

marinamuca01

www.wuolah.com/student/marinamuca01

13850

Tema-2-Procesos-y-hebras.pdf

Resumenes Teoría



2º Sistemas Operativos



Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación
Universidad de Granada



OPENSKY
CONECTANDO IDIOMAS

We prepare for

Cambridge

English Qualifications™

iAT GOOD HOURS, GREEN SLEEVES!
(A BUENAS HORAS, MANGAS VERDES)

Inglés Alemán
TODOS LOS NIVELES



Clases en
DIRECTO



Máximo
10 PERSONAS



Audio y video
PROFESIONAL



Pizarra digital
COMPARTIDA

@Academia OpenSky

691 243 557

info@academiaopensky.es

Exámenes, preguntas, apuntes.

12:48

WUOLAH

Join the student revolution.

MULTI

Conéctate dónde y cómo prefieras.

Guarda tus apuntes en un lugar seguro y ordenado, y accede a ellos desde tu pc, móvil o tablet.

Acceder

Registrarse

GET IT ON
Google Play

Download on the
App Store

TEMA 2

CONCEPTOS FUNDAMENTALES SOBRE PROCESOS

Concepto de proceso

Proceso = programa en ejecución, cuya ejecución se debe realizar sin interferencias debidas a la ejecución de otros programas.

- **Programa** → fichero estático ejecutable.
 - **Proceso** → programa + estado de ejecución.

Ejecución de un programa se caracteriza por su traza de ejecución

Distintos procesos pueden ejecutar el mismo programa.

Ejecutar un programa, un proceso requiere recursos → al menos { memoria para texto (código)
SO se encarga de controlar dichos recursos. datos y pila
CPU (conjunto de registros)

Ej.: SO multiplexa la CPU para permitir ejecución de varios procesos \Rightarrow necesario almacenar contexto de ejecución en CPU de un proceso para cargarlo posteriormente.

El SO es un programa que también se ejecuta en CPU \Rightarrow ¿Cómo se gestiona la ejecución del programa SO y los distintos programas asociados a procesos.

Concepto sobre kernel y programa de usuario

Ejecución en CPU = dos niveles de privilegio (modos de ejecución) {
 user (ring 3)
 kernel (ring 0)}

espacio protegido para el Kernel

CPU opera sobre él sólo en modo Kernel

Trabajar sobre este espacio requiere una secuencia especial de instrucciones para cambiar de modo usuario a kernel.

Código de este espacio es **reentrante** (puede ser interrumpido en medio de su ejecución y volver a llamarse de forma segura antes de que las invocaciones anteriores completen su ejecución)

Necesario info sobre estado de la memoria asociada a cada proceso. Esta info se guarda en el espacio de memoria del kernel.

Todos los procesos comparten el mismo espacio de Kernel

Existe una pila de Kernel por cada proceso ya que el kernel es un código reentrant.

Modo	User	Kernel
Contexto		
Proceso	Código de usuario	Llamadas al sistema y excepciones
Kernel	(No permitido)	Tratamiento de Interrupciones, Tareas del sistema.

Ejecución del SO

Núcleo fuera de todo proceso :

- Ejecuta el núcleo del SO fuera de cualquier proceso.
- El código de SO se ejecuta como una entidad separada que opera en modo privilegiado.

Ejecución dentro de los procesos de usuario :

- Software del SO en el contexto de un proceso de usuario.
- Un proceso se ejecuta en modo privilegiado (modo Kernel) cuando se ejecuta el código del SO.

Idea de proceso

Unidad de actividad caracterizada por ejecución de una secuencia de instrucciones (traza de ejecución)
estado de computación actual (contexto de registros)
conjunto de recursos del SO asociados.

En el sistema hay muchos procesos simultáneamente. Cuando el SO decide que un proceso ejecutándose en CPU debe abandonarla, tiene que salvar los valores actuales de los registros de CPU (contexto de registros) en el PCB de dicho proceso.

La acción de comutar la CPU de un proceso a otro se denomina cambio de contexto. El tiempo que el SO emplea en cada cambio de contexto va en detrimento de la productividad del sistema.

Process Control Block (PCB)

Estructura de datos que contiene la info relativa al concepto de proceso.

El PCB es creado, gestionado y destruido por el Kernel.

```
PCB {
    PID;
    ESTADO = {"Listo", "Ejecutándose", "Bloqueado",
              "Nuevo", "Finalizado"};
    LISTA_LISTOS *next;
    LISTA_LISTOS *previous;
    CONTEXTO REGISTROS CPU {
        PC, PSW,
        BP, SP,
        BR, LR (registro base y registro límite);
    }
}
```

Info que contiene PID (Process Identifier)
Estado del proceso
Contexto de registros
Información de memoria
Lista de recursos utilizados

Identifier
State
Priority
Program counter
Memory pointers
Context data
I/O status information
Accounting information

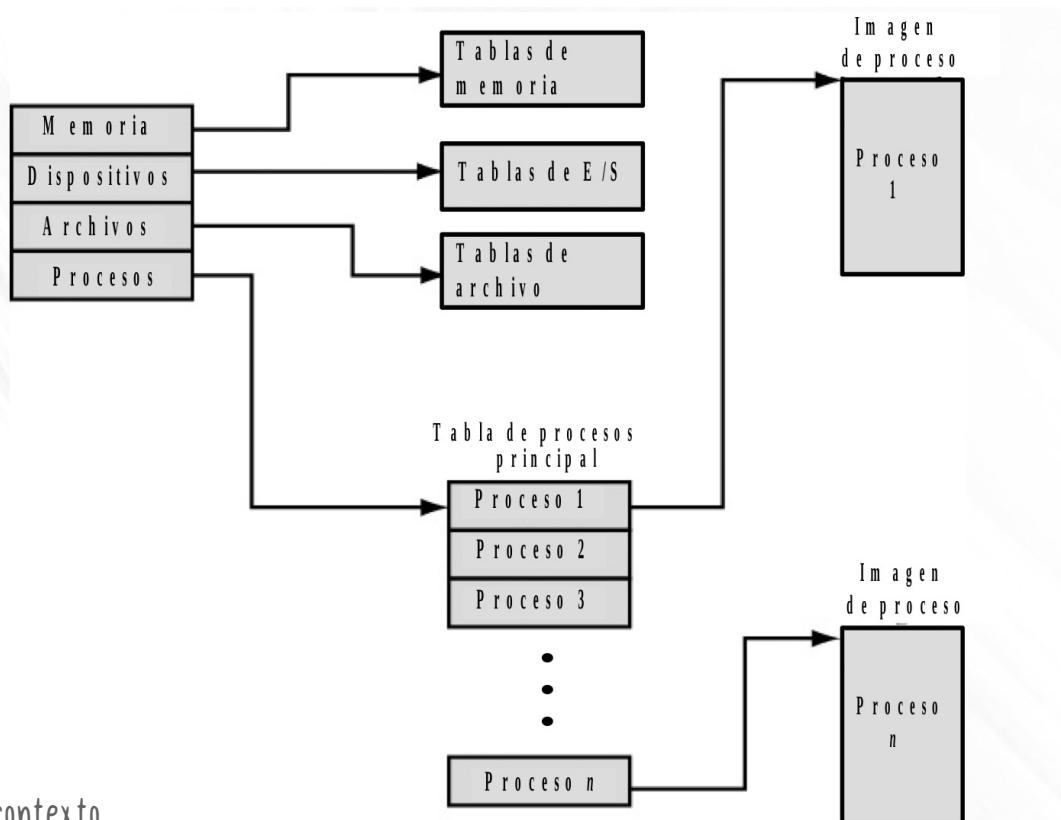
Un proceso tiene

- text (código)
- datos asociados (el programa a ejecutar)
- una pila asociada para poder realizar llamada a funciones

SO almacena el estado de ejecución del programa en el PCB.

text, datos y pila + metadatos = PCB → residen en memoria

Tablas del kernel del SO



Cambio de contexto

Cuando un proceso se está ejecutando, el programa asociado está utilizando registros de la CPU: PC, SP, PSW, ... Los registros contienen el estado actual de la computación.

Cuando el SO detiene un proceso que está ejecutándose, salva los valores actuales de estos registros (contexto registros) en el PCB de dicho proceso.

Cambio contexto → acción de cambiar al proceso que está usando la CPU por otro proceso.

→ se lleva a cabo sólo en modo Kernel.

Es fundamental en SOs multitarea (multitasking SO).

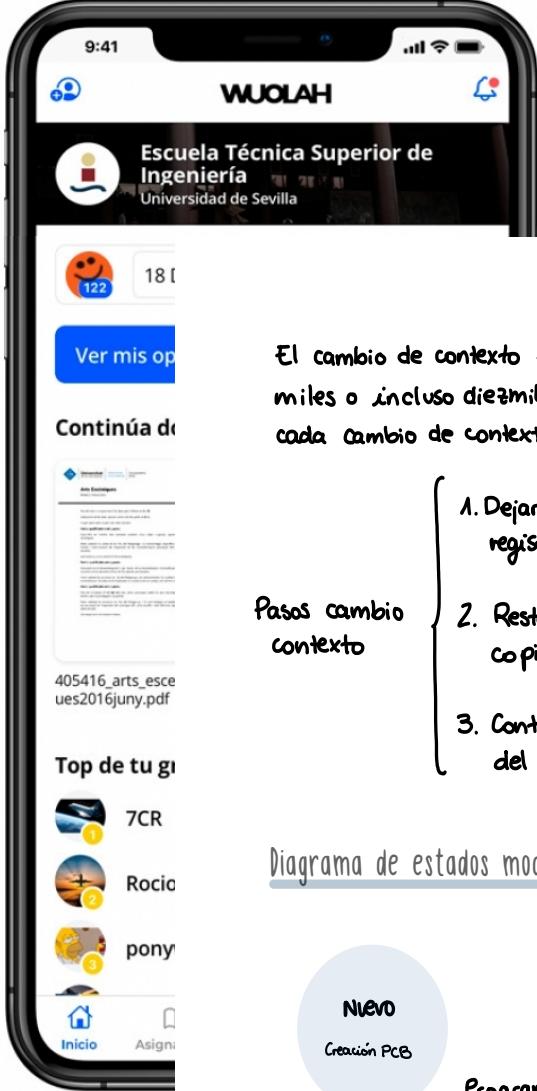
¿Cuándo se produce cambio de contexto?

- Proceso en ejecución ha agotado su time slice.
- Como consecuencia de una llamada al sistema bloqueante.
- Como consecuencia de una interrupción.
- El proceso decide por sí mismo abandonar la CPU.

```

Context_switch() {
    PID=Planificador_CPU();
    Dispatch(PID_CPU,PID);
}
  
```

WUOLAH



Descarga la APP de Wuolah.

Ya disponible para el móvil y la tablet.

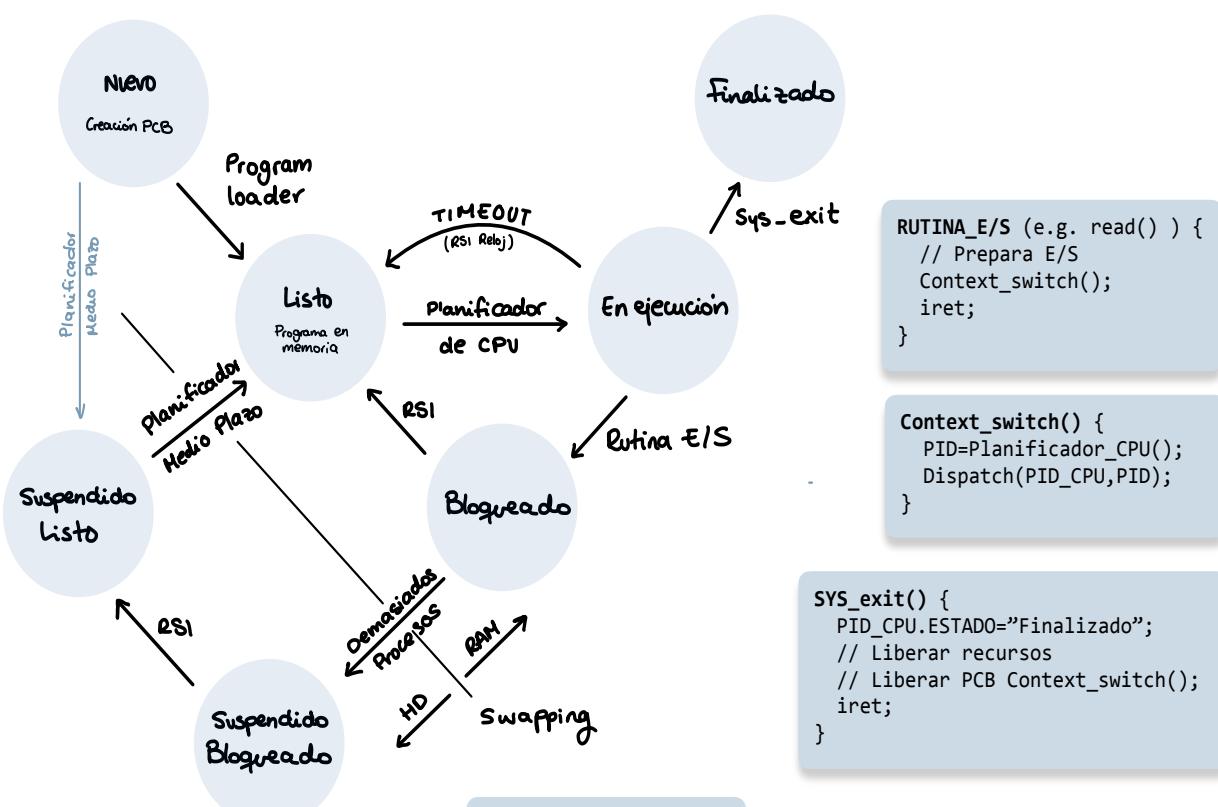
Available on the
App Store

GET IT ON
Google Play

El cambio de contexto representa un alto coste para el sistema en términos de CPU. Se dan cientos, miles o incluso diezmiles de cambio de contexto por segundo (se necesitan nano segundos para cada cambio de contexto).

- Pasos cambio contexto
1. Dejar en suspenso la "ejecución" de un proceso almacenando el contexto de registros en el PCB.
 2. Restaurar el contexto de registros del proceso que va a ejecutarse en CPU copiándolo del que se había guardado previamente en su PCB.
 3. Continuar el ciclo captación-ejecución de instrucciones utilizando el nuevo valor del registro PC.

Diagrama de estados modificado



```

RSI {
    // OK o error
    // Transferencia de info. ...
    // ... ModuloE/S-CPU-RAM
    PID.ESTADO="LISTO";
    encolar(PID,LISTA_LISTOS); iret;
}
  
```

```

RSI_Reloj {
    ...
    // TIMEOUT
    PID_CPU.cont++;
    if (cont >= Q) {
        context_switch();
    }
    ...
}
  
```

```

RUTINA_E/S (e.g. read() ) {
    // Prepara E/S
    Context_switch();
    iret;
}
  
```

```

Context_switch() {
    PID=Planificador_CPU();
    Dispatch(PID_CPU,PID);
}
  
```

```

SYS_exit() {
    PID_CPU.ESTADO="Finalizado";
    // Liberar recursos
    // Liberar PCB
    Context_switch();
    iret;
}
  
```

```

Program_loader {
    // Crea/Reserva un PCB
    // Inicializa PCB
    // Crear la región de PILA
    // Cargar el programa en RAM
    PID.ESTADO="LISTO";
    encolar(ID_proceso,LISTA_LISTOS);
}
  
```

OPERACIONES SOBRE PROCESOS

Creación de procesos

¿Qué significa crear un proceso?

- { Asignarle el espacio de direcciones que utilizará.
- Crear las estructuras de datos para su administración (PCB)

¿Por qué motivos se prede crear un proceso?

- { En sistemas batch → en respuesta a la recepción y admisión de un trabajo
- "Logon" interactivo → un usuario se autentica desde un terminal (log on), el SO crea un proceso que ejecuta el intérprete de órdenes asignado.
- El SO puede crear un proceso para llevar a cabo un servicio solicitado por un proceso de usuario.
- Un proceso puede crear otros procesos formando un árbol de procesos (relación padre-hijo).

Proceso crea nuevo proceso ⇒ ¿Cómo obtiene sus recursos el nuevo proceso (hijo) ?

- Los obtiene directamente del SO ⇒ padre e hijo no comparten recursos
- Comparte todos los recursos con el padre.
- Comparte un subconjunto de recursos con el padre

Possibilidades de ejecución

- { Padre e hijo se ejecutan concurrentemente
- Padre espera a que el hijo termine

Possibilidades de espacio de direcciones

- { Espacio direcciones hijo es un "duplicado" del padre (UNIX-like OS)
- Espacio del hijo → programa nuevo distinto al del programa asociado al proceso padre (VMS)

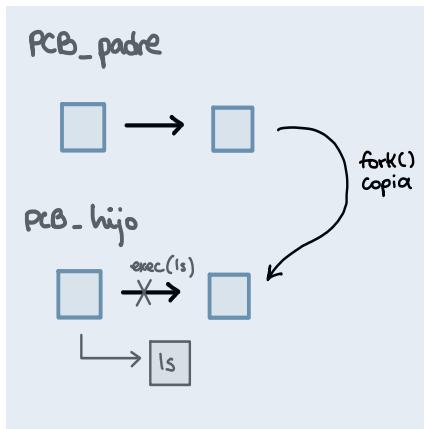
¿Cómo funciona UNIX-like OS?

- Llamada al sistema fork() → crea un nuevo proceso (hijo).
- Llamada exec → reemplaza el espacio de direcciones con el programa pasado como argumento.
- Si se realiza una llamada fork() seguida de una exec(), se consigue un proceso hijo que ejecuta el programa pasado como argumento.

`fork()` crea un nuevo hijo que hereda una copia idéntica de la memoria del padre y los registros de la CPU del padre

Procesos padre e hijo se ejecutan en el mismo punto de ejecución: tras el retorno de `fork()`.

`fork()` devuelve { 0 en el proceso hijo
PID_proceso_hijo en el proceso padre }



pid = fork();
if (pid == 0)
 exec("ls", "ls", NULL);
else
 // código padre

OR

pid = fork();
if (pid == 0)
 exec("ls", "ls", NULL);
// código padre

Como `exec` carga un programa nuevo libera el espacio de direcciones copiado del padre. Por lo que no seguiría ejecutando el resto del código del programa donde se creó.

Ejemplo: Uso de fork

```
main(int argc, char* argv[]) {
    pid_t forkRetVal; //Stores the value returned by fork() syscall
    //fork() creates a new process named child process.
    //The process which creates is named the parent process
    forkRetVal = fork();
    //fork() returns 0 in the child process and the child process' PID in the parent process.
    if (forkRetVal == 0) { //This code is ONLY executed by the child process
        //Things to do exclusively by child process
        printf("Child -- Hello, I was born! forkRetVal = %d\n", forkRetVal);
    } else { //forkRetVal != 0, then this code is ONLY executed by the parent process
        //Things to do exclusively by the parent process.
        printf("Parent -- Hello, I created a child! forkRetVal = %d\n", forkRetVal);
    } //This code is executed both by the child and parent processes
    printf("Bye, bye!\n");
    return EXIT_SUCCESS;
}
```

Pasos a realizar por el Kernel cuando se solicita la creación de un proceso

Asignar un PID único para identificarlo.

Asignar espacio en RAM y/o en memoria secundaria (SWAP) para el programa que se ejecutará.

Crear el PCB e inicializar los campos de información.

Insertar el PCB en la Tabla de Procesos y enlazarlo en la cola de planificación correspondiente, en caso de que el programa resida en RAM.

Terminación de procesos

¿Qué situaciones determinan la finalización de un proceso?

- Proceso ejecuta su última instrucción, solicita al SO su finalización mediante la llamada `exit()`, lo que implica:
 - Aviso de finalización al padre (`SIGCHLD`) y guardar estado de finalización.
 - Recursos asociados al proceso son liberados por el SO.
- El proceso padre puede finalizar la ejecución de sus procesos hijos mediante `Kill()`.
- El proceso padre va finalizar y el SO no permite a los procesos hijos continuar con la ejecución (Terminación en cascada).
- El SO puede terminar la ejecución de un proceso porque se hayan producido errores o condiciones de fallo.

THREADS (HEBRAS O HILOS)

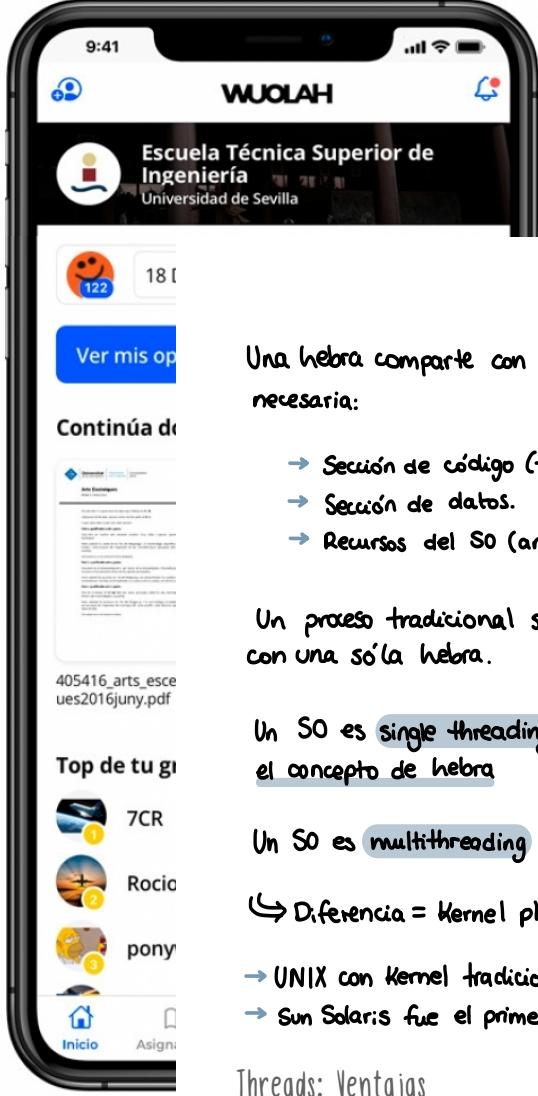
Hebra → sub básica de utilización de la CPU

El Kernel requiere para gestionarlas
↳ TCB

Contexto de registros → CP, SP, PSW, ...
Pila de ejecución
Estado (diagrama de estados).

```
// PCB multithreading
PCB {
    PID (el ID_proceso);
    ESTADO = {"Nuevo", "Finalizado",
              "Suspendido_LISTO", "Suspendido_BLOQUEADO"};
    // Espacio de direcciones (TEXTO + DATOS)
    // SWAPPING
    Bloques de disco ocupados
    // Resto recursos (archivos, señales,...)
    LISTA de Thread Control Block (TCB)
    main_thread_TCB
    ...
}
```

```
TCB {
    TID (el ID_hebra);
    ESTADO = {"LISTO", "Ejecutándose", "Bloqueado",
              "Nuevo", "Finalizado", ...};
    CONTEXTO REGISTROS CPU {
        PC, PSW
        BP, SP //Pila de ejecución
    }
    LISTA_LISTOS *next; LISTA_LISTOS *previous;
}
```



Descarga la APP de Wuolah.

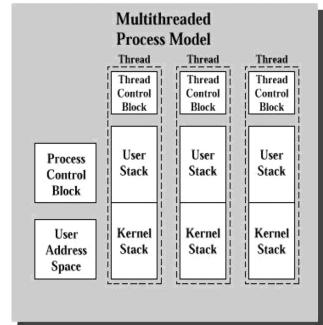
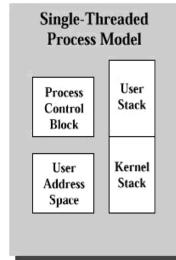
Ya disponible para el móvil y la tablet.

Available on the App Store GET IT ON Google Play

Una hebra comparte con sus hermanas pares una tarea (proceso) que mantiene el resto de información necesaria:

- Sección de código (texto).
- Sección de datos.
- Recursos del SO (archivos abiertos, señales, ...).

Un proceso tradicional se puede ver como una tarea con una sola hebra.



Un SO es **single threading** cuando el Kernel no reconoce el concepto de hebra

Un SO es **multithreading** cuando el Kernel soporta múltiples hebras dentro de un proceso (tarea).

↳ Diferencia = Kernel planifica o no hebras.

- UNIX con Kernel tradicional soporta múltiples procesos de usuario pero sólo una hebra por proceso.
- Sun Solaris fue el primero en soportar múltiples hebras por proceso.

Threads: Ventajas

- Se reduce el tiempo de

Cambio de contexto (entre hebras de la misma tarea)

Cada vez que hay que cambiar de hebra (de una tarea distinta) no se reduce, porque hay que hacer flush de cachés.

Creación

Lo que más cuesta al crear un PCB es que hay que cargar programa en RAM.
Las hebras no necesitan cargarse en RAM.

Terminación

No hace falta liberar ya que el proceso se asocia en PCB no en TCB

- Mientras una hebra de una tarea está bloqueada, otra hebra de la misma tarea puede ejecutarse, siempre que el Kernel sea multithreading.
- Usan los sistemas multiprocesador de manera eficiente y transparente, a mayor número de procesadores, mayor rendimiento (performance).
- Comunicación entre hebras de una misma tarea se realiza a través del espacio de direcciones asociado a la tarea por lo que no necesitan utilizar los mecanismos del núcleo.

Threads: funcionalidad

Al igual que los procesos, las hebras poseen un estado (diagrama de estados) y pueden sincronizarse.

Estados de hebras

- Nuevo
- Ejecutándose
- Listo
- Bloqueado
- Finalizado

Operaciones básicas de cambio de estado en hebras

- Creación
- Bloqueo
- Desbloqueo
- Terminación

WUOLAH

Threads: Tipos

Hebras de biblioteca de usuario, User-level threads (ULT).

Toda la gestión = a nivel de usuario mediante uso de biblioteca de hebras.

Biblioteca → proporciona operaciones para

- Crear/finalizar hebras.
- Gestionar el modelo de estados de las hebras.
- Planificar hebras y salvar/cargar el contexto de hebra.
- Permitir la comunicación entre hebras.

El Kernel no es consciente de las actividades relacionadas con hebras ya que sólo gestiona el proceso (tarea).

La entidad de planificación para el Kernel es el proceso.

Ventajas:

- Cambio de hebra no provoca cambio de modo.
- Planificación de hebras se adapta a las necesidades de la aplicación.
- Las aplicaciones se pueden ejecutar en cualquier So.

Inconvenientes:

- Mayoria de llamadas al sistema son bloqueantes ⇒ si una hebra realiza una llamada el resto de hebras se bloquean.
- El Kernel sólo asigna procesos a procesadores ⇒ no se puede asignar más de un procesador a más de una hebra de la misma tarea.

Hebras a nivel de kernel, Kernel-level threads (KLT)

Toda la gestión = a nivel de Kernel ⇒ tiene que mantener info para procesos (tareas) y hebras.

El So proporciona un conjunto de llamadas al sistema para la gestión se realiza a nivel Kernel por lo que este tiene que mantener información para procesos y hebras.

La entidad de planificación para el Kernel es la hebra.

Muchas de las funciones que realiza el Kernel son gestionadas mediante hebras Kernel.

Ventajas:

- Kernel puede planificar distintas hebras de la misma tarea en distintos procesadores.
- Bloqueo de una hebra de una tarea no provoca el bloqueo del resto de "hebras pares".
- Rutina del Kernel pueden ser multihébra.



Inconvenientes:

- El cambio de hebras dentro de la misma tarea se realiza en modo kernel, por lo que provoca un cambio de modo.

Enfoques híbridos

Solaris OS = ejemplo combinación ULT y KLT.

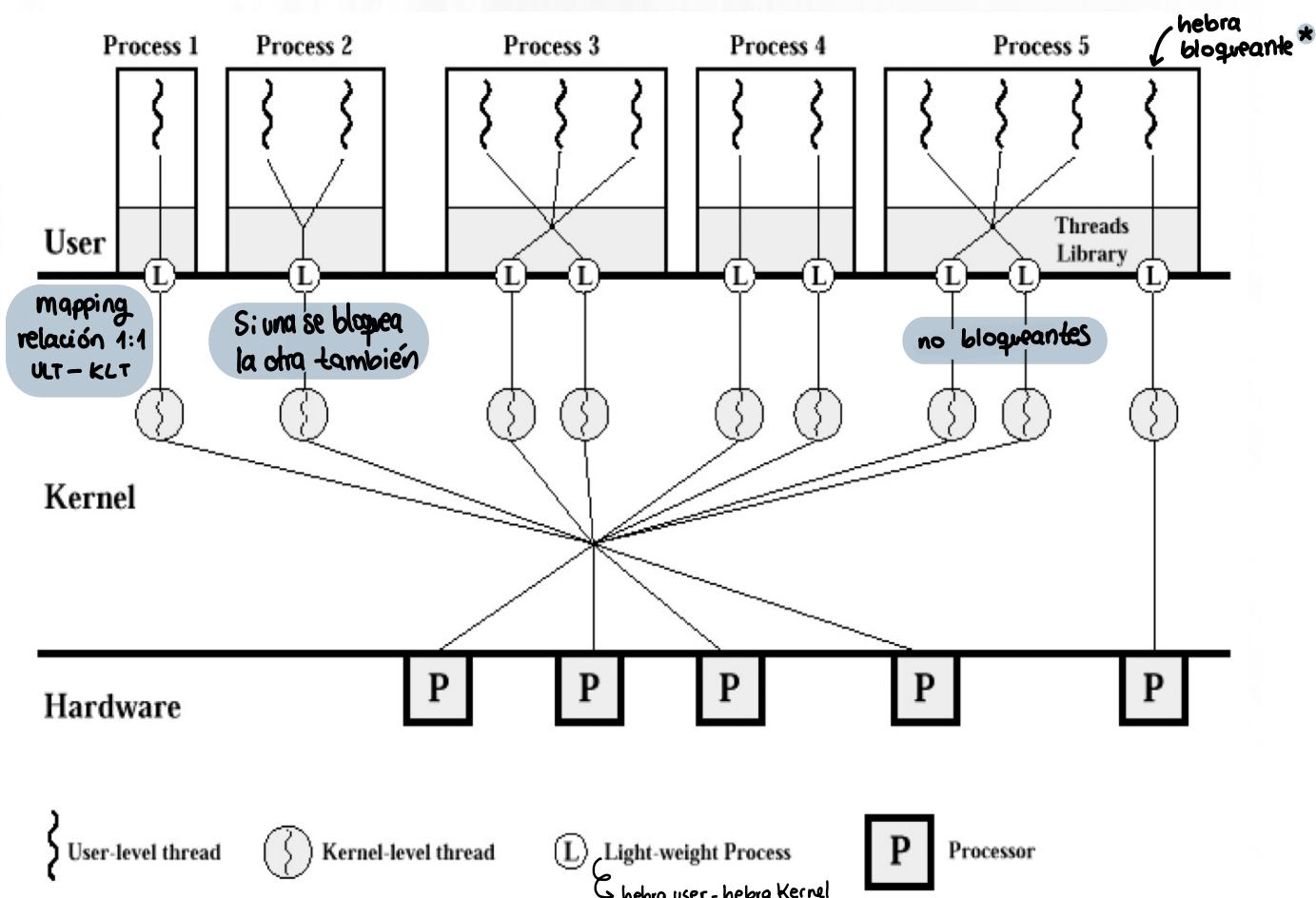
Las ULT proporcionadas por la biblioteca de hebras son totalmente invisibles para el Kernel (API de hebras).

Creación de hebras y mayor parte de la planificación y sincronización de hebras se realiza en modo usuario. Las KLT son la unidad de planificación del Kernel.

El programador puede decidir el nº de KLTs.

Los "procesos ligeros" (Lightweight processes, LWP) soportan una o más ULT y se asocian con una KLT.

El concepto LWP permite que las hebras de una tarea potencialmente bloqueantes no bloquen al resto de las hebras de la tarea.



* Separamos la hebra que realizará E/S para que no bloquee al resto, va directa a CPU

CONCEPTOS SOBRE PLANIFICACIÓN

Planificación

Idea → Disponemos de 'n' clientes que desean acceder a un recurso y tenemos que decidir a qué cliente le asignamos el recurso

Problema de Planificación de CPU

- { SO dispone de n procesos/hebras en estado 'LISTO'.
- SO dispone $K \geq 1$ CPUs (o cores) para ejecutar procesos/hebras.
- SO debe decidir qué proceso/hebra asignar a qué CPU.

PCB's y colas de estados

MODELO GENÉRICO DE COLAS

- { SO mantiene una colección de colas que representan el estado de todos los procesos en el sistema.
- Hay una cola por estado del proceso (sólo en Listo, Suspenido-Listo y Bloqueado)
- Cada PCB está encolado en una cola de estado acorde a su estado actual.
- Conforme un proceso cambia de estado, su PCB es retirado de una cola y encolado en otra.

COLAS DE ESTADOS

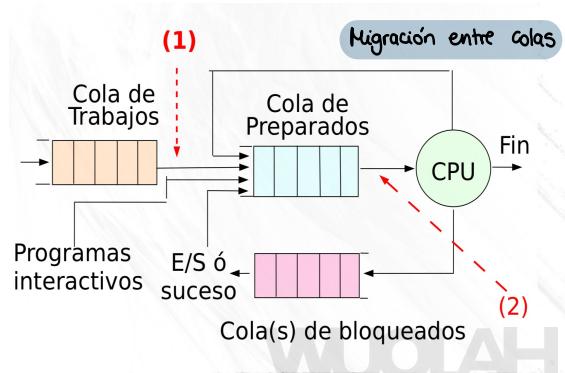
- { Cola de trabajos → Trabajos pendientes de ser admitidos (especiales con info de qué recursos necesitamos) ~ trabajos por lotes (batch)
- Cola de LISTOS (Preparados para Ejecutar) → Conjunto de todos los procesos, cuyos programas asociados residen en memoria principal, esperando para poder ejecutar en la CPU.
- Cola de BLOQUEADOS → Conjunto de todos los procesos esperando por un dispositivo de E/S particular o por un evento que debe producirse para poder continuar su ejecución

Planificador: tipos

PLANIFICADOR → Parte del SO que controla la utilización de un recurso

Tipo

- { Planificador a Largo Plazo → Selecciona trabajos para llevarlos a la cola de preparados (1)
- Planificador a Corto Plazo (de CPU) → Selecciona el proceso que debe ejecutarse a continuación y le asigna la CPU (2)
- Planificador a Medio plazo





Descarga la APP de Wuolah.

Ya disponible para el móvil y la tablet.

Available on the
App Store

GET IT ON
Google Play

Características de los planificadores

Planificador de CPU

- Trabaja con la Cola de LISTOS
- Se invoca muy frecuentemente (milisegundos) por lo que debe ser rápido.

Planificador a Largo plazo

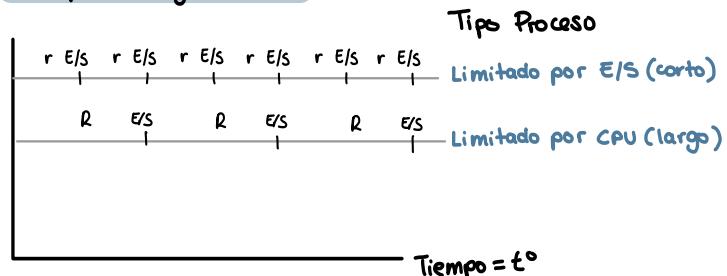
- Permite controlar el grado de multiprogramación → recurso que hay disponibles.
- Se invoca poco frecuentemente (segundos o minutos), por lo que puede ser más lento.

Clasificación de procesos

Procesos limitados por

- E/S o procesos cortos
 - Dedican más tiempo a realizar E/S que cómputo
 - Muchas ráfagas de CPU cortas
 - Largas procesos de espera
- CPU o procesos largos
 - Dedican más tiempo a cómputos que a E/S
 - Pocas ráfagas de CPU pero largas.

Concepto: Ráfaga de CPU



Mezcla de trabajos

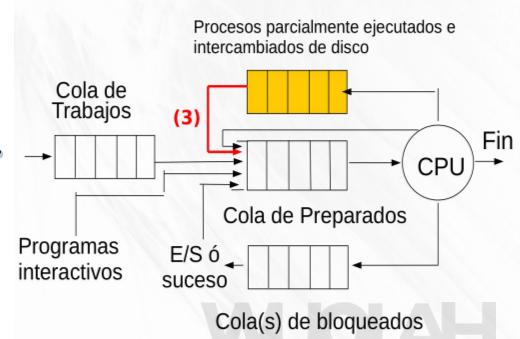
Importante que Planificador a Largo plazo seleccione una buena mezcla de trabajos, ya que:

- Si todos los procesos están Limitados por
 - E/S ⇒ cola de preparados estará casi siempre vacía y planificador a Corto plazo tendrá poco que hacer = CPU inactiva mucho tiempo
 - CPU ⇒ cola E/S casi siempre vacía = sistema desequilibrado de nuevo.

Planificador a Medio Plazo

En algunos SO's (por ejemplo en tiempo compartido), a veces es necesario sacar procesos de la memoria (reducir el grado de multiprogramación), bien para mejorar la mezcla de procesos, bien por cambio en los requisitos de memoria, y luego volverlos a introducir ⇒ Intercambio (swapping).

El Planificador a Medio plazo se encarga de devolver los procesos a memoria (3).



Despachador (dispatcher())

El Dispatcher = función del SO que da el control de la CPU al proceso seleccionado por el planificador a corto plazo (CPU).

Involucra { Cambio contexto de registros desde el punto de vista de la CPU (en modo Kernel)
Comutación a modo usuario o a modo Kernel dependiendo del nuevo contexto de registros.

Latencia de Despacho

→ tiempo que emplea el dispatcher en detener un proceso y comenzar a ejecutar otro

Activación del dispatcher

actúa cuando

{ Proceso no quiere o no puede continuar ejecutando instrucciones. En el segundo caso, la solicitud de recurso o comunicación por parte del proceso provoca que el SO lo pase a Bloqueado.
Un elemento del SO determina que el proceso no puede seguir ejecutándose (E/S o retirada de HP)
El proceso agota el quantum de tiempo asignado (time slice)
Un suceso cambia el estado de un proceso de Bloqueado a Listo.

Políticas de planificación: medidas

Objetivos { Buen rendimiento (productividad)
Buen servicio

Para saber si un proceso obtiene un buen servicio, definiremos un conjunto de medidas:

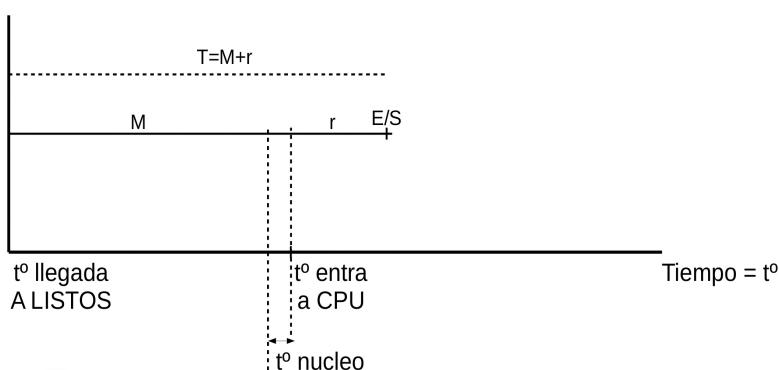
Dado un proceso P que necesita un tiempo de servicio (ráfaga) r.

Tiempo de respuesta (T) → tiempo transcurrido desde que se remite una solicitud (entra en la cola de listos) hasta que se produce la primera respuesta (no se considera el tiempo que tarda en dar la salida).

Tiempo de espera (M) → tiempo que un proceso ha estado esperando en la cola de listos (preparados): $T - r$

Penalización (P) → T/r

Índice de respuesta (R) → r/T : fracción de tiempo que P está recibiendo servicio.



$$\text{Penalización} \rightarrow P = \frac{T}{r} = \frac{(M+r)}{r}$$

Si $M=0 \Rightarrow P=1$
Si $M \rightarrow \infty \Rightarrow P \rightarrow \infty$
(\hookrightarrow Dominio $[1, \infty)$)

$$\text{Índice respuesta} \rightarrow R = \frac{r}{T} = \frac{r}{M+r}$$

Si $M=0 \Rightarrow R=1$
Si $M \rightarrow \infty \Rightarrow R \rightarrow 0$
(\hookrightarrow Dominio $[0, 1]$)

A = va bien
Si pasa de 0.5 no tiene buen servicio

Otras medidas interesantes

- tiempo de planificación CPU
tiempo de dispatch } context switch
- Tiempo del núcleo → tiempo perdido por el SO tomando decisiones que afectan a la planificación de procesos y haciendo los cambios de contexto necesarios. En un sistema eficiente debe representar entre el 10% y el 30% del total del tiempo del procesador.
- no suele ocurrir si no hay nada que hacer hay → procesos estadísticos.
- Tiempo de inactividad → tiempo en el que la coda de ejecutables está vacía y no se realiza ningún trabajo productivo.
- Tiempo de retorno (turnaround time) → cantidad de tiempo necesario para ejecutar un proceso completo.

Políticas de planificación

Se comportan de distinta manera dependiendo de la clase de procesos (cortos o largos).

Ninguna política de planificación es completamente satisfactoria, cualquier mejora en una clase de procesos supone perder eficiencia en las de otra clase.

Se clasifican en

- No apropiativas (no expulsivas) → una vez que se le asigna el procesador a un proceso, no se le puede retirar hasta que éste voluntariamente lo deje (finalice o se bloquee).
- Apropiativas (expulsivas) (preemptive) → el SO puede apropiarse del procesador cuando lo decide

Reparso context_switch()

context_switch() ↗
schedule () == planificador CPU
dispatch () == despachador

(a) Context_switch() {
PID=Planificador_CPU();
Dispatch(PID_CPU,PID);
}

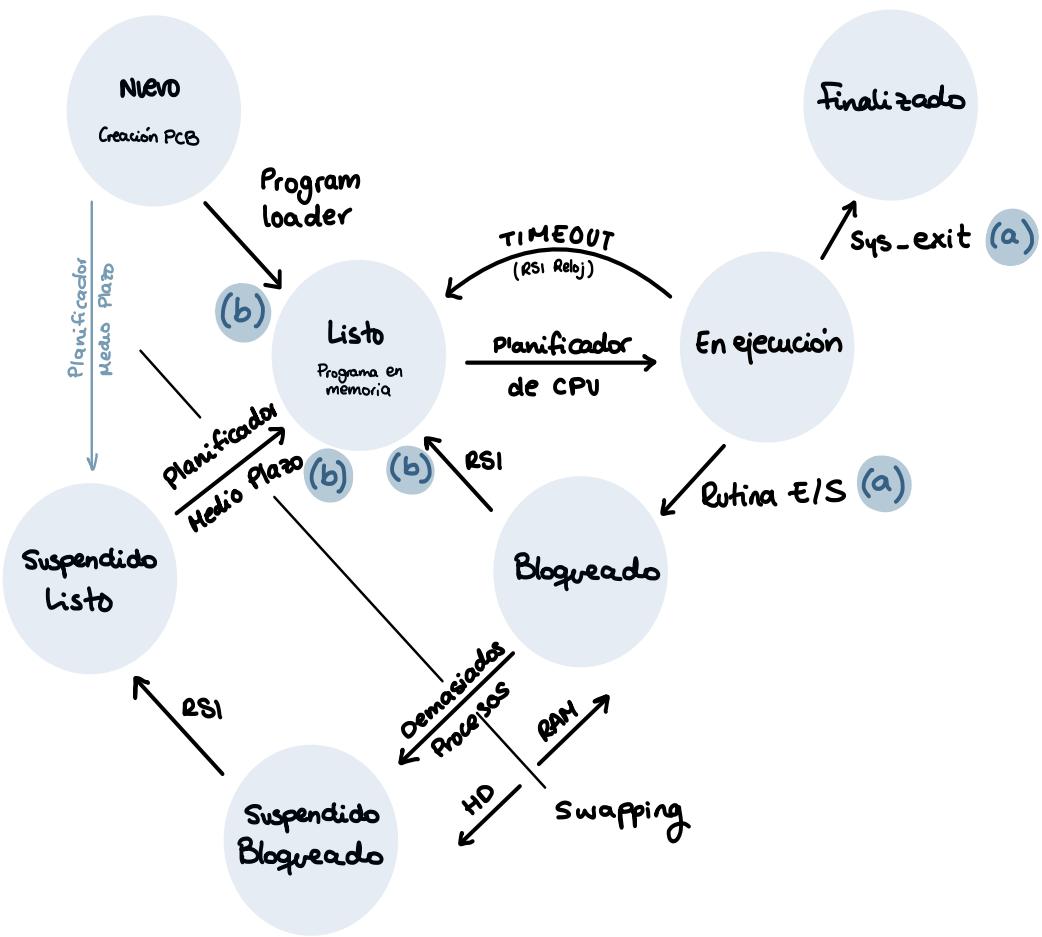
¿Cuándo toma decisiones de planificación el SO?

- Ejecutándose → Bloqueado ⇒ dentro de Rutina E/S() o wait() syscall.
- Ejecutándose → finalizado ⇒ sys_exit() kernel algorithm
- Ejecutándose → Listo ⇒ dentro de RSI_reloj().
- { Nuevo, Bloqueado, Suspended Listo } → Listo ⇒ implementando CPU preemption (apropiación CPU)

Si el planificador es ↗
no apropiativo → sólo se tienen en cuenta 1 y 2.
apropiativo → se tienen en cuenta todas 1-4

(b) Preemptive Kernel:
RSI_Reloj() || program_loader() || planif_medio-plazo(){
...
if(Planif_CPU (PID_VaEntrarAListos, PID_CPP) == true)
 dispatch(PIC_CPU, PID_VaEntrarAListos);
else{
 PID_ESTADO = "LISTO";
 encolar(PID.LISTA_LISTOS);
}
}





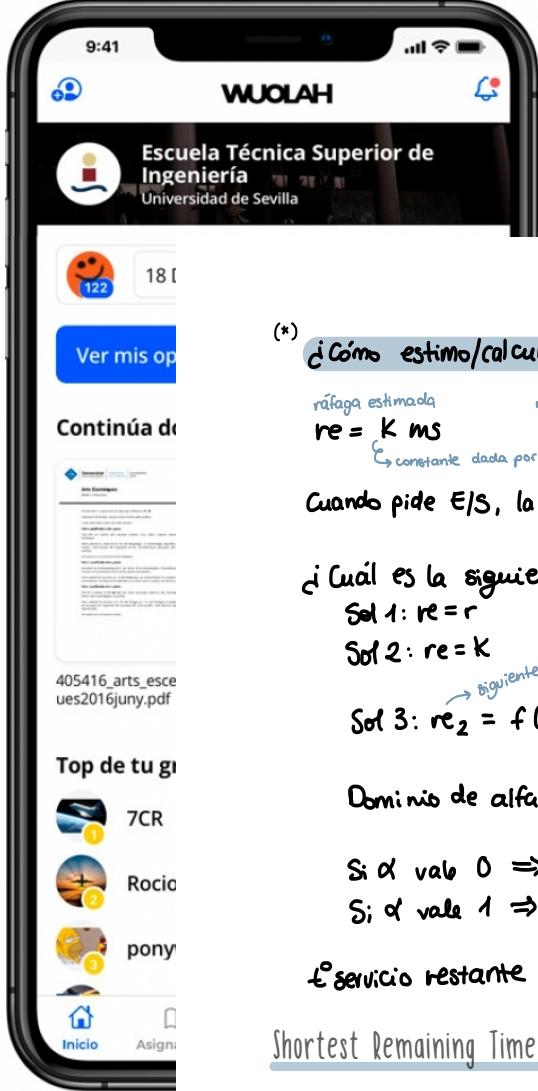
POLÍTICAS DE PLANIFICACIÓN DE LA CPU

First Come First Served (FCFS)

- Proceso son servidos según el orden de llegada a la cola de ejecutables.
- No apropiativo → cada proceso se ejecutará hasta que finalice o se bloquee (decisiones → (a) context-switch)
- Fácil de implementar pero pobre en cuanto a prestaciones.
- Todos los procesos pierden la misma cantidad de tiempo esperando en la cola de ejecutables independientemente de sus necesidades.
- Procesos cortos muy penalizados.
- Procesos largos poco penalizados.

Shortest Job first (SJF)

- No apropiativo
- Cuando procesador queda libre, selecciona el proceso que requiera un **tiempo de servicio menor**.
- Si existen dos o más procesos en igualdad de condiciones, se sigue FCFS.
- Necesita conocer explícitamente el tiempo estimado de ejecución (+° servicio) ¿Cómo? (*)
- Disminuye el tiempo de respuesta para los procesos cortos y discrimina los largos.
- Tiempo medio de espera bajo
↳ mejor en minimizar tiempo de espera medio ⇒ Si se ordena la lista por T° servicio ⇒ O(1) algoritmo de Planificación.



Descarga la APP de Wuolah.

Ya disponible para el móvil y la tablet.

Available on the
App Store

GET IT ON
Google Play

(*) ¿Cómo estimo/calculo el tº servicio?

$$\begin{array}{ll} \text{ráfaga estimada} & \text{ráfaga real} \\ \text{re} = K \text{ ms} & r = ?? \\ \text{constante dada por el sistema (todos iguales)} \end{array}$$

PCB {

...
// Parámetros de planificación de SJF
tº de servicio = ráfaga estimada = re = k ms inicialmente
r=?

}

Cuando pide E/S, la ráfaga real que ha realizado en CPU la sabe el kernel (r_{ms})

¿Cuál es la siguiente re?

Sol 1: $re = r$

Sol 2: $re = K$

$$\begin{aligned} \text{Sol 3: } re_2 &= f(re_1, r) = (1-\alpha) re_1 + \alpha r \\ &\quad \xrightarrow{\text{siguiente}} \xrightarrow{\text{previa}} \\ &\quad \text{establecido por el sistema} \end{aligned}$$

$$re_i = (1-\alpha) re_{i-1} + \alpha r_i$$

Dominio de alfa [0, 1]

Si α vale 0 $\Rightarrow re_i = re_{i-1}$

Si α vale 1 $\Rightarrow re_i = r_i$

tº servicio restante = tº restante - tiempo que llevaba en CPU

Shortest Remaining Time first (SRTF)

Cada vez que entra un proceso a la cola de listos se comprueba si su tiempo de servicio es menor que el tiempo de servicio que le queda al proceso que está ejecutándose.

↓ { Si es menor → cambio de contexto y proceso de menor tiempo de servicio se ejecuta.
No es menor → continúa el proceso que estaba ejecutándose.

- El tiempo de respuesta es menor excepto para procesos muy largos.
- Se obtiene la menor penalización en promedio (mantiene cola de ejecutables con la mínima ocupación posible).

```
PCB {
    ...
    // Parámetros de planificación de SJRF
    tº de servicio = re
    tº de servicio restante = re inicialmente
    Para evitar starvation (Si M para un umbral)
    tº de servicio restante = 1 ms
}
```

SOLUCIONAR STARVATION (inanición)

Cuando lleva un tiempo en cola (M) mayor a un umbral se le asigna un tiempo restante de 1ms para que pase directamente a ejecutarse recuperando su tiempo restante real.

Planificación por prioridades

- Asociamos a cada proceso un número de prioridad (entero).
- Se asigna la CPU al proceso con mayor prioridad (enteros menores = mayor prioridad)
- Se puede implementar con modalidades { Apropiativa
No apropiativa }

PROBLEMA: Inanición → procesos baja prioridad pueden no ejecutarse nunca.

SOLUCIÓN: Envejecimiento → con el paso del tiempo se incrementa la prioridad de los procesos (prioridad 1 cuando pasa un tiempo determinado en cola)

WUOLAH

Planificación por turnos (Round-Robin)

- CPU asigna a los procesos en intervalos de tiempo (quantum)
- Es apropiativo.
- Se implementa en la RSI_reloj().

- Procedimiento
 - Si el proceso finaliza o se bloquea antes de agotar el quantum, libera la CPU. Se toma el siguiente proceso de la cola de ejecutables (FIFO) y se le asigna un quantum completo.
 - Si el proceso no termina durante ese quantum, se apropia y se coloca al final de la cola de ejecutables.
- Quantum entre 10 y 100 ms (normalmente)
- Penaliza a todos los procesos en la misma cantidad, sin importar si son cortos o largos.
- Ráfagas muy cortas están más penalizadas de lo deseable.

dónde elegir el valor del quantum?

- Muy grande (excede del t^o de servicio de todos los procesos) \Rightarrow se convierte en FCFS
- Muy pequeño \Rightarrow sistema monopoliza la CPU haciendo cambios de contexto (t^o de núcleo muy alto).

Ejemplo:

rrrrRRRrrr $\rightarrow n=9$
va a tardar como mucho $H \leq (n-1)*Q = 8Q$ (quantums)

- $r < Q$ /
- $2Q < r < 3Q$ regular \Rightarrow gasta mucho t^o nucleo que no debería
- R sin problema

Colas múltiples (Multilevel Queue)

- Cola de listos se divide en varias colas y cada proceso es asignado permanentemente a una cola concreta. Por ejemplo \rightarrow 2 colas: Procesos interactivos y procesos batch.
- Cada cola puede tener su algoritmo de planificación P. Ejemplo \rightarrow interactivos RR y batch con FCFS.

- Requiere planificación entre colas
 - Planificación con prioridades fijas. Por ejemplo \rightarrow 1º servimos a los interactivos luego a los batch
 - Tiempo compartido entre colas. Cada cola obtiene cierto tiempo de CPU que debe repartir entre sus procesos. Por ejemplo 80% interactivos en RR y 20% a los batch con FCFS.

WUOLAH

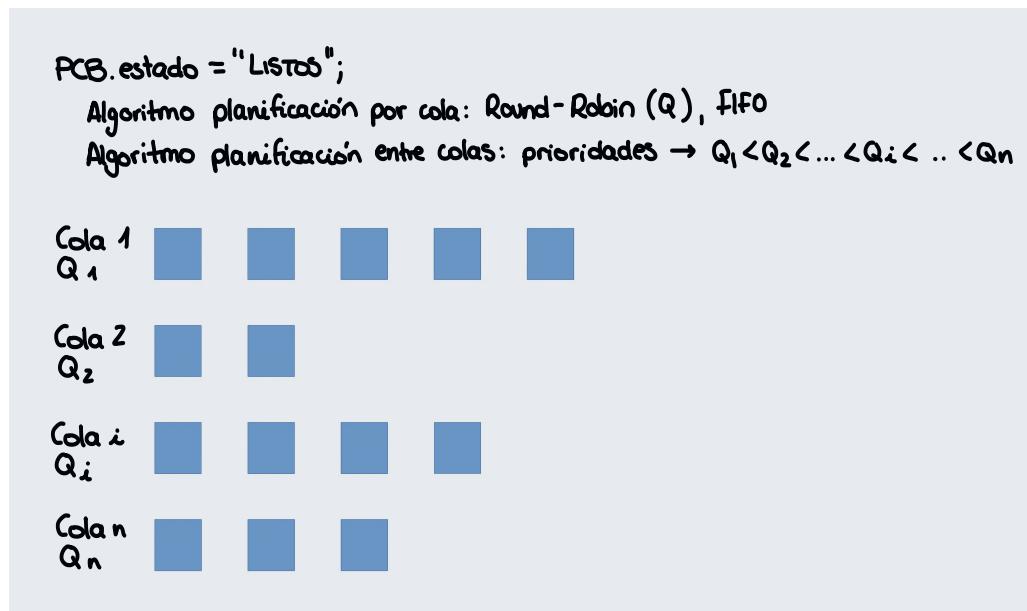
Colas múltiples con realimentación (Multilevel Feedback Queue)

- Proceso se puede mover entre las distintas colas.

Requiere definir los siguientes parámetros

- Número de Colas.
- Algoritmo de planificación para cada cola.
- Método utilizado para determinar cuando trasladar a un proceso a otra cola.
- Método utilizado para determinar en qué cola se introducirá un proceso cuando necesite un servicio.
- Algoritmo de planificación entre colas.

- Mide el tiempo de ejecución el comportamiento real de los procesos.
- Planificación generalmente usada en Unix, Linux, Windows,...



Planificación en multiprocesadores

- Planificación de procesos → Igual que en monoprocesadores pero teniendo en cuenta
 - no de CPUs
 - Asignación/Liberación proceso-procesador
- Planificación por hilos → Permiten explotar el paralelismo real dentro de una aplicación con múltiples hebras
- Granularidad = frecuencia de sincronización entre entidades planificables
 - Grano grueso → procesos con baja sincronización.
 - Grano medio → Hebras con alta sincronización.

Tres aspectos de diseño del planificador interrelacionados

Asignación de procesos a procesadores:

- Cola dedicada para cada procesador.
- Cola global para todos los procesadores.

Uso de multiprogramación en cada procesador individual. En aplicaciones multihilo (grano medio) es más importante aumentar el tiempo de respuesta que el uso de CPU → todas las hebras deben tener asignado un procesador independiente.

Activación del proceso: `schedule()`. Nuevos aspectos de diseño debido a aplicaciones multihilo.

→ Planificador CPU

Cuatro Soluciones

1) Compartición de carga

Cola global de hilos preparados (LISTOS)

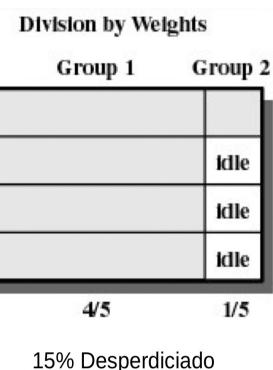
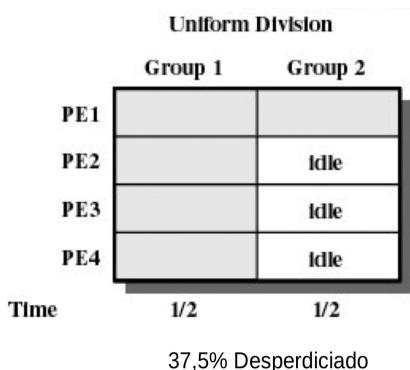
Cuando un procesador está ocioso, se selecciona un hilo de la cola (método muy usado)

2) Planificación en pandilla

Se planifica un conjunto de hilos afines (de un mismo proceso) para ejecutarse sobre un conjunto de procesadores al mismo tiempo (relación 1 a 1)

Útil para aplicaciones cuyo rendimiento se degrada mucho cuando alguna parte no puede ejecutarse (los hilos necesitan sincronizarse).

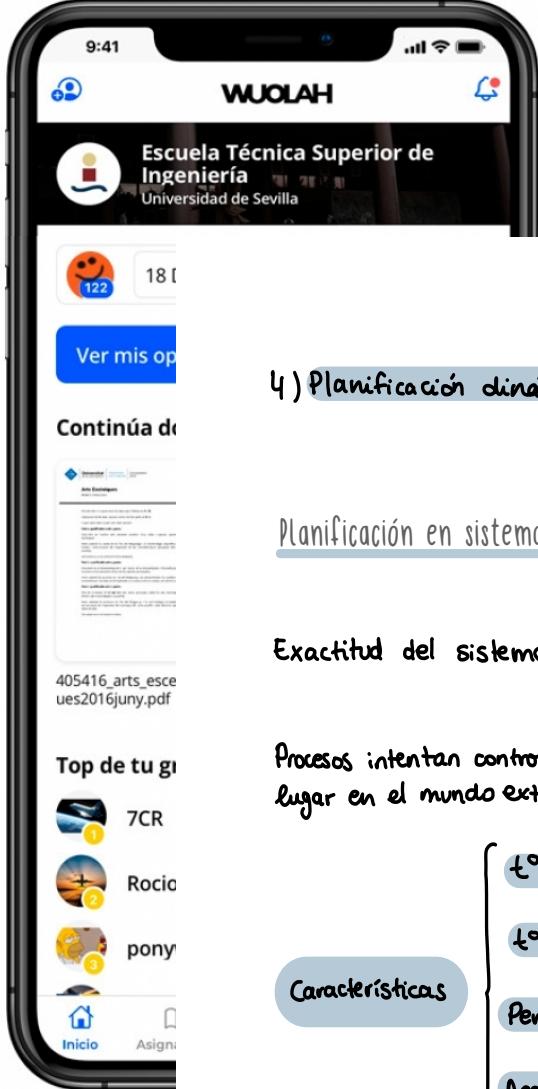
Ejemplo planif. pandilla:



3) Asignación de procesador dedicado

Cuando se planifica una aplicación, se asigna un procesador a cada uno de sus hilos hasta que termine la aplicación.

Algunos procesadores pueden estar ociosos → no hay multiprogramación de procesadores



Descarga la APP de Wuolah.

Ya disponible para el móvil y la tablet.

Available on the App Store

GET IT ON Google Play

4) Planificación dinámica

La aplicación permite que varíe dinámicamente el número de hilos de un proceso.
El SO ajusta la carga para mejorar la utilización de los procesadores

Planificación en sistemas de tiempo real

Exactitud del sistema depende de

resultado lógico de un cálculo
+
instante en que se produce el resultado

Procesos intentan controlar o reaccionar ante sucesos que se producen en "tiempo real" y que tienen lugar en el mundo exterior.

Características

tº real duro (hard rt) → tarea debe cumplir su plazo límite
tº real suave (soft rt) → tarea tiene un tiempo límite pero no es obligatorio cumplirlo
Periódicas → se sabe cada cuánto tiempo se tiene que ejecutar.
Aperiódicas → Tiene un plazo en el que debe comenzar o acabar o restricciones respecto a esos tiempos pero son impredecibles.

Distintos enfoques dependen de

Cuando el sistema realiza un análisis de visibilidad de la planificación → Estudia si puede atender a todos los eventos periódicos dado el tiempo necesario para ejecutar la tarea y el periodo
Si el análisis se realiza estáticamente o dinámicamente
→ antes de comenzar → en tiempo de ejecución ejecución
Si el resultado produce un plan de planificación o no
→ reordenación de las tareas en la cola de listos.

Tipos de enfoques:

Enfoques estáticos dirigidos por tabla

Análisis estático de la factibilidad de la planificación

Resultado = planificación que determina cuándo, en tiempo de ejecución, debe comenzar a ejecutarse cada tarea.

Aplicable a tareas periódicas

Datos entrada del análisis

tiempo periódico de llegada.
tiempo de ejecución.
plazo periódico de finalización.
prioridad relativa de cada tarea.

Planif Q plan que permita cumplir todos los requisitos de todas las tareas periódicas

Enfoque predecible pero no flexible

→ cualquier cambio en requisitos = rehacer toda la planificación

Enfoques estáticos expulsivos dirigidos por prioridad

Análisis estático de la factibilidad de la planificación

No se obtiene una planificación

Tareas periódicas

Analisis se utiliza para asignar prioridades a las tareas y poder utilizar un planificador expulsivo tradicional basado en prioridades

Asignación de prioridades relacionada con las restricciones de tiempo asociadas a cada tarea.

Enfoques dinámicos basados en un plan

factibilidad se determina en tiempo de ejecución

Resultado de análisis = plan para decidir cuándo poner en marcha la tarea.

Nueva tarea aceptada como ejecutable sólo si es posible satisfacer sus restricciones de tiempo

Si nueva tarea puede cumplir sus plazos sin que ninguna otra tarea planificada anteriormente pierda un plazo, es aceptada poniéndose en marcha de nuevo el plan de planificación.

Enfoques dinámicos de mejor esfuerzo

No se realiza análisis de factibilidad

Tareas no periódicas \Rightarrow no posible análisis estático

Sistema intenta cumplir todos los plazos y aborta la ejecución de cualquier proceso cuyo plazo haya fallado.

Cuando llega una tarea \rightarrow sistema asigna prioridad basada en las características de ésta

Se utiliza planificación basada en plazos como planificación del plazo más cercano.

Desventaja \rightarrow No sabemos si la restricción de tiempo será satisfecha hasta vencimiento del plazo o tarea completada.

Ventaja \rightarrow + fácil de implementar.

Problemas de inversión de prioridad

Se da en un esquema de planificación de prioridad cuando:

- Una tarea de mayor prioridad espera por otra de menor prioridad debido al bloqueo de un recurso de uso exclusivo (no compatible)

Enfoques para evitarla

Heredencia de prioridad → tarea menos prioritaria hereda la prioridad de la más prioritaria.

Techo de prioridad → Se asocia una prioridad a cada recurso de uso exclusivo que es más alta que cualquier prioridad que pueda tener una tarea, y esa prioridad se le asigna a la tarea a la que se le da el recurso.