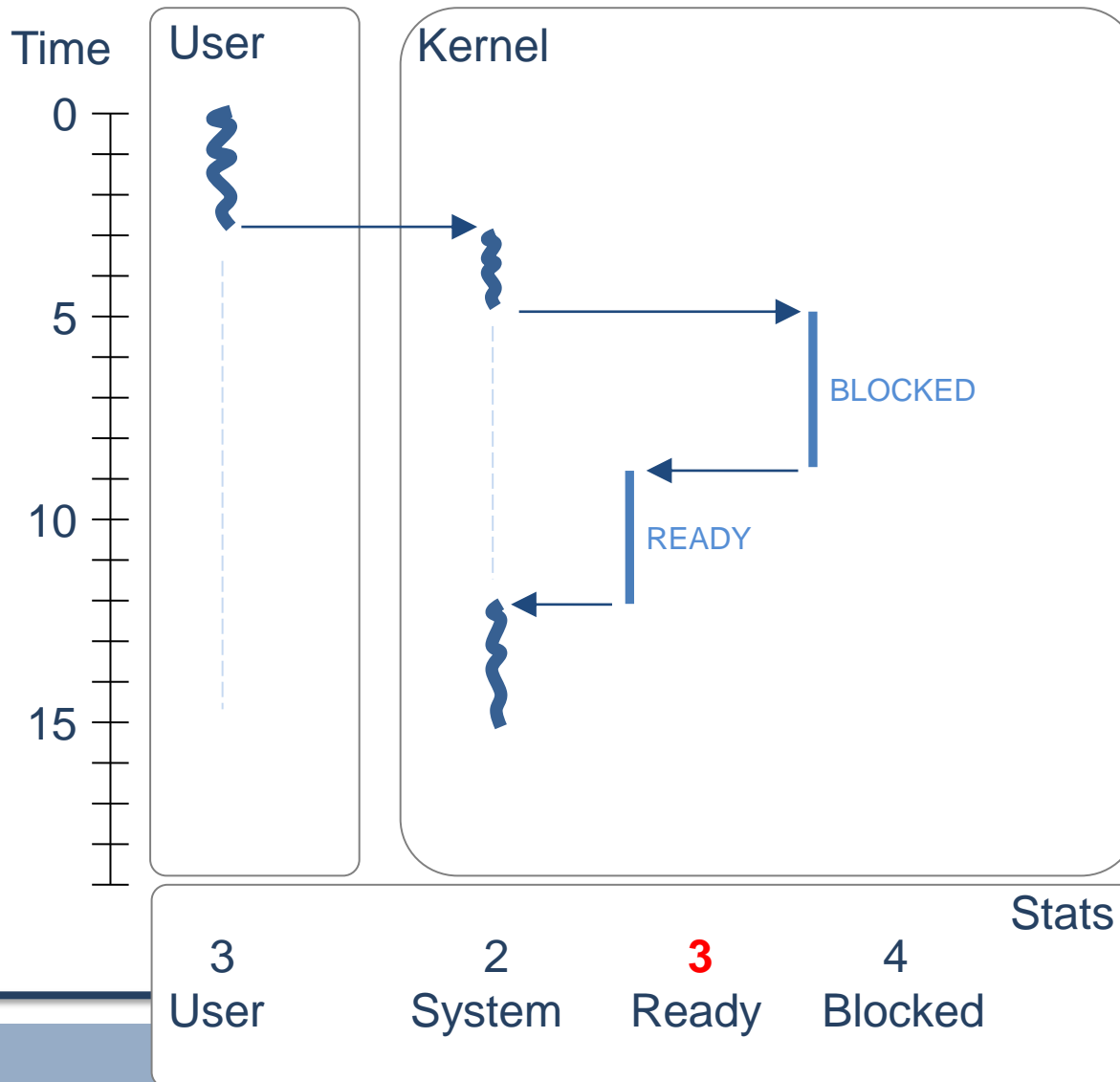
Zeos: Statistics



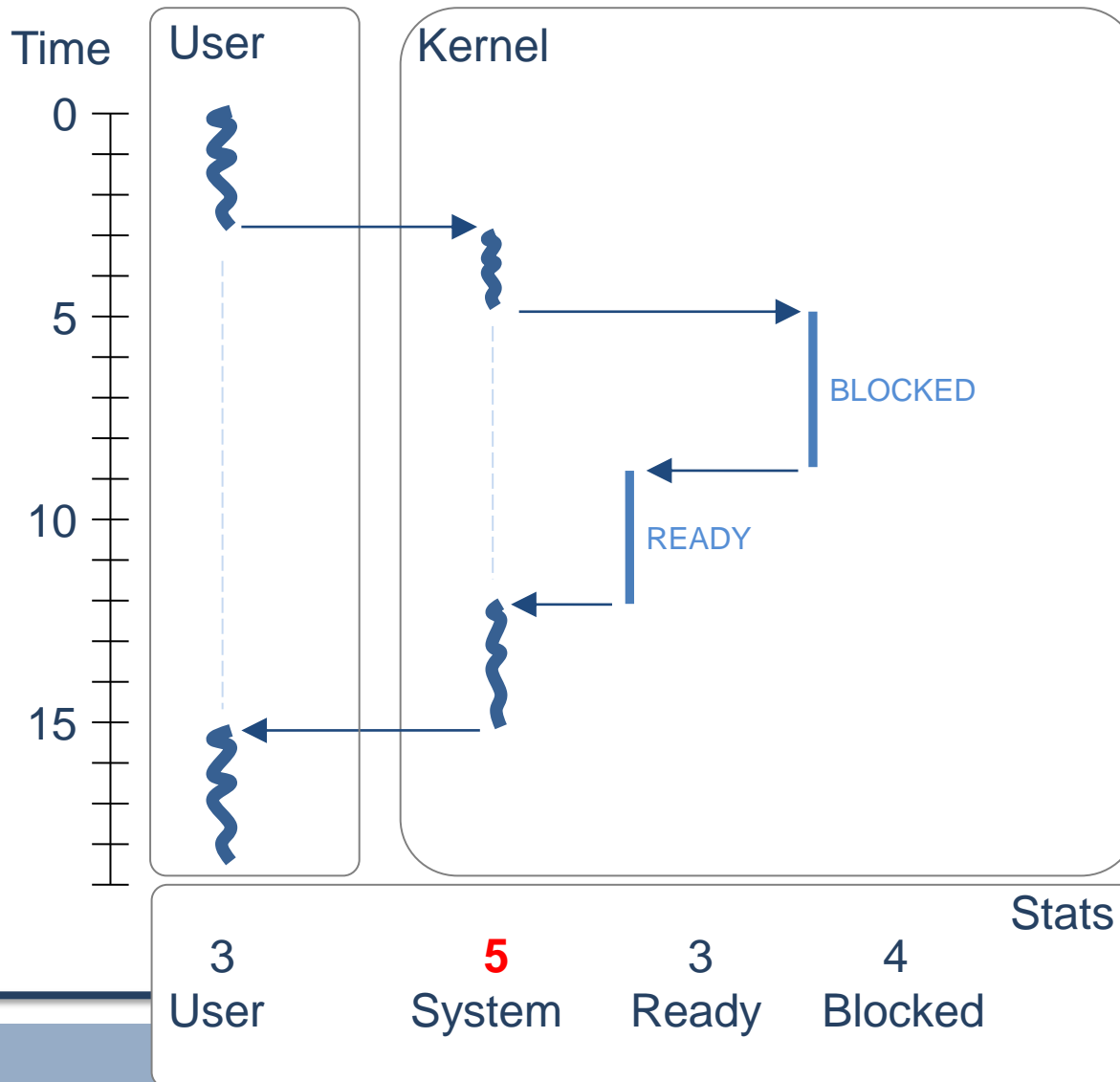
Zeos: Statistics



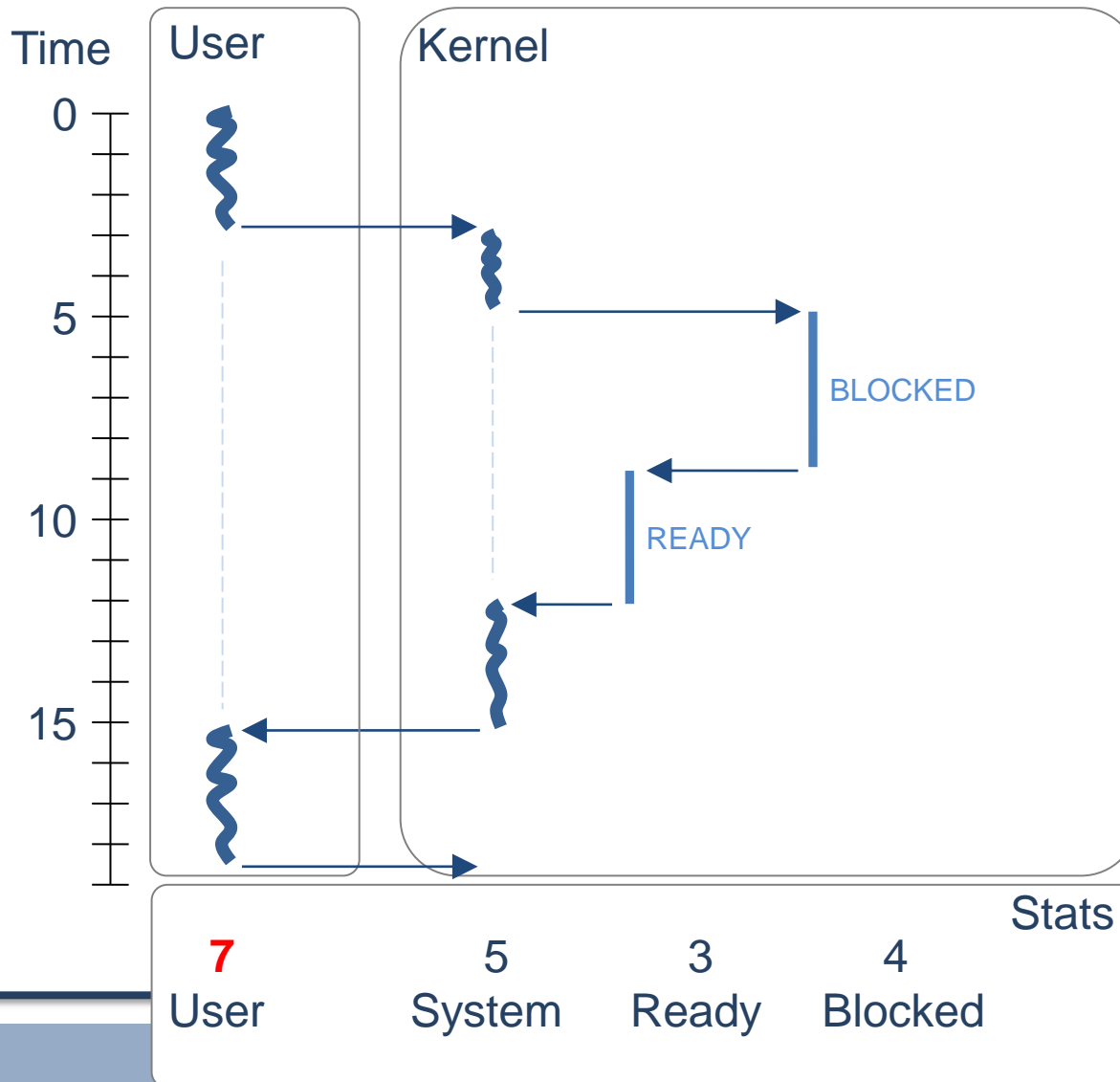
Zeos: Statistics



Zeos: Statistics



Zeos: Statistics



Índice

- Conceptos previos
 - Procesos
- Visión de usuario
- Estructuras de datos
- Operaciones
 - Identificación
 - Cambio contexto
 - Creación
 - Destrucción
 - Planificación
 - **Flujos (Threads)**
 - Concurrencia y paralelismo
 - Sincronización entre procesos

Flujos (threads)

- Procesos vs flujos
- Librerías de flujos

Concurrencia de procesos

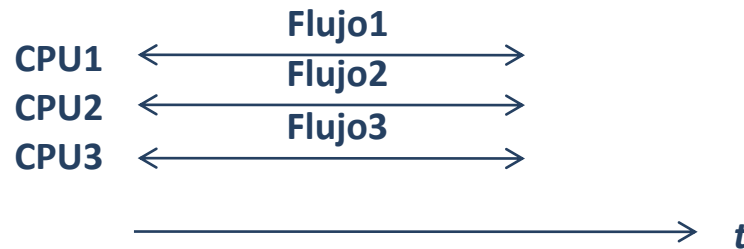
- ¿Porqué nos puede interesar ejecutar múltiples procesos simultáneamente?
 - Cada proceso hace una tarea diferente simultáneamente
 - Aprovechar el tiempo de E/S de un proceso
 - Si tenemos varios procesadores podemos ejecutar más rápido la misma tarea paralelamente
- Compartir recursos entre diferentes usuarios
 - Hay que dar la ilusión de que cada uno tiene su(s) procesador(es)

Concurrencia y paralelismo

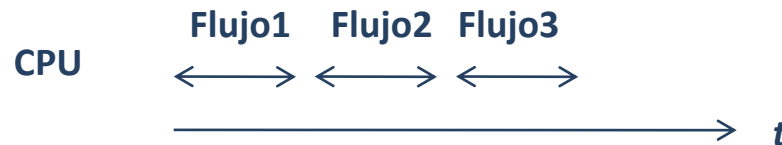
- Los recursos físicos son limitados.
 - El S.O. reparte los recursos.
- Como repartir/distribuir:
Concurrencia/Paralelismo

Concurrencia y paralelismo

- Paralelismo: 1 flujo asociado a un procesador

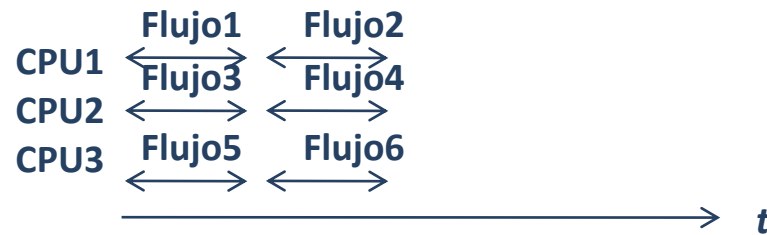


- Concurrencia: Cuando un procesador es compartido en el tiempo por varios flujos



Concurrencia y paralelismo

- Normalmente combinaciones de las dos



Concurrencia de procesos

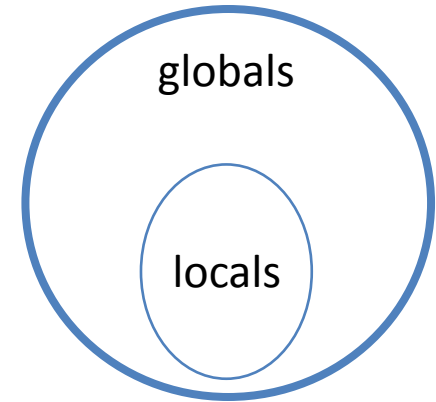
- Los procesos *por defecto* son independientes
 - no comparten nada
- Pero pueden compartir algunos recursos
 - Memoria (código, datos, ...)
 - Ficheros
 - ...
 - El nivel de compartición puede variar
 - Al menos habrá una pila privada para cada proceso

Ejemplo

- Escenario: aplicación servidor
 - **Monoproceso**: Sólo 1 cliente a la vez
 - Sin concurrencia ni paralelismo
 - **Multiproceso**: 1 proceso para atender cada cliente
 - Ejecución concurrente y/o paralela
 - Pero... se desaprovechan recursos
 - Replicación innecesaria de estructuras de datos que almacenan los mismos valores
 - Replicación del espacio lógico de memoria
 - Mecanismos para intercambiar información
 - ...

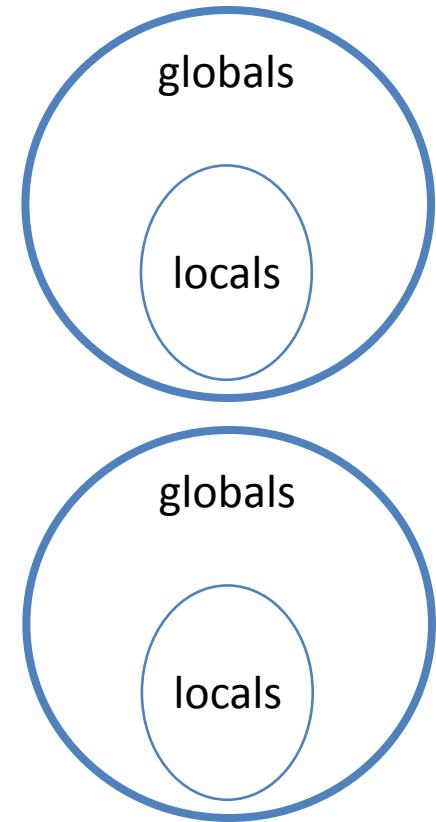
Monoprocess

```
int ko = 0; /* global */  
while (!ko) {  
    struct req r = getRequest();  
    processRequest( &r );  
    ko = r.malFormedRequest;  
}
```



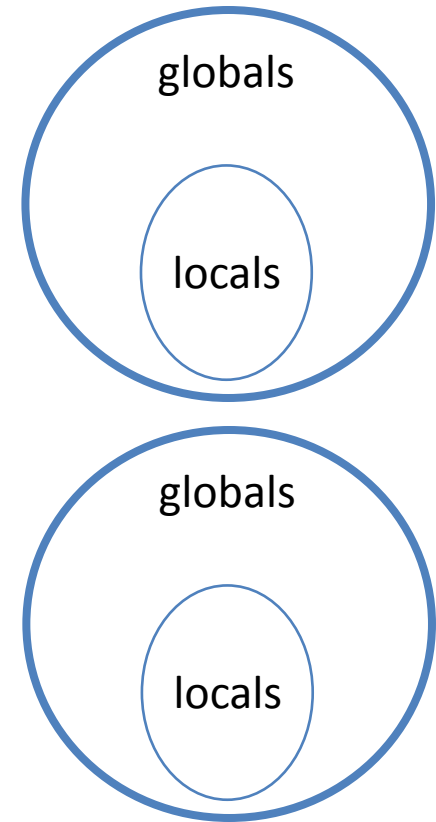
Multiprocess

```
int ko = 0; /* global */
while (!ko) {
    struct req r = getRequest();
    int p = fork();
    if (p==0) {
        processRequest( &r );
        ko = r.malFormedRequest;
    }
}
```



Multiprocess

```
int ko = 0; /* global */
while (!ko) {
    struct req r = getRequest();
    int p = fork();
    if (p==0) {
        processRequest( &r );
        ko = r.malFormedRequest;
        UUUPS!!!
    }
}
```



Procesos vs flujos

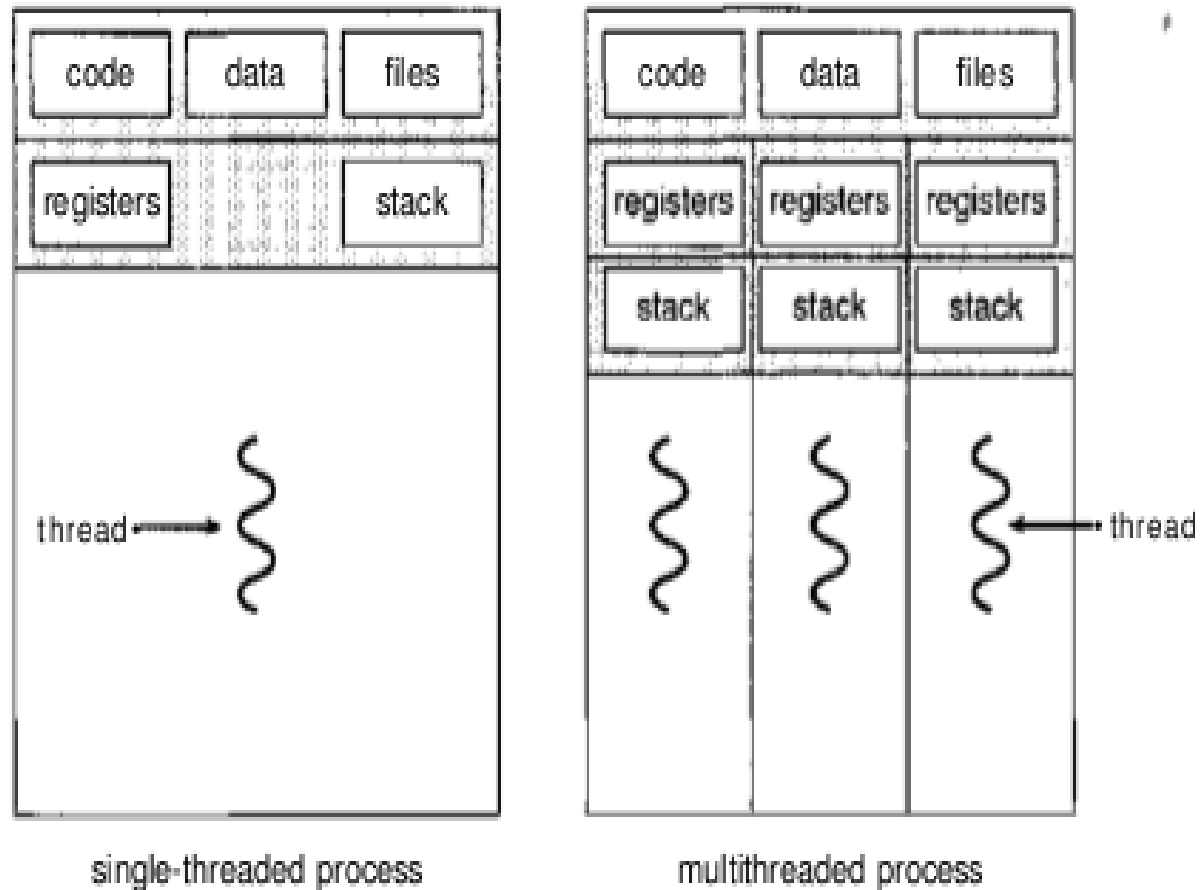
- Alternativa: procesos multiflujo

- Permitir diferentes secuencias de ejecución simultáneas asociadas a **un mismo proceso**
- ¿Que necesitamos para describir una secuencia de ejecución?
 - Pila
 - Program Counter
 - Valores de los registros
- El resto de características del proceso puede ser única (resto del espacio lógico, información sobre dispositivos, gestión de signals, ...)

Procesos vs flujos

- Flujo (thread): unidad mínima de planificación de CPU
- Los flujos de un proceso comparten **todos los recursos asignados al proceso y todas las características**
 - Cada flujo del proceso tiene asociado:
 - Siguiendo instrucción a ejecutar (valor del PC)
 - Zona de memoria para la pila
 - El estado de los registros
 - Un identificador
- Proceso tradicional: un sólo flujo de ejecución

Procesos vs flujos

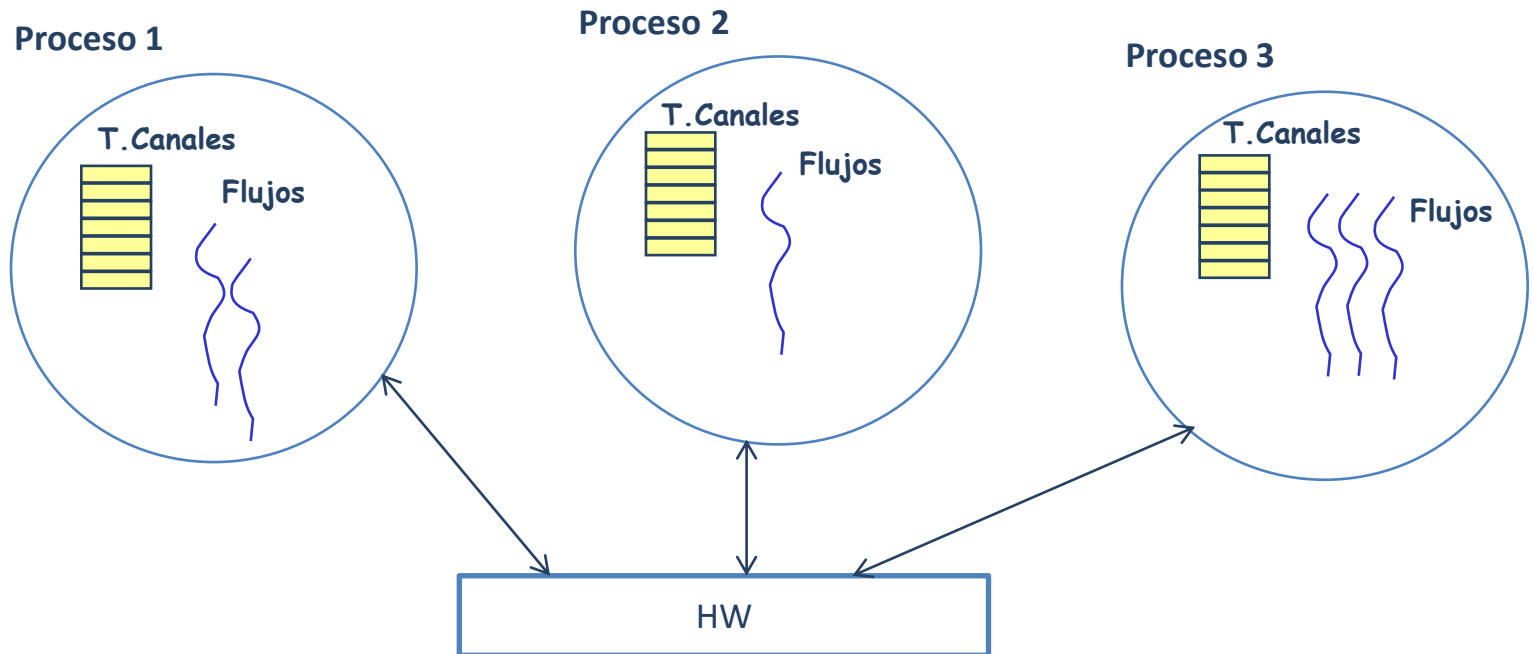


Procesos vs flujos

- Ventajas de usar flujos en lugar de procesos:
 - Coste en el tiempo de gestión: creación, destrucción y cambio de contexto
 - Aprovechamiento de recursos
 - Simplicidad del mecanismo de comunicación:
memoria compartida
 - Ya que los threads comparten memoria y ficheros, se puede intercambiar información sin llamar a rutinas de sistema
 - provoca la necesidad de **exclusión mutua y sincronización**.

Procesos y flujos

- Visión



Procesos vs flujos

- Utilización de procesos multiflujo
 - Explotar paralelismo y concurrencia
 - Mejorar la modularidad de las aplicaciones (encapsular tareas)
 - Aplicaciones intensivas en Entrada/Salida
 - Usar flujos dedicados SOLO a acceder a dispositivos
 - Aplicaciones servidores (atender varias peticiones de servicio)

Un ejemplo: Linux

- Linux usa la llamada a sistema *clone* para crear threads.
 - No portable (sólo funciona en Linux)
 - Internamente se usa esta llamada tanto para threads como para procesos
 - Se indica grado de compartición con el proceso que la usa
 - En Linux no se hace distinción threads/procesos a la hora de la planificación: todo son tasks que pueden compartir (o no) recursos con otras tasks.
 - Task_struct contiene punteros a los datos en lugar de los datos en sí.

Un ejemplo: Linux

- ```
int clone(int (*fn)(void *),
 void *child_stack,
 int flags,
 void *arg);
```

  - Devuelve el PID del proceso creado
  - El proceso ejecuta la rutina `fn(arg)` – es diferente de `fork()`!!
  - Se le debe pasar una zona de memoria `child_stack` para ser usada como pila (lo único que debe tener cada task y que no puede compartir)
  - `flags`:
    - `CLONE_PARENT`: el padre del proceso creado es el mismo que el del proceso creador
    - `CLONE_FS`: compartición de la información de file system
    - `CLONE_FILES`: compartición de la tabla de canales
    - `CLONE_SIGHAND`: compartición de la tabla de gestión de SIGNALS
    - ...

# Otro ejemplo: Win32

- 1 aplicación se ejecuta en 1 proceso
- Cada proceso puede tener 1 o N threads
- Thread incluye
  - Thread Id
  - Conjunto de registros
  - Pila de usuario y pila de kernel
  - Area de almacenamiento privada

# Otro ejemplo: Win32

- HANDLE ThreadHandle = CreateThread (  
    NULL, // Default Security attributes  
    0,     // Default Stack size  
    rutina, // Routine to be executed  
    &param, // Routine parameter  
    0,     // Default creation flags  
    &ThreadId); // Returns Thread Identifier
- WaitForSingleObject (ThreadHandle, INFINITE)
- CloseHandle ( ThreadHandle )

# Zeos

- Similar a Linux
- 1 PCB x thread (igual que procesos)
- Los threads compartirán todo el espacio de direcciones del padre
  - Comparten directorio y tabla de páginas
- Inicialización del contexto hardware
  - Cada thread empieza con los valores de registros de código y pila indicados como parámetro
  - Sólo necesitan heredar los valores de los registros de segmentos de datos de usuario

# Zeos: clone

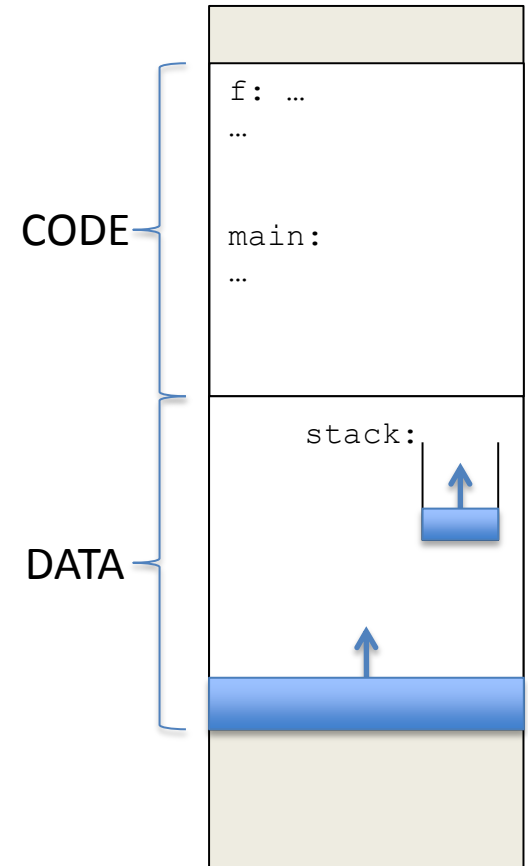
- Nueva llamada a sistema para crear threads:

```
int clone(void (*fn)(void),
 char *stack);
```

- Parámetros:
  - *fn* es la función a ejecutar por el nuevo thread
  - *stack* es la base de una zona de memoria para ser usada como pila
  - El thread creado ejecutará *fn()*.
- Devuelve:
  - El pid del nuevo thread creado
  - -1 en caso de error

# Ejemplo flujos

```
#define S 512
void f(void){
 write(1, "Hello cruel world", 17);
 exit();
}
int main() {
 char stack[S];
 int p = clone(f, &stack[S]);
 exit();
}
```



# Librerías de flujos

- POSIX Threads (Portable Operating System Interface, definido por la IEEE)
  - Interfaz de gestión de flujos a nivel de usuario
    - Creación y destrucción
    - Sincronización
    - Gestión de la planificación
  - Estándar definido para conseguir portabilidad (POSIX 1003.1c – 1995)
    - Cada SO debe implementar código para esta interfaz
    - Existe implementación para la mayoría de los SO (p.ej. Linux y W2K)



# Interfaz de pthreads

- Creación

```
pthread_create (pthread_t* tid, pthread_attr_t * attr,
 void *(* start_routine) (void *), void* arg)
```

- Identificación

```
pthread_self ()
```

- Finalización

```
pthread_exit (void* status)
```

- Sincronización fin de flujo

```
pthread_join (pthread_t tid, void **status)
```

# Creación

```
#include <pthread.h>
pthread_create (pthread_t* tid, pthread_attr_t * attr,
 void *(* start_routine) (void *), void* arg)
```

- Parámetros
  - *tid* Identificador de thread.
  - *attr* Atributos del thread, si ponemos NULL se inicializa con los atributos por defecto
  - *start\_routine* Función que ejecutará el nuevo thread
  - *arg* Dirección que recibe como parámetro el nuevo thread.
- Qué devuelve?
  - 0 si OK
  - Código de error

# Identificación y destrucción

- Identificación del pthread

```
#include <pthread.h>
pthread_t pthread_self (void);
```

- Devuelve el identificador del pthread

- Destruir flujo

```
#include <pthread.h>
void pthread_exit(void * status);
```

- Lo realiza el flujo que va a morir (equivalente al exit de procesos)
- *status*, valor que recibirá la función que espera la finalización del flujo

# Esperar flujo

```
#include <pthread.h>
int pthread_join (pthread_t thread, void ** status);
```

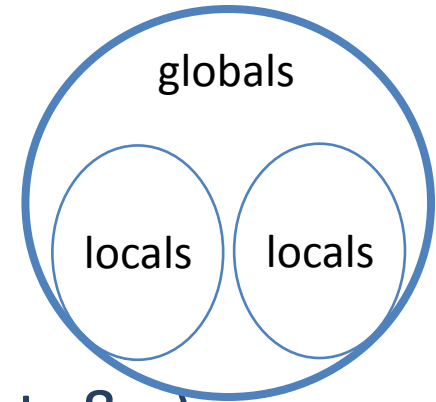
- Parámetros
  - *thread* Identificador del flujo a esperar.
  - *status* Estado de finalización del flujo que estábamos esperando.
- Qué devuelve?
  - 0 si OK
  - Código de error
- Join puede bloquearse (hasta que acabe el flujo esperado)

# Implementación Pthreads

- Como se implementaria el pthread\_create en ZeOS?

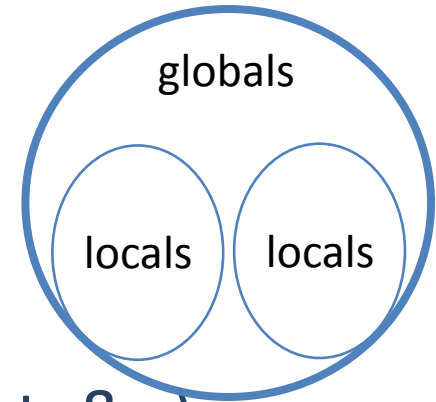
# Threads

```
int ko = 0; /* global */
while (!ko) {
 struct req r = getRequest();
 int p = createThread(processRequest, &r);
 ko = r.malFormedRequest;
}
```



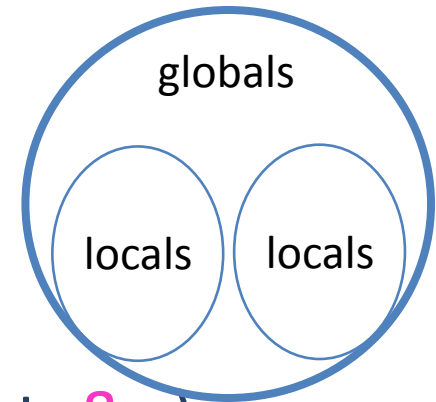
# Threads

```
int ko = 0; /* global */
while (!ko) {
 struct req r = getRequest();
 int p = createThread(processRequest, &r);
 ko = r.malFormedRequest;
 UUUPS!!!
}
```



# Threads

```
int ko = 0; /* global */
while (!ko) {
 struct req r = getRequest();
 int p = createThread(processRequest, &r);
 ko = r.malFormedRequest;
 UUUPS!!!
}
```



UUUPS!!!



# Índice

- Conceptos previos
  - Procesos
- Visión de usuario
- Estructuras de datos
- Operaciones
  - Identificación
  - Cambio contexto
  - Creación
  - Destrucción
  - Planificación
  - Flujos (Threads)
  - Concurrencia y paralelismo
  - **Sincronización entre procesos**

# Sincronización entre procesos

- Condición de carrera
- Acceso en exclusión mutua
  - Busy wait
  - Semáforos

# Condición de carrera

- Flujos pueden intercambiar información a través de la memoria que comparten
  - Accediendo más de uno a las mismas variables
- Problema que puede aparecer: condición de carrera (**race condition**)
  - Cuando el resultado de la ejecución depende del orden en el que se alternen las instrucciones de los flujos (o procesos)
  - Asociado a leer/escribir la misma posición de memoria a la vez
- Solución: **Exclusión mutua**

# Condición de carrera: Ejemplo

```
int primero = 1 /* variable compartida */
crear_2_flujos_identicos();
```

```
/* flujo 1 */
if (primero) {
 primero--;
 tarea_1();
} else {
 tarea_2();
}

/* flujo 2 */
if (primero) {
 primero--;
 tarea_1();
} else {
 tarea_2();
}
```

| tarea 1           | tarea 2 |
|-------------------|---------|
| flujo 1           | flujo 2 |
| flujo 2           | flujo 1 |
| flujo 1 y flujo 2 |         |

Resultado incorrecto

# Región crítica

- Región crítica

- líneas de código que contienen condiciones de carrera que pueden provocar resultados erróneos
  - líneas que acceden a variables compartidas cuyo valor cambia durante la ejecución

- Solución

- Garantizar el acceso en exclusión mutua a estas regiones de código
- ¿Evitar cambios de contexto?

# Región crítica

- Condiciones que garantizan un acceso correcto (Dijkstra)
  - Si un flujo está ejecutando una sección crítica, ningún otro flujo podrá entrar en la misma sección crítica
  - Un flujo no puede esperar indefinidamente para entrar en una sección crítica
  - Un flujo que está ejecutando fuera de una sección crítica no puede impedir que los otros flujos entren en ella
  - No se pueden hacer hipótesis sobre el número de procesadores, ni la velocidad de ejecución

# Exclusión mutua

- Acceso en exclusión mutua
  - Se garantiza que el acceso a la región crítica es secuencial
    - Mientras un flujo está ejecutando código de esta región ningún otro flujo lo hará (aunque haya cambios de contexto)
  - El programador debe:
    - Identificar regiones críticas de su código
    - Marcar inicio y fin de la región usando las herramientas del sistema
  - El sistema operativo ofrece llamadas a sistema para marcar inicio y fin de región crítica
    - Inicio: Si ningún otro flujo ha pedido acceso a la región crítica se deja que continúe accediendo, sino se hace que el flujo espera hasta que se libere el acceso a la región crítica
    - Fin: se libera acceso a la región crítica y si algún flujo estaba esperando el permiso para acceder, se le permite acceder

# Exclusión mutua: ejemplo

```
int primero= 1; /* variable compartida */
mutex_t hay_alguien;
mutex_init(&hay_alguien);
crear_2_flujos_identicos ();
```

```
/* flujo 1 */
mutex_lock(&hay_alguien);
if (primero) {
 primero --;
 mutex_unlock(&hay_alguien);
 tarea_1();
} else {
 mutex_unlock(&hay_alguien);
 tarea_2();
}
```

```
/* flujo 2 */
mutex_lock(&hay_alguien);
if (primero) {
 primero --;
 mutex_unlock(&hay_alguien);
 tarea_1();
} else {
 mutex_unlock(&hay_alguien);
 tarea_2();
}
```



# Exclusión mutua

- El programador debe tener en cuenta:
  - Las regiones críticas deben ser lo más pequeñas posibles para maximizar concurrencia
  - El acceso en exclusión mutua viene determinado por el identificador (variable) que protege el punto de entrada
    - No hace falta que tengan las mismas líneas de código
    - Si hay varias variables compartidas independientes puede ser conveniente protegerlas mediante variables diferentes

# Implementación mutex: busy waiting (i)

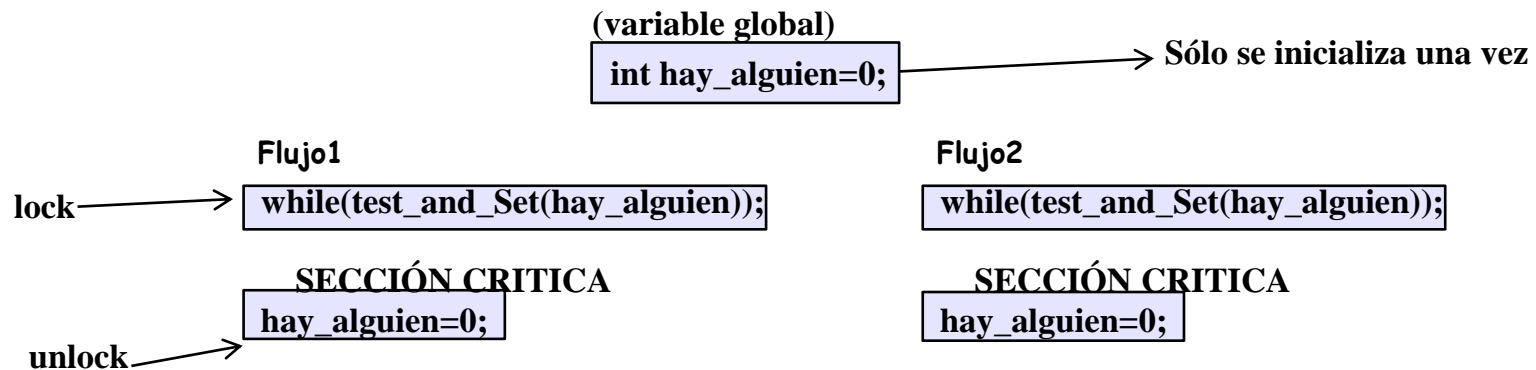
- Espera activa (busy waiting)
  - Necesitaremos soporte de la arquitectura: instrucción atómica, es decir, ininterrumpible (instrucción de lenguaje máquina)
    - Consulta y modificación de una variable de forma atómica
  - El equivalente en alto nivel sería...

```
int test_and_set(int *a)
{
 int tmp=*a;
 if (tmp==0) *a=1;
 return(tmp);
}
```

**Es necesario hacerlo  
Por hardware para que  
Sea atómico**

# Implementación mutex: busy waiting (ii)

- ¿Como se usa?



# Implementación mutex: busy waiting (iii)

- Inconvenientes:
  - Ocupamos la CPU (mientras esperamos poder entrar en una sección crítica) con trabajo inútil
    - Podríamos no dejar avanzar al flujo que ha de liberar la sección crítica
  - Podríamos colapsar el bus de memoria (siempre accediendo a la misma instrucción y la misma variable)
  - El resto de los usuarios no pueden aprovechar la CPU
- Posible solución
  - Consultar si podemos acceder a la sección crítica, y si no podemos, **bloquear el proceso** hasta que nos avisen que podemos acceder, en lugar de estar consultando continuamente como en espera activa

# Implementación: bloqueo

- Idea:
  - Evitar el consumo inútil de tiempo de cpu.
  - Reducir el número de accesos a memoria
- Propuesta:
  - Bloquear a los flujos que no pueden entrar en ese momento
  - Desbloquearlos cuando quede libre la sección crítica
- Utilizaremos SEMAFOROS
  - *sem\_init(sem,n)*: crea un semáforo
  - *sem\_wait(sem)*: entrada en exclusión mutua (equivale al lock)
  - *sem\_signal(sem)*: salida de exclusión mutua (equivale al unlock)

# Semáforos

- Qué es un semáforo?
  - Es una estructura de datos del SO para proteger el acceso a recursos. Tendrá asociado
    - un contador
    - y una cola de procesos bloqueados.
  - El contador indica la cantidad de accesos simultáneos que permitimos al recurso que protege el semáforo
    - Si usamos el semáforo para hacer exclusión mútua:  
Recurso=sección crítica,  $n=1$ .
    - Se pueden usar para más cosas

# Una posible implementación de semáforos

- `sem_init(sem,n);`

```
sem->count = n;
ini_queue (sem->queue);
```

- `sem_wait(sem);`

```
sem->count --;
If (sem->count < 0){
 bloquear_flujo (sem->queue);
}
```

**/\* bloquea al flujo que hace la llamada\***

- `sem_signal(sem);`

```
sem->count ++;
If (sem->count <= 0){
 despertar_flujo (sem->queue);
}
```

**/\* despierta un flujo de la cola \*/**

# Uso de semáforos

- En función del valor inicial del contador usaremos el semáforo para distintos fines
  - *sem\_init(sem,1)*: **MUTEX** (permitimos que 1 flujo acceda de forma simultanea a la sección crítica)
  - *sem\_init(sem,0)*: **SINCRONIZACIÓN**
  - *sem\_init(sem,N)*: **RESTRICCIÓN DE RECURSOS**, genérico
- Habitualmente usaremos:
  - Espera activa si los tiempos de espera se prevén cortos
  - Bloqueo si se prevén largos
    - Bloquear un flujo es costoso (entrar a sistema)



# Problemas concurrencia: deadlock

- Se produce un abrazo mortal entre un conjunto de flujos, si cada flujo del conjunto está bloqueado esperando un acontecimiento que solamente puede estar provocado por otro flujo del conjunto

**Flujo 1**  
**Conseguir(impresora)**  
**Conseguir(fichero)**  
**imprimir\_datos()**  
**Liberar(fichero)**  
**Liberar(impresora)**

**Flujo2**  
**Conseguir(fichero)**  
**Conseguir(impresora)**  
**imprimir\_datos()**  
**Liberar(fichero)**  
**Liberar(impresora)**

# Condiciones del deadlock

- Se han de cumplir 4 condiciones a la vez para que haya abrazo mortal
  - Mutual exclusion: mínimo de 2 recursos no compartibles
  - Hold&Wait: un flujo consigue un recurso y espera por otro
  - No preempción: si un flujo consigue un recurso sólo él puede liberarlo y nadie se lo puede quitar
  - Circular wait: ha de haber una cadena circular de 2 o más flujos donde cada uno necesita un recurso que esta siendo usado por otro de la cadena

# Evitar deadlocks

- ¿Como evitarlos?, evitar que se cumpla alguna de las condiciones anteriores
  - Tener recursos compartidos
  - Poder quitarle un recurso a un flujo
  - Poder conseguir todos los recursos que necesitas de forma atómica
  - Ordenar las peticiones de recursos (tener que conseguirlos en el mismo orden)

# References

- [1] Understanding Linux Kernel 3rd ed. Chapter 3 Processes and Chapter 7 Process Scheduling.
- [2] Intel® 64 and IA-32 architectures software developer's manual volume 3A: System programming guide. Chapter 7 Task Management.
- [3] Petr Hudeček and Michal Pokorný. The Deadlock Empire. HackCambridge 2016.  
<https://deadlockempire.github.io/>