Respuestas Final Sistemas Operativos

Tomás Felipe Melli

March 1, 2025

Índice

1 2da fecha febrero 2024

1.1 Procesos

1.1.1 Por qué el cambio de contexto entre threads es menos costoso que entre procesos?

Dado que los threads comparten gran parte de la memoria del proceso, menos el stack, la TCB y el PC, sólo se deberán cargar pocos registros en comparación a traer toda la información de la PCB de otro proceso. Considerar adicionalmente el costo de seleccionar el segmento en el que se encuentra toda la información de la tarea que debe ser cargada, los chequeos de permisos, entre otras cosas que inncrementan este overhead.

1.1.2 Por qué una syscall es más costosa que una función normal en C?

Dado que una syscall la atiende el Kernel, se produce un **context switch** y un cambio de nivel de privilegio. Por el contrario, ejecutar una función en C, no requiere de cambiar de privilegio, ya que esta se almacena en el stack del proceso. Dicho esto, el overhead en el que incurre la tarea para hacer una llamada al sistema es mucho más grande porque debe ser desalojada de la cpu, mientras que con un call a una función, no.

1.2 Sincronización

1.2.1 Explicar deadlock. Nombrar las 4 condiciones de Coffman

Deadlock es una situación que ocurre en sistemas multiprogramación donde los procesos realizan operaciones sobre recursos compartidos y los mecanismos que sincronizan el acceso a los mismos falla porque cuando analizamos la traza de la ejecución, un proceso A espera que cierto proceso B haga algo que B puede hacer sólo si A hace algo. Esta situación obstaculiza el progreso del sistema. Las condiciones de coffman son condiciones que ocurren en sistemas donde se puede dar deadlock y en caso de que se cumplan las 4, hay deadlock:

1. Exlusión Mutua : un recurso no puede estar asignado a más de un proceso (hay contención digamos en el sistema y uno lo tiene y los otros no)
2. Hold and Wait : el proceso que tiene un recurso en su poseción (hold), está esperando que se libere otro
3. No-Preemption : no existe mecanismo capaz de sacarle el recurso al proceso que lo tiene en su poseción
4. Espera Circular : un proceso espera algo de otro que ese otro espera algo del primero

1.3 Administración I/O

1.3.1 Cuáles son las dos disferencias de administración entre HDD y SSD?

SSD (Solid State Drive) y HDD(Hard Disk Drive) son dos tecnologías diferentes. En los discos, para leer los bloques de datos el cabezal debe posicionarse sobre el sector adecuado. Esto introduce dos problemas : desgaste y latencia. Mover el cabezal de un sector a otro toma tiempo y tanto mover daña el disco. Por supuesto que se han pensado algoritmos para reducir estas falencias como el *elevator*, *shortest-seek-time-first*, entre otros. En las memorias SSD, las celdas de memoria se componen de *flash nand* y la lectura es electrónica. Los problema que introduce esta memoria son el desgaste y la write amplification. El desgaste se debe a que las celdas de memoria flash nand tiene un número limitado de lecturas/escrituras por la tensión eléctrica aplicada sobre sí. El concepto de write amplification aparece en este tipo de tecnología ya que las escrituras/lecturas son de a bloques. Entonces si se modifica una pequeña cosa se termina escribiendo una locura en la ssd. Las estrategias en esta tecnología incluyen utilizar *trim* para al SO avisar a la ssd que bloques no están en uso, como también wear leveling que es para distribuir de manera más equitativa las escrituras y reducir el impacto del desgaste.

1.4 Sistema de archivos

1.4.1 Describa la creación de un comando 'snapshot' que dado un path de un archivo genere un snapshot del mismo sin duplicar memoria de más basado en un sistema de inodos

La idea es hacer uso de la técnica **copy-on-write**, en esencia, el archivo 'snapshot' apunta a los mismos bloques de memoria que el archivo que se encuentre en el path que nos pasan. Sólo se **crearán nuevos bloques de datos** para los bloques que hayan sido modificados.

1.5 Seguridad

1.5.1 Explique las diferencias entre MAC y DAC.

En seguridad, existe este concepto de control de acceso en el que el sistema operativo debe controlar que los sujetos del sistema tengan los permisos adecuados para acceder a cierta información. MAC (Mandatory Access Control) y DAC (Discretionary Access Control) son dos modelos de control de acceso.

- En MAC, el control de acceso se da de forma centralizada. Es decir, existe una política de seguridad que establece el administrador y los usuarios no pueden modificarla. En particular, se implementa con un sistema basado en etiquetas (labels) que cada archivo tiene asociada.
- En DAC el control de acceso lo define el owner del archivo (cualquier usuario owner) y es a discreción de él.

2 1era fecha febrero 2024

2.1 Procesos

2.1.1 Cómo hace un proceso padre para enterarse que su hijo finalizó ? Cómo puede saber si terminó normalmente o con un error?

Cuando el proceso termina su ejecución, mediante la syscall **EXIT()** envía una señal al padre llamada **SIGCHLD** o señal nro.17. La syscall exit toma como parámetro el **status** :

```
if (pid == -1)exit(EXIT_FAILURE);
...
exit(0)
```

Donde el 0 indica que el proceso finalizó correctamente y el 1 que hubo un error.

2.2 Sincronización

2.2.1 Tengo dos procesos con memoria compartida. Cómo hago para que no accedan a memoria al mismo tiempo? Dar 2 formas

Si tengo dos procesos que comparten memoria, como una base de datos, para controlar los accesos concurrentes puedo :

- Mutex : es un objeto que permite la exclusión mutua entre los dos procesos. Es decir, el owner del mutex podrá realizar la operaciones que quiera sobre el recurso compartido y devolverlo una vez que termine (unlock).
- Semáforo binario : se utiliza el tipo de semáforo que tomá valores en el dominio binario. Cuando el proceso que esté haciendo uso del recurso, luego de hacer sem.wait(), pase a la sección crítica y ejecute lo que necesite, podrá al final avisarle al otro proceso con sem.signal() para que pueda ingresar a la sección crítica.

2.3 Scheduling

2.3.1 Qué es una política de scheduling con prioridades? Cuál es el mayor problema que tiene y cómo se resuelve

Una política de scheduling con prioridades es un mecanismo para asignar ráfagas de cpu a procesos basado en cierta cualidad del proceso. Por ejemplo, EDF (Earliest Deadline First) donde el scheduler priorizará los procesos que tengan el deadline más cercano en tiempo. Puede ser útil en sistemas real-time donde si no se cumple, algo malo sucede.
El mayor problema que tiene esta política es que no está libre de inanción, ya que si aparecen procesos con mayor prioridad, en el caso de deadline por ahí no es tan intuitivo, pero con SJF(Shortest-Job-First), si entran procesos con ráfagas más cortas, se adelantan y otros con ráfaga larga nunca se ejecutarán.

2.4 Sistema de Archivos

2.4.1 Qué se modifica al crear un archivo en EXT2 (a nivel inodos y estructuras)? Se puede crear un archivo nuevo si me quedé sin inodos en EXT2?

La creación de un archivo en un FS basado en inodos como ext2 conlleva una serie de pasos.

1. Ya conocemos la estructura de ext2, *boot_block*/*block_group_*0/.../*block_group_n* y dentro del bloque de grupo tenemos más información sobre el grupo, en particular, para la creación de un archivo, tenemos el **bit map de bloques y de inodos**. Es decir, estructuras que con 0 indican que el bloque/inodo está libre u ocupado(1). Dicho esto, el primer paso es encontrar un inodo disponible.
2. Luego de setear el bitmap de inodo en 1, tenemos que asignarle un bloque de datos a ese inodo, poniendo 1 en el bitmap de datos en algún bloque disponible.
3. Modificar en la tabla de inodos, el inodo conseguido asignando la información que se quiere almacenar al crearlo.
4. Modificar el dir_entry apropiado para ordenarlo adecuandamente en el sistema de archivos

En el caso de no contar con inodos disponibles en el grupo, se intentará hacer lo mismo en otro grupo. Supongamos que todos los grupos tienen todos los inodos ocupados. El sistema fallará. No necesariamente no hay más espacio disponible en el sistema. Hay soluciones como limitar la cantidad de bytes por inodo :

```
mkfs.ext2 -i 1024 /dev/sdX
```

El problema asociado a tener muchos inodos ocupado con espacio de sobra para seguir creando archivos se llama **fragmentación de inodos**.

2.5 Seguridad

2.5.1 Qué es el SETUID? Qué riesgos tiene? Dar 2 ejemplos de vulnerabilidades

EL SETUID es un atributo de ejecución de programas en sistemas basados en Unix que permite a un programa correr con privelgio de owner. Recordemos que Linux maneja el control de acceso con el modelo DAC donde los privilegios de owner son los más elevados. Los riesgos que conlleva que un usuario pueda ejecutar con permisos de owner, supongamos un programa donde el owner es root, podría ser escalar privilegios o mismo aprovechar alguna vulnerabilidad del programa para hacer un ataque de integridad si el programa recibe información por pantalla y esos datos al ser almacenanados, se modifican a propósito.

3 3era fecha Diciembre 2024

3.1 Procesos y Syscalls

3.1.1 ¿Qué es una syscall? ¿Cómo difiere de una llamada a una función de usuario?

Una syscall, como su nombre lo indica, es una llamada al sistema. Es la forma que tienen los procesos de hacer uso de los servicios que ofrece el sistema operativo. La razón es que muchas veces los procesos corren con niveles de privilegio inferiores y quieren realizar ciertas acciones que no puede suceder directamente. La syscall es la interfaz para que interactúe cierto proceso a nivel usuario con los servicios del SO, como abrir un archivo, leer, escribirlo, hacer un fork, ... entre otras miles de funcionalidades. En esencia, una syscall es la API del SO.
Una llamada a una función al estilo :

```
1    CALL FOO
```

No requiere de un **context switch** ni de cambiar de privilegios. Esto es porque la función *foo* se ejecuta en el mismo espacio de memoria que el proceso que la llama, con el mismo nivel de privilegio. Por el contrario, la syscall necesita hacer el cambio de contexto al Kernel del SO para poder y evidentemente, escalar en privilegio, para realizar la acción pedida por el proceso que hace la llamada al sistema.

3.1.2 Explique el proceso de transición de modo usuario a modo kernel cuando ocurre una syscall.

La transición del modo usuario al modo kernel al momento de una syscall ocurre como sigue :

1. Interrupción : sí o sí se interrumpe para poder pasar al modo privilegiado y poder responder al pedido de la tarea que corre a nivel de usuario.
2. Cambio de contexto : necesitamos guardar la info de la tarea que estaba en ejecución en su PCB correspondiente. Cargar el kernel para que atienda el pedido. Realiza lo solicitado.
3. Retorna a la tarea a su ejecución, para ello, guarda el estado de las variables del kernel en su PCB, carga la PCB de la tarea solicitante en el proce y cambia el nivel de privilegio.

3.1.3 Mencione al menos tres syscalls comunes en sistemas Linux y describa brevemente su propósito.

Existen muchísimas, por mencionar :

- read()/write() son llamadas al sistema muy comunes que lo que hacen es recibir un file descriptor (para saber dónde leer/escribir), el buffer con la data y el tamaño. Podemos agregar open()/close() para abrir un archivo y para cerrar descriptores de archivos.
- fork() es otra syscall muy utilizada que crea un proceso hijo, es decir lo que hace es clonar un proceso y retorna el identificador del proceso hijo.
- exit() se utiliza para terminar un proceso y devolver un estado de terminación
- también está toda la familia **exec** que es una syscall utilizada por los programas para correr otros, reemplazando el actual (razón por la cuál, si hay algo escrito en líneas inferiores, nunca se ejecuta)

3.2 IPC

3.2.1 Compare los mecanismos de IPC que se utilizan en ambientes locales (como memoria compartida) con los usados en ambientes remotos (como RPC).

Veamos cómo funciona cada uno :

- **RPC** permite que un proceso ejecute una función o procedimiento en otro proceso que se ejecuta en una máquina diferente, como si fuera una llamada local.
  El proceso cliente invoca una función en el servidor remoto a través de una llamada local. El sistema de RPC maneja la serialización de los datos (marshalling), la transmisión de la solicitud a través de la red y la deserialización (unmarshalling) en el lado del servidor. El servidor procesa la solicitud y devuelve el resultado al cliente.
- Con **Memoria compartida** los procesos pueden mapear una región de memoria a su espacio de direcciones y leer/escribir directamente en esta memoria compartida. No requiere comunicación a través de la red, lo que lo hace más rápido en comparación con métodos que requieren transmisión de datos.

3.2.2 ¿Qué desafíos únicos presenta cada caso?

Según el mecanismo se presentan los siguientes desafíos :

- Cuando los procesos interactúan con **memoria compartida** aparecen problemas de contención ya que los procesos compiten por acceder al recurso compartido.
- En cuanto a procesos que se comunican remotamente con **RPC** los problemas se vinculan a la conexión como puede ser una falla de red. Además del tiempo que tardan los mensajes en llegar de un punto a otro (latencia).

3.2.3 ¿Cómo se resuelven los problemas de rendimiento y confiabilidad?

Para solucionar estos problemas de rendimiento:

- En el caso de la **memoria compartida** se utilizan mecanismos de sincronización para controlar el acceso al recurso compartido.
- En el caso de la comunicación vía **RPC** se pueden implementar técnicas de multiplexación de solicitudes y optimización del uso de la red (TCP congestion control). Así como también técnicas de compresión de datos para reducir la cantidad de información que debe ser transmitida a través de la red.

En el caso de la confiabilidad (que la comunicación sea efectiva):

- Para los procesos que **comparten memoria** con implementar bien la sincronización y que no haya deadlocks alcanza
- Para procesos que están **remotamente aloojados** se puede utilizar la persistencia de mensajes para asegurarse de que las solicitudes importantes no se pierdan o mismo protocolos como heartbeat (latido) para monitorear la disponibilidad de los servidores y garantizar que los clientes solo realicen RPC a servidores activos.

3.3 Scheduling

3.3.1 ¿Cuál es la diferencia entre scheduling preemtive y no preemtive?

La diferencia entre un scheduler preemptive y uno no-preemptive en esencia es que uno, el preemptive, tiene la capacidad de desalojar al proceso que está haciendo uso del procesador en 'cualquier' momento durante su ejecución (cualquier significa, siempre que el reloj haga tick (entre 50 y 70 veces por segundo)). Por el contrario, el no-preemptive no tiene esa facilidad y está a merced de que el proceso que está en poseción del proce lo ceda, o en el momento en que se interrumpe (ya que el kernel toma el control), lo desaloje. Muchas veces estos últimos, como requieren de hw (el reloj), en sistemas embebidos no están presentes.

3.3.2 Dé un ejemplo práctico donde sea esencial usar scheduling preemtive.

En sistemas donde los procesos tienen que cumplir un deadline es fundamental. Supongamos los sistemas de navegación aérea necesitan cumplir plazos muy estrictos en el procesamiento de información relacionada con la ubicación y la trayectoria de los aviones, la gestión del tráfico aéreo, y las decisiones que se deben tomar en tiempo real para evitar colisiones y garantizar la seguridad.

3.3.3 ¿Qué desafíos adicionales introduce el scheduling preemtive en términos de diseño del sistema operativo?

Los desafíos que introduce el scheduling preemptive son :

- Sincronización : se incurriría en race-conditions ya que sin el manejo adecuado de acceso a recursos se modificarían datos generando resultados inconsistentes. En scheduling no preemptive no pasa en general, ya que se ejecutan de principio a fin de manera secuencial.
- Starvation : si la política de scheduling prioriza ciertos procesos, podría ocasionar que algunos procesos no se ejecuten nunca. En no-preemptive no pasa, ya que todos se ejecutan de principio a fin una vez que tienen el proce a disposición.

3.4 Sistema de Archivos

Un usuario eliminó por accidente un archivo en un filesystem ext2.

3.4.1 Explique qué ocurre con el inodo y los bloques de datos.

Cuando se elimina un archivo, ocurre lo siguiente :

1. Se busca en el bitmap de bloque de datos del respectivo grupo al que pertenece, y se marca con un 0 aquellos bloques que están apuntados por el archivo en cuestión. Es decir, recorremos el inodo, y por cada bloque apuntado, le marcamos 0 en el bitmap asociado a los bloques de datos.
2. Una vez liberados los bloques, liberamos el inodo en sí, esto es en el bitmap de inodos, le ponemos un 0.
3. La entrada de directorio asociada a este inodo, la eliminamos.

3.4.2 ¿Qué estructuras pueden analizarse para intentar recuperar el archivo?

Como la información dentro de los bloques de datos del archivo no es que se borra, podríamos ver, en caso de saber el número de inodo, la tabla de inodos y recorrer el mismo de forma que podamos, en el bitmap de bloques de datos, poner un 1 en aquellos apuntados por el inodo y consecuentemente, recuperar la información.
En caso de no conocer el nro de inodo, lo que podríamos hacer es tratar de encontrar la entrada de directorio si aún no fue sobreescrita en el directorio en el que estaba el archivo en cuestión. Si la encontramos, vemos qué nro de inodo tenía asociado y ahí podemos hacer lo de arriba.

3.5 Seguridad

3.5.1 Diseñe un sistema de autenticación de usuarios al sistema operativo de manera remota usando criptografía de clave pública/privada y sin usar contraseñas.

Un sistema basado en asimetría cripto hace algo como : El servidor genera un desafío aleatorio. El cliente firma el desafío con su clave privada. El cliente envía la firma al servidor. El servidor usa la clave pública del cliente para verificar la firma. Si la verificación es exitosa, el servidor autentica al cliente.

3.5.2 ¿Es su sistema más seguro que usar contraseñas? Justificar.

El sistema basado en criptografía de clave pública/privada es más seguro que el uso de contraseñas debido a la ausencia de transmisión de información sensible (como contraseñas)

4 1era fecha Diciembre 2024

4.1 Procesos

4.1.1 ¿Cual es la diferencia entre un proceso y un thread?

Primero vamos a definir cada uno :

- **Proceso** : un proceso es un programa en ejecución que tiene un espacio de memoria donde se almacena su código, el área de datos (heap) que es la memoria dinámica donde se almacenan los datos y el stack que guarda las variables locales, llamadas a funciones (stack frame).
- **Thread** : un hilo de ejecución o thread es una unidad de ejecución dentro de un proceso. Pensemos lo siguiente, el proceso es descargar un archivo de internet, procesar la data y devolver un resultado. Si fuese cada una de las partes una cajita, tendríamos la unidad de descarga, procesamiento y resultado. Supongamos que dentro de la misma tarea, en vez de ejecutarla en secuencia, el proce le asigna un quantum a cada unidad de estas. Intercalandolas para no perder tiempo en descarga cuando ya tengo, aunque poca, data ya lista para ser procesada. La idea es que, aunque sólo tenga un core, el context switch es más simple porque los threads comparten el mismo espacio de memoria (mejor comunicación) y sólo el stack es diferente. En sistemas multicore, se paraleliza y es un golazo porque puede ejecutarse en simultáneo muchos threads, lo que mejora incluso más el throughput del sistema.

Dicho esto, las diferencias son, en principio, que un proceso puede ejecutarse en un sólo hilo de ejecución de manera secuencial, pero también puede distribuirse en varios para mejorar el throughput y evitar que el proceso se bloquee en caso de una operación de I/O.

4.1.2 ¿Qué pasaría si los threads compartieran el stack?

Mencionamos que en el stack se guarda toda la información relacionada a las variables locales y llamadas a funciones. Problemón si, supongamos un programa que calcula el promedio de 1000 datos, y tenemos 4 threads que tienen que procesar de a cuartos. Si para preprocesar esos 250 datos un thread pisa los datos del otro, tendríamos un problema de consistencia. Y como dijimos que también las llamadas a funciones, podría modificarse la dirección de retorno de otro thread, lo que causaría un problema de integridad.

4.2 Concurrencia

4.2.1 Hablar de sincronización entre procesos. ¿Como se implementan?

La sincronización entre procesos es un desafío de los sistemas con multiprogramación (sistemas que pueden ejecutar varias tareas en el mismo procesador) con scheduling preemtive. Es un desafío ya que como podemos intuir, ejecutar tareas diferentes en el mismo proce podría ocasionar problemas de consistencia de datos cuando entre ellas comparten un recurso de memoria como una tabla de datos o mismo una variable entera. Esto sucedería pues el scheduler desalojaría una tarea y seguiría con otra, y este interleaving de procesos podría dar trazas de ejecución donde las variables sean inconsistentes (el ejemplo de donación con el nro de ticket de la clase).
Para abordar estas problemáticas hay una serie de herramientas. Locks y semáforos. Dentro de locks hay varios tipos, mutexes, spin-locks. La idea detrás es proveer a un proceso del ownership de cierto recurso que incurre en contención, es más estricto. En cambio, los semáforos, hay dos tipos, binario y counter. Se suele utilizar el counter ya que permite a cierta cantidad de procesos acceder a cierto recurso compartido, es más flexible en cuanto al acceso. La idea de estos es coordinar accesos.

4.2.2 ¿Que es un spinlock?

Un spin-lock es un tipo especial de lock, implementado con TASLock (Test-and-Set-Lock) en el que el proceso hace un testeo compulsivo del lock sin bloquearse. Es decir que está 'girando' haciendo ese spin de chequeo constante hasta que en algún momento lo tiene. Esto incurre en busy-waiting y es consumo de cpu. Este chequeo es atómico, y esa es la optimización que se hace del spin-lock con TTASLock que en vez de hacer el spin de manera atómica lo hace normal, y si ve que está libre, recién ahí se manda a hacer el test de manera atómica.

4.3 Scheduling

4.3.1 Mencionar 3 politicas de scheduling.

1. **EDF(Earliest Deadline First)** : se priorizan procesos con deadline más temprana
2. **SJF(Shortest Job First)** : se priorizan procesos con ráfagas cortas

3. **FCFS(Fisrt-Come-First-Served)** : se ordenan como una cola fifo, el primero en llegar se ejecuta primero y por eso sale primero.

4. **Multilevel-Feedback-Queue** : es un sistema de colas donde según el nivel los procesos tienen una prioridad asociada. Normalmente se implementa con aging para evitar inanición de procesos menos prioritarios y se suele bajar de prioridad pero dando mayor quantum a procesos intensivos en CPU y priorizando a procesos intensivos en I/O con quantum más corto.

5. **Round Robin** : es una política que asigna un quantum a cada proceso que se encuentra en una lista de procesos en estado ready por orden de llegada.

4.3.2 ¿Como diseñarías un scheduler para un dispositivo móvil teniendo en cuenta cuestiones de hardware?

Para el caso de un dispositivo donde habrá procesos interactivos y la carga del sistema debe ser minimizada para que el usuario reciba una response time bien rápida utilizaría una política de scheduling del tipo multilevel-feedback queue con aging para priorizar procesos interactivos y que la experiencia del usuario sea lo más eficiente posible. Todos los procesos de tipo bactch como copias de seguridad podrían recibir mayor quantum pero con menor prioridad.

4.4 Memoria

4.4.1 ¿Cual puede ser la razón de que el proceso no encuentre una página en memoria? Explicar los 3 casos.

Caso	Descripción	Solución
Página no está en memoria	La página solicitada por el proceso no está cargada en la memoria RAM.	Cargar la página desde el disco o archivo de intercambio.
Acceso no permitido a la página	El proceso intenta acceder a una página en memoria, pero no tiene los permisos adecuados.	Notificar el error o corregir los permisos de la página.
Página desalojada (swapping)	La página fue movida a disco para hacer espacio en memoria, y el proceso intenta acceder a ella.	Traer la página de nuevo desde el disco a la memoria.

4.5 Sistema de archivos

4.5.1 Explicar las diferencias entre soft links y hard links.¿Cuales son las ventajas y desventajas entre ambos?

Definamos primero qué hace cada uno :

- Hard : es un enlace que funciona de alias de un archivo, pero que se vincula directamente con el inodo del cuál enlaza. Sólo funciona dentro del mismo FS. Cuando se elimina el archivo, los datos asociados al mismo no se liberan hasta que se elimine el enlace duro al mismo. Por eso es hard-link

- Soft : también llamado simbólico, enlaza con el nombre del archivo, a su path digamos. Es por ello que puede apuntar a un archivo por fuera del propio fs. En caso de que se elimine el apuntado, este enlace queda apuntando a la nada.

Característica	Hard Link	Soft Link
Referencia	Apunta al mismo inodo	Apunta a una ruta del archivo (nombre)
Dependencia del archivo original	No depende del archivo original, el archivo sigue existiendo si hay más enlaces	Depende del archivo original, se rompe si el archivo original se elimina
Creación de enlaces a directorios	No se puede crear un hard link a un directorio (excepto en casos especiales)	Puede apuntar a archivos y directorios
Visibilidad	No se puede distinguir del archivo original	Se distingue claramente como un enlace simbólico
Comportamiento si el archivo se elimina	El archivo sigue existiendo si hay más hard links apuntando a él	El enlace se rompe si el archivo original se elimina
Tamaño	Igual al archivo original	El tamaño es el tamaño de la ruta de referencia
Identificación	No se diferencia del archivo original	Puede identificarse fácilmente como un enlace (usando `ls -l`)
Comando de creación	`ln archivo_original archivo_hardlink`	`ln -s archivo_original archivo_symlink`

4.6 Seguridad

4.6.1 ¿Qué es y para qué sirve el permiso de SetUID?

El atributo/permiso setUID es un atributo especial que permite a cierto sujeto del SO como un usuario que no es owner de un objeto, ser ejecutado como si lo fuese. Esto sirve para que temporalmente pueda correr con elevados privilegios y no haya problemas de permisos en la ejecución del programa.

4.6.2 ¿Como crees que se implementa el comando "sudo" de linux?

Archivo /etc/sudoers : qué usuarios o grupos pueden ejecutar comandos como root o con privilegios elevados. Solicitud de ejecución de comandos (sudo ¡comando¿) Autenticación del usuario : el sistema solicita la contraseña del usuario para verificar su identidad Generación del token de tiempo (caché) Ejecución del comando con privilegios elevados Auditoría y registro : se registra en el archivo de logs del sistema (normalmente /var/log/auth.log), lo que permite auditar el uso de privilegios elevados.

5 3era fecha Marzo 2023

5.1 Procesos

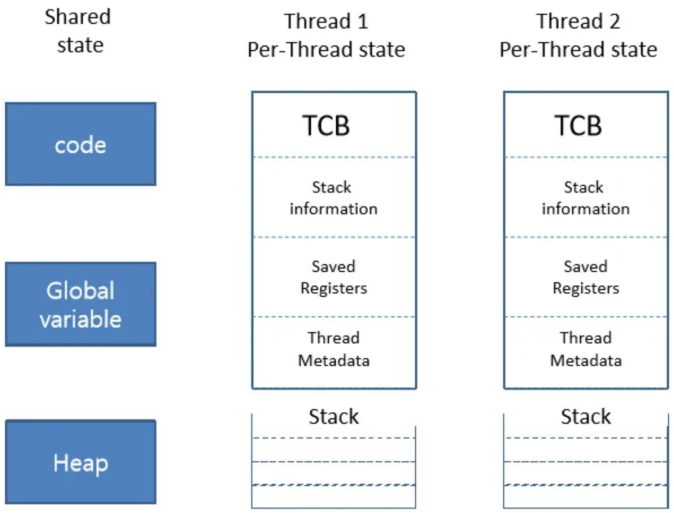
5.1.1 ¿Qué es la comunicación entre procesos en sistemas operativos y por qué es importante? ¿Cuáles son lo métodos más comunes?

La comunicación entre procesos en los sistemas operativos es la interacción que existe entre procesos que ocurren ya sea en el mismo equipo, o entre procesos que se encuentran en distintos equipos. Es importante ya que permite que los procesos puedan intercambiar información o comunicarse la ocurrencia de eventos. Los mecanismos de comunicación son :

1. **Pasaje de Mensajes** : el sistema operativo le define un canal de comunicación para que los procesos puedan intercambiar mensajes, como una cola o un socket (está asociado a una dirección IP y un puerto en el sistema operativo).
2. **Memoria compartida** : normalmente puede ser una base de datos u otro espacio de memoria.
3. **Pipes** : archivos anónimos temporales que pueden funcionar como un canal de comunicación.

5.1.2 ¿Qué estructuras de la PCB es necesario duplicar para el caso de un thread_fork?

Cuando creamos un thread, no copiamos la PCB completa, ya que los hilos comparten la memoria del proceso (en particular la PCB) no el Stack. Por tanto, necesitamos un stack para el hilo, el TCB (thread control block) de cero para este hilo, y de la PCB necesitamos definirle un PC, un PID, los registros de la cpu, el estado del proceso.



5.2 Scheduling

5.2.1 ¿Qué política de scheduling de procesos simplifica la implementación de semáforos? Justifique.

Las políticas de scheduling **no-preemptive** simplifican la sincronización ya que como no deaslojan al proceso en ejecución, este se ejecuta de principio a fin, es decir que si necesita hacer uso de un recurso compartido estará a disposición de sí de principio a fin sin que otro proceso haga una modificación en el medio.

5.2.2 ¿Qué es la inversión de prioridad y cómo se puede evitar en la planificación de procesos?

La inversión de prioridad es un fenómeno que ocurre cuando un proceso de baja prioridad interfiere en la ejecución de un proceso de alta prioridad, debido a la forma en que el sistema maneja las prioridades y los recursos compartidos. Este problema surge principalmente en sistemas que utilizan scheduling preemptivo con prioridades estáticas. Supongamos que el Proceso A necesita acceder a un recurso compartido (como una sección crítica de código o una variable global), pero está esperando que Proceso B libere ese recurso. El Proceso B tiene una prioridad baja y, mientras tanto, está siendo interrumpido por Proceso C (que tiene prioridad media). Como el Proceso A no puede ejecutar hasta que el Proceso B libere el recurso, Proceso A termina esperando mucho tiempo, a pesar de tener una prioridad mucho más alta que Proceso B.
Existen una serie de mecanismos para evitar esto :

1. **Herencia de Prioridad (Priority Inheritance)** : cuando un proceso de baja prioridad (que está bloqueado esperando un recurso) necesita ser interrumpido por un proceso de alta prioridad, herede la prioridad alta temporalmente hasta que libere el recurso que está bloqueando al proceso de alta prioridad.

Si el Proceso B (de baja prioridad) está bloqueando el recurso que el Proceso A (de alta prioridad) necesita, el sistema eleva temporalmente la prioridad de Proceso B hasta la de Proceso A (o una prioridad intermedia). Esto permite que Proceso B termine más rápidamente y libere el recurso. Una vez que Proceso B termine su tarea y libere el recurso, su prioridad vuelve a su valor original.

2. **Precedencia de Prioridad de Recursos (Priority Ceiling Protocol)** : se asigna a cada recurso un techo de prioridad, que es la prioridad más alta de los procesos que pueden usar ese recurso. Cuando un proceso quiere acceder a un recurso, se le asigna automáticamente la prioridad del recurso (el techo de prioridad) si el proceso tiene una prioridad inferior a la de ese techo. Si el proceso de baja prioridad intenta acceder al recurso, su prioridad será aumentada al techo del recurso mientras lo esté utilizando. Esto evita que los procesos de alta prioridad tengan que esperar si el recurso está siendo utilizado por un proceso de baja prioridad.

3. **Protocolos de Control de Recursos en Tiempo Real (Real-Time Resource Access Control)** : en los sistemas de tiempo real, es común usar protocolos específicos que gestionan la prioridad de los recursos y cómo se asignan. Algunos de estos protocolos, como el Protocolos de Acceso de Recursos Basado en Prioridad (como el Priority-Driven Locking), implementan políticas estrictas de asignación de recursos para prevenir la inversión de prioridad y garantizar que los procesos más críticos tengan acceso a los recursos sin ser bloqueados por procesos menos importantes.

4. **Planificación con Prioridades Dinámicas (Priority Aging)** : a medida que los procesos esperan más tiempo en la cola, su prioridad se incrementa gradualmente, evitando que procesos de baja prioridad bloqueen indefinidamente a los de alta prioridad.

5.3 Sistemas de archivos

5.3.1 Se decide implementar un nuevo comando llamada "timestump" que recibe como parámetro el nombre de un archivo y que sirve para modificar la fecha de la última modificación del mismo, en un filesystem de tipo ext2. Describa como lo implementaría, indicando las operaciones que se realizan y la/s estructura/s que se toca/tocan. Tenga en cuenta la modificación del contenido y la modificación de la metadata.

Como sólo tenemos el nombre del archivo, ese dato estará también en la dir_entry del archivo. Esto nos permitirá comparar con las dir_entries que estén en ese directorio y sacar el nro de inodo. Ahora con el nro de inodo, tendremos que buscar en el grupo, dentro de él en la tabla de inodos para cargar nuestro inodo en memoria y luego, como los inodos tienen los timestamps dentro de su estructura, bastará con modificarle aquella que nos interese.

5.4 Memoria

5.4.1 Explique el mecanismo por el cuál la paginación protege al espacio de memoria de un proceso.

La protección se logra mediante bits de permisos en las entradas de la tabla de páginas. Estos bits pueden especificar si una página es de solo lectura, de lectura y escritura o ejecutable.
Cuando un proceso intenta acceder a una página de manera no permitida (por ejemplo, escritura en una página de solo lectura), se produce una excepción o fallo de página, protegiendo así el espacio de memoria del proceso.

5.4.2 Si una página es compartida por dos procesos. ¿Es posible que esta página sea de solo lectura para un proceso y de escritura para el otro? Justificar.

Sí, es posible. Esto se puede lograr utilizando permisos de página diferenciados: La tabla de páginas puede asignar permisos distintos para cada proceso. Por ejemplo, un proceso puede tener la página marcada como solo lectura, mientras que el otro proceso puede tenerla marcada como lectura y escritura.

5.5 Seguridad

5.5.1 ¿Qué es y para qué sirve el permiso de SetUID? Proponga algún mecanismo que provea una funcionalidad similar y, de ser necesario, identifique potenciales debilidades del mismo.

El setuid es un atributo que permite a cualquier usuario ejecutar un programa con privilegios del owner. El permiso va directamente en el programa y por tanto, es persistente en el tiempo. Existe otro mecanismo, el uso de **sudo**. Este comando permite a cierto usuario (se autentica con su contraseña) ejecutar algo con privilegios de root. Es más controlado ya que los privilegios son temporales a la ejecución de ese programa, una vez finalizado se termina el privilegio. Además, los usuarios que pueden hacer uso de sudo están especificados en etc/sudoers. Es fácil de auditar.

6 2da fecha Marzo 2023

6.1 Procesos

6.1.1 Describa como funciona la lectura de un archivo desde un proceso de usuario indicando las llamadas al sistema, sus parámetros y los controles que se hacen. Relacione el concepto de monitor de referencias.

Cuando un proceso de usuario quiere leer un archivo se realizan una serie de llamadas al sistemas.

1. **open()** : hay que abrir el archivo, le pedimos al sistema que le asigne un file descriptor que nosotros vamos a capturar desde el usuario. Le pasamos un PATH, y le decimos con qué intención, si es lectura, escritura, lecto-escritura,...
2. **read()** : queremos leer del file descriptor que nos devolvió open, y para ello usamos esta syscall, necesitamos definir el lugar donde vamos a almacenar la infor leída (con un buffer) y luego la cantidad de bytes que vamos a leer (count).
3. **close()** : es la syscall que nos permitir decirle al SO que ya puede liberar los recursos asociados a la lectura del archivo.

Los controles que hace el sistema, básicamente es, si el proceso tiene los permisos adecuados para realizar la lectura.
El monitor de referencias es la herramienta que tiene el SO para mantener un conteo de los procesos que tienen una referencia a cierto descriptor. Si ve que no hay referencias, libera los recursos asociados a tenerlo abierto. Es por ello, que el close es importante al trabajar con ellos, para que el sistema sepa que tiene que liberar recursos.

6.2 Concurrencia

6.2.1 ¿Para qué se usa TAS en semáforos? Explicar que alternativas puede haber con el hardware y utilidades de SO correspondientes.

TAS(Test-and-Set) es una operación que se ejecuta atómicamente y lo que hace es capturar el valor de una variable, le setea un valor y devuelve el que capturó. En un semáforo binario, es útil para excluir a otros proceso del uso de un recurso compartido. Actualmente, se puede hacer uso de TTAS que hace un testeo no atómico del semáforo binario hasta que tiene 'evidencia' de que si lo hace atómicamente, logrará conseguirlo. Esto parece que no cambia nada, pero sí, ya que no se desperdician ciclos de clock en un recurso inaccesible. Esta optimización se llama *local spinning* y la espera es pasiva.

6.3 Sistema de Archivos

6.3.1 ¿Qué modificaciones se hacen en el sistema de archivos cuando se hace rm a un archivo en ext2?

El comando rm lo que hace es básicamente realizar la eliminación de un archivo:

1. Elimina la entrada de directorio asociada a ese inodo
2. Marcar los bloques de datos del archivo en el bitmap de data blocks del bloque de grupo en 0.
3. En el bitmap de inodo, le pone un 0

6.3.2 ¿Por qué rm no es seguro? ¿Por qué esta implementado así?

No es 'seguro' en términos de que la información está todavía en el disco, es decir, si no se sobreescribe está. Imagino q por performance, como que lo dejás así y no gastás energía en ocuparte del borrado de cada bloque que ocupaba ese archivo. Y pienso en tal vez lo que vimos de administración de memoria que hacer esto también reduce la vida útil del dispositivo de almacenamiento.

6.4 Seguridad

6.4.1 ¿Qué es y para qué sirve el permiso de SetUID? Proponga algún mecanismo que provea una funcionalidad similar y, de ser necesario, identifique potenciales debilidades del mismo.

Ya respondimos esto varias veces.

6.5 Sistemas Distribuidos

6.5.1 Explicar la relación entre consistencia, disponibilidad y tolerancia a fallos en sistemas distribuidos.

Teorema CAP