

ÍNDICE

1. Conceptos Fundamentales sobre procesos	3
1.1. Concepto de proceso	3
1.2. Conceptos sobre Kernel y programa de usuario	3
1.3. Modos de ejecución, espacios de direcciones y contexto	4
1.4. Ejecución del SO	4
1.5. Nuestra idea de proceso	4
1.6. Process Control Block	5
1.7. Un Gato	5
1.8. Tabla del Kernel del SO	6
1.9. Cambio de contexto	6
1.10. Cambio de contexto de registros de CPU	6
1.11. Diagrama de estados modificado	7
2. Operaciones sobre procesos	7
2.1. Creación de procesos	7
2.2. Terminación de procesos	8
3. Threads (Hilos o hilos)	9
3.1. Funcionalidad	9
3.2. Tipos de hilos	9
3.3. Enfoques híbridos	10
4. Conceptos fundamentales sobre planificación	11
4.1. Planificación	11
4.1.1 PCB's, colas y estados	11
4.1.2 Colas de estados	11
4.2. Tipos de planificadores	11
4.3. Características de planificadores	12
4.4. Clasificación de procesos	12
4.5. Mixta de procesos	12
4.6. Planificador a media plazo	12
4.7. Dispatcher	13
4.7.1. Activación del dispatcher	13
4.8. Políticas de planificación: medidas	13
5. Políticas de planificación de la CPU	14
5.1. Política "First Come First Served"	14
5.2. Política "Shortest Job First"	15

5.3. Política "Shortest Remaining Time First"	15
5.4. Planificación por prioridades	16
5.5. Planificación por turnos	16
5.6. Colas múltiples	17
5.7. Colas múltiples con realimentación	17

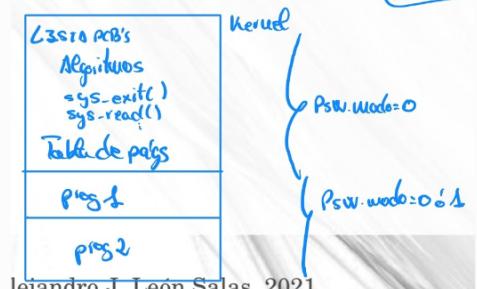
1. Conceptos fundamentales sobre procesos.

1.1. Concepto de proceso

Un proceso es un programa en ejecución sin interrupciones debido a la ejecución de los programas, entendiendo estos como ficheros estáticos ejecutables. Sin embargo, un proceso es un programa junto a su ejecución.

Se define la ejecución de un proceso como la función de ejecución de cada uno de sus programas junto con las distintas interrupciones, excepciones o llamadas al sistema producidas.

Como mínimo, un proceso requiere; para la ejecución de un programa, memoria para código, datos y pila junto con la CPU para ejecutarse. Por lo general, requiere de recursos que el SO controla. Por ejemplo, el so multiplexa (reparte la CPU en el tiempo) la CPU para ejecutar varios procesos de forma simultánea lo que implica almacenar los contextos de ejecución en CPU de los procesos para poder cargarlos posteriormente. Por último, el Sistema Operativo no dejó otro proceso.



1.2 Conceptos sobre Kernel y programa de usuario

La ejecución en CPU de un proceso, programa o simplemente código puede hacerse en dos niveles de privilegio: Kernel (más privilegiado) o usuario.

Con respecto a la memoria:

Hay un espacio protegido para Kernel donde para usar dicho espacio es necesario una secuencia de instrucciones para cambiar de modo usar el modo Kernel y la CPU opera solo en dicho modo (Kernel).

Como información adicional, cabe destacar que el código de este espacio es reservado, es decir, sirve para poder hacer varias llamadas a la vez de manera que varias unidades de ejecución pueden hacer uso de él. Lo que implica la existencia de una pila kernel por proceso.

Por último, es importante conocer que cada proceso está dotado de un espacio de direcciones (fragmento de memoria) protegido. Por tanto, es necesario tener información sobre el ámbito de la memoria reservada a cada proceso (esta información se guarda en espacio de memoria del Kernel, el cual es el mismo para todos los procesos).

1.3. Modos de ejecución, espacio de direcciones y contexto

Modo	User	Kernel
Contexto		
Proceso	Código de usuario ($PSW=1$)	Llamadas al sistema y excepciones
Kernel	(No permitido)	Tratamiento de Interrupciones, Tareas del sistema.

↗ Peticiones explícitas del SO
 ↗ Gestión de código Kernel que no tiene modo privilegiado con el proceso.
 ↗ Hechos kernel.
 ↗ No necesitamos tablas de páginas
 ↗ Almacend → gestión Kernel de Hebrews Kernel

1.4. Ejecución del SO.

Vemoslo por partes:

- Núcleo: Está fuera de todo proceso y por tanto, se ejecuta total; es decir, como una entidad separada que opera en modo Kernel.
- Ejecución dentro de los procesos de usuario: Esto quiere decir a aquellos casos cuando se activa en modo Kernel para la ejecución de código del sistema operativo, como son las llamadas al sistema, las RSFC) ...

1.5. Nuestra idea de proceso

Para nosotros, un proceso es una unidad de actividad caracterizada por la ejecución de una secuencia de instrucciones, un estado de computación actual y un juego de recursos del SO asociados.

Supongamos que hay un proceso ejecutándose en CPU, en el caso de que el SO decide que otro proceso debe abandonar la CPU, se ejerce un cambio de contexto (tarea 1) donde se salvan los registros actuales de CPU en el PCB de dicho proceso. El context switch produce la entrada de un nuevo proceso a ejecución y un uso de tiempo que dependería del detrimento de la productividad del sistema.

1.6. Process Control Block

Es una estructura de datos que contiene la información relativa al ciclo de proceso. Dicho bloque es creado, gestionado y destruido por el kernel.

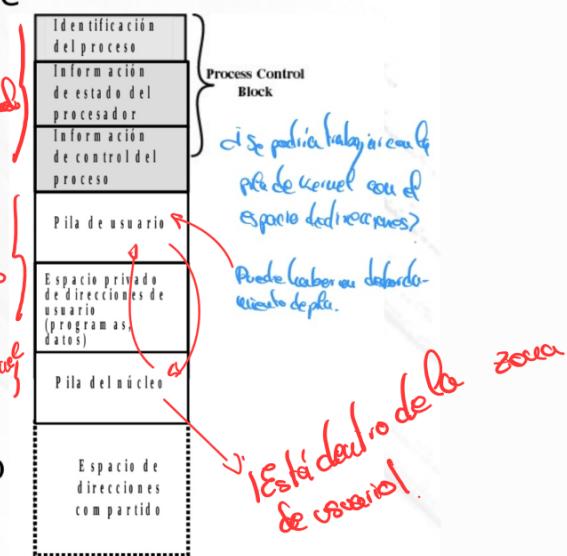
Cómo información básica está dotada de:

- PSO
- Estado del proceso.
- Contenido de Registros
 - { .PSW
· PC
· SP}
 - Sufragencia de memoria
 - { - Datos
- Metadatos (PCB)
- Pila esperada
 - Lista de recursos utilizados

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

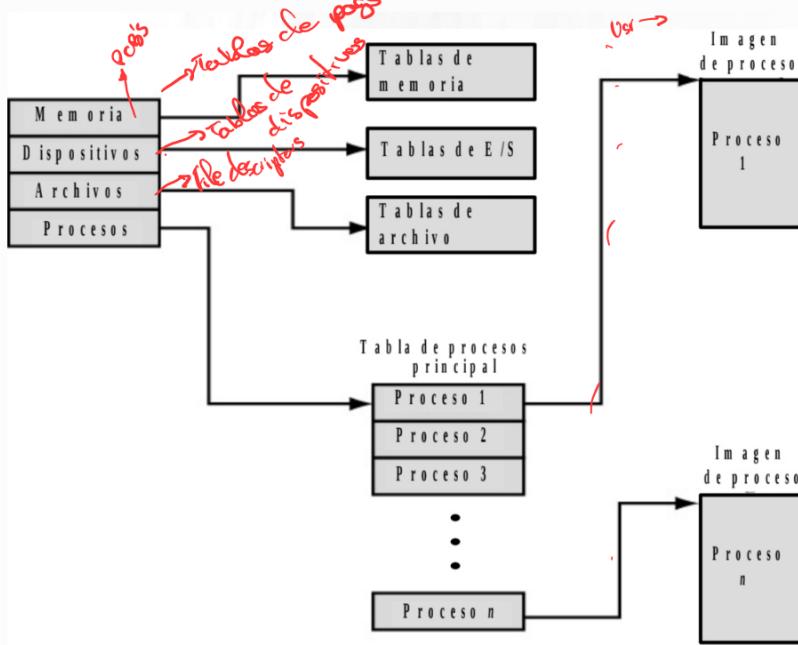
1.7. Un fallo

- La figura muestra una imagen de un libro de introducción a los sistemas operativos en la que se describe el concepto de proceso...
- ¿Hay algo que "choca"?
- ¿Dónde reside el PCB en el espacio de direcciones del proceso?
- ¿Dónde residen las pilas de usuario y núcleo en el espacio de direcciones del proceso?



Este tipo de organización proporcionada por el stallings presenta un error muy grave de diseño de SO, presentando una gran vulnerabilidad frente a la pila del núcleo debido a estar completamente disponible para que forme ataques. (d?)

1.8. Tablas del Kernel del SO.



1.9. Cambio de contexto

Un cambio de contexto no es más que un cambio de ejecución de programa, es decir, el SO detejue un proceso que estaba ejecutándose, guarda sus registros en el PCB del proceso, el plificador de CPU elige un nuevo proceso de la cola de listos y el dispatcher (que ya ha salvado los registros) cambia el orden del proceso detenido a bloqueado y lo sacuda en la cola de bloqueados; mientras tanto, el proceso seleccionado por el plificador pasa a ejecutarse.

Este tipo de operaciones son típicas de sistemas multithreading, solo pueden ser ejecutadas en modo kernel y representan un alto coste para el sistema en términos de tiempo de CPU (Págs. 8.2).

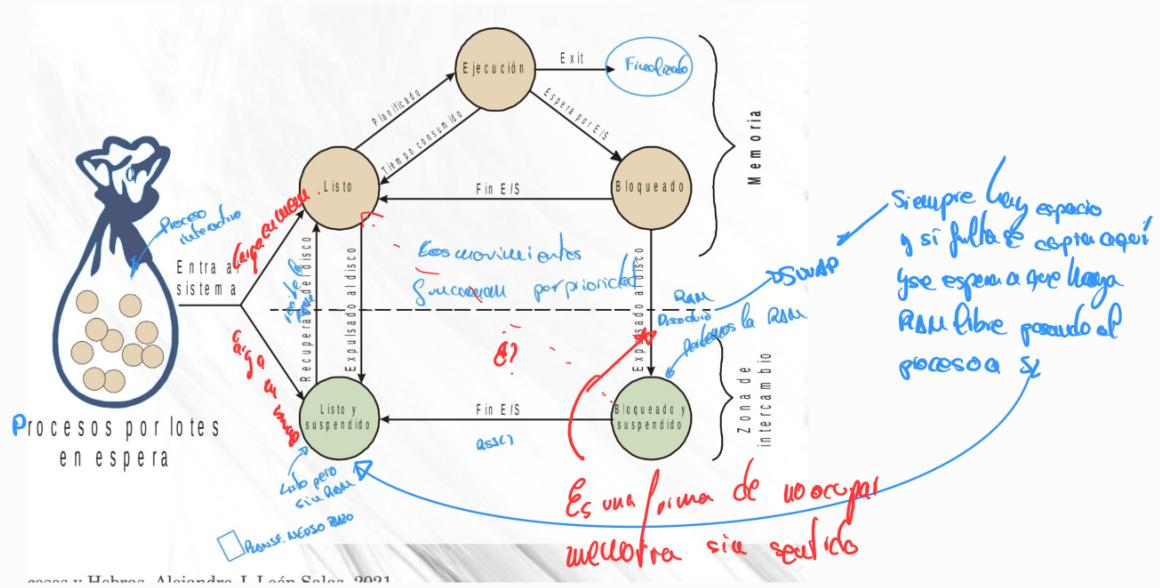
Caso común de cambios de contexto tienen las interrupciones (que se producen cuando se agota el tiempo de ejecución sin interrupción o time slice), agotamiento de fine slice, el usuario al sistema bloqueantes o abandono de CPU voluntario.

1.10. Cambio de contexto de registros de CPU

Vamos a realizar una descripción general:

1. Suspender 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, copiando el contexto de registros que se encuentra en su PCB.
3. Continuar la ejecución usando el nuevo valor del PC.

1.11. Diagrama de estados modificado



2. Operaciones sobre procesos

2.1. Creación de procesos

Crear un proceso consiste en asignarle el espacio de direcciones que utilizará y crear las estructuras de datos para su administración. → PCB ...

Como **causes** que provoquen dicha creación feudales:

- En sistemas batch, como respuesta a la recepción y adquisición de un trabajo.
 - "logou" interactive; es decir, cuando el usuario se identifica desde un terminal, el sistema activa proceso que ejecuta el intérprete de órdenes asignado
 - Llevar a cabo un servicio solicitado por un proceso de usuario
 - Creación de un árbol de procesos por parte de un proceso padre.

En este último caso, los recursos se pueden obtener de 3 maneras:

1. Direchamente del so (no comparten recursos)
 2. Comparten, padre - hijo, todos los recursos
 3. Comparten un subconjunto de los recursos.

{ Compartirle todo - algo

Como posibilidades de ejecución: 

- ## 1. Egeweide concurrentie.

2. Padre espera que hijo acabe

Como posibilidades del espacio de direcciones del programa tipo bucle:

1. Duplicación del padre. Para ello usamos fork() que crea el proceso si queremos reemplazar el espacio de dirección, podemos hacerlo con exec() pasando como argumento el espacio de dirección a copiar. crea el proceso
reemplaza el proceso

Si encadenamos `fork()` + `exec()` conseguiremos un hijo que ejecute el argumento de `exec()`.

fork() crea un hijo que hereda:
pero ~~comparte~~ ^{comparte memoria}
~~per separado~~ } - Copia idéntica de la memoria del padre
- Copia idéntica de los registros de CPU
del padre.

Por último, `fork()` devuelve 0 en el proceso hijo y el PID del hijo en el proceso padre.

- Ejemplo de programa simple que usa **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!\n");
        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;
}

Crear ceras: for(i=0; i<5; i++) { ret = spawn(); if(ret==0) // hijo
    sleep(1); else //padre
    sleep(1); }

```

2. Programma nuovo distinto dal programma associato al precedente padre.

Por último, el kernel debe realizar una serie de pasos para crear un proceso:

1. Identificar con un PID único.
 2. Asignar RAM y/o almacenamiento secundario (swap).
 3. Crear e inicializar PCB. → Estas la estructura declarada para cada proceso y elaborarla en la tabla de planificación.
 4. Insertar PCB en Tabla de Procesos correspondiente.

- ↳ si reside en RAM \Rightarrow LIFO
- ↳ si solo swap \Rightarrow Suspended-lista

2.2. Terciación de procesos

Cuando se termina su ejecución, un proceso solicita su eliminación al sistema mediante `exit()`, que produce, un aviso a su proceso padre manipulando marcador de señal `SIGCHLD`.
Luego, el guardado de su estado de finalización. Además, se liberan sus recursos asociados al proceso.

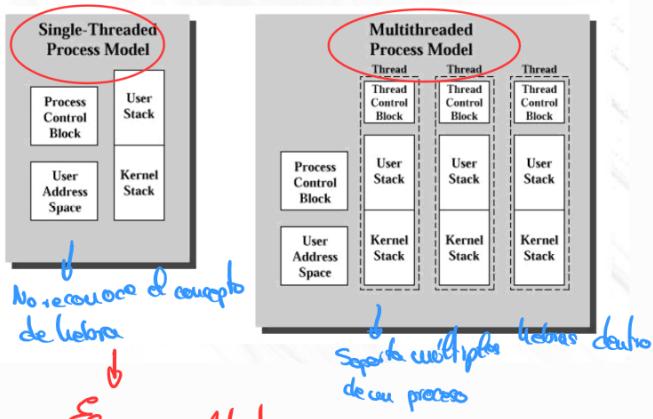
Por otro lado hay otros factores de finalizar un proceso: el padre lo manda con null (cosa), el padre va a acabar y el 50 no permite que los hijos continúen la ejecución (por lo general si se permite pero pasan a ser hijos de null) ó se produce una excepción no recuperable ocausada en su sistema.

3. Threads (Hilos o hilos)

Una hilera es la unidad básica de utilización de tu CPU, y el kernel requiere cierta información para gestionarla como es el contenido de los registros, de la pila o del estado. Al igual que un proceso tiene un "PCB" llamado "TCB". Como una tarea está dividida en varios hilos, cada uno de ellos comparte cierta información de la tarea como secciones de código, de datos o recursos del SO.

Como consecuencia, un proceso finalmente se puede ver como una tarea con una sola hilera donde PID y TID coinciden. TID es el identificador de hilera.

Hay dos tipos de computación de hilos que son excluyentes:



Es como 1 hilera = 1 proceso

3.1. Funcionalidad

Al igual que cumplían la funcionalidad del PCB con los procesos, cumplen la existencia de estados y pueden sincronizarse. Como operaciones básicas tenemos la creación, el bloqueo y el desbloqueo y la terminación.

3.2. Tipos de hilos

Gracias a la distinción entre single threading kernel y multithreading kernel tenemos:

- **ULT** o hilos en biblioteca de usuario, donde toda la gestión se realiza a nivel de usuario con una biblioteca de hilos que se encarga de realizar todas las operaciones y facilitar la comunicación entre ellos. Además, el kernel no es consciente de lo que hacen las hilas puesto que sólo gestionan la tarea; por tanto, la entidad de planificación para el kernel es el proceso. → single threaded.

- **MKT** o hilos a nivel de kernel, donde la gestión se hace a nivel de kernel por lo que mantiene la información de todo. Por otro lado, el se proporciona un conjunto de llamadas al sistema para la gestión de hilos y por tanto la entidad de planificación es la hilera que gestiona la ejecución de una tarea kernel luego es una hilera kernel.

Pg 3

Tarea id;
Lista de hilos: MainThread
Memoria
Recursos
Estado: {NF, EN.PLAN, EJECUTANDO}

TOS & LWP
TSD:
Contador
Estado
Sustituir

Misma
digna
defectos
del proceso blog + proc. bloqueado

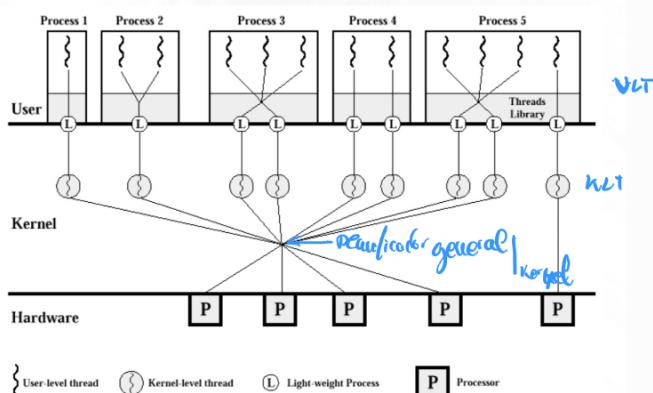
	Ventajas	Inconvenientes
ULT	<ul style="list-style-type: none"> Cambio de hilo \Rightarrow cambio de modo Cambio de hilos adaptable a la app. Aplicaciones ejecutables para todo SO. <p style="text-align: center;">Cada hilo que se pase la biblioteca de hilos</p>	<ul style="list-style-type: none"> Mayoría de llamadas bloqueantes \Rightarrow si una llamarla de más se bloquea. Si no hay asignación de procesos a procesadores \Rightarrow un procesador ejecuta una hilera de forma forzada. \rightarrow Los hilos esto son más lentos
KLT	<ul style="list-style-type: none"> Plaificacióndistintos hilos de una tarea en distintos procesadores (kernel) Bloqueo de una hilera \Rightarrow bloqueo de más Plataforma kernel puede ser multihilera 	<ul style="list-style-type: none"> Cambio de hilos \Rightarrow cambio de modo

3.3. Enfoques híbridos

Un enfoque híbrido consiste en una mezcla entre hilos user y kernel, donde las KLT proporcionadas por la biblioteca de hilos son totalmente invisibles para el kernel.

Por una parte, la creación de hilos y la mayoría del manejo de los hilos se hace en modo usuario y los KLT son la unidad de planificación del kernel que pueden ser elegidas en tiempo por el programador.

Tenemos el concepto de "procesos ligeros", son aquellos que sostienen una o más KLT y se asocian con una KLT. Este concepto facilita que los hilos de una tarea potencialmente bloqueantes no bloquen al resto de los hilos de la tarea.



El procesador asigna KLT a KLT de procesos

4. Conceptos fundamentales sobre planificación

4.1. Planificación

La idea general es desarrollar es a qué cliente de los que piden un recurso debe concederle dicho recurso.

Expresado en hebras: Nosotros tenemos un SO que dispone de "n" hebras en estado listo; sin embargo, disponemos de mejores (o cores) con cuca para ejecutar hebras. Por tanto, el SO debe decidir qué hebras se deben ejecutar.

4.1.1. PCB's, cores y estados (nos vemos a procesos)

Modelo GENÉRICO DE CORES

En este caso tenemos una relación de roles que significan los estados de los respectivos procesos del sistema, donde lo más habitual es que cada role esté asociado a un estado dando sus elementos son los PCB's de los distintos procesos con ese estado. Si medida que un proceso va cambiando de estado, se va deseandando de su role y encolumnado en la siguiente.

4.1.2. CORES DE ESTADOS

Hay 3 roles:

Caso si fuera la de cores

- Trabajos: representa al conjunto de los trabajos que están siendo ejecutados en el sistema.
- Listos: Son aquellos procesos cuyos programas están cargados en memoria principal esperando su ejecución.
- Bloqueados: Son aquellos que esperan el fin de una T/S o de un evento bloqueante.

También se llaman a esto y bloquedo susp.

• listos

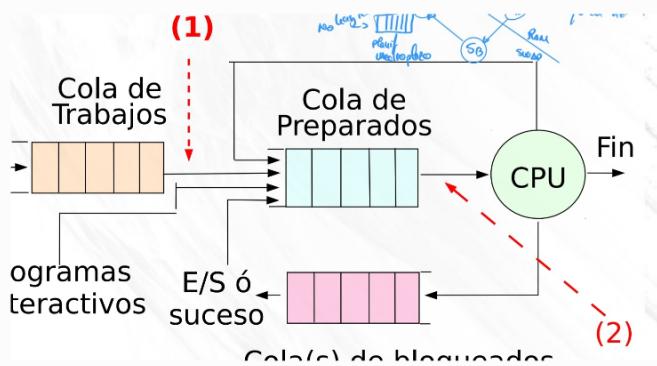
• interrupciones

• excepciones recurrentes

4.2. TIPOS DE PLANIFICADORES

Un planificador es la parte del SO que controla la utilización de un recurso.

- A largo plazo: selecciona los trabajos que deben llevarse a la caja de preparados. "Listos"
- A corto plazo (Planificador de CPU): selecciona el proceso que debe ejecutarse en CPU y se le asigna.
- A medio plazo



4.3. Características de los planificadores

DE CPU

- Trabaja en la cola de trabajos
- Interrupción muy frecuente \Rightarrow debe ser rápido

LARGO PLAZO

- Controla el grado de multiprogramación
- Interrupción poco frecuente \Rightarrow puede ser más lento

4.4. Clasificación de procesos

- Limitados por E/S o procesos cortos (Más tiempo en E/S que cálculo con ráfagas de cortas y largas periodos de espera)
 - Limitados por CPU o procesos largos (Más cálculo que E/S con pocas ráfagas de CPU largas)
- $\cancel{\text{Ritmo de instrucción}} \cancel{\text{de CPU}}$

4.5. Modelos de trabajos

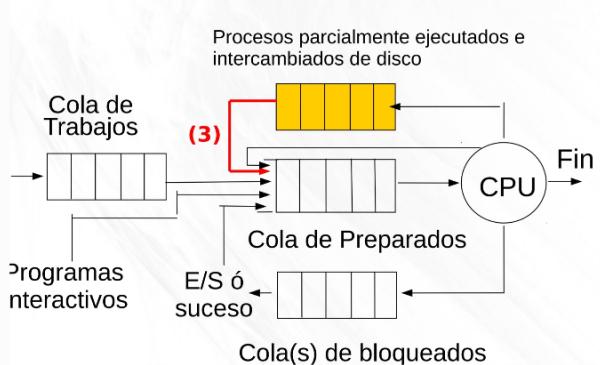
Hay un problema que restriñe al planificador a largo plazo a seleccionar mezclas de trabajos:

- Si todos tienen limitaciones de E/S \Rightarrow cola preparados casi vacía y poco trabajo para el planificador a corto plazo.
- Caso opuesto todo desequilibra el sistema. Todos tienen limitaciones de CPU

colas de preparados
 → vacía
 = espacio disponible para planificar

4.6. Planificador a medio plazo

Es el encargado de desarrollar procesos a memoria. Por ejemplo, se sacan para mejorar la mezcla de procesos o por un cambio en los requisitos de memoria. Esto utiliza se conoce como swapping. Este planificador es el encargado de realizar los cambios de swap a memoria principal.



4.7. Dispatching

Es la función del so que da el control de la cpu al proceso asignado por el planificador.
Realiza el cambio de contexto de registros desde el punto visto de la cpu (modo kernel) y la conversión a modo usuario o a modo kernel dependiendo del nuevo contexto de registros.

Se define la latencia de dispatching como el tiempo que emplea el despatcher en detener un proceso y comenzar a ejecutar otro.

4.7.1 Activación del Despatcher

Puede ocurrir cuando:

- No ejecución de instrucciones → finalización
o por bloqueo
- Determinación de ejecución por el sistema operativo.
- Agotamiento del quantum de tiempo asignado.
- Cambio de estado Bloqueado - Liso → suele no pasar excepto en prioridades

4.8 Políticas de planificación: métricas

Un buen planificador siempre busca un buen rendimiento y un buen servicio. Pero...

¿Cómo sé que hay un buen servicio? Para ello vamos a definir una serie de métricas:

- a) **Retorno:** Para nosotros será el tiempo de servicio, es decir, el tiempo que usa la cpu. Notación (r)
- b) **Tiempo de respuesta:** tiempo transcurrido desde que se produce una solicitud de uso de la cpu hasta que comienza a usarse. Notación (T)

- o) **Tiempo de espera:** tiempo que un proceso ha esperado en la cola de listos antes de ejecutarse. Notación (n)
- o) **Penalización:** Tiempo de respuesta por retraso = $\frac{\text{Tiempo que tarda la CPU en terminar una tarea y empezar otra}}{T} \cdot \text{Notación (P)}$
- o) **Tiempo de respuesta:** es la fracción de tiempo que un proceso está recibiendo servicio.
- o) **Tiempo de nulos:** tiempo perdido por eso en la ejecución de tareas señaladas, ya sean llamadas, cambios de contexto, rutinas de kernel... lo cual es que sea 10-30% del total del tiempo del procesador.
- o) **Tiempo de inactividad:** No es cuando sea distinto de 0 puesto que es el tiempo en el que la cola de listos está vacía = procesador ocioso.
- o) **Tiempo de retorno:** cantidad de tiempo necesario para ejecutar un proceso completo.

Centrándonos más en las políticas; como en todo, nada es perfecto, es decir, ninguna política satisface al 100% a los procesos largos y cortos, lo que pasa es que los trae de forma excluyente. Hay dos tipos:

- o) **No apropiativas:** con prioridades no exigentes permiten un cambio de contexto hasta que el programa en ejecución acabe o lo pida.
- o) **Apropiativas:** con prioridades exigentes si se dan una serie de condiciones, el proceso en ejecución para a lista y el proceso listo que cumplía dichas condiciones lo sucede.

5. Políticas de planificación de la CPU

5.1. Política "First Come First Served"

Literalmente, simula el funcionamiento de una estructura FIFO, proceso que entra a la cola de listos, debe esperar hasta su turno al igual que tú lo haces en un supermercado.

Es un tipo de planificación no apropiativa, ya que un proceso se aleja de ejecutarse si no es por finalizar o bloqueo. Presenta peores prestaciones, equidad en la espera a ejecución y fácil implementación. gusca de

los procesos "cortos" son perjudicados puesto que si hay un proceso largo en ejecución, deberá esperar todo su tiempo de ejecución hasta fin o bloqueo. Dicho modo de espera se producirá siempre hasta que el proceso largo termine su ejecución. Por tanto, los procesos largos están penalizados.

5.2. Política "Shortest Job First"

Como dice su nombre, cuando un proceso realiza un context-switch() por motivos propios, el plificador elige el proceso con la ráfaga más corta, es decir, aquel que tardará menos en CPU. Como se puede intuir, es un apropiativa.

Excepción: Dos procesos con mismo tiempo de servicio estandar (cosa que es necesario establecer para cada proceso de la cola de listos) tendrán la misma prioridad. Su cálculo se encuentra en un pcf de las siguientes:

$$T_{servicio.est-i} = (1-\alpha)T_{serv.est-i} + \alpha rafegas. \quad \alpha \in [0,1] \text{ sirve como función de peso.}$$

Importante: Todos los campos usados para determinar la prioridad de un proceso se guardan en su PCB; por tanto cada política tendrá asociado un modelo PCB distinto.

En este caso, los procesos grandes resultan perjudicados, a veces incluso pueden no llegar a ejecutarse. En dichos casos, su prioridad es modificada a mano fijando su ejecución; un buen modo sería poner su tiempo de servicio en el umbral mínimo cuando supera un umbral máximo.

5.3. Política "Shortest Remaining Time First"

Consiste en una copia de la política anterior pero de forma apropiativa, es decir, en cada encolado de un proceso listo se comprueba la prioridad, y si es mayor se da una apropiación de la CPU.

Presenta el mismo problema y tiene la misma solución.

Este modelo está dotado de la mejor penalización en producido.

5.4 Planificación por prioridades

Consiste en una representación del comando top usando la columna PR, NI. Consiste en asignar una prioridad a cada proceso mediante un valor de manera que un proceso entraría en CPU si su prioridad es mayor a la del resto de procesos.

Permite dos formas de implementación:

- Apropiativa (cada ejecutado provoca un context-switch dependiendo de la prioridad del nuevo proceso).
- No apropiativa (se hay revisión de prioridades cuando el proceso en ejecución se sale de los recursos propios).

Se veo presenta el problema de la inactividad, es decir, puede haber procesos con baja prioridad que nunca se ejecutan. Para solucionarlo, se va a fijar un límite del tiempo de espera, de manera que si el tiempo es "muy grande", se aumentará la prioridad de dicho proceso.

5.5 Planificación por turnos

Se establecerá una serie de intervalos de tiempo de ejecución de CPU (quantum) que van aclarar la determinación de prioridades de los respectivos procesos.

El procedimiento a seguir es el siguiente:

- Si un proceso acaba antes de su quantum, se libera la CPU y se anuncia el siguiente de la cola a ejecución por criterio FIFO.
- Si se superan los quantum, se fuerza una apropiación y el proceso expulsado pasa al final de la cola de ejecutados. → Timeout

Como entre colas proveer es una política apropiativa implementada, normalmente, con valores 10 y 100 ms de quantum. Presenta las siguientes características:

- o) Equidad en las penalizaciones (son de un quantum)
- o) Mayor penalización de ráfagas cortas que lo deseado.
- o) Complejidad elección de quantum; ya que si es muy grande podríamos crear un FCFS y si es muy pequeña provocar una excesiva ejecución de context-switches disminuyendo el tiempo de utilizo.

5.6. Círcos múltiples

Causas: lleva a aplicar las políticas de planificación a las distintas etapas de los procesos de forma que se permite el cambio de etapa de un proceso. ✓ Y esto es lo que nos lleva a la necesidad de tener la planificación entre etapas:

- Cores prioritarios fijos: no usada
 - Tiempo cuantificado entre cores: se divide el tiempo de CPU entre los diferentes cores donde cada uno lo distribuye según su política de planificación.

5.7. Colas múltiples con realimentación.

Consciente en el modelo anterior permitiendo el cambio de cola de los procesos. Para ello, es necesario usar una serie de variables como: número de colas, algoritmo de planificación de cada cola, criterio de traslado de cola, criterio inicial de inserción en cola y algoritmo de planificación entre colas. Además, se necesita la medición en tiempo de ejecución del comportamiento real de los procesos.

5.8. Rauficación en multiprescridores

Antes de cada, conviene hablar del concepto de **granularidad**; consiste en la sincronización entre entidades planificables, es decir, cuando se buscan mutuamente para ver si son sincronizadas. Puede que haya **grano grueso** (baja sincronización) o **grano fino** (alta sincronización)
↳ entre procesos ↳ entre hilos

Si nos valemos a procesos, su planificación consiste en ampliar el concepto de monoprogramado a multiprogramado. Sin embargo; en hilos se puede explotar el paralelismo real dentro de una aplicación con múltiples hilos.

Identificar tres aspectos de diseño del planificador que se relacionan:

1. Distinguición de procesos a procesadores
 2. Uso de multiprogramación en cada procesador individual

3. Activación del planificador de CPU, que va a soñar cambios gracias al uso de hilos. El proceso por el planificador

Como datos y cosas a tener en cuenta, todas las implementaciones disponen de la granularidad de los aplicaciones y del uso de procesadores

1. Asignación de procesos a procesadores

Es una de las estrategias más simples que consiste en pensar que el procesador es un almacén de recursos que pueden asignarse directamente o estaticamente.

La segunda provoca que cada procesador sólo se encargue de los hilos de un proceso, es decir, 1 procesador - 1 proceso. Sin embargo, es preferible la primera, que pese a generar una sobre carga de planificación entre los procesadores gracias al uso de una sola cota para todos los procesadores (sistema 1) o al uso de un balanceo dinámico de carga (sistema 2)

2. Uso de multiprogramación en cada procesador individual

Lo único a apartar de esta idea es que si hay muchos procesadores y granularidad media, hay que equilibrar la productividad en tiempo frente a maximizar la utilización de CPU entre los aplicaciones

3. Activación de procesos

No todos los algoritmos de planificación son igual de buenas; en este caso, SRTF es más eficiente que FCFS en sistemas multiprocesados.

En el caso de usar hilos con respecto a su planificación, hay nuevos aspectos que pueden ser más importantes que las prioridades o las listas de ejecución.

5.9. Planificación de hilos en multiprocesadores

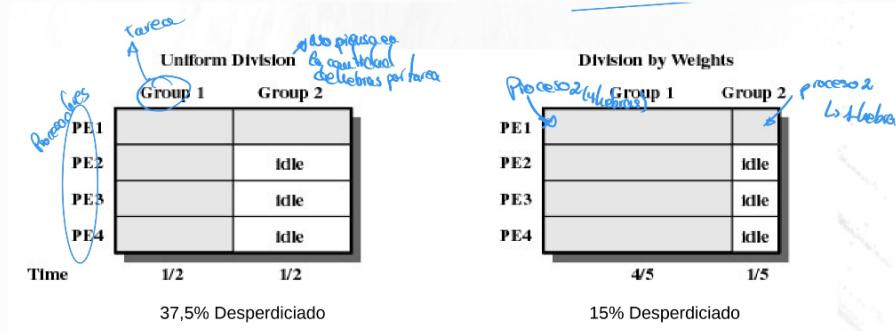
Hay varias formas:

La primera de ellas es la compartición de carga, donde hay una sola global de hilos con sentido a esto, de manera que usando "FCFS" podemos evitar procesadores ociosos.

La segunda consiste en la planificación en paralelo, donde se eligen sobrecargos de hilos de una tarea, se asignan a hilos y que hay mas procesadores y ellos

Consiste en usar mas de los procesadores para garantizar la sincronización de las tareas, como inconveniente, hay procesadores que se usan hasta acabar su ejecución. Resulta interesante no pensar en la optimización de este algoritmo a sistemas de tiempo compartido.

Este algoritmo presenta una gran utilidad en tareas que necesitan mucha sincronización.



Como tercer algoritmo, aparece la asignación de procesador dedicado, donde a cada hilo de un proceso se le asigna un hilo, dicho procesador no suelta el proceso hasta que acaba todos los hilos. En este caso, se presenta la posibilidad de tener procesadores ociosos ya que no hay multiprogramación de procesadores.

Por último, encontramos la planificación dinámica, donde la aplicación permite que varíe dinámicamente el número de hilos de un proceso. Es el sistema encargado de ajustar la carga para mejorar la utilización de los procesadores.

5.10. Planificación en tiempo real

En este caso, la exactitud del sistema no depende sólo del resultado lógico de un cálculo sino también del instante en el que se procesa el resultado. Se caracteriza, además, por el hecho de que las tareas deben de reaccionar ante sucesos causados en tiempo real del mundo exterior. Como características presentan:

- Ser de "hard rt": si no cumples el umbral de tiempo no lo ejecuta.
- Ser de "soft rt": aunque no lo cumplas puede que lo ejecutes.
- Ser periódicas: se sabe cada cuanto tiempo se tiene que ejecutar.
- Ser aperiodicas: tiene umbrales pero la iniciación de una tarea es impredecible.

La distinción en los enfoques depende de:

- Comprobación de la viabilidad de la planificación
- Implementación estática o dinámica.
- Obtención de un plan de planificación o no.

Hay varios tipos:

- Estatísticos dirigidos por tarea (se genera una planificación que determina el momento de inicio de una tarea).
- Estatísticos explorativos dirigidos por prioridad (no genera planificación, sólo conjunta prioridades).
- Dinámicos basados en un plan (se ve la viabilidad en ejecución, se actualiza las tareas que cumplen sus tiempos).
- Dinámicos de menor esfuerzo (no hace análisis de viabilidad, se intenta cumplir los plazos y abortar ejecuciones si su plazo ha finalizado)

5.11. Problema de Inversión de Prioridad.

Consiste en momentos en los que una tarea con prioridad u recibe un recurso para ella, dicho recurso es suyo hasta que acabe lo que tenga que hacer. El problema surge cuando una tarea de prioridad usa apropiada con y dicho recurso es necesario, pero este siendo ocupado por otra. = un bucle colgando tu -> él. Producidose así, un bucle colgando pasó de avance.

Cómo solucionarlos se proponen dos:

- Herencia de prioridad, donde la tarea tiene heredad de prioridad del recurso
 $u_i = \max \{ i / \text{tareas} \}$
- Telón de prioridad; consiste en asociar un número de prioridad a cada recurso haciendo que solo puedan usar tareas con prioridad superior. Además, a la tarea que requiere el recurso se le asigna esa prioridad.

Una vez que se libera el recurso, la tarea retoma su prioridad inicial.

