¿Qué es el process control block (PCB)?

•Contador de programa: contiene la dirección de la próxima instrucción que debe ejecutarse para el proceso.

•Información de programación: incluye la prioridad del proceso, punteros a colas de planificación y cualquier otro parámetro necesario para la planificación por parte del sistema operativo.

•Información de gestión de memoria: incluye los valores de registros base y límite, tablas de páginas o tablas de segmentos, dependiendo del sistema de memoria utilizado por el sistema operativo.

¿Qué es una llamada al sistema? ¿Cúales son su propósito?

Una llamada al sistema (system call) es el mecanismo controlado por el cual un programa en modo usuario solicita un servicio al sistema operativo (en modo kernel).

Propósito:

Permitir que los programas usen funcionalidades críticas del sistema (leer archivos, usar red, asignar memoria, etc.)

Proteger el sistema: solo a través de system calls controladas un proceso puede acceder a recursos.

Ejemplos de llamadas al sistema:

read(), write() → acceso a archivos

fork(), exec() → creación de procesos

exit() → terminación de proceso

kill() → enviar señales

¿Qué es un módulo del kernel?

Un módulo del kernel (kernel module) es una parte del código del núcleo que puede ser cargada o descargada dinámicamente sin necesidad de reiniciar el sistema operativo.

Características:

Permite agregar funcionalidades al kernel en tiempo de ejecución (por ejemplo, nuevos drivers de dispositivos).

Se maneja con comandos como insmod, rmmod, lsmod en Linux.

Ejemplos: drivers de red, módulos de sistemas de archivos, extensiones de seguridad.

¿Por qué son útiles?

El kernel puede mantenerse pequeño y modular.

Solo se cargan los módulos necesarios.

¿El modo usuario y modo kernel, son modos de trabajo del hardware o del sistema? Explique

El **modo usuario** y el **modo kernel** son modos de trabajo del sistema que se utilizan para distinguir qué tareas son ejecutadas por parte del usuario, y cuáles por parte del sistema operativo. Dado que los usuarios en un sistema comparten recursos con el SO, el propósito de estos modos es proteger a otros programas, e incluso al sistema operativo en sí mismo, de la ejecución de código que les pueda causar daño y/o impedir su correcto funcionamiento. Si bien a este tipo de código se lo suele llamar "malicioso", no necesariamente se trata únicamente de programas que intencionalmente intenten impedir el correcto funcionamiento de otros, sino que también puede provenir de programas "incorrectos". Por ejemplo, un programa que acceda fuera de su espacio de memoria asignado por un error en su programación puede provocar daños a otros programas en la memoria.

Para lograr esta protección, los modos restringen qué instrucciones puede ejecutar el procesador dependiendo de aquel en que se encuentra el sistema en un momento dado. A aquellas instrucciones que pueden causar daño a otros programas si se las usa incorrectamente, se las llama **instrucciones privilegiadas**, y sólo se pueden ejecutar cuando el procesador se encuentra en modo kernel.

La implementación más sencilla de los modos es mediante hardware, a partir de establecer un bit en un registro del procesador que indica el modo actual, conocido como el **bit de modo** (*mode bit*, en inglés). Así, si el sistema se encuentra en modo kernel, se lo suele representar con un `0` en el bit de modo, y si se encuentra en modo usuario, se lo representa con el bit `1`.

De esta forma, cuando arranca el sistema, éste se encuentra en modo kernel y se carga el sistema operativo, el cual luego inicia las aplicaciones a nivel usuario en el modo con el mismo nombre. Cuando alguna aplicación requiere de algún recurso de parte del SO, lo hacen a través de una **llamada al sistema** (*syscall*, en inglés) que ejecuta un *trap*[1] para establecer el bit de modo en 0, o sea, en modo kernel. En este modo, el SO determina si la operación solicitada por la aplicación es válida y "legal" (es decir, que se encuentra permitida dentro del contexto del programa que la solicitó), y en ese caso la ejecuta, seguido de devolverle el control, habiendo ya revertido el bit de modo a `1` (modo usuario).

Existen implementaciones más complejas donde el bit de modo puede tomar más de dos valores, conocidos como **anillos de protección** (*protection rings*, en inglés), que permiten distinguir entre modos adicionales, como aquellos destinados a servicios del sistema operativo, o para virtualización. El uso de estos anillos implementados en el procesador depende del sistema operativo, siendo a veces poco común el uso de algunos, como es el caso de los anillos 1 y 2 en procesadores Intel con 4 rings, donde los únicos normalmente utilizados son el anillo 0 (modo kernel) y el 3 (modo usuario).

En el caso en el que algún programa intente ejecutar una instrucción privilegiada en modo usuario, el hardware impide su ejecución y le informa al SO mediante un "hardware trap" (una "excepción" por hardware síncrona) para que que éste lo maneje. Típicamente, el resultado de esto es que el programa sea finalizado.

Dicho todo esto, el modo usuario y el modo kernel son, al fin y al cabo, modos conceptuales de trabajo del sistema, en particular utilizados por el SO, cuyos detalles de implementación varían según el sistema. Casi siempre su funcionalidad se implementa por hardware (en particular, en el procesador), y permiten proteger al sistema operativo y al resto de programas, de otros programas que intenten ejecutar instrucciones que puedan impedir el correcto funcionamiento de los primeros.

¿Cúales de las siguientes operaciones deben ser privilegiadas?

```
a) Setear un valor del timer.
b) Leer el clock.
c) Limpiar la memoria.
d) Apagar las interrupciones.
e) Modificar la entrada de la table de estados de los dispositivos.
f) Cambiar de modo usuario a modo kernel.
g) Acceder a un dispositivo de entrada y salida.
```

Las operaciones que afectan al sistema completo o a otros procesos deben ser privilegiadas (solo ejecutables en modo kernel).

a) Setear un valor del timer → ✅ Sí, privilegiada (controla interrupciones de tiempo)

b) Leer el clock → ❌ No, no es privilegiada (solo lectura de datos)

c) Limpiar la memoria → ✅ Sí, privilegiada (puede afectar otros procesos)

d) Apagar las interrupciones → ✅ Sí, privilegiada (afecta control de CPU)

e) Modificar la entrada de la tabla de estados de los dispositivos → ✅ Sí, privilegiada (cambia dispositivos del sistema)

f) Cambiar de modo usuario a modo kernel → ❌ No directamente. El cambio a kernel se produce solo mediante un trap controlado (no puede forzarlo un programa).

g) Acceder a un dispositivo de entrada y salida → ✅ Sí, privilegiada (los I/O devices se manejan solo desde kernel)

¿Qué mecanismos para lograr exclusión mutua podemos usar?

CAS

Una forma de implementar exclusión con una variable de lock, que funcione correctamente, es usando **CAS**(Compare and swap). Esta es una instrucción atómica provista por varias arquitecturas, tiene la forma `CAS(l,a,b)` ; Lo que hace es leer el valor en la dirección de memoria `l` , si es igual a `a` se almacena `b` en la dirección a la que apunta `l` , de lo contrario no hace nada. Al mismo tiempo devuelve un booleano indicando si ocurre el cambio o no. En el caso de x86-64 **CAS** se puede implementar usando la instrucción `cmpxchg` . Esto nos permite usar una variable de lock, ya que podemos hacer la comparación y la asignación a la bandera de forma atómica, lo que resuelve el problema del punto anterior. Presentamos una implementación de este metodo de exclusion usando C y `cmpxchg` :

Definición de CAS	Uso de

`int CAS(volatile int *ptr, int expected, int new_val) {`
`  unsigned char success;`
 	
`  __asm__ volatile (`
`  "lock cmpxchg %2, %1\n\t"`
`  "sete %0"`
`  : "=q" (success),`
`  "+m" (*ptr),`
`  "+r" (new_val)`
`  : "a" (expected)`
`  : "memory"`
`  );`
 	
`  return success;`
`}`
	`//La bandera debe ser una var`
	`int flag = 0;`
	
	`void *funcionThread(void* arg`
	`  //Lock de zona crítica. A`
	`  while(!CAS(&flag, 0, 1));`
	
	`  /*`
	`  Zona critica`
	`  */`
	
	`  flag = 0;`
	
	`  return NULL;`
	`}`

Mutexes

La solución que vimos en clase es utilizar un *mutex*; Un *mutex* es una variable compartida que tiene dos estados, bloqueada o desbloqueada. Cuando un proceso quiere ingresar a la zona crítica esté consulta el estado del *mutex*, si está desbloqueado lo bloquea e ingresa a la zona crítica, de lo contrario el proceso deja de ejecutarse, liberando el CPU. La próxima vez que el proceso se ejecute volverá a revisar el estado del *mutex*.

¿Cuáles son los mecanismos de comunicación entre procesos?

Si el proceso es cooperativo, luego este requiere un mecanismo de comunicación entre procesos (IPC), de los cuales hay 2 modelos fundamentales, memoria compartida y envío de mensajes. En el modelo de memoria compartida o shared memory, una región en la memoria es establecida, luego los procesos pueden intercambiar información, escribiendo y leyendo lo que esté en la región compartida.En general el sistema operativo evita que los procesos puedan acceder a la información

de otros procesos, pero si 2 o más procesos aceptan remover esta restricción se creará una región compartida, notar que este método no está bajo el control del sistema operativo y hace responsable a los procesos de asegurarse de no estar escribiendo en la misma ubicación simultáneamente. En cambio en el modelo de pasaje de mensajes o message passing la comunicación toma forma de mensajes entre los procesos cooperativos.Este modelo provee un mecanismo que permite a los procesos comunicarse y sincronizar sus acciones sin ningún espacio compartido en memoria, por ejemplo en el uso de comunicación entre diferentes computadoras conectadas a la red. Este tipo de modelo tiene al menos 2 funciones:

```
send(message)

receive(message)
```

Los mensajes enviados entre procesos pueden ser fijos de tamaño o variables, con la complicación que cada uno conlleva. Si un proceso P y Q se quieren comunicar, luego se deberá crear un link de comunicacion (communication link), este link se puede implementar de muchas formas: -Comunicación directa o indirecta -Comunicación sincrónica o asincrónica -Buffer automático o específico

En el caso de comunicación directa, ambos procesos se quieren comunicar de forma explícita, es decir saben los nombres de los procesos, esto denota simetría, hay variantes simétricas, donde un proceso manda un mensaje a un proceso en específico y otros procesos escuchan sin especificar a quién están escuchando, la desventaja de ambos esquemas es el hecho de que saber los nombres de los procesos requiere hard-coding.

Cuando se habla de comunicación indirecta, los mensajes se envían y se reciben por los llamados puertos o mailbox, cada uno de estos tiene una forma única de identificarse.Dos procesos sólo se pueden comunicar su ambos tiene un puerto compartido, este esquema trae un problema de concurrencia, donde uno debería revisar qué pasa si más de un proceso quiere recibir información del mismo puerto. Los procesos pueden ser los creadores de los mailbox tanto como el sistema operativo, una vez que el proceso dueño de un puerto termine, su puerto terminará con el, y se deberá notificar a los usuarios de ese puerto que el mismo ya no existe.

Sincronización, hay 2 tipos de send() y receive() los bloqueantes y no bloqueantes, también conocidos como sincrónico y asincrónicos.Básicamente si se ejecuta alguna función (send o receive) y esta es bloqueante, el proceso frenada hasta poder recibir u/o escribir donde sea pertinente. El hecho de que sea bloqueante puede ayudar a resolver muchos problemas, por ejemplo el de consumidor y productor, pero a su vez generar otros, ya que bloquear procesos no es algo que siempre sea factible.

Por último Buffering.Tanto como directa como indirecta, la comunicación entre procesos se puede definir como una cola, y estas pueden ser implementadas de 3 formas:

```
-Capacidad cero:La cola no tiene capacidad, por lo que el link no puede tener mensajes esperan

-Capacidad limitada:Esta cola tiene un tamaño finito n, luego puede contener hasta n mensajes.
```

```
-Capacidad ilimitada:La capacidad de la cola es prácticamente infinita luego entran una cantid
```

Ambos modelos mencionados son comunes en los sistemas operativos y generalmente ambos están implementados en los mismo, ya que tiene cualidades y beneficios diferentes.Por ejemplo la comunicación a través de mensajes es útil a la hora de transferir poca información, ya que hay menos factores a tener en cuenta. La memoria compartida por otro lado puede ser mas rapida, ya que los mensajes enviados por los procesos son escritos y leídos vía llamadas a sistema, lo cual hace que sea más costosa, la memoria compartida solo necesita establecer una región compartida, una vez establecida todos los procesos que participen de esta memoria, podrán acceder, sin asistencia del kernel.

Describir el mecanismo por el cúal se refuerza la protección para prevenir que un programa modifique la memoria asociada a otro programa

El sistema operativo y el hardware protegen la memoria de los procesos usando:

Registros base y límite: Cada proceso tiene:

un registro base que marca el inicio de su memoria permitida

un registro límite que marca el tamaño permitido. → El hardware chequea automáticamente que toda dirección de memoria accedida esté entre base y base+límite.

Memoria paginada / segmentada: La MMU (Memory Management Unit) traduce direcciones virtuales a físicas. Cada proceso ve su propia "memoria virtual", aislada de otros.

Permisos de página: Cada página de memoria tiene permisos (lectura, escritura, ejecución). Si un proceso accede mal, el hardware lanza una excepción → el SO puede terminar el proceso ofensivo.

Todo esto evita que un proceso pueda dañar o leer la memoria de otro proceso.

¿Cúal es la diferencia entre hilos y procesos en Linux?

Las principales diferencias entre los procesos y los threads son:

- Los Hilos dentro de un mismo proceso comparten su espacio de direcciones, memoria global y otros atributos como el PID, lo cual hace que sea más fácil compartir información entre estos. Basta con que copien los datos a compartir en variables compartidas (en el segmento de heap o el de data). En cambio con los procesos se dificulta, ya que al realizar un fork, el padre y el hijo no comparten memoria. En estos casos se debe recurrir a mecanismos de memoria compartida (shm).
- La inicialización de un nuevo Proceso mediante fork() es mucho más costosa que la creación de un nuevo Hilo, ya que hay que duplicar varios de sus atributos como sus páginas de tablas y file

descriptors (aún utilizando la técnica de copy-on-write), mientras que al crear un Hilo estos son simplemente compartidos entre ellos.

- Como los hilos comparten su memoria, un error en alguno de ellos puede propagarse al resto de los hilos sobre el mismo Proceso, en el peor de los casos terminando con el mismo. En procesos, esta situación se evita ya que cada uno tiene su propia memoria.
- Al programar con Hilos hay que asegurarse que las funciones sean `Thread-Safe`, mediante el uso de Mutex y otros métodos para evitar condiciones de carrera, Deadlocks y otros.
- Los Hilos de un mismo proceso compiten por el uso de una única memoria virtual, lo cual puede causar problemas en casos con una gran cantidad de Hilos o cuando estos necesiten abundante memoria. Mientras, en cambio, los Procesos tienen la totalidad de su memoria virtual disponible.
- Todos los Hilos deben correr sobre el mismo programa, mientras que con los procesos estos pueden ser resultado de correr diferentes programas.
- Realizar un cambio de contexto entre hilos suele ser más rápido que un cambio entre procesos, ya que los hilos comparten gran parte de su entorno. En particular, como comparten memoria al realizar un CC no se requiere un intercambio de páginas virtuales, una de las operaciones más costosas.

Algunas CPUs proveen más de 2 modos de operación. ¿Cúales son los dos usos posibles de esos múltiples modos?

Cuando una CPU ofrece más de 2 modos (más allá de "usuario" y "kernel"), estos se usan para:

Separar niveles de privilegio en servicios del sistema operativo Ejemplo: drivers de hardware pueden correr en un modo intermedio entre kernel y usuario.

Soportar virtualización segura Ejemplo: hipervisores o máquinas virtuales corriendo en un "modo de virtualización" diferente del sistema operativo principal.

(📍 Los famosos protection rings: anillo 0 = kernel, anillo 3 = usuario; anillos 1 y 2 poco usados)

1. Cabe aclarar que la definición de *trap* es ambigüa y puede variar según la bibliografía o su definición en una arquitectura, pudiéndose referir de forma general a una interrupción, como a interrupciones por software, o incluso aquellas por software que específicamente son síncronas. En este caso, se lo usa en el contexto de una definición común para la arquitectura x86, donde un *trap* es una "excepción" por software, comúnmente utilizada para implementar llamadas al sistema. ↩️