Sistemas Operativos - Apuntes para final

Gianfranco Zamboni

15 de febrero de 2021

Índice

1. Introducción

Un sistema informático tiene cuatro componentes:

- El hardware (CPU, memoría y dispositivos de entrada salida) proveen los recursos básicos del sistema.
- Las aplicaciones definen la forma en que estos recursos va a ser usados para resolver los problemas del usuario.
- El sistema operativo controla el hardware y coordina su uso entre las distintas aplicaciones de los usuarios.

1.1. El sistema operativo

Un sistema operativo provee un entorno de ejecución de programas. Para esto tiene ciertos componentes que difieren de sistema en sistema pero que entre lo más comunes se encuentran:

- **Drivers:** Programas manejan los detalles de bajo nivel relacionados con la operación de los distintos dispositivos.
- **Núcleo**: (o Kernel) Es el sistema operativo, propiamente dicho. Se encarga de las tareas fundamentales y contiene los diversos sub-sistemas que iremos viendo a lo largo de la materia.
- **Sistema de archivos**: Forma de organizar los datos en el disco para gestionar su acceso, permisos, etc.
  - **Archivo:** Secuencia de bits con un nombre y una serie de atributos que indican permisos.
    - **Binario del sistema**: Son archivos que no forman parte del kernel pero suelen llevar a cabo tareas muy importantes o proveer las utilidades básicas del sistema.
    - **Archivos de configuración:** Son archivos especiales con información que el sistema operativo necesita para funcionar.
  - **Directorio**: Colección de archivos y directorios que contiene un nombre y se organiza jerárquicamente.
    - **Directorios del sistema**: Son directorios donde el propio SO guarda archivos que necesita para su funcionamiento.
  - **Dispositivo virtual**: Abstracción de un dispositivo físico bajo la forma, en general, de un archivo de manera tal que se pueda abrir, leer, escribir, etc.
  - **Usuario**: La representación, dentro del propio Sistema Operativo, de las personas qo entidades que pueden usarlo. Sirve principalmente como una forma de aislar información entre distintos usuarios reales y de establecer limitaciones.
  - **Grupo**: Una colección de usuarios

Systems calls (Syscalls): Son rutinas que proveen una interfaz a los servicios disponibles en el sistema. Generalmente, son rutinas escritas en C/C++ y llamarlas implican un cambio de contexto y privilegios del proceso a ser ejecutado. Algunos de estos servicios son:

- Creación y control de procesos.
- Pipes.
- Señales
- Operaciones de archivos y directorios
- Excepciones
- Errores del bus
- Biblioteca C

Bibliografía

(1) Sistemas operativos - Clases Teóricas, 2019.

Parte I
Procesos y Scedulling

2. Procesos y API

Proceso: Es un programa en ejecución y todas las estructuras que debe mantener el sistema para su correcto funcionamiento. Entre ellas, los valores de los registros, el program counter, el stack del proceso, el área de memoria en la que se cargan las instrucciónes a ejecutar (el programa propiamente dicho) y el área de datos (contiene variables globales) y un heap de memoria (que es donde reserva memoria dinámica).

2.1. Creación y control de un proceso

2.1.1. Creación de un proceso

Durante su ejecución, un proceso puede crear nuevos procesos. En este caso llamamos **proceso padre** al proceso que crea nuevos procesos y **procesos hijos** a los procesos creados. Además, cada proceso hijo podrá crear otros procesos, formando así un **árbol** de procesos.

En la mayoría de los sistemas operativos, se asigna a cada proceso un identificador numérico único llamado **process id** o **pid** que sirve como índice para poder acceder a varios de sus atributos.

El nuevo proceso hijo creado necesitara de ciertos recursos para poder llevar a cabo su objetivo. Estos recursos podrán ser obtenidos directamente del sistema operativo o estar restringidos a los recursos del proceso padre.

Cuando un proceso crea un nuevo proceso (`fork()`) hay dos posibilidades:

- El padre y su hijo continúan ejecutando concurrentemente.
- El padre espera a que alguno o todos sus hijos terminen para seguir ejecutando (`wait()`).

Además, el nuevo proceso puede ser un duplicado de su padre o cargar un nuevo programa (`exec()`)

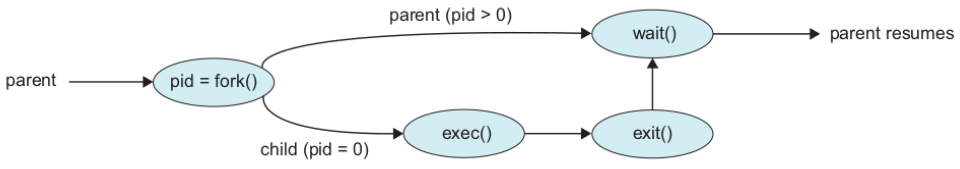

Figure 3.10 Process creation using the `fork()` system call.

2.1.2. Terminación

Un proceso termina cuando ejecuta su última instrucción y pide al sistema operativo que lo borre usando la syscall `exit()`. En este momento, el proceso envía el código de estado con el que terminó a su padre y el sistema operativo libera todo los recursos que le había asignado.

La terminación puede ocurrir por varios motivos. Un proceso puede causar la terminación de otro usando las syscalls adecuadas (usualmente, esto puede hacerlo solo el proceso padre).

2.1.3. Un poco más sobre las syscalls mencionadas

- `pid_t fork()`: Crea un nuevo proceso. En el caso del creador se retorna el PID del hijo. En el caso del hijo, retorna 0.
- `int execve(const char* filename, char* const argv[], char* const envp[] )`: Sustituye la imagen de memoria del programa por la del programa ubicado en filename.
- `pid_t forkv()`: Crea un hijo sin copiar la memoria del padre, el hijo tiene que hacer `exec`.

  Un proceso creado con esta syscall comienza con sus páginas de memoria apuntando a las mismas que las de su padre. Recién cuando alguno escriba en memoria, se hace la copia. Esto se llama **Copy-On-Write**.
- `pid_t wait(int* status)`: Bloquea al padre hasta que algún hijo termine o hasta que alcance el status indicado.
- `pid_t waitpid(pid_t pid, int* status)`: Igual al anterior pero espera a que el hijo con PID `pid` llegue a ese status.
- `void exit(int status)`: Finaliza el proceso actual.
- `clone(...)`: Crea un nuevo proceso. El hijo comparte parte del contexto con el padre. Es usado en la implementación de threads.

Fork Bomb: Un proceso crea infinitos hijos.

2.2. Estados de un proceso

Durante su ejecución, un proceso va modificando su estado acorde al siguiente diagrama:

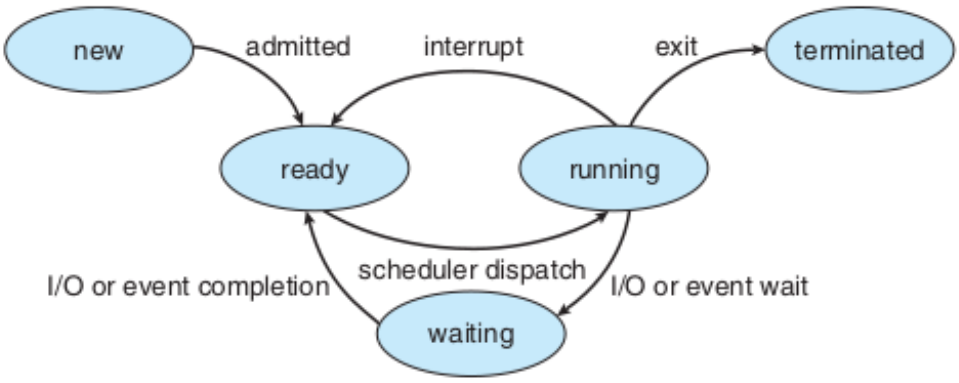

Figure 3.2 Diagram of process state.

- **Nuevo (New):** El proceso se está creando.
- **Listo (Ready):** El proceso está listo para ser ejecutado.
- **Bloqueado (Waiting):** El proceso está esperando a que ocurra algo (por ejemplo a que se complete una operación de entrada salida)
- **Ejecutando (Running):** El proceso se está ejecutando.
- **Terminado (Terminated):** El procesó termino de ejecutarse.

Es importante notar que solo un proceso puede estar **ejecutando** en un instante de tiempo de dado. Sin embargo, muchos procesos pueden estar bloqueados o listos simultáneamente.

2.3. Process Control Block (PCB)

Cada proceso está representado, en el sistema operativo, por un Process Control Block o Task Control Block. El mismo contiene información asociada a cada proceso, entre ellas:

- El estado del proceso
- El program counter (la dirección de memoría de la próxima instrucción a ser ejecutada)
- Los registros del CPU,
- Información de Scheduling, como puede ser prioridad de ejecución
- Información de manejo de memoria, como las direcciones de los directorios de paginas o las tablas de segmentación.
- Estadísticas de uso del sistema
- Información de E/S (que dispositivos está usando, lista de archivos abiertos, etc)

2.4. Scheduler

Multiprocesador: Un equipo con más de un procesador.

Multiprogramación: La capacidad de un SO de tener varios procesos en ejecución.

Multiprocesamiento: Se refiere al tipo de procesamiento que sucede en los multiprocesadores.

Multitarea: Es una forma especial de multiprogramación, donde la conmutación entre procesos se hace de manera tan rápida que da la sensación de que varios programas están corriendo en simultáneo.

Multithreaded: Son procesos en los cuales hay varios *mini procesos* corriendo en paralelo (de manera real o ficticia).

2.4.1. Context Switching

Una interrupción hace que el sistema operativo cambie la tarea de una CPU y corra un rutina del kernel. Cuando esto ocurre, el sistema, debe guardar el **contexto** del proceso que se está ejecutando para poder retornar a la ejecución del mismo cuando termine de atenderla. Este contexto es el PCB del proceso.

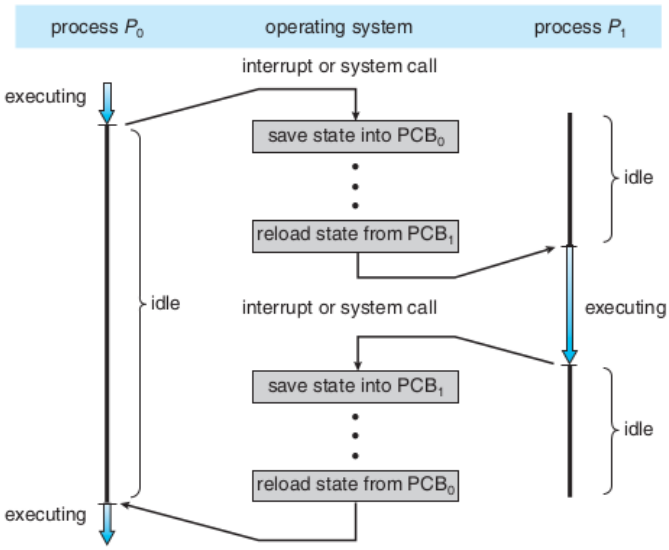

Figure 3.4 Diagram showing CPU switch from process to process.

2.4.2. Colas de procesos

Una de las características de los sistemas operativos actuales es que nos permiten correr varias aplicaciones a la vez, sin embargo solo un proceso puede estar ejecutando en la CPU. Para darnos la sensación de simultaneidad, el SO hace lo que se llama **preemption**.

Preemption: A cada proceso le asigna un **quantum** o una cantidad de tiempo durante la cual es ejecutado. Una vez que transcurrido ese tiempo, se guarda el contexto del proceso en memoria y se carga el contexto del próximo proceso a ejecutar.

El objetivo de la multiprogramación es intercambiar las tareas que se ejecutan en la CPU lo suficientemente rápido como para que el usuario pueda interactuar con dada programa como si estuviesen ejecutándose simultáneamente.Para esto existe un proceso especial llamado **Scheduler** que selecciona el próximo proceso que debe ser ejecutado.

El mismo dispone de varias colas de procesos en los que se encuentran los PCB de los procesos en ejecución. Los PCB se van encolando y desencolando de cada cola dependiendo del estado de cada proceso.

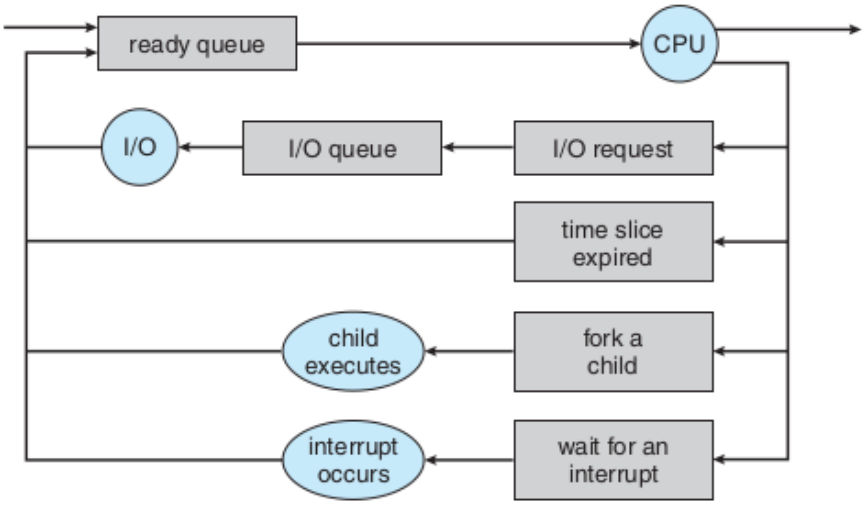

Figure 3.6 Queueing-diagram representation of process scheduling.

2.5. Inter Process Communication (IPC)

Los procesos que se están ejecutando concurrentemente en el sistema operativa pueden ser **procesos independientes** o **cooperativos**.

Procesos independiente: Es un proceso que no puede afectar o ser afectado por otros procesos ejecuntandose en el sistema operativo.

Procesos cooperativos: Son procesos que no son independientes (pueden afectar o ser afectados por otros procesos).

Los procesos cooperativos requieres de un mecanismo de comunicación que les permita intercambiar datos e información. Hay dos modelos fundamentales para hacer esto: **memoria compartida** y **Pasaje de mensajes**.

En el modelo de memoria compartida, se establece un área de memoría que es compartida entre procesos. Los procesos pueden intercambiar información leyendo y escribiendo en este área.

En el modelo de pasaje de mensajes, la comunicación se realiza a través de mensajes entre los procesos cooperativos.

2.5.1. Pasaje de mensaje

Los mensajes proveen a los procesos un mecanismo que permite que los procesos se comuniquen entre si y sincronicen sus acciones. Un sistema que permite el pasaje de mensajes provee al menos dos operaciones:

```
send(message)      receive(message)
```

Los mensajes pueden tener una longitud fija o variable. La primera opción es más fácil de implementar pero hace que la programación de tareas sea mas tediosa. Los mensajes de longitud variable necesitan un sistema más complejo pero permite más agilidad a la hora de programar tareas.

Si un proceso P y Q se quieren comunicar, entonces debe existir un **link de comunicación** entre ellos. Para crear este link, se deben tener en cuenta el tipo de comunicación que se desea ofrecer:

- **Direccionamiento**
  - **Conexión directa:** Cada proceso debe explicitar el nombre del proceso destinatario/receptor:
    - `send(P, message)` - Envía un mensaje al proceso P
    - `receive(Q, message)` - Recibe un mensaje del proceso Q

    En este caso, el link de comunicación se establece automáticamente entre cada par de procesos que quiere comunicarse. Y cada proceso sabe la identidad del otro.
  - **Comunicación indirecta:** Los mensajes son enviados a buzones (**mailboxes**) o puertos (**puertos**). Cada buzón tiene un identificador único. Un proceso puede comunicarse con otro a través de varios buzones.
    - `send(A, message)` - Envía un mensaje al buzón A
    - `receive(A, message)` - Recibe un mensaje del buzón A

    En este esquema, un link se establece entre dos procesos solo si comparten un buzón. Además un link puede estar asociado a varios pares de procesos e incluso puede haber varios links entre dos mismos procesos.

- **Sincronización:** El pasaje de mensajes puede ser bloqueante (síncrono) o no bloqueante (asíncrono):
  - **Envío bloqueante:** El proceso que envía un mensaje espera a que el mismo sea recibido por su destinatario.
  - **Envío no bloqueante:** El proceso envía el mensaje y sigue su ejecución.
  - **Recepción bloqueante:** El proceso se bloquea hasta que recibe un mensaje.
  - **Recepción no bloqueante:** El receptor recibe un mensaje válido o null.
- Buffering: Los mensajes intercambiados entre procesos deben almacenarse en una cola temporal. Estas colas pueden ser de tres tipo:
  - **Capacidad Cero:** El link no puede tener mensajes en espera. El remitente debe bloquearse hasta que el mensaje sea recibido.
  - **Capacidad acotada:** La cola tiene una capacidad finita n por lo que puede haber a lo sumo n en espera. Si la cola no está llena se puede enviar un mensaje, sino el remitente debe bloquearse hasta que se que haya espacio disponible.
  - **Capacidad infinita:** El remitente nunca se bloquea.

2.5.2. Sockets

Los sockets son los extremos de una comunicación que usan dos procesos para comunicarse a través de una red. Cada socket está identificado por una dirección IP y un número de puerto.

En general, se usa una arquitecutra cliente-servidor: El servidor espera a que un cliente haga un pedido y, una vez que lo recibe, acepta la conexión del socket del cliente para completar la conexión. El número del socket del servidor, en general, va a ser menor o igual a 1024. Estos puerstos son los puertos conocidos (*well known ports*) e implementan

Cuando el proceso cliente inicia la conexión, la computadora que lo está ejecutando le asigna un puerto arbitrario cuyo número es mayor a 1024.

2.5.3. Pipes

Un **pipe** es un canal que provee una de las formas más simples de comunicación entre dos procesos aunque tienen sus limitaciones. Al implementar un pite, se debe tener ciertas consideraciones:

1. ¿El pipe permite comunicación bidireccional o unidireccional?
2. Si es bidireccional, es **half-duplex** (para mandar información se debe esperar a que el otro extremo del pipe termina de hacerlo) o **full-duplex** (la información puede viajar de un lado a otro y viceversa simultaneamente).
3. ¿Debe existir alguna relación entre los procesos que se están comunicando (por ejemplo, padre-hijo)?

4. ¿Los procesos se van a poder comunicar dentro de una red o tienen que estar en la misma maquina?

Ordinary pipes: Permiten que dos procesos se comuniquen en modo servidor-consumidor: El servidor escribe en un extremo del pipe (extremo de escritura) y el consumidor lee desde el otro extremo (extremo de lectura). Estos pipes son unidireccionales. Si se necesita una comunicación bidireccional se debe crear otro pipe que permita mandar datos en la otra dirección.

Este tipo de pipes requieren que los procesos que se comunican tengan una relación padre-hijo y dejan de existir una vez que la comunicación termina.

Named pipes: Proveen comunicación bidireccional y no necesitan que los procesos estén relacionados. Una vez que es establecido, varios procesos pueden usarlo para comunicarse y continuan existiendo incluso después de que las comunicaciones hayan finalizado.

2.6. E/S bloqueante / no bloqueante

Cundo un proceso necesita escribir o leer información de algún dispositivo necesita realizar operaciones de **entrada/salida**

Estas operaciones son muy lentas por lo que quedarse bloqueado es un desperdicio de tiempo.

Busy Waiting: El proceso no hace nada pero no libera el CPU. Se gastan ciclos de procesamiento en hacer nada.

Para evitar esto se utilizan algunas técnicas que permiten al SO operativo seguir ejecutando mientras espera la respuesta de los dispositivos:

- **Polling**: El proceso libera la CPU per todavía recibe un quantum que desperdicia hasta que la E/S esté terminada.
- **Interrupciones:** Esto permite la multiprogramación. El SO no le otorgá más quantum al proceso hasta que su E/S esté lista. El hardware comunica esto mediante una interrupción que hace que el proceso se despierte.

2.7. Manejo básico de un shell Unix

2.7.1. File descriptors:

En Unix, cada proceso se crea con una tabla que le permite identificar cuales son los archivos que tiene abiertos. Cada índice es un **file descriptors** que usa el Kernel para saber como leer/escribir datos en los distintos archivos (en Unix, el teclado y la pantalla se modelan como archivos).

Además, cada proceso hereda de su proceso padre tres archivos abiertos que ocupan los file descriptors 0, 1 y 3 y representan la entrada estándar (**stdin**), la salida estandar (**stdout**) y el error estandar (**stderr**), respectivamente.

Linux provee de dos llamadas al sistema que nos permiten leer/escribir a un archivo usando su file descriptor `fd`:

```
ssize_t read(int fd, void *buf, size_t count)
ssize_t write(int fd, const void *buf, size_t count);
```

Aquí `buf` es un puntero a donde se almacenan los datos a leer o escribir y `count` la cantidad de bytes que hay escribir/leer.

2.7.2. Comandos de consola

- `echo` escribe lo que le pasemos como parámetro en su **stdout**.

```
echo ‘‘Esto es un mensaj’’
```

- `>` Se le indica a la consola que el **stdout** se redirija a un archivo:

```
echo ‘‘Esto es un mensaje’’ >mensaje.txt
```

- `|` Redirige el **stdout** de un proceso hacia el **stdin** de otro:

```
echo ‘‘Esto es un mensaje’’ | wc -c
```

  En este caso, el primer proceso ejecuta el comando `echo` que imprime en **stdout** el mensaje `‘‘Esto es un mensaje’’`. El segundo proceso recibe por **stdin** lo que se escribió en el **stdout** del proceso que ejecutó `echo`

Bibliografía

(3) Silberschatz, A.; Galvin, P. B. y Gagne, G. en *Operating System Concepts - International Student Version, 9th Edition.* Wiley: 2014; cap. 3. Processes, págs. 105-162.

3. Scheduling

En sistemas con un único procesados, solo se puede correr de a un proceso por vez. Uno de los objetivos de la multiprogramación es que todo el tiempo se esté ejecutando un proceso para maximizar el uso de CPU.

Cada vez que el CPU entra en estado IDLE, el sistema debe seleccionar alguno de los procesos que estén listos para ser ejecutados. El proceso de selección es llevado a cabo por el **scheduler** usando la que se llama **cola de ejecución**.

3.1. Objetivos de la política de scheduling

Diferentes algoritmos de schedulling tienen diferentes propiedades y la elección de los mismos depende de las situación particular del sistema. Por lo general, un algoritmo de schedulling busca optimizar alguna combinación de las siguientes propiedades:

- **Eficiencia**: Maximizar la cantida de tiempo que el CPU esté ocupadp.
- **Rendimiento (Throughput)**: Maximizar el número de procesos terminados por unidad de tiempo.
- **Tiempo de ejecución (Turnaround time)**: Minimizar el tiempo total que le toma a un proceso ejecutar completamente (el intervalo de tiempo desde que el proceso se crea hasta que temina, incluye tiempo de esperas en la cola de procesos).
- **Ecuanimidad (Fairness)**: Que Cada proceso reciba una dosis "justa"de CPU (para alguna definición de justicia).
- **Carga del sistema (Waiting time)**: Minimizar la cantidad de tiempo que un proceso esté en la cola de espera.
- **Tiempo de respuesta (Response time)**: Minimizar el tiempo para que un proceso empieze a dar resultados.
- **Latencia**: Minimizar el tiempo requerido para que un proceso comienze a dar resultados.
- **Liberación de recursos:** Hacer que terminen cuanto antes los procesos que tiene reservados más recursos.

Muchos de estos objetivos son contradictorios. Si los usuarios del sistema son heterogéneos, pueden tener distintos intereses por lo que cada política de scheduling debe buscar maximizar una función objetivo que es una combinación de estas metas tratando de impactar lo menos posible en el resto.

3.2. Scheduling con y sin desalojo

El sistema puede tomar decisiones de scheduling en alguna de las siguientes situaciones:

1. Cuando un proceso pasa de estado *Ejecutando* al estado de espera (por ejemplo cuando hace un request de entrada salida)
2. Cuando pasa de *Ejecutando* a *Listo*
3. Cuando pasa de *Esparando* a Listo
4. Cuando termina.

Scheduling sin desalojo: Se da cuando las decisiones de scheduling solo toman lugar en las situaciones 1 y 4, es decir se espera a que el proceso haya terminado o esté inactivo. Tiene como desventaja que si un proceso muy largo toma control del procesador se puede generar un cuello de botella y, dependiendo de como se elijan los procesos, starvation.

Starvation (Bloqueo indefinido) : Un proceso sufre de starvation cuando está listo para ser ejecutado pero la CPU nunca le asigna clocks de reloj en lo que ejecutar.

Scheduling sin desalojo: Tambien conocido como **coperativo** o **nonpremptive**, se da cuando las decisiones de scheduling solo toman lugar en las situaciones 1 y 4, es decir se espera a que el proceso haya terminado o esté inactivo. Tiene como desventaja que si un proceso muy largo toma control del procesador se puede generar un cuello de botella y otros procesos mas cortos tardarían demasiado en ser ejecutados.

Scheduling con desalojo: Tambien llamado scheduling *apropiativo* o *preempitve*, se vale de la interrupción del clock para decidir si el proceso actual debe seguir ejecutando o le toca a otro. No da garantías de continuidad a los procesos.

Por lo general se usa una combinación de los dos tipos de scheduling para decidir las políticas adecuadas.

3.3. Políticas de scheduling

3.3.1. First In/First Out (FIFO)

El algoritmo FIFO (también conocido como First Come, First Served) es una de las políticas de scheduling más simples. Los process control block (PCB) se ubican en una cola, el próximo proceso a ejecutar es el que está en la cabeza. Cuando un proceso está listo para ser ejecutado se lo encola al final.

Por un lado, es simple de implementar. Por otro, el tiempo de espera promedio de un proceso es bastante largo y hay que tener en cuenta que es un algoritmo sin delajo. Si llega un proceso que requiera mucho tiempo de CPU, tapona todos los demás, esto se llama **convoy effect**.

3.3.2. Shortest Job First (SJF)

Este algoritmo asocia cada proceso con su duración y ejecuta primero aquellos que duran menos. Está ideado para sistemas donde predominan los trabajos batch y está orientado a maximizar el throughput.

Si los procesos ejecutados tienen un comportamiento regular, se puede usa el historial de ejecución para predecir los tiempos ejecución de los procesos actuales. Sin embargo, en sistemas con procesos heterogéneos no es posible saber cuanto tiempo de ejecución va a necesitar cada uno por lo que no es posible implementar este algoritmo.

3.3.3. Round Robin

Este algoritmo está especialmente diseñado para sistemas de tiempo compartido (varios procesos deben usar el cpu al mismo tiempo). Se comporta de manera similar al FIFO, solo que se agrega desalojo para permitir al sistema ejecutar otros procesos.

Para esto se define una pequeña unidad de tiempo llamada **quantum** durante la cual puede correr cada proceso. Si la ráfaga de procesamiento (CPU Burst) de un proceso es mas chico que el quantum, entonces el mismo la liberará y el scheduler eligirá el siguiente proceso a ejecutar.

Si el CPU Burst toma más de un quantum, entonces se enviará una interrupción al sistema. Éste desalojará el proceso, cambiará el contexto y el proceso que se estaba ejecuntando se pondrá al final de la cola de ejecución.

El rendimiento de este tipo de algoritmos depende del tamaño del quantum. Por un lado, si el quantum es demasiado largo, la política de Round Robind termina siendo una FIFO. Por el otro, si el quantum es extremadamente pequeño se pueden producir una gran cantidad de cambios de contexto, por lo que una gran parte del tiempo del CPU sería gastado solo en esto.

En general, se debe elegir el quantum de tal manera que la mayoria de los procesos termine su CPU Burst durante el mismo pero no tan largo como para que sea un FIFO.

3.3.4. Multiples colas

Otra idea, es serparar los procesos en distintas colas de acuerdo a la duración de su CPU Burst. Cada cola tendrá mas prioridad sobre la otra. Si un proceso toma demasiado tiempo, entonces se lo mueve a una cola con una prioridad menor.

Este esquema, da mayor prioridad a los procesos interactivos y aquellos procesos que estén esperando demasiado tiempo pueden ser movidos a colas de mayor prioridad para evitar starvation.

Normalmente, los sistemas operativos tratan de prevenir que un proceso acceda a la memoria de otro. Sin embargo, si dos o mas procesos deciden remover esta restricción pueden definir una región de memoria mediante la cual podrán intercambiar información. En este caso, los mismos procesos son los responsables de asegurar que no escriben simultáneamente en esta región.

Contención y concurrencia: Ocurre cuando uno o más procesos tratan de acceder al mismo recurso de manera simultánea. Esto puede causar que algunas secuencias de ejecución terminen con resultados erróneos.

Bibliografía

(5) Silberschatz, A.; Galvin, P. B. y Gagne, G. en *Operating System Concepts - International Student Version, 9th Edition.* Wiley: 2014; cap. 6. CPU Schedulling, secciones 1 a 6, págs. 261-290.

Parte II
Sincronización entre procesos con memoria compartida

4. Sincronización bloqueante

4.1. Modelo Productor-Consumidor

Uno de los paradigmas clásicos de procesos cooperativos con este tipo de intercomunicación es el de **productor-consumidor**. En este esquema, un proceso **productor** debe producir información que va a ser consumida por un proceso **consumidor**.

Una de las soluciones a este problema, es definir un buffer en una región de memoria compartida entre ambos procesos en la cual el productor pueda encolar elementos mientras que el consumidor los desencola. Ambos procesos deben estar sincronizados de tal manera que el consumidor no trate de desencolar elementos si la cola está vacía.

Se pueden usar dos tipos de buffers para implementar esta solución:

- **Unbounded buffer** (buffer infinito): No posee límites de espacio. El consumidor tiene que esperar a que el buffer tenga algo y el productro siempre puede agregarle elementos.
- **Bounded buffer** (buffer limitado): Tiene un tamaño fijo. Si está vacío, el consumidor debe esperar a que haya algo. Si está lleno, el productor debe esperar a que se haya consumido por lo menos un elemento antes de agregar otro.

Vamos a concentrarnos, en una solución con buffer limitado:

Zona de memoria compartida

```
buffer in[BUFFER_SIZE];
int cant = 0;
```

Proceso productor

```
while(true){
  while(counter == BUFFER_SIZE) {};
  in.push(item);
  counter++;
}
```

Proceso consumidor

```
while(true){
  while(counter == 0) {};
  item = pop(in);
  counter--;
}
```

4.1.1. Condiciones de carrera (race conditions)

En el ejemplo anterior, si bien el código para cada proceso puede parecer correcto, puede haber errores si ambos se ejecutan de manera concurrente. Supongamos que las lineas `counter++` y `counter--` tienen la siguiente forma en lenguaje maquina:

`counter++`:

```
register1 = counter;
register1 = register+1;
counter = register1
```

`counter--`:

```
register2 = counter;
register2 = register2-1;
counter = register2;
```

Supongamos que el valor inicial de `counter` es 5, el proceso productor agrega un elemento al buffer, el consumidor quita el primer elemento encolado y ambos procesos ejecutan `counter++` y `counter--` de manera concurrente, entonces una posible secuencia de ejecución de estas instrucciones podria ser:

Productor	**Consumidor**	`counter`
		5
`register1 = counter`		5
`register1 = register + 1`		5
	`register2 = counter`	5
	`register2 = register - 1`	5
`counter = register1`		6
	`counter = register2`	4

En este caso, el consumidor comienza a modificar la variable `counter` antes de que el productor guarde el valor correspondiente en memoria y no se .[en]tera"que un nuevo elemento fue agregado a la cola (por lo que todavía habría 5 elementos en ella). Entonces se genera una inconsistencia entre el valor de `counter` y la cantidad elementos en el buffer.

Este tipo de situaciones se llama **race condition**: Se da cuando varios procesos pueden acceder y modificar la misma variable de manera concurrente y el valor final de ésta depende del orden particular en el que se hayan realizado los accesos a la misma.

4.2. Secciones críticas

La idea es que cada proceso que se esté ejecutando en el sistema tenga un segmento de código, llamado **sección crítica**) que solo se puede ejecutar cuando ningún otor proceso esté en ella. Es decir, no puede haber mas de un proceso ejecutando su sección crítica al mismo tiempo. Para poder implementar esto, por lo general, se divide el código del segmento en tres partes:

1. **Entry section:** En donde el proceso pide permiso para acceder a la sección crítica.
2. **Critical section:** La zona crítica per se, en la que se puede manejar la memoria compartida
3. **Exit section:** Donde el proceso avisa al sistema que dejó de realizar operaciones críticas.

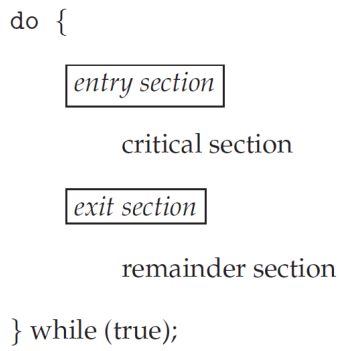

Figura 1: Estructura general de un proceso que implementa sección crítica

Una buena implementación de esta método debe sastifacer los siguientes requerimientos:

1. **Exclusión mutua**: Si un proceso está ejecutando su sección crítica, entonces ningún otro debe estar ejecutando la suya.
2. **Progreso**: Si un proceso necesita entrar a su sección crítica, entonces se le dará permiso para hacerlo en algún momento.
3. **Bounded waiting (espera acotada/no bloqueante)**: Si un proceso P pide entrar a la sección crítica, entonces hay un límite en la cantidad de veces que se la mayor prioridad a otros procesos sobre P.

4.2.1. Instrucción TestAndSet

Los sistemas opertivos modernos proveen una instrucción de hardware especial que nos permite testear y/o modificar una palabra de manera **atómica**, es decir como unidad no interrumpible de ejecución. Cuando se utiliza, el sistema no ejecuta ninguna otra instrucción hasta que ésta haya terminado. Para abstraernos, de sistemas operativos espécificos, llamemos esta instrucción `testAndSet()`, la misma toma como paramétro una variable booleana que se va a setear en `true` y va devolver el valor anterior:

```
bool testAndSet(bool* source) {
  bool result = *source;
  *source = true;
  return result;
}
```

En este caso, un sistema de procesos que utilice esta instrucción podría tener una variable booleana compartida llamada **lock** que controlaria el acceso a la secciónes criticas de cada proceso, si `lock == false` entonces es posible entrar, si es `true` el proceso debe esperar:

```
bool lock;

void main() {
  while(true) {
    ...
    while(testAndSet(&lock)) {};

    /* Sección crítica */

    lock = false;
    ...
  }
}
```

4.2.2. Instrucción CompareAndSwap

Es una instrucción primitiva ofrecida por varios porcesadores que toma 3 argumentos: una dirección de memoria, un valore esperado y un valor que se debe escribir en esa posición si se encuentra el valor esperado:

```
bool testAndSet(int* value, int expected, int new_value) {
  int temp = *value;
  if(*value == expected) *value = new_value
  return temp;
}
```

4.2.3. Mutex Lock o SpinLock (busy waiting)

La solución basada en hardware presentada en la sección anterior, generalmente es inaccesible a los programadores. Por esta razón, los sistemas operativos diseñan herramientas básicas para implementar secciones críticas: La más simple de ellas es el **mutex lock**. El mismo contiene de una variable booleana `avaliable` que indica si el lock está disponible o no y provee dos funciones:

- `acquire()`: Permite a un proceso adquirir el lock y bloquear otros procesos el acceso a su sección crítica. Si un proceso llama a esta función y el lock ya estaba tomado, el proceso queda en espera hasta que el lock se libere.

  ```
  void aquire() {
    while(!avaliable) {}; // Busy wait
    avaliable = false;
  }
  ```

- `release()`: Libera el lock para que los procesos bloqueados puedan continuar con su ejecución.

```
void release() {
  avaliable = true;
}
```

Ambas funciones deben ejecutarse de manera atómica por lo que generalmente son implementadas usando la instrucción de hardware `testAndSet()`. La principal desventaja es que requiere de **busy waiting** (mientras un proceso está su sección crítica, cualquier proceso que llame a `aquire()`) debe ciclar continuamente hasta que el lock se libere. Esto es un problema en los sistemas de multiprogramación, donde los ciclos de CPU son compartidos entre varios procesos.

4.3. Semáforos

Un semáforo `S` es una estructura que contiene una variable entera `value` y una lista `list` de procesos a las que se puede acceder mediante dos operaciones atómicas:

- `wait()`: Cuando un proceso llama a esta operación, si `S.value` es negativo entonces debe esperar. Sin embargo, en vez de hacer busy waiting, el proceso se bloquea (entra en estado `waiting`, ver sección 2.2) y se encola en `S.list`. Luego, el control es transferido al scheduler que selecciona otro proceso para ejecutar.

  ```
  void wait(S) {
    S.value--;
    if(S.value < 0) {
      agregar este proceso a S.list;
      block();
    }
  }
  ```

- `signal()`: Cuando un proceso termina de ejecutar su sección crítica, llama a esta operación que indica al semáforo que puede despertar alguno de los procesos en espera. Para esto, se quita algun proceso de la lista y se lo pasa a estado `ready` para que que el scheduler lo vuelva a tener en cuenta.

  ```
  void signal() {
    S.value++
    if(S.value <= 0) {
      quitar proceso P de S.list;
      wakeup(P);
    }
  }
  ```

Este tipo de semáforos está pensado para administrar la asignación de recursos (de los cuales se tiene una o más instancias) del sistema.

La lista de procesos en espera puede ser implmentada por un campo `link` en cada process control block (sección 2.3) y la lista de procesos del semáforo en realidad es una lista punteros a PCBs.

4.3.1. Monitores y variables de condición

Aunque los semáforos proveen un mecanismo conveniente y efectivo para realizar la sincronización de proceso, usarlos incorrectamente puede conllevar a errores dificiles de detectar, ya que estos pueden ocurrir en una secuencia de ejecución particular.

Para resolver estos errores, se dasorrollo un tipo abstracto de dato llamado **monitor** que contiene un conjunto de operaciones que ya aseguran mutual exclusion. El monitor declara variables locales que definen su estado junto con las funciones que operan esas variables (y son la única forma de accederlas).

```
Monitor M {
/** Declaración de variables compartidas **/
var x;
var y;
var z;

/** operaciones sobre esas variables **/
function op1(...) { ... }
function op2(...) { ... }
function op3(...) { ... }
function constructor(...) { ... }
}
```

La implementación del monitor debe asegurar que nunca hay más de un proceso activo dentro del mismo. De esta forma, el programador no debe preocuparse por los requerimientos de sincronización. Sin embargo, la construcción presentada no alcanza para modelar algunos mecanismos de sincronización, por lo que se agrega los mismos lo que se llama **variables de condición**.

La únicas operaciones invocadas en una variable de condición son `signal()` y `wait()`. `wait()` suspende el proceso hasta que otro proceso llame al `signal()`. Si no hay procesos esperando, entonces `signal()` no tiene efecto. Este tipo de variables se puede implementar con semáforos.

Supongamos que definimos una variable de condición $c1$ sobre el monitor M y que un proceso P invoca $c1.signal()$. Si existe un proceso Q suspendido asociado a $c1$ entonces Q debería poder ingresar al monitor, sin embargo, P sigue estando dentro del mismo. Tenemos dos posibilidades:

1. **Signal and wait:** P se pausa y espera a que Q salga del monitor o espera a que se cumpla otra condición.

2. **Signal and continue:** Q espera a que P deje el monitor o espera a que se cumpla otra condición.

Por un lado, tiene sentido dejar que P siga ejecutandose dentro del monitor. Por otro, si permitimos esto, puede llegar a pasar que para el momento en el que Q sea activado, la condición lógica que estaba esperando deje de valer. En muchos casos, lo que se hace es hacer que la ultima instrucción que se ejecuta dentro del monitor sea un `signal()`. De esta manera, P deja la sección crítica, puede seguir su ejecución y Q puede entrar en su propia sección.

4.3.2. Deadlock

La implementación de semáforos puede resultar en una situación donde dos o más procesos se pueden quedar esperando por un evento que solo puede ser causado por otro proceso en espera. Cuando esto sucede, se dice que estos procesos están en **deadlock**.

Decimos que un conjunto de procesos está en estado de deadlock cuando cada proceso del conjunto está esperando por un evento que solo puede causado por otro proceso de ese conjunto.

Ejemplo: Supongamos que tenemos dos procesos P_0 y P_1 que hacen uso de los semáforos binarios S y Q y se produce la siguiente secuencia de comandos:

P_0	P_1	Efecto
		Ambos semáforos comienzan habilitados
`wait(S)`		P_0 continua ejecución y reserva S
	`wait(Q)`	P_1 continua ejecución y reserva Q
`wait(Q)`		P_0 se suspende hasta que P_1 libere Q
	`wait(S)`	P_1 se suspende hasta que P_0 libere S

Al final de la secuencia, ambos procesos están suspendidos porque uno necesita un recurso que tiene en el otro y el sistema entra en deadlock.

4.3.3. Condiciones de Coffman

Un sistema puede entrar en deadlock si se cumplen las siguientes condiciones de manera simultanea:

1. **Mutual exclusion**: Hay al menos un recurso que no puede ser compartido. Es decir, es recurso no puede estar asignado a mas de un proceso al mismo tiempo.
2. **Hold and Wait**: Un proceso debe tener asigado al menos un recurso y estar esperando por otro que está asignado a otro proceso.
3. **No preemption**: El sistema no implementa un mecanismo que le permita quitarle los recursos a los procesos.
4. **Circular wait**: Existe un conjunto de procesos $\{P_0, \ldots, P_n\}$ tal que P_i espera un recurso que está asignado a P_{i+1} y P_n espera un recurso que está asignado a P_0.

Livelock: Se da cuando dos o más procesos no pueden progresar porque hay otros procesos esperando para conseguir un recurso. En este caso, todos los procesos involucrados esperan para poder obtener el recurso y ninguno lo acepta.

4.4. Correctitud de sistemas concurrentes

4.4.1. Modelo del proceso

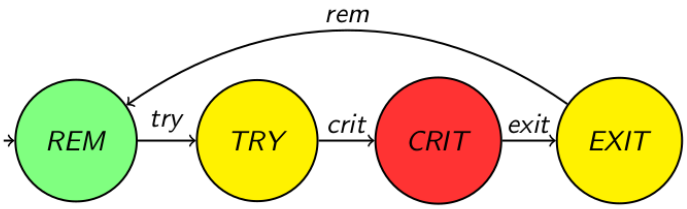

Figura 2: Modelo de un proceso según Lynch

Un proceso está definido por un aútomata finito con cuatro estados:

- **REM** es el estado en el que se encuentra cuando no está ejecutando en su sección crítica
- **TRY** es el estado en el que se encuentra cuando está ejecutando los chequeos necesarios para saber si puede entrar en su sección
- **CRIT** es cuando está ejecuntando su sección crítica
- **EXIT** cuando está ejecutando los pasos necesarios para salir correctamente de la sección crítica.

Ejecución: Es una secuencia de estados $\tau = \tau_0 \xrightarrow{l_1} \tau_1 \xrightarrow{l_2} \dots$ donde cada τ_i es un posible estado del sistema y l_i el nombre de la transición utilizada para hacer el pasaje de estados.

Una de las dificultades para demostrar correctitud sobre programas concurrentes es que tiene infinitas posibles ejecuciones. Por lo que la noción de *correcto* deja de ser únivoca y pasa a transformarse en comprobar que el sistema cumple ciertas propiedades que en conjunto deben asegurar el comportamiento deseado. Hay tres tipos de propiedades que podemos plantear:

- **Propiedades de Safety:** Aseguran que no ocurren cosas malas. Son propiedades tales que si no se cumplen entonces existe una ejecución finita del sistema en las que ocurre el evento no deseado. Por ejemplo:
  - No hay deadlocks
  - La función f nunca a devolver null

La idea es enon

- **Propiedades de Liveness (Progreso):** Aseguran que, en algún momento, van a ocurrir cosas buenas. Por ejemplo:
  - Si se presiona el botón de stop, el tren frena.
  - Cada vez que el sistema recibe un estímulo, el sistema responde en X tiempo.
- **Propiedades de Fairness:** Los procesos ejecutandose en el sistema reciben su turno con infinita frecuencia. Es decir, los procesos que componen el sistema se ejecuntan regularmente y no son postergados para siempre.

  En general, se asume que el sistema estudiado cumple este tipo de propiedades para poder demostrar las propiedades de liveness.

4.4.2. Formalización de algunas propiedades

Fairness: Para toda ejecución τ y todo proceso i, si i puede hacer una transición l_i en una cantidad infinita de estados de τ entonces existe un k tal que $\tau(i) \xrightarrow{l_i} \tau_{k+1}$. Es decir, que si un proceso puede pasar de un estado a otro, entonces en algún momento lo va a hacer.

Exlusión mutua: Para toda ejecución τ y estado τ_k, no puede haber más de un proceso i tal que $\tau_k(i) = CRIT$

Progreso: Para toda ejecución τ, si en τ_k hay un proceso i en TRY y ningún otro proceso se encuentra en $CRIT$ entonces $\exists$ un momento $k' > k$ tal que $\tau_{k'}(i) = CRIT$.

Progreso global dependiente (deadlock-free): Para toda ejecución τ, si para todo proceso que esté en estado $CRIT$ en el momento k, en algun momento k' pasa a REM entonces, va a valer que todo proceso i' tal que $\tau_{k'}(i) = TRY$ va entrar en su sección crítica en algun momento $k'' > k'$.

Progreso global absoluto (WAIT-FREEDOM): Para toda ejecución τ, estado τ_k y proceso i, si $\tau_k(i) = TRY$ entones existe $k' > k$ tal que $\tau_{k'} = CRIT$

Bibliografía

(1) Sistemas operativos - Clases Teóricas, 2019.

(4) Silberschatz, A.; Galvin, P. B. y Gagne, G. en *Operating System Concepts - International Student Version, 9th Edition.* Wiley: 2014; cap. 5. Process Synchronization, págs. 203-261.

(6) Silberschatz, A.; Galvin, P. B. y Gagne, G. en *Operating System Concepts - International Student Version, 9th Edition.* Wiley: 2014; cap. 7. Deadlock, Sección 2, págs. 317-322.

5. Programación concurrente (no bloqueante)

Todos los métodos de sincronización vistos hasta ahora son métodos **bloquantes** porque un delay inesperado en alguno de los threads puede bloquear el progreso de otros threads. Este tipo de delays es común en multiprocesadores en los que suelene ocurrir cache misses, page faults, cambios de contexto, etc.

5.1. Algoritmos wait-free y lock-free

Wait-Free Algorithm: Un método es *wait-free* si garantiza que termina en una cantidad finita de pasos cada que es llamado. Si la cantidad de pasos es acotada, entonces se lo llama *bounded wait-free*. Este tipo de algoritmos aseguran la condición de progreso no bloqueante (el delay en un thread, no neceseriamente bloquea la ejecución de otros threads.)

Population-oblivious Algorithm: Un algoritmo *wait-free* cuyo rendimiento no depende de la cantidad de threads activos.

Wait-Free Object: Un objeto tal que todos sus métodos son *wait-free*

Lock-Free Algorithms: Algoritmos que garantizan que, infinitamente amenudo, alguna llamada a un método termina en un número finito de pasos.

Esta condición es más débil que la condición de *wait-free*, por lo que cualquier algoritmo *wait-free* es un algoritmo *lock-free* pero no vale la vice versa. Los últimos admiten la posibilidad de que algunos threads sufran de inanición.

Obstruction-Free Algorithm: Son métodos que terminan en una cantidad finita de pasos si se ejecutan de manera aislada. Este tipo de métodos, obliga a pausar todos los threads que comparten el objeto/región de memoria que va a ser modificado.

Todos los tipos de algoritmos mencionados en esta sección anterior garantizan que la computación realiza progreso como un todo, independiente de como el sistema maneja los threads.

5.2. Problema ABA

Supongamos que tenemos tres threads (A, B y C) que operan sobre una cola implementada con algoritmos *wait-free*. Y se produce la siguiente situación:

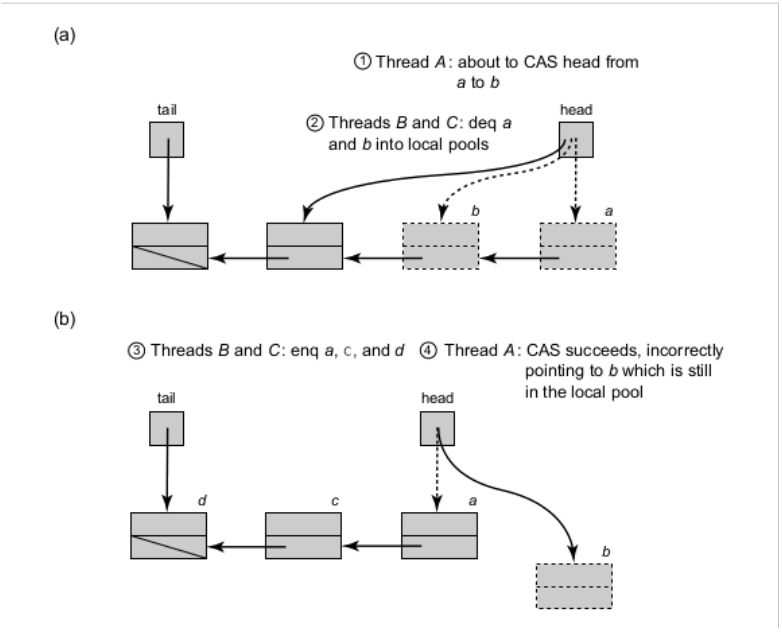

Figura 3: Problema ABA

1. El thread A necesita deseconlar el primer elemento de la cola. Lee el valor almacenado en `head` y observa que la cabeza es el nodo a y que el siguiente nodo es b. Entonces se prepara para hacer la llamada `compareAndSwap(&head, a, b)` (ver sección 4.2.2) pero es desalojado.

2. Los thread B y C desencolan los nodos a y b. y vuelven a encolar el nodo a, entonces en la cabeza queda el nodo a y c como el nodo siguiente.

3. El thread A ejecuta la llamada `compareAndSwap(&head, a, b)`. En este caso, $head = a$ por lo que la condición de la instrucción se cumple y A termina remplazando a por b. Dejando incosistente la cola.

Este tipo de situacioens se da a menudo en algoritmos que usan memoria dinámica y operaciones de sincronización condicionales. Usualmente, una referencia que está por ser modificada por `compareAndSwap` cambia de a a b y vuelve a ser a por lo que la instrucción termina exitosamente a pesar que la estructura cambió y ya no tiene el efecto deseado.

Una forma facil de resolver este problema es agregar a cada referencia atómica una estampa única que permite definir si el valor fue cambiado o no en el interval de tiempo entre que se llamo a la función y efectivamente se ejecuta.

5.3. Programación de multicores

5.4. Invalidación de caché

5.5. Reorden de instrucciones

Bibliografía

(8) Herlihy, M. y Shavit, N. en *The Art of Multiprocessor Programming, Revised Reprint*, 1st; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2012; cap. 3. Concurrent Object, Sección 7, págs. 59-61.

(9) Herlihy, M. y Shavit, N. en *The Art of Multiprocessor Programming, Revised Reprint*, 1st; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2012; cap. 10. Memory Reclamation and the ABA Problem, Sección 6, págs. 233-238.

6. Administración de entrada/salida

6.1. Polling, interrupciones, DMA

6.2. Almacenamiento secundario

6.3. Drivers

6.4. Políticas de scheduling de E/S a disco

6.5. Gestión del disco (formateo, booteo, bloques dañados)

6.6. RAID

6.7. Copias de seguridad

6.8. Spooling

6.9. Clocks

7. Sistemas de archivos

7.1. Responsabilidades del FS

7.2. Punto de montaje

7.3. Representación de archivos

7.4. Manejo del espacio libre

7.5. FAT, inodos

7.6. Atributos

7.7. Directorios

7.8. Caché

7.9. Consistencia, journaling

7.10. Características avanzadas

7.11. NFS, VFS

8. Protección y seguridad

8.1. Conceptos de protección y seguridad

8.2. Matrices de permisos

8.3. MAC vs. DAC

8.4. Autenticación, autorización y auditoría

8.5. Funciones de hash de una vía

8.6. Encriptación simétrica

8.7. RSA

8.8. Privilegios de procesos