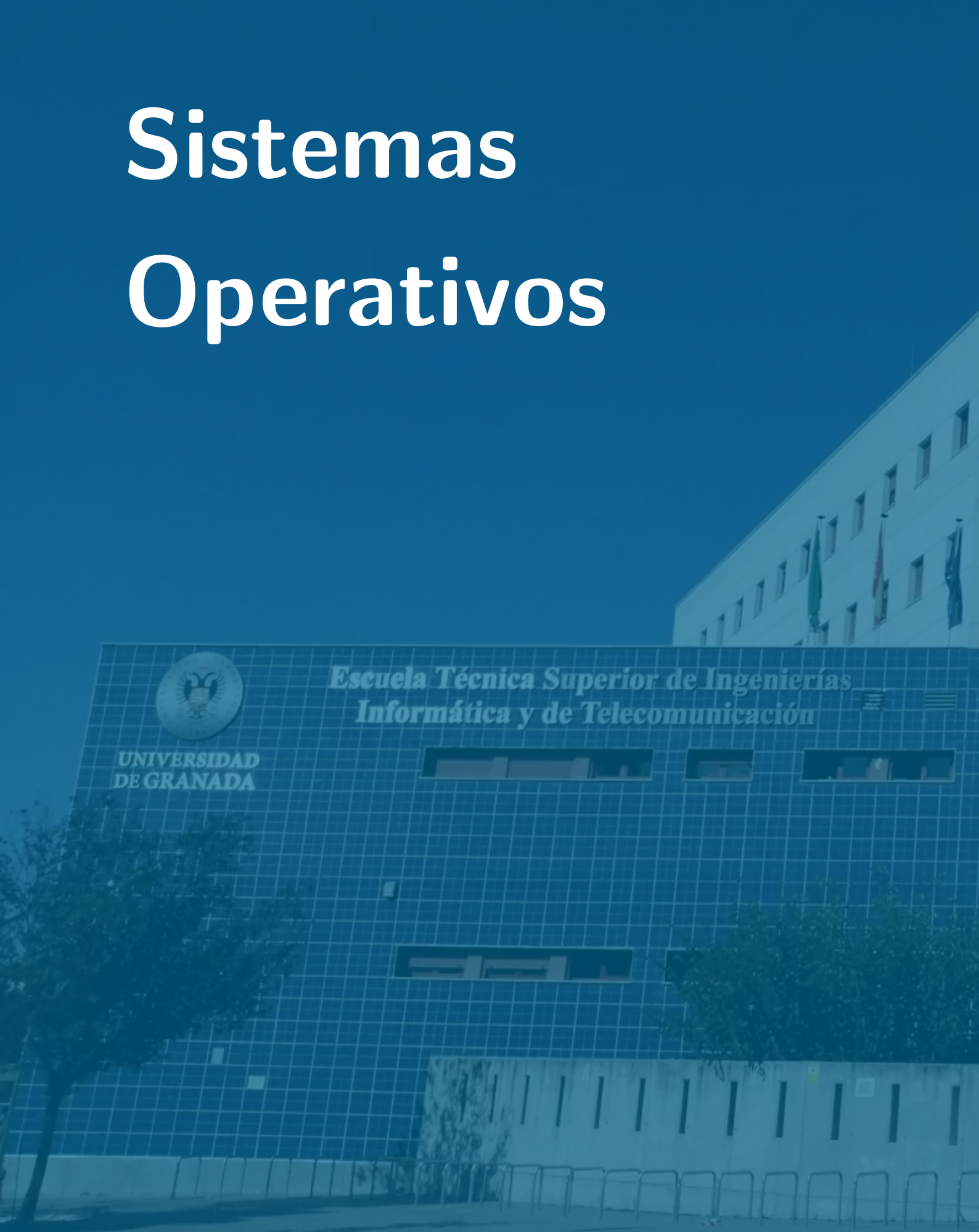
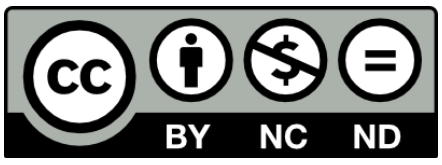
Sistemas Operativos



Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Sistemas Operativos

Los Del DGIIM, `losdeldgiim.github.io`

Arturo Olivares Martos

Granada, 2023-2024

Índice general

1. Relaciones de Problemas

1.1. Estructuras de SO's

Ejercicio 1.1.1. Cuestiones generales relacionadas con un SO:

1. ¿Qué es el núcleo (kernel) de un SO?

   El kernel de un SO es un programa que reside en RAM que contiene las llamadas al sistema (funciones relacionadas directamente con el hardware del ordenador).

   Como son funciones a tan bajo nivel, no se puede acceder a ellas en modo usuario. Cuando se ejecuta una llamada al sistema, se cambia el bit de modo de la PSW a 0, y se entra en modo kernel.

2. ¿Qué es un modelo de memoria (interpretación del espacio de memoria) para un programa? Explique los diferentes modelos de memoria para la arquitectura IA-32.

   El modelo de memoria es la forma que tiene la CPU de interpretar los accesos a memoria. El espacio de direcciones son todas las direcciones disponibles en un ordenador, y va desde 0 hasta $2^n - 1$, siendo n el número de bits del bus de direcciones. Los diferentes modelos de memoria del IA-32, cuyo espacio de direcciones es $\{0, \ldots, 2^{32} - 1\}$, son:

   a) Flat memory model:
   Es un espacio lineal que direcciona todo el espacio de direcciones, desde 0 hasta $2^{32} - 1$. Puede direccionar cada byte, por lo que se dice que lo hace con granularidad de un byte.

   b) Segmented memory model:
   Usa segmentación. Cada dirección lógica consta de un selector de segmento y de un desplazamiento dentro de dicho segmento.

   Consta también de una tabla de segmentos en la que, entre otros aspectos, se encuentra dónde comienza cada segmento.

   Se puede identificar una gran cantidad de segmentos, cada uno con un tamaño límite de 2^{32} bytes (tamaño de la memoria).

   c) Real-adress mode memory model:
   Muy similar al modelo de memoria segmentada, aunque este se mantiene por compatibilidad hacia atrás con otras arquitecturas. En este caso hay limitaciones de tamaño tanto para los segmentos como para la memoria total.

3. ¿Cómo funciona el mecanismo de tratamiento de interrupciones mediante interrupciones vectorizadas? Explique que parte es realizada por el hardware y que parte por el software.

   En primer lugar, el módulo de E/S envía la interrupción a la CPU, que la recibe junto con un `id_disp`, para saber qué dispositivo se queda libre.

   En ese momento, el procesador terminará de ejecutar la instrucción por la que estaba, ya que al final del ciclo de cada instrucción hay una fase en la que el procesador comprueba si tiene instrucciones pendientes.

   En ese momento, el procesador le envía al módulo de E/S una señal avisando de que se ha recibido la señal. Se hace mediante el bus `INTA` (*interruption acknowledgement*).

   Tras enviar dicha señal, se guarda el contexto del proceso que se estaba ejecutando (se apilan la `PSW` y `PC`). Una vez apilada la `PSW`, se cambia el bit de modo a 0, ya que hay que indicar que se entra en el modo kernel. Es importante apilar dicha información, ya que esta será potencialmente modificada en el tratamiento de la interrupción, y será necesario que; al finalizar la `RSI`, el programa continúe por el mismo sitio.

   Una vez realizado el cambio de modo, se establece el valor del contador de programa al inicio de la `RSI`. Para poder encontrar dicho valor del `PC`, es necesario acceder al vector de interrupciones con el `id_disp`.

   Con el nuevo valor del `PC` establecido, se iniciará la rutina de servicio de la interrupción, que es donde entra el software. En primer lugar, la rutina almacenará en pila el resto de información del proceso, como el `PCB`.

   Posteriormente, se tratará la interrupción, que dependerá de la `RSI` correspondiente. Tras finalizar, se desapilará el `PCB`, restaurándolo. Además, también se restaurará la `PSW` y `PC`. Esto restaura el valor de bit de modo, ya que se cambió a 0 tras apilarse.

   Por último, toda interrupción termina con `iret`. Además, al haber restaurado el contador de programa, el programa continuará por la instrucción donde se quedó.

4. Describa detalladamente los pasos que lleva a cabo el SO cuando un programa solicita una llamada al sistema.

   En primer lugar, cuando se hace una llamada al sistema (por ejemplo, `open()`), en el código de dicha función es necesario que se modifiquen los registros con el valor de la llamada al sistema correspondiente (en Linux, se usa el registro `%rax`). Posteriormente, se llamará a `int 0x80h` o `syscall`, produciéndose entonces la llamada al sistema

   En primer lugar, se almacenará el contexto de los registros del proceso (`PSW`, `PC`, etc.) Posteriormente, se cambiará el bit de modo a kernel, ya que el código de las llamadas al sistema es código kernel. Además, en el manejador de llamadas al sistema se obtendrá del vector de llamadas al sistema el `PC` correspondiente a la llamada que hemos hecho, entrando entonces en dicha función (`sys_read`, por ejemplo).

Una vez ha terminado la ejecución de la llamada al sistema, el manejador de llamadas volverá a restaurar el contexto apilado, incluyendo entonces el cambio del bit de modo a usuario.

Ejercicio 1.1.2. Explique tres responsabilidades asignadas al gestor de memoria de un SO y tres asignadas al gestor de procesos.

Al gestor de procesos, se le asignan varias tareas. En primer lugar, es el encargado de la creación del PCB asociado a un programa que va a ejecutarse, y el encargado de eliminarlo una vez el programa termine.

Además, también se encarga de bloquear o reanudar los procesos dependiendo de los eventos que se produzcan.

Por último, es el responsable de proporcionar recursos para que los procesos se ejecuten de forma sincronizada.

Respecto al gestor de memoria, por ejemplo es el encargado de la protección de las zonas de memoria que han de ser protegidas; como la asociada al kernel o las que solo son de escritura o lectura.

Además, también se encarga de asignar o liberar la memoria asociada a un proceso en cualquier nivel de la jerarquía. También mantiene dicha información transparente al programador.

Por último, se encarga de establecer algoritmos que decidan cuándo es necesario expulsar un programa de memoria principal a secundaria o al revés (planificador a medio plazo).

Ejercicio 1.1.3. ¿Cómo gestionaría el sistema operativo la posibilidad de anidamiento de interrupciones?

Esto no supondría problema, ya que gracias a la gestión de la pila se podría llevar a cabo.

En primer lugar, mientras se ejecuta la `RSI` de la interrupción A, se está ejecutando en modo kernel. Al llegar la interrupción B, se lleva a cabo el mismo proceso, tan solo que se apila en la pila del kernel asociada a dicho proceso (la interrupción A se trataba en modo kernel). Al finalizar la `RSI` de la interrupción B, se hará una `iret`, por lo que se desapilará la información que se había guardado en la pila del kernel y se continuará con la `RSI` de la interrupción A.

Ejercicio 1.1.4. Contraste las ventajas e inconvenientes de una arquitectura de SO monolítica frente a una arquitectura microkernel.

En el caso de la arquitectura de SO monolítica, el SO es un único programa, que se ejecuta entero en modo kernel. No se oculta información, por lo que las dependencias entre distintos módulos del SO son complejas y difíciles de entender y modificar para el desarrollador. Además, al ser un único programa cargado en memoria, un fallo en dicho programa provocaría la caída entera del SO. Por último, en el caso de realizar una pequeña actualización en un módulo del kernel, habría que cambiar el programa completo.

Respecto a la arquitectura microkernel, tenemos que el SO se implementa como diversos módulos separados. Una pequeña parte, la esencial, se implementa como

código kernel, por lo que tan solo lo esencial se ejecutará con estos privilegios. El resto, se implementan como módulos independientes con privilegios de usuario.

Además, al haber ocultación de la información, no hay dependencias completas entre los distintos módulos, lo que simplifica en gran medida la programación. También hay que tener en cuenta que, al ser módulos independientes, un fallo en uno de los servicios tan solo provoca error ahí, no en el sistema entero. Igualmente, cuando se realiza una actualización, tan solo afecta al correspondiente módulo, no al sistema operativo entero.

En esta última arquitectura, dos servicios se comunican mediante la parte del sistema en kernel. Cuando una aplicación solicita un servicio al sistema operativo con `send(Server, &d)`, espera a la respuesta con `recieve(Server, &r)`. Al recibir el microkernel la petición, envía el mensaje `m` al servicio correspondiente, quien contestará al microkernel con los resultados necesarios. El microkernel se los devolverá a la aplicación en `r`.

El único inconveniente que tiene la arquitectura microkernel es que para una petición se han de realizar dos cambios de modo, pero esto aporta principalmente fiabilidad y extensibilidad, por lo que suele ser preferible.

Ejercicio 1.1.5. Cuestiones relacionas con virtualización:

1. ¿Qué se entiende actualmente por virtualización mediante hipervisor?

   La virtualización consiste en la abstracción del hardware real de un computador mediante software, de forma que una máquina virtual pueda ejecutarse. El software que permite esta abstracción se denomina hipervisor, y es el encargado de abstraer y, especialmente, gestionar los recursos hardware (CPU, memoria principal, dispositivos, etc.).

   Mediante la virtualización, es posible que en una única máquina real (*host machine*) se ejecuten varias máquinas virtualizadas (*guest machine*). El ratio de consolidación indica el número de máquinas virtuales que puede gestionar un hipervisor.

   Hay dos tipos, como veremos en la siguiente pregunta.

2. ¿Qué clases de hipervisores existen de manera general y que ventajas e inconvenientes plantea una clase con respecto a la otra?

   Hay dos tipos de hipervisores, el tipo 1 y el tipo 2.

   El tipo 1, también conocido como *native* p *bare-metal*, es un hipervisor que se ejecuta directamente sobre el hardware de la *host machine*, sin ninguna capa software entre medias.

   El tipo 2, también conocido como *hosted*, es un hipervisor que se ejecuta sobre el SO convencional de la máquina real. Es decir, es como un programa más que se ejecuta sobre el SO. Un ejemplo de este tipo de hipervisor es *Virtual Box*.

   El tipo 1 tiene como ventaja que dispone de la gestión de todos los recursos para las máquinas virtuales. Además, debido a que hay un SO como capa intermedia en el tipo 2, el tipo 1 es bastante más eficiente. No obstante, tiene como principal desventaja es que en estos casos hay que dedicar la máquina

real íntegra para la virtualización, mientras que en el tipo 2 no es necesario que esté la virtualización constantemente.

Ejercicio 1.1.6. Cuestiones relacionadas con RTOS:

1. ¿Qué característica distingue esencialmente a un proceso de tiempo real de otro que no lo es?

   Un proceso de tiempo real se distingue de un proceso normal en que, no solo han de garantizar la corrección del resultado lógico (de la ejecución del proceso), sino que también han de gestionar el tiempo empleado en ello.

   Hay dos tipos de procesos de tiempo real, los duros y los suaves. En el primer caso, el proceso tiene que cumplirse de forma obligada en determinado intervalo de tiempo, ya que en caso contrario puede desenvolver en un error fatal del sistema. Los suaves; en cambio tienen asociado un plazo límite deseable, pero tiene sentido seguir planificando dicho proceso si el plazo no se cumple.

2. ¿Cuáles son los factores determinantes del tiempo de respuesta en un RTOS, e.d. define determinismo y reactividad?

   El tiempo de respuesta viene determinado en primer lugar por el determinismo, que consiste en el tiempo que tarda el proceso en detectar una interrupción. Por el otro lado, la reactividad se refiere al tiempo que tarda el proceso en ejecutar la `RSI()` correspondiente.

1.2. Procesos y Hebras

Ejercicio 1.2.1. Cuestiones generales sobre procesos y asignación de CPU:

1. ¿Cuáles son los motivos que pueden llevar a la creación de un proceso?

   Hay cuatro motivos principales para la creación de un proceso:

   a) En sistemas por lotes de trabajo:
   Cuando, en estos tipos de sistemas, se recibe otro trabajo a ejecutar, se crea un nuevo proceso.

   b) *Log on* interactivo:
   Cuando el usuario inicia una terminal, el SO crea un nuevo proceso que ejecuta el *shell* correspondiente.

   c) Para proporcionar un servicio:
   El sistema operativo puede crear un proceso para dar servicio a un proceso creado por el usuario. Por ejemplo, el SO puede crear un proceso distinto cuando el proceso del usuario solicita, por ejemplo, imprimir un documento.

   d) Árbol de procesos:
   Un proceso existente puede crear otro proceso, manteniendo una relación de padre-hijo y; por consiguiente, creando un árbol de procesos.

2. ¿Es necesario que lo último que haga todo proceso antes de finalizar sea una llamada al sistema para finalizar de forma explícita, por ejemplo `exit()`?

   Cuando un proceso termina, es necesario que abandone el procesador y que se llame al `context_switch()`. Por tanto, es necesario que el procesador tenga constancia de el proceso ha terminado, por lo que es necesario una llamada al sistema para indicarlo.

   No obstante, la función `exit()` no es la única opción que tenemos. Podemos usar `sys_exit` o, directamente, ajustar los registros necesarios y emplear `syscall`.

3. Cuando un proceso pasa a estado "BLOQUEADO", ¿Quién se encarga de cambiar el valor de su estado en el descriptor de proceso o PCB?

   El `context_switch()`, que es la función que llama cuando se produce un cambio de contexto. Dentro de él, `planif_CPU()` no se encarga de esto; sino que actualizar el estado en el PCB del proceso y guardar el contexto también en el PCB es función del activador o `dispatcher`.

4. ¿Qué debería hacer cualquier planificador a corto plazo cuando es invocado pero no hay ningún proceso en la cola de ejecutables?

   Siempre hay un proceso kernel que se introduce.

5. ¿Qué algoritmos de planificación quedan descartados para ser utilizados en sistemas de tiempo compartido?

Al ser un sistema de tiempo compartido, puede ser usado por más de un usuario a la vez, por lo que no se puede permitir que un proceso acapare la CPU. Por tanto, las no apropiativas, por norma general se podrían descartar.

Por ejemplo, el algoritmo de planificación FCFS queda descartado, ya que en el caso de que un proceso con una gran ráfaga llegue antes que muchos procesos cortos, el procesador quedará acaparado por un gran tiempo; no permitiendo entonces que sea de tiempo compartido.

El algoritmo SJB también queda descartado, ya que en un momento dado puede ser que el único proceso disponible tenga una ráfaga grande, pero que posteriormente entren varios de ráfaga pequeña que no puedan disponer de procesador hasta que termine el proceso largo.

Una buena implementación podría ser mediante RR, ya que así se obliga a que ningún proceso acapare la CPU.

Ejercicio 1.2.2. Cuestiones sobre el modelo de procesos extendido:

1. ¿Qué pasos debe llevar a cabo un SO para poder pasar un proceso de reciente creación de estado "NUEVO" a estado "LISTO"?

   Al estar en estado "nuevo", implica que ya tiene creado el PCB. No obstante, todavía no ha sido cargado en memoria. Para pasar un proceso de "nuevo" a "listo", el cargador debe asignarle espacio en memoria principal y cargarlo.

   En el caso de que no haya suficiente espacio en memoria, el procesador a largo plazo puede decidir no cargarlo en listos y mantenerlo en "nuevo". Otra opción es que el planificador a largo plazo considere que es necesario ejecutarlo, pero que el procesador a medio plazo opte por descartarlo a "listo y suspendido" por no haber espacio suficiente en memoria.

2. ¿Qué pasos debe llevar a cabo un SO para poder pasar un proceso ejecutándose en CPU a estado "FINALIZADO"?

   En primer lugar, tras la llamada al sistema `sys_exit`, se han de liberar todos los recursos que estuviese usando, como los dispositivos de E/S o los descriptores de archivo que estuviese usando.

   Además, enviará la señal `SIGCHLD` al padre, avisándole de que ha terminado su ejecución.

   Por último, en el caso de que este tuviese hijos, el padre ha de asignarles un proceso padre a estos. Por norma general, los cuelga del `init` o proceso líder de la sesión.

   Posteriormente, se llama al `context_switch()` para seleccionar otro proceso. El `dispatcher` cambiará su estado a "finalizado". Esto indicará al SO que puede eliminar dicho proceso.

   Tras esto, el SO eliminará el proceso, liberando la memoria que tenía asignada y borrándolo de la tabla de procesos.

3. Hemos explicado en clase que la función `context_switch()` realiza siempre dos funcionalidades y que además es necesario que el kernel la llame siempre

cuando el proceso en ejecución pasa a estado "FINALIZADO" o "BLOQUEADO". ¿Qué funcionalidades debe realizar y en qué funciones del SO se llama a esta función?

El `context_switch()` simplemente llama en primer a `Planif_CPU()`; y dentro de él, el planificador a corto plazo decidirá qué proceso ejecutar.

En ese momento, el `dispatcher()` arreglará los estados; es decir, modificará los PCB de ambos procesos, pasando el que se estaba ejecutando al nuevo proceso y el que estaba en la cola de listos a "ejecutándose". Además, debe guardar el contexto del proceso que ha dejado de ejecutarse y restaurar el del proceso que pasa a ejecutarse.

Al `context_siwtch()` se le llama cada vez que hay un cambio de contexto, y esto puede ser:

- `sys_exit`.

  Cuando un proceso finaliza, para que el procesador no quede ocioso se llama al `context_switch()`.
- Rutinas de E/S.

  Cuando se produce una rutina de E/S (el proceso envía el `IORB` al módulo de E/S), el proceso entra a "bloqueado" hasta que el procesador no reciba una interrupción. Por tanto, se ha de llamar al `context_switch()`, para elegir un proceso y ejecutarlo.
- Al final de `wait()`:
  - Si el padre ha recibido la señal `SIGCHLD`, implica que el hijo ha terminado y, entonces, no se produce un cambio de contexto y continúa con su ejecución.
  - Si no ha terminado el hijo, se produce un cambio de contexto ya que el padre entra a bloqueados.

Además, en el caso de que el algoritmo de planificación sea apropiativo, tenemos que también se le llama desde las siguientes funciones dependiendo del algoritmo:

- `RSI` de reloj, en el caso de RR.

  Cuando un proceso ha agotado su *Quantum* o rodaja de tiempo, debe producirse un cambio de contexto para ver qué proceso ejecutar.
- En otros algoritmos, cada vez que un proceso pasa a "Listos" se ha de llamar al planificador de la CPU, y en el caso de que decida que se debe realizar una apropiación, se llamará al despachador.

4. Indique el motivo de la aparición de los estados "SUSPENDIDO-BLOQUEADO" y "SUSPENDIDO-LISTO" en el modelo de procesos extendido.

   Puede darse el caso de que todos los procesos no puedan estar cargados a la vez en memoria principal. Uno podría pensar que esto se podría evitar no cargando en memoria principal más procesos cuando esta estuviese prácticamente llena, pero podría darse el caso de que todos los procesos que estuviesen en memoria principal fuesen de tipo E/S (ráfagas cortas, interrumpidos por frecuentes

procesos largos de E/S) y, en cierto momento, todos estuviesen bloqueados. Entonces, el procesador estaría ocioso, pero tampoco podrían ejecutarse más programas ya que estos no cabrían en memoria. (Análogamente, el problema podría ser que todos los procesos fuesen de una ráfaga larga, por lo que el módulo de E/S no estaría haciendo nada, y los procesos de E/S estarían esperando cuando podrían estar bloqueados esperando al evento correspondiente).

Llegados a este punto, aparece la memoria *swap* y el intercambio o *swapping* de procesos entre memoria principal y memoria secundaria. Cuando un proceso no se está ejecutando, parte de su imagen (como su programa o sus datos) pueden ser expulsados de memoria y almacenados en memoria secundaria. El `PCB` nunca podrá ser expulsado (ya que se perdería el control del registro), pero al liberar parte de la memoria principal ya podrían entrar nuevos procesos en memoria. El planificador a largo plazo se tendría que encargar de decidir qué procesos de los "nuevos" cargar en memoria (traer a "listos"), pero sería relevante que fuesen de ráfagas largas y pocas esperas de E/S, para equilibrar la mezcla.

Es por esto que aparecen los dos nuevos estados. Cuando un proceso está bloqueado y el planificador a medio plazo (que es el encargado del *swapping*) considera que la espera puede ser larga, puede decidir expulsarlo a memoria secundaria. De igual forma, si considera que la llegada del evento puede ser inminente, puede traerlo a memoria principal. De igual forma, si un proceso en "listos" va a tardar mucho en ser planificado por el planificador a corto plazo, el planificador a medio plazo puede decidir expulsarlo a memoria secundaria.

Ejercicio 1.2.3. ¿Tiene sentido mantener ordenada por prioridades la cola de procesos bloqueados? Si lo tuviera, ¿en qué casos sería útil hacerlo? Piense en la cola de un planificador de E/S, por ejemplo el de HDD, y en la cola de bloqueados en espera del evento "Fin E/S HDD".

Mantener una única cola para todos los procesos bloqueados no termina de tener sentido, ya que cuando un evento llegase, se tendría que recorrer la cola entera buscando a los procesos esperando por dicho evento.

Por ello, se guarda una cola por cada evento. Además, sí tendría sentido mantener dicha cola ordenada por prioridades, ya que si hay n procesos esperando para usar un dispositivo de E/S (el HDD, por ejemplo), cuando este se libere tan solo podrá usarlo uno, por lo que habrá que planificar cuál es el proceso elegido; y esto se puede hacer mediante planificación mediante prioridades.

Ejercicio 1.2.4. Explique las diferentes formas que tiene el kernel de ejecutarse en relación al contexto de un proceso y al modo de ejecución del procesador.

Respecto al modo de ejecución, el código kernel siempre ha de ejecutarse en modo kernel, ya que ejecuta tareas a bajo nivel que requieren de permisos.

Respecto al contexto, hay dos opciones:

1. Contexto de usuario:

   Un proceso de usuario ha realizado una llamada al sistema o ha producido una excepción. En ese momento, se ejecuta la llamada al sistema o la `RSE()`, ambas parte del código kernel.

Lo mismo ocurre con las interrupciones. Aunque estas sean debidas a otro proceso, en ningún momento se llama al `context_switch()`, ya que el proceso ejecutándose sigue ejecutándose. Por tanto, no se produce ningún cambio de contexto y sigue en contexto usuario.

2. Contexto kernel:

   Los casos restantes en los que se ejecuta código kernel son las tareas del sistema. Estas se ejecutan en procesos independientes a los que ya hay en ejecución, por lo que se considera el contexto kernel.

Ejercicio 1.2.5. Responda a las siguientes cuestiones relacionadas con el concepto de hebra:

1. ¿Qué elementos de información es imprescindible que contenga una estructura de datos que permita gestionar hebras en un kernel de SO? Describa las estructuras `task_t` y la `thread_t`.

   Las estructuras `task_t` y la `thread_t` son el PCB y TCB que emplea linux para gestionar las hebras en los ordenadores multihilo. Describamos dichas estructuras:

   - PCB o `task_t`:
     - PID, que es un identificador único para el proceso.
     - Lista de hebras, donde se identifica también la *Main Thread*.
     - Estado del proceso, dentro del modelo multihilo {N,L,EN_SWAP, EN_RAM}.
     - Zona de memoria del proceso (puntero a la sección de datos y código).
     - Controlador de recursos del sistema.
   - TCB o `thread_t`:
     - TID, que es un identificador único para cada hebra.
     - Estado de la hebra, dentro del modelo de 5 estados {N,F,L,B,E}.
     - Zona de memoria de la hebra (puntero a la pila de la hebra y del kernel).
     - Contexto de los registros, entre los que se encuentra la palabra de estado de la hebra (TSW) o el PC.

     Notemos que para las hebras no se distingue si están en RAM o SWAP, sino que se hace a nivel de proceso. Se podría considerar llevarse alguna hebra a SWAP, pero tan solo podrías llevarte ambas pilas; por lo que no se hace, ya que no es eficiente.

2. En una implementación de hebras con una biblioteca de usuario en la cual cada hebra de usuario tiene una correspondencia N:1 con una hebra kernel, ¿Qué ocurre con la tarea si se realiza una llamada al sistema bloqueante, por ejemplo `read()`?

   Si todas las hebras de usuario se corresponden con una única hebra kernel, en el momento en el que dicha una de las hebras realice una llamada al sistema bloqueante, dicha hebra kernel se bloqueará y, por tanto, la tarea entera estará

bloqueada, ya que no tendrá acceso al procesador. Para evitarlo, se puede usar correspondencia 1:1 o híbrida.

3. ¿Qué ocurriría con la llamada al sistema `read()` con respecto a la tarea de la pregunta anterior si la correspondencia entre hebras usuario y hebras kernel fuese 1:1?

   La correspondiente hebra kernel se bloquearía, pero el resto de hebras no estarían bloqueadas y, por tanto, podrían ser elegidas por el planificador a corto plazo.

Ejercicio 1.2.6. ¿Puede el procesador manejar una interrupción mientras está ejecutando un proceso sin hacer `context_switch()` si la política de planificación que utilizamos es no apropiativa? ¿Y si es apropiativa?

Cuando se maneja una interrupción, aunque se apile la PSW y se cambie el valor del PC, en ningún momento se hace `context_switch()`, ya que el proceso que estaba ejecutándose seguirá ejecutándose al finalizar la RSI y, por tanto, en ningún momento abandona el estado de "Ejecutándose".

Por tanto, ya sea apropiativa o no apropiativa, en ningún momento se hace un `context_switch()`.

Ejercicio 1.2.7. Suponga que es responsable de diseñar e implementar un SO que va a utilizar una política de planificación apropiativa (*preemptive*). Suponiendo que el sistema ya funciona perfectamente con multiprogramación pura y que tenemos implementada la función `Planif_CPU()`, ¿qué otras partes del SO habría que modificar para implementar tal sistema? Escriba el código que habría que incorporar a dichas partes para implementar apropiación *(preemption)*.

Al querer cambiar a una planificación apropiativa, es necesario que cada vez que un proceso cambie su estado a "listo", se invoque al planificador y, en su caso, al `dispatcher`.

Por tanto, cada rutina del kernel que cambiase el estado de cualquier proceso a "listo", tendría que ser modificada y recompilada añadiéndole al final una llamada a `Planif_CPU()` y, en el caso de que esta función determine que el proceso que ha pasado a listo ha de ocupar la CPU, una llamada al `dispatcher()`.

Ejercicio 1.2.8. Para cada una de las siguientes llamadas al sistema explique si su procesamiento por parte del SO requiere la invocación del planificador a corto plazo (`Planif_CPU()`):

1. Crear un proceso, `fork()`.

   Si usamos una planificación apropiativa, sí sería necesario; ya que puede ser que el nuevo proceso creado sea el elegido por el planificador, pasando el que se estaba ejecutando a "listos", y el nuevo proceso a "ejecutándose".

   En el caso de que la planificación sea no apropiativa, no sería necesario llamar al planificador a corto plazo; ya que la creación de un proceso nuevo no provoca que el proceso que estaba ejecutándose deje de hacerlo.

2. Abortar un proceso, es decir, terminarlo forzosamente, `abort()`.

   En este caso, no habría ninguno proceso ejecutándose. Por tanto, para que el procesador no esté ocioso, se llama al planificador a corto plazo para decidir qué proceso de los listos pasará a ejecutarse.

3. Bloquear (suspender) un proceso, `read()` o `wait()`.

   En todos esos casos el proceso que estaba ejecutándose pasa a "bloqueado". Por tanto, no habría ninguno proceso ejecutándose y; análogamente al caso anterior, se llamaría al planificador a corto plazo.

4. Desbloquear (reanudar) un proceso, `RSI` o `exit()` (complementarias a las del caso anterior).

   El caso de `exit()` suponemos que es una errata, ya que no tiene que ver con desbloquear un proceso, sino finalizarlo. Por tanto, ocurriría lo mismo que al abortar un proceso.

   Respecto a desbloquear un proceso, si la política no es apropiativa, entonces no es llamará al planificador porque no se puede cambiar de proceso.

   No obstante, si la política sí es apropiativa, entonces se llamará al planificador para ver si el proceso que ha sido desbloqueado se apropiará del procesador.

5. Modificar la prioridad de un proceso.

   Suponemos que se está usando planificación por prioridades, ya que en caso contrario la pregunta carecería de sentido.

   Si la planificación es apropiativa, sí sería necesario; ya que puede ser que al modificar la prioridad de un proceso listo, este deba pasar a ejecutarse, o al revés.

   En el caso de que la planificación sea no apropiativa, no sería necesario llamar al planificador a corto plazo; ya que no se va a cambiar el proceso que se esté ejecutando en dicho momento.

Ejercicio 1.2.9. En el algoritmo de planificación FCFS, el índice de penalización, $P = \frac{M+r}{r}$, ¿es creciente, decreciente o constante respecto a r (ráfaga de CPU: tiempo de servicio de CPU requerido por un proceso)? Justifique su respuesta.

Al ser FCFS, tenemos que el tiempo de espera M es constante e independiente respecto a r. Por tanto, cuanto menor es la ráfaga, mayor es la penalización. Es decir, es decreciente respecto a r.

Matemáticamente, como M es constante, podemos argumentarlo mediante derivación:

$$\frac{\partial P}{\partial r}(M, r) = \frac{r - (M + r)}{r^2} = -\frac{M}{r^2} < 0$$

Como la primera derivada es negativa y la función es diferenciable, tenemos que P es decreciente respecto a r.

Ejercicio 1.2.10. Sea un sistema multiprogramado que utiliza el algoritmo Por Turnos (Round-Robin, RR). Sea S el tiempo que tarda el despachador en cada cambio de contexto. ¿Cuál debe ser el valor de quantum Q para que el porcentaje de

uso de la CPU por los procesos de usuario sea del 80 %?

Necesitamos $Q = 4 \cdot S$, ya que así por cada cuatro unidades de tiempo que se está ejecutando el proceso, se ejecuta una vez el kernel. Es decir, suponiendo que el proceso se ejecuta todo el tiempo en modo usuario, es necesario que $4/5$ de su tiempo de retorno se ejecuten seguidos sin consumir el "time slice"; por lo que tendría que usarse $Q = 4 \cdot S$.

Otra forma de verlo es, sabiendo que el tiempo de retorno es $T_{ret} = Q + S$ y $Q = 0{,}8T_{ret}$, tenemos que:

$$T_{ret} = 0{,}8T_{ret} + S \Longrightarrow 0{,}2T_{ret} = S \Longrightarrow T_{ret} = 5S \Longrightarrow Q = 4S$$

Ejercicio 1.2.11. Para la siguiente tabla que especifica una determinada configuración de procesos, tiempos de llegada a cola de listos y ráfagas de CPU; responda a las siguientes preguntas y analice los resultados:

Proceso	Tiempo de Llegada	Ráfaga CPU
A	4	1
B	0	5
C	1	4
D	8	3
E	12	2

Tabla 1.1: Configuración de procesos del Ejericio 1.2.11.

Observación. En los diagramas de ocupación de la CPU, las marcas de tiempo deberían ir justo en las líneas verticales, no en las casillas. Por ello, se opta por señalar justo antes de la marca de tiempo i y justo después de la i con i^- e i^+ respectivamente.

1. FCFS. Tiempo medio de respuesta, tiempo medio de espera y penalización.

																		T	**M**	**P**
A						L		L		L	L	L		E				6	5	6
B	E		E		E		E											5	0	1
C			L		L		L		L		E		E		E			8	4	2
D													L		L		E	5	2	5/3
E																	L	3	1	3/2
	0^+					5^-		5^+					10^-		10^+			15^-		15^+		

Tabla 1.2: Diagrama de ocupación de memoria para FCFS.

Las medias aritméticas son:

$$\overline{T} = 5{,}4 \qquad \overline{M} = 2{,}4$$

2. SJF (ráfaga estimada coincide con ráfaga real). Tiempo medio de respuesta, tiempo medio de espera y penalización.

																					T	**M**	**P**
A						L	E														2	1	2
B	E	E	E	E	E																5	0	1
C		L	L	L	L	L	E	E	E	E											9	5	$^9/_4$
D										L	L	E	E	E							5	2	$^5/_3$
E														L	E	E					3	1	$^3/_2$
	0^+				5^-	5^+				10^-	10^+					15^-	15^+						

Tabla 1.3: Diagrama de ocupación de memoria para SJB.

Las medias aritméticas son:

$$\overline{T} = 4{,}8 \qquad \overline{M} = 1{,}8$$

3. SRTF (ráfaga estimada coincide con ráfaga real). Tiempo medio de respuesta, tiempo medio de espera y penalización.

																					T	**M**	**P**
A						L	E														2	1	2
B	E	E	E	E	E																5	0	1
C		L	L	L	L	L	E	E	E	E											9	5	$^9/_4$
D										L	L	E	E	E							5	2	$^5/_3$
E														L	E	E					3	1	$^3/_2$
	0^+				5^-	5^+				10^-	10^+					15^-	15^+						

Tabla 1.4: Diagrama de ocupación de memoria para SRTF.

Las medias aritméticas son:

$$\overline{T} = 4{,}8 \qquad \overline{M} = 1{,}8$$

4. RR ($q = 1$). Tiempo medio de respuesta, tiempo medio de espera y penalización.

																					T	**M**	**P**
A						L	E														2	1	2
B	E	L	E	L	E	L	L	E	L	L	E										11	6	$^{11}/_5$
C		E	L	E	L	L	E	L	E												8	4	2
D										L	E	L	E	L	E						6	3	2
E														E	L	E					3	1	$^3/_2$
	0^+				5^-	5^+				10^-	10^+					15^-	15^+						

Tabla 1.5: Diagrama de ocupación de memoria para RR($q = 1$).

Las medias aritméticas son:

$$\overline{T} = 6 \qquad \overline{M} = 3$$

5. RR ($q = 4$). Tiempo medio de respuesta, tiempo medio de espera y penalización.

																					T	**M**	**P**
A						L	L	L	L	E											5	4	5
B	E	E	E	E	L	L	L	L	L	E											10	5	2
C		L	L	L	E	E	E	E													7	3	7/4
D									L	L	E	E	E								5	2	5/3
E													L	E	E						3	1	3/2
	0^+				5^-	5^+				10^-	10^+				15^-	15^+							

Tabla 1.6: Diagrama de ocupación de memoria para RR($q = 4$).

Las medias aritméticas son:

$$\overline{T} = 6 \qquad \overline{M} = 3$$

Ejercicio 1.2.12. Utilizando los datos de la tabla 1.1, dibuje el diagrama de ocupación de CPU para el caso de un sistema que utiliza un algoritmo de colas múltiples con realimentación con las siguientes colas:

Cola	Prioridad	Quantum
1	1	1
2	2	2
3	3	4

Tenga en cuenta las siguientes suposiciones:

1. Todos los procesos inicialmente entran en la cola de mayor prioridad (menor valor numérico).
2. Cada cola se gestiona mediante la política RR y la política de planificación entre colas es por prioridades no apropiativa.
3. Un proceso en la cola i pasa a la cola $i + 1$ si consume un quantum completo sin bloquearse.
4. Cuando un proceso llega a la cola de menor prioridad, permanece en ella hasta que finaliza.

Planificación entre cola con prioridades apropiativa :

																					T	**M**	**P**
A						E															1	0	1
B	E	L	E	E	L	L	L	E	L	L	L	L	L	L	E						15	10	5
C		E	L	L	L	E	E	L	L	L	L	E									11	7	11/4
D									E	E	E										3	0	1
E													E	E							2	0	1
	0^+				5^-	5^+				10^-	10^+				15^-	15^+							

Tabla 1.7: Diagrama de ocupación de memoria para el Ejercicio 1.2.12 con planificación entre cola con prioridades apropiativa

Notemos que, en naranja, se señala cuando un proceso pasa de la cola 1 a la cola 2; mientras que en rojo se señala cuando pasa de la 2 a la 3.

Planificación entre cola con prioridades no apropiativa :

																		T	**M**	**P**
A					E													1	0	1
B	E	L	E	E	L	L	L	E	E									9	4	$^9/_5$
C		E	L	L	L	E	E	L	L	L		L	L	L	L	E		14	10	$^7/_2$
D									L	E		E	E					4	1	$^4/_3$
E														E	E			2	0	1
	0^+				5^-	5^+				10^-	10^+					15^-	15^+			

Tabla 1.8: Diagrama de ocupación de memoria para el Ejercicio 1.2.12 con planificación entre cola con prioridades no apropiativa

1.3. Gestión de Memoria

Ejercicio 1.3.1. Si un computador no posee hardware de reubicación, e implementa intercambio (*swapping*), entonces el gestor de memoria necesita utilizar un cargador para recalcular las direcciones físicas de un proceso. ¿Sería posible para el sistema de intercambio reubicar los segmentos de datos y pila? Explique cómo funcionaría este sistema, o si es imposible que funcione.

En un sistema de computación que no posee hardware de reubicación y que implementa intercambio (swapping), la gestión de la memoria se vuelve más compleja, especialmente cuando se trata de reubicar segmentos como los de datos y pila de un proceso. Vamos a desglosar cómo funcionaría o no este sistema:

- **Ausencia de Hardware de Reubicación:**

  El hardware de reubicación permite al sistema operativo mover procesos en la memoria física sin cambiar sus direcciones lógicas. Si un sistema no cuenta con este hardware, cada proceso debe ser consciente de su ubicación en la memoria física. Esto significa que las direcciones en el código del programa deben ser absolutas o deben recalcularse cada vez que el proceso se mueve en la memoria.

- **Implementación de Intercambio (Swapping):**

  El intercambio implica mover procesos completos desde y hacia la memoria principal y secundaria (usualmente un disco). Sin hardware de reubicación, las direcciones en el programa deben ser recalculadas cada vez que se mueve el proceso, lo cual es una operación costosa en términos de tiempo de CPU.

- **Reubicación de Segmentos de Datos y Pila:**

  Reubicar los segmentos de datos y pila es técnicamente posible, pero implica un esfuerzo considerable del gestor de memoria y del cargador del sistema operativo. Cada vez que un proceso es intercambiado de vuelta a la memoria principal, el gestor de memoria tendría que usar un cargador para ajustar todas las direcciones en los segmentos de datos y pila del proceso, asegurándose de que apunten a las nuevas ubicaciones físicas.

- **Viabilidad del Sistema:**

  Aunque es posible implementar un sistema de intercambio sin hardware de reubicación, sería ineficiente y lento debido a la necesidad de recalcular constantemente las direcciones físicas. Esto puede ser especialmente problemático en sistemas con alta carga, donde los procesos se intercambian con frecuencia.

En resumen, aunque es posible implementar intercambio sin hardware de reubicación, el costo en términos de rendimiento y complejidad del gestor de memoria sería considerable. En la práctica, este enfoque sería poco práctico para sistemas modernos donde la eficiencia y la velocidad son críticas.

Ejercicio 1.3.2. Considere un sistema con un espacio lógico de memoria de 128K páginas (máximo espacio de memoria virtual) con 8 KB cada una, una memoria física de 64 MB y direccionamiento al nivel de byte. ¿Cuántos bits hay en la dirección lógica? ¿Y en la física?

Calculemos en primer lugar cuántas palabras se pueden direccionar en la memoria física:

$$2^6 \cdot 2^{20} \cdot \frac{1 \text{ palabra}}{1\ B} = 2^{26} \text{ palabras}$$

Por tanto, necesitamos 26 bits para direccionar la memoria física.

Ahora, para calcular el número de bits necesarios para direccionar la memoria virtual. Cada página es de 8 $KB = 2^{13}$, por lo que el desplazamiento ocupa 13 bits. Respecto a la página, como hay 2^{17} páginas, necesitamos 17 bits para direccionarlas, por lo que en total necesitamos 30 bits para direccionar la memoria virtual.

Ejercicio 1.3.3. Considérese un sistema con memoria virtual en el que el procesador tiene una tasa de utilización del 15 % y el dispositivo de paginación está ocupado el 97 % del tiempo, ¿qué indican estas medidas? ¿Y si con el mismo porcentaje de uso del procesador el porcentaje de uso del dispositivo de paginación fuera del 15 %?

Si el dispositivo de paginación está ocupado el 97 % del tiempo, significa que la mayor parte del tiempo se están produciendo faltas de página, por lo que es necesario que el dispositivo de paginación traiga páginas de memoria secundaria a memoria principal. Asimismo, los datos dan a entender que el procesador está la mayor parte del tiempo ocioso, ya que está a la espera de que se traigan las páginas necesarias. Este problema se conoce como *hiperpaginación*.

En el otro caso, tanto el procesador como el dispositivo de paginación están ocupados el 15 % del tiempo. El hecho de que el dispositivo de paginación esté ocupado el 15 % del tiempo significa que la mayoría de referencias a páginas se encuentran en memoria principal, por lo que la gestión de memoria está funcionando correctamente. No obstante, el hecho de que el procesador esté ocupado tan solo el 15 % del tiempo significa que el algoritmo de planificación no está funcionando correctamente, ya que hay que aumentar la multiprogramación para que el procesador esté más tiempo ocupado.

Ejercicio 1.3.4. Sea un sistema de memoria virtual paginada con direcciones lógicas de 32 bits que proporciona un espacio virtual de 2^{20} páginas y con una memoria física de 32 Mbytes ¿cuánta memoria requiere en total un proceso que tenga 453Kbytes, incluida su tabla de páginas cuyas entradas son de 32 bits?

Calculemos el tamaño de página del sistema. Como, de los 32 bits de la dirección lógica, 20 son para la página, tenemos que el tamaño de página es de $2^{12}\ B = 4\ KB$. Veamos ahora cuántas páginas ocupa el proceso:

$$453\ KB \cdot \frac{1 \text{ página}}{4\ KB} = 113{,}25 \text{ páginas}$$

Es decir, necesita 114 páginas. Por tanto, el tamaño de la tabla de páginas es de $114 \cdot 32\ b = 114 \cdot 4\ B = 456\ B$. Por tanto, el tamaño total del proceso es de:

$$456\ B + 114 \text{ páginas} \cdot \frac{2^{12}\ B}{1 \text{ página}} = 456\ B + 114 \cdot 2^{12}\ B = 456\ B + 466944\ B = 467400\ B$$

Ejercicio 1.3.5. Un ordenador tiene 4 marcos de página. En la siguiente tabla se muestran: el tiempo de carga, el tiempo del último acceso y los bits R y M para cada página (los tiempos están en tics de reloj). Responda a las siguientes cuestiones justificando su respuesta.

Página	Tº de carga	Tº última referencia	R (Referencia)	M (Modificación)
0	126	279	1	0
1	230	235	1	0
2	120	272	1	1
3	160	200	1	1

- ¿Qué página se sustituye si se usa el algoritmo FIFO?

  Con el algoritmo FIFO, se sustituye la página que se cargase antes. No obstante, hay dos posibilidades. Si se da prioridad absoluta a que se sustituyan antes las páginas limpias que las sucias, entonces se sustituiría la página 0, aunque la primera en cargarse fue la 2. Si se usa el algoritmo FIFO puro, entonces se sustituiría la página 2 por ser la primera en cargarse.

- ¿Qué página se sustituye si se usa el algoritmo LRU?

  Con el algoritmo LRU, se sustituye la página que lleve más tiempo sin ser referenciada. No obstante, en este caso también hay dos posibilidades. Si se da prioridad absoluta a que se sustituyan antes las páginas limpias que las sucias, entonces se sustituiría la página 1, aunque la 3 es la que lleva más tiempo sin ser referenciada. Si se usa el algoritmo LRU puro, entonces se sustituiría la página 3 por ser la que lleva más tiempo sin ser referenciada.

Ejercicio 1.3.6. ¿Depende el tamaño del conjunto de trabajo de un proceso directamente del tamaño del programa ejecutable asociado a él? Justifique su respuesta.

No, ya que el conjunto de trabajo depende de las páginas que se hayan referenciado en un determinado momento. Si, por ejemplo, en dicho momento se está ejecutando un bucle, puede ser que el conjunto de trabajo sea muy pequeño, porque dicho bucle se encuentre entero en la misma página.

Ejercicio 1.3.7. ¿Por qué una cache (o la TLB) que se accede con direcciones virtuales puede producir incoherencias y requiere que el sistema operativo la invalide en cada cambio de contexto y, en cambio, una que se accede con direcciones físicas no lo requiere?

La TLB es una memoria caché que almacena las traducciones de direcciones virtuales a direcciones físicas. Contiene la información que se ha traído de la tabla de páginas que, como es una estructura de datos que pertenece a cada proceso, cambiará en cada cambio de contexto. Por tanto, al cambiar el proceso que se está ejecutando, la tabla de páginas cambiará, y por tanto la TLB contendrá información que ya está desfasada.

En cambio, las direcciones físicas no cambiarán en un cambio de contexto. Por tanto, la caché que se accede con direcciones físicas no requiere que el sistema operativo la invalide en cada cambio de contexto, pero las direcciones con las que se

accederá serán distintas y, por tanto, generará una gran cantidad de fallos de caché. No obstante, no son inválidos.

Ejercicio 1.3.8. Un ordenador proporciona un espacio de direccionamiento lógico (virtual) a cada proceso de 65536 bytes de espacio dividido en páginas de 4096 bytes. Cierto programa tiene un tamaño de región de texto de 32768 bytes, un tamaño de región de datos de 16386 bytes y tamaño de región de pila de 15878 bytes. ¿Cabría este programa en el espacio de direcciones? (Una página no puede ser utilizada por regiones distintas). Si no es así, ¿cómo podríamos conseguirlo, dentro del esquema de paginación?

Calculamos cuántas páginas ocupa cada región:

- Región de texto: $32768\ B \cdot \frac{1\ \text{página}}{4096\ B} = 8$ páginas
- Región de datos: $16386\ B \cdot \frac{1\ \text{página}}{4096\ B} \approx 4{,}0004$ páginas
- Región de pila: $15878\ B \cdot \frac{1\ \text{página}}{4096\ B} \approx 3{,}8764$ páginas

Por tanto, en total se necesitan $8+5+4 = 17$ páginas. Como cada página ocupa $4096\ B$, el tamaño total del programa es de $17 \cdot 4096\ B = 69632\ B$. Por tanto, vemos claramente que el programa no cabe en el espacio de direcciones.

El espacio que en realidad necesitamos es de $32768\ B + 16386\ B + 15878\ B = 65032\ B$, por lo que defectivamente cabe en el espacio de direcciones. El problema con el que nos hemos encontrado es el de la fragmentación interna. Para solucionarlo, la mejor solución es disminuir el tamaño de página, de forma que la fragmentación interna sea menor.

Ejercicio 1.3.9. Analice qué puede ocurrir en un sistema que usa paginación por demanda si se recompila un programa mientras este está ejecutando. Proponga soluciones a los problemas que pueden surgir en esta situación.

El comportamiento que se produce es que aumentarán los fallos de página, ya que las páginas que se habían cargado en memoria principal ya no son válidas, por lo que se producirán fallos de página cada vez que se intente acceder a ellas. Además, el resultado no será el deseado.

Para solucionar este problema, se debe esperar a que el proceso termine de ejecutarse para recompilarlo, o bien forzar que termine de ejecutarse antes de recompilarlo.

Ejercicio 1.3.10. Para cada uno de los siguientes campos de la tabla de páginas, se debe explicar si es la MMU o el sistema operativo quién los lee y escribe (en este último caso si se activa o desactiva), y en qué momentos:

1. Número de marco

   Lo escribe el SO cuando se carga la página en memoria principal, y lo lee la MMU cuando se produce una referencia a dicha página para poder traducir la dirección virtual a dirección física.

2. Bit de presencia

   Lo escribe el SO cuando se carga la página en memoria principal y al retirarla de la memoria principal, y lo lee la MMU cuando se produce una referencia a dicha página para poder verificar si está en memoria principal o no.

3. Bit de protección

   Lo escribe el SO cuando se carga la página en memoria principal, y lo lee la MMU cuando se produce una referencia a dicha página para poder verificar si se puede acceder a dicha página o no.

4. Bit de modificación

   Lo escribe (activa) la MMU cuando se produce una operación de escritura en dicha página, y lo escribe también (desactiva) el SO cuando se carga una página en memoria principal, y lo lee el SO cuando va a sustituir una página para saber si es necesario escribirla en memoria secundaria o no.

5. Bit de referencia

   Lo escribe el MMU cuando se produce una referencia a dicha página (activándolo) y el SO cuando se carga la página en memoria principal (activándolo). Además, para desactivarlo, lo hace el SO cuando, en el algoritmo de sustitución de Reloj, está buscando qué página sustituir y se encuentra con que el bit de referencia está activado. También lo lee el SO en dicho proceso.

Ejercicio 1.3.11. Suponga que la tabla de páginas para el proceso actual se parece a la de la figura. Todos los números son decimales, la numeración comienza en todos los casos desde cero, y todas las direcciones de memoria son direcciones en bytes. El tamaño de página es de 1024 bytes.

Nº Página	Bit de validez	Bit de referencia	Bit de modificación	Nº Marco
0	0	1	0	4
1	1	1	0	7
2	1	0	1	1
3	0	0	0	-
4	1	0	0	2
5	1	1	1	0

¿Qué direcciones físicas, si existen, corresponderán con cada una de las siguientes direcciones virtuales? (no intente manejar ninguna falta de página, si las hubiese)

1. 999

   En primer, buscamos en qué pagina se encuentra la dirección virtual. Como $999 = 0 \cdot 1024 + 999$, se encuentra en la página 0. Ahora, como el bit de validez es 0, no se encuentra en memoria, por lo que no existe dirección física para ella.

2. 2121

   En primer, buscamos en qué pagina se encuentra la dirección virtual. Como $2121 = 2 \cdot 1024 + 73$, se encuentra en la página 2. En este caso si tiene el bit de validez activado (1), y su marco es el 1. Por tanto, la dirección física es $1 \cdot 1024 + 73 = 1097$.

3. 5400

   En primer, buscamos en qué pagina se encuentra la dirección virtual. Como $5400 = 5 \cdot 1024 + 280$, se encuentra en la página 5. En este caso si tiene el bit de validez activado (1), y su marco es el 0. Por tanto, la dirección física es $0 \cdot 1024 + 280 = 280$.

Ejercicio 1.3.12. Sea la siguiente secuencia de números de página referenciados: 1,2,3,4,1,2,5,1,2,3,4,5. Calcule el número de faltas de página que se producen utilizando el algoritmo FIFO y considerando que el número de marcos de página de que disfruta nuestro proceso es de:

1. 3 marcos

Pag. Referenciada	1	2	3	4	1	2	5	1	2	3	4	5
Está 1	1	1	1	-	1	1	1	1	1	-	-	-
Está 2	-	2	2	2	-	2	2	2	2	2	-	-
Está 3	-	-	3	3	3	-	-	-	-	3	3	3
Está 4	-	-	-	4	4	4	-	-	-	-	4	4
Está 5	-	-	-	-	-	-	5	5	5	5	5	5
Falta	*	*	*	*	*	*	*			*	*	

Tabla 1.9: Algoritmo FIFO con 3 marcos

   En este caso, el número de faltas de página es 9.

2. 4 marcos

Pag. Referenciada	1	2	3	4	1	2	5	1	2	3	4	5
Está 1	1	1	1	1	1	1	-	1	1	1	1	-
Está 2	-	2	2	2	2	2	2	-	2	2	2	2
Está 3	-	-	3	3	3	3	3	3	-	3	3	3
Está 4	-	-	-	4	4	4	4	4	4	-	4	4
Está 5	-	-	-	-	-	-	5	5	5	5	-	5
Falta	*	*	*	*			*	*	*	*	*	*

Tabla 1.10: Algoritmo FIFO con 4 marcos

   En este caso, el número de faltas de página es 10.

¿Se corresponde esto con el comportamiento intuitivo de que disminuirá el número de faltas de página al aumentar el tamaño de memoria de que disfruta el proceso?

Claramente no, ya que en el segundo caso tenemos más faltas de página que en el primero aun habiendo aumentado el número de marcos. Esto se debe a que dicha regla no es cierta siempre, sino que depende de la secuencia de páginas referenciadas. Por norma general, sí que es cierta.

Ejercicio 1.3.13. ¿Por qué la localidad no es un factor que se tiene en cuenta en los sistemas con segmentación?

La localidad no es un factor que se tiene en cuenta en los sistemas con segmentación porque, en estos sistemas, el tamaño de los segmentos es variable. Por tanto, se supone que el tamaño se ha escogido de forma que se aproveche la localidad. Es decir, la localidad no se considera porque se presupone intrínseca en el tamaño de los segmentos.

Como se habrá traído a memoria principal el segmento completo, se aprovechará la localidad en el interior del segmento, sin necesidad de traernos los segmentos contiguos.

Ejercicio 1.3.14. En la gestión de memoria en un sistema paginado, ¿qué estructura/s de datos necesitará mantener el Sistema Operativo para administrar el espacio libre?

En paginación, se usan tres estructuras de datos:

- Tabla de páginas
- Tabla de marcos de página
- Tabla de ubicación en disco

La primera estructura de datos es la que contiene la información para realizar la traducción de direcciones virtuales a direcciones físicas. La segunda es la que contiene la información sobre cada marco de página, y en concreto se puede ver si está libre o no. Por tanto, es esta la estructura de datos que nos interesa para administrar el espacio libre. Por último, la tercera es la que contiene la información sobre la ubicación de cada página en memoria secundaria.

Por tanto, la estructura de datos que necesitará el sistema operativo para administrar el espacio libre es la tabla de marcos de página, ya que con esa tabla sabremos qué marcos de página están libres y cuáles no.

Ejercicio 1.3.15. ¿Cuánto puede avanzar como máximo la aguja del algoritmo de reemplazo de páginas del reloj durante la selección de una página?

Como máximo, puede avanzar hasta la página que se está referenciando en ese momento, es decir, tantos pasos como marcos de página haya. Suponiendo que todos los marcos tienen el bit de referencia $R = 1$ activado, al avanzar la aguja se irán desactivando los bits de referencia de los marcos que se vayan encontrando, hasta que se encuentre el primero, que ya lo había desactivado, por lo que se detendrá y esa será la página seleccionada para ser sustituida.

Ejercicio 1.3.16. Situándonos en un sistema paginado, donde cada proceso tiene asignado un número fijo de marcos de páginas. Supongamos la siguiente situación: existe un proceso con 7 páginas y tiene asignados 5 marcos de página. Indica el contenido de la memoria después de cada referencia a una página si como algoritmo de sustitución de página utilizamos el LRU (la página no referenciada hace más tiempo). La secuencia de referencias es la indicada en la Tabla 1.11.

Referencias	2	1	3	4	1	5	6	4	5	7	4	2
Marcos de página	2	2	2	2	2	2	6	6	6	6	6	6
		1	1	1	1	1	1	1	1	1	1	2
			3	3	3	3	3	3	3	7	7	7
				4	4	4	4	4	4	4	4	4
						5	5	5	5	5	5	5
Falta de Página	*	*	*	*		*	*			*		*

Tabla 1.11: Ejercicio 1.3.16

Se producen 8 faltas de página.

Ejercicio 1.3.17. ¿Cuál es la ventaja del algoritmo de frecuencia de faltas de página (FFP) sobre el algoritmo basado en el modelo del conjunto de trabajo utilizando el tamaño de ventana w? ¿Cuál es la desventaja?

La ventaja del algoritmo de faltas de página es que, en el caso de tener muchos marcos de página que lleven tiempo sin referenciarse, el algoritmo liberará bastantes marcos en la siguiente falta de página; mientras que si tenemos muchas faltas de página (será por tener pocos marcos de página ocupados), el algoritmo incrementará el número de marcos de página empleados, logrando siempre un equilibrio. Además, es un modelo bastante más sencillo y menos costoso, ya que tan solo se llama cuando se produce una falta de página, mientras que en el modelo del conjunto de trabajo se llama cada vez que se referencia una página.

La desventaja es que en ciertos momentos, tenemos cargadas en memoria páginas que no se usan, cosa que evitamos con el algoritmo basado en los conjuntos de trabajo.

Ejercicio 1.3.18. Supongamos que tenemos un proceso ejecutándose en un sistema paginado, con gestión de memoria basada en el algoritmo de sustitución frecuencia de faltas de página. El proceso tiene 5 páginas (0, 1, 2, 3, 4). Represente el contenido de la memoria real para ese proceso (es decir, indique que páginas tiene cargadas en cada momento) y cuándo se produce una falta de página. Suponga que, inicialmente, está cargada la página 2, el resto de páginas están en memoria secundaria y que no hay restricciones en cuanto al número de marcos de página disponibles. La cadena de referencias a página es: 0 3 1 1 1 3 4 4 2 2 4 0 0 0 0 3 y el parámetro es $\tau = 3$.

Pag. Referenciada	0	3	1	1	1	3	4	4	2	2	4	0	0	0	0	3
Está 0	0	0	0	0	0	0	-	-	-	-	-	0	0	0	0	0
Está 1	-	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Está 2	2	2	2	2	2	2	-	-	2	2	2	2	2	2	2	2
Está 3	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
Está 4	-	-	-	-	-	-	4	4	4	4	4	4	4	4	4	4
Falta de Página	*	*	*				*		*			*				

Tabla 1.12: Ejercicio 1.3.18

Notemos que en (*) no se ha eliminado ninguna página, ya que la anterior falta de página había ocurrido en $t - 2$, y la ventana es de $\tau = 3$. Además, es interesante

notar que, como no hay restricciones en cuanto al número de marcos de página disponibles, una vez todas las páginas tengan su marco de página correspondiente, no se producirán más faltas de página. Esto es lo que ocurre en la última parte de este ejercicio.

Ejercicio 1.3.19. Describa el funcionamiento del algoritmo de sustitución basado en la frecuencia de faltas de página, con los siguientes datos: 4 marcos de página, en $t = 0$ la memoria contiene a la página 2. El tamaño de la ventana es $\tau = 3$ y se produce la secuencia de referencias de páginas, 1 4 2 2 2 4 5 5 3 3 5 1 1 1 1 4

Pag. Referenciada	1	4	2	2	2	4	5	5	3	3	5	1	1	1	1	4
Marco 0	1	1	1	1	1	1	5	5	5	5	5	!				
Marco 1	2	2	2	2	2	2	2	2	2	2	2	!				
Marco 2	-	4	4	4	4	4	4	4	4	4	4	!				
Marco 3	-	-	-	-	-	-	-	-	3	3	3	!				
Falta de Página	*						*		*			*				

Tabla 1.13: Ejercicio 1.3.19

Notemos que en (*) no se ha eliminado ninguna página, ya que no había transcurrido el tiempo suficiente para que se cumpliera la condición de la ventana $\tau = 3$. En el último caso, cabe destacar que no hay marco de página disponible para la página 1, por lo que se produce una falta de página y el proceso se bloquea.

Ejercicio 1.3.20. Describa el funcionamiento del algoritmo de sustitución global basado en el algoritmo basado en el modelo del conjunto de trabajo, con los siguientes datos: 4 marcos de página, en $t = 0$ la memoria contiene a la página 2 que se referenció en dicho instante de tiempo. El tamaño de la ventana es $\tau = 3$ y se produce la secuencia de referencias de páginas, 1 4 4 4 2 4 1 1 3 3 5 5 5 5 1 4

Pag. Referenciada	1	4	4	4	2	4	1	1	3	3	5	5	5	5	1	4
Marco 0	1	1	1	-	2	2	2	-	3	3	3	3	-	-	1	1
Marco 1	2	2	-	-	-	-	1	1	1	1	-	-	-	-	-	4
Marco 2	-	4	4	4	4	4	4	4	-	-	5	5	5	5	5	5
Marco 3																
Falta de Página	*	*			*		*		*		*				*	*

Tabla 1.14: Ejercicio 1.3.20

Ejercicio 1.3.21. Una computadora con memoria virtual paginada tiene un bit U por página virtual, que se pone automáticamente a 1 cuando se realiza un acceso a la página. Existe una instrucción `limpiar_U (dir_base_tabla)` que permite poner a 0 el conjunto de los bits U de todas las páginas de la tabla de páginas cuya dirección de comienzo pasamos como argumento. Explica cómo puede utilizarse este mecanismo para la implementación de un algoritmo de sustitución basado en el modelo del conjunto de trabajo.

Ejercicio 1.3.22. Un Sistema Operativo con memoria virtual paginada tiene el mecanismo `fijar_página(np)` cuyo efecto es proteger contra la sustitución al marco

de página en que se ubica la página virtual `np`. El mecanismo `des_fijar (np)` suprime esta protección.

1. ¿Qué estructura/s de datos son necesarias para la realización de estos mecanismos?

2. ¿En qué caso puede ser de utilidad estas primitivas?

3. ¿Qué riesgos presentan y qué restricciones deben aportarse a su empleo?

Ejercicio 1.3.23. Disponemos de un ordenador que cuenta con las siguientes características: tiene una memoria RAM de 4KBytes, permite usar memoria virtual paginada, las páginas son de 1KBytes de tamaño y las direcciones virtuales son de 16 bits. El primer marco de página (marco 0) se usa únicamente por el Kernel y los demás marcos están disponibles para su uso por los procesos que se ejecutan en el sistema. Supongamos que tenemos sólo dos procesos, P1 y P2, y que utilizan las siguientes direcciones de memoria virtual y en el siguiente orden:

Proceso	Direcciones Virtuales
P1	0-99
P2	0-500
P1	100-500
P2	501-1500
P1	3500-3700
P2	1501-2100
P1	501-600

Tabla 1.15: Direcciones virtuales utilizadas por los procesos P1 y P2

1. ¿Cuántos marcos de página tiene la memoria RAM de este ordenador?

   Tenemos que:

   $$4 \text{ KBytes} \cdot \frac{1 \text{ marco de página}}{1 \text{ KBytes}} = 4 \text{ marcos de página}$$

2. ¿Cuántos bits necesitamos para identificar los marcos de página?

   Necesitamos 2 bits para identificar los marcos de página, ya que $2^2 = 4$.

3. Describe los fallos de página que tendrán lugar para cada intervalo de ejecución de los procesos, si la política de sustitución de páginas utilizada es LRU. Suponga que se dicho algoritmo es de asignación variable y sustitución global.

Intervalo	Proceso	Direcciones Virtuales	Páginas
I_1	P1	0-99	P1, Página 0
I_2	P2	0-500	P2, Página 0
I_3	P1	100-500	P1, Página 0
I_4	P2	501-1500	P2, Páginas 0 y 1
I_5	P1	3500-3700	P1, Página 3
I_6	P2	1501-2100	P2 Página 1 y 2
I_7	P1	501-600	P1, Página 0

Tabla 1.16: Páginas asociadas a cada proceso

Notaremos la página i del proceso j como P_{ij}. Como el marco 0 está reservado para el kernel, solo contamos con 3 marcos:

Intervalo	I_1	I_2	I_3	I_4	I_5	I_6	I_7
Referencias	P_{01}	P_{02}	P_{01}	$P_{02},\ P_{12}$	P_{31}	$P_{12},\ P_{22}$	P_{01}
Marco 1	P_{01}	P_{01}	P_{01}	P_{01}	P_{31}	P_{31}	P_{01}
Marco 2		P_{02}	P_{02}	P_{02}	P_{02}	P_{22}	P_{22}
Marco 3				P_{12}	P_{12}	P_{12}	P_{12}
Falta de Página	$*$	$*$		$*$	$*$	$*$	$*$

Tabla 1.17: Ejercicio 1.3.23

Ejercicio 1.3.24. Se tiene un sistema de memoria virtual con paginación a dos niveles que permite agrupar las páginas en "directorios de páginas". Cada tabla de páginas puede contener hasta 1024 páginas. Los espacios de direcciones lógicas de este sistema son de 4 GiB y el tamaño de página es de 4 KiB. El espacio de direcciones físicas puede tener hasta 1 GiB. Describa la estructura de las direcciones lógicas y de las direcciones físicas de este sistema de memoria virtual.

En este ejercicio nos dan información de más, así que intentaremos dar distintas formas de calcular lo pedido. En primer lugar, como el espacio de direcciones lógicas es de 4 GiB = 2^{32} B, necesitamos 32 bits para representar las direcciones lógicas.

Como cada tabla de páginas puede contener hasta $1024 = 2^{10}$ páginas, necesitamos 10 bits para representar el directorio de páginas (primer nivel) y 10 bits para representar la tabla de páginas (segundo nivel). Por tanto, nos quedan $32-10-10 = 12$ bits para el desplazamiento. Es decir:

- 10 bits para el directorio de páginas
- 10 bits para la tabla de páginas
- 12 bits para el desplazamiento

De forma análoga, el desplazamiento se podría haber obtenido del tamaño de página, ya que 4 KiB = 2^{12} B, por lo que necesitamos 12 bits para representar el desplazamiento. Vemos que, efectivamente, coincide.

Respecto a las direcciones físicas, como el espacio de direcciones físicas es de 1 GiB = 2^{30} B, necesitamos 30 bits para representar las direcciones físicas. Como el tamaño de marco de página y de página es el mismo, el desplazamiento será el mismo que en el caso de las direcciones lógicas, es decir, 12 bits. Por tanto, nos quedan $30 - 12 = 18$ bits para el marco de página. Es decir:

- 18 bits para el marco de página
- 12 bits para el desplazamiento

Ejercicio 1.3.25. Suponga un sistema que utiliza paginación a dos niveles. Las direcciones son de 8 bits con la siguiente estructura: 2 bits en la tabla de páginas de primer nivel, 2 bits en la tabla de páginas de segundo nivel y 4 bits para el desplazamiento. El espacio de direccionamiento virtual de un proceso tiene la estructura de la Figura 1.1. Represente gráficamente las tablas de páginas y sus contenidos, suponiendo que cada entrada de la tabla de páginas ocupa 8 bits y que todas las páginas están cargadas en memoria principal (elige tú mismo la ubicación en memoria principal de dichas páginas, suponiendo que la memoria principal es de 160 Bytes). Dada esa asignación traduce la dirección virtual 47.

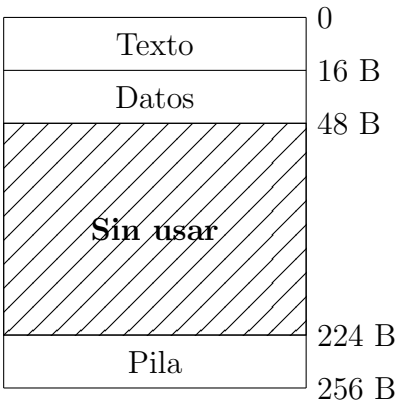


Figura 1.1: Espacio de direccionamiento virtual del ejercicio 1.3.25

Como hay 4 bits para el desplazamiento, tenemos que el tamaño de página es de $2^4 = 16$ Bytes. Por tanto, la sección de texto ocupa 1 página, y la sección de datos y la pila ocupan 2 páginas cada una.

Ejercicio 1.3.26. Considere la siguiente tabla de segmentos:

Segmento	Presencia o validez	Dirección base	Longitud
0	0	219	600
1	1	2300	14
2	1	90	100
3	0	1327	580
4	1	1952	96

Tabla 1.18: Tabla de segmentos

¿Qué direcciones físicas corresponden a las direcciones lógicas (nº segmento, desplazamiento) siguientes? Si no puede traducir alguna dirección lógica a física, explique el por qué.

1. 0, 430

   Corresponde al segmento 0, pero como su bit de validez es 0, no se encuentra en memoria, por lo que no tiene dirección física. Se produce una falta de segmento.

2. 1, 10

   La dirección base es 2300, y como $10 < 14$, la dirección física es $2300 + 10 = 2310$.

3. 3, 400

   Corresponde al segmento 3, pero como su bit de validez es 0, no se encuentra en memoria, por lo que no tiene dirección física. Se produce una falta de segmento.

4. 4, 112

   La dirección base es 1952, y como $112 > 96$, se produce una excepción de acceso indebido.

Ejercicio 1.3.27. Respecto a la gestión de memoria que se hace en Linux: suponga que un proceso realiza una llamada al sistema fork creando un proceso hijo. Represente gráficamente como quedan las estructuras de datos relacionadas con ambos procesos.

Ejercicio 1.3.28. ¿Qué información comparten un proceso y su hijo en un sistema Linux después de ejecutar el siguiente código? Justifique su respuesta e indique qué hace este trozo de código.

```
if ( fork() != 0 )
    wait (&status);
else
    exec (B);
// usamos una llamada al sistema exec genérica con un único
// argumento, el nombre de un archivo ejecutable
```

1.4. Sistema de Archivos

Ejercicio 1.4.1. Sea un Sistema Operativo que solo soporta un directorio (es decir, todos los archivos existentes estarán al mismo nivel), pero permite que los nombres de archivos sean de longitud variable. Apoyándonos únicamente en los servicios proporcionados por este Sistema Operativo, deseamos construir una "utilidad" que "simule" un sistema jerárquico de archivos. ¿Es esto posible? ¿Cómo?

Ejercicio 1.4.2. En un entorno multiusuario, cada usuario tiene un directorio inicial al entrar en el sistema a partir del cual puede crear archivos y subdirectorios. Surge, entonces, la necesidad de limitar el tamaño de este directorio para impedir que el usuario consuma un espacio de disco excesivo. ¿De qué forma el Sistema Operativo podría implementar la limitación de tamaño de un directorio?

Ejercicio 1.4.3. En la siguiente figura se representa una tabla FAT. Al borde de sus entradas se ha escrito, como ayuda de referencia, el número correspondiente al bloque en cuestión. También se ha representado la entrada de cierto directorio. Como simplificación del ejemplo, suponemos que en cada entrada del directorio se almacena: Nombre de archivo/directorio, el tipo (F=archivo, D=directorio), la fecha de creación y el número del bloque inicial.

Nombre	Tipo	Fecha	Nº Bloque
DATOS	F	8-2-90	3

(a) Directorio

1		10	
2		11	
3	15	12	
4		13	
5		14	
6		15	*
7		16	
8		17	
9		18	

(b) FAT

Tenga en cuenta que:

- El tamaño de bloque es de 512 bytes.
- El asterisco indica último bloque.
- Todo lo que está en blanco en la figura está libre.

Rellene la figura para representar lo siguiente:

1. Creación del archivo DATOS1 con fecha 1-3-90, y tamaño de 10 bytes.
2. Creación del archivo DATOS2 con fecha 2-3-90, y tamaño 1200 bytes.
3. El archivo DATOS aumenta de tamaño, necesitando 2 bloques más.
4. Creación del directorio D, con fecha 3-3-90, y tamaño 1 bloque.
5. Creación del archivo CARTAS con fecha 13-3-90 y tamaño 2 KBytes.

Nombre	Tipo	Fecha	Nº Bloque
DATOS	F	8-2-90	3
DATOS1	F	1-3-90	1
DATOS2	F	2-3-90	2
D	D	3-3-90	8
CARTAS	F	13-3-90	9

(a) Directorio

1	*	10	11
2	4	11	12
3	15	12	*
4	5	13	
5	*	14	
6	7	15	6
7	*	16	
8	*	17	
9	10	18	

(b) FAT

Ejercicio 1.4.4. Si usamos un Mapa de Bits para la gestión del espacio libre, especifique la sucesión de bits que contendría respecto a los 18 bloques del ejercicio anterior.

1	1	1	1	1	1	1	1	1
1	1	1	0	0	1	0	0	0

Tabla 1.19: Mapa de bits de los bloques del Ejercicio 1.4.3.

Ejercicio 1.4.5. Si se pierde el primer puntero de la lista de espacio libre, ¿podría el Sistema Operativo reconstruirla? ¿Cómo?

Ejercicio 1.4.6. El espacio libre en un disco puede ser implementado usando una lista encadenada con agrupación o un mapa de bits. La dirección en disco requiere D bits. Sea un disco con B bloques, en que F están libres. ¿En qué condición la lista usa menos espacio que el mapa de bits?

El mapa de bits ocupa B bits, ya que contiene un bit por cada bloque. La lista encadenada tan solo necesita almacenar un puntero, D bits, ya que tan solo se almacena la dirección del primer bloque libre; el resto de bloques libres se obtienen a partir de este. Por tanto, la lista encadenada ocupa menos espacio que el mapa de bits cuando $D < B$.

No obstante, hay que tener en cuenta que el acceso a la lista encadenada es más lento que el acceso al mapa de bits, ya que para acceder al mapa de bits no es necesario acceder a memoria.

Ejercicio 1.4.7. Entre los posibles atributos de un archivo, existe un bit que marca un archivo como temporal y por lo tanto está sujeto a destrucción automática cuando el proceso acaba ¿Cuál es la razón de esto? Después de todo un proceso siempre puede destruir sus archivos, si así lo decide.

Ejercicio 1.4.8. Algunos SO proporcionan una llamada al sistema (`RENAME`) para dar un nombre nuevo a un archivo existente. ¿Existe alguna diferencia entre utilizar este mandato para renombrar un archivo y copiar el archivo a uno nuevo, con el nuevo nombre y destruyendo el antiguo?

Sí, hay una gran diferencia. Al renombrar el archivo, tan solo se cambia el nombre del archivo en la entrada del directorio, pero el puntero a su estructura de datos (i-nodo en el caso de Linux) no cambia. Por tanto, la estructura de datos que sostiene el archivo no cambia, por lo que en esencia el archivo sigue siendo el mismo. En cambio, si se copia el archivo a uno nuevo, con el nuevo nombre y destruyendo el antiguo, se crea una nueva entrada en el directorio, se crea un nuevo i-nodo, y se copia el contenido del i-nodo antiguo al nuevo. Por tanto, el archivo antiguo y el nuevo son dos archivos distintos y es mucho más costoso, ya que se copia todo el contenido del archivo.

Ejercicio 1.4.9 (Ejercicio de Evaluación Continua DGIIM 2023-2024)**.** Un i-nodo de UNIX tiene 10 direcciones de disco para los diez primeros bloques de datos, y tres direcciones más para realizar una indexación a uno, dos y tres niveles respectivamente. Si cada bloque índice tiene 256 direcciones de bloques de disco, ¿cuál es el tamaño del mayor archivo que puede ser manejado, suponiendo que 1 bloque de disco es de 1KByte?

En total el número de bloques de disco que puede manejar el sistema de archivos es:

$$10 + 256 + 256^2 + 256^3 = 10 + 2^8 + 2^{16} + 2^{24} = 16843018$$

Por tanto, el tamaño máximo de archivo que puede manejar el sistema de archivos es:

$$16843018 \text{ bloques} \cdot \frac{1 \text{ KiB}}{1 \text{ bloque}} = 16843018 \text{ KiB} = 16{,}062 \text{ GiB}$$

Ejercicio 1.4.10. Sobre conversión de direcciones lógicas dentro de un archivo a direcciones físicas de disco. Estamos utilizando la estrategia de indexación a tres niveles para asignar espacio en disco. Tenemos que el tamaño de bloque de índices es igual a 512 bytes, y el tamaño de puntero es de 4 bytes. Se recibe la solicitud por parte de un proceso de usuario de leer el carácter número N de determinado archivo. Suponemos que ya hemos leído la entrada del directorio asociada a este archivo, es decir, tenemos en memoria los datos PRIMER-BLOQUE y TAMAÑO.

Calcule la sucesión de direcciones de bloque que se leen hasta llegar al bloque de datos que posee el citado carácter.

Ejercicio 1.4.11. ¿Qué método de asignación de espacio en un sistema de archivos elegiría para maximizar la eficiencia en términos de velocidad de acceso, uso del espacio de almacenamiento y facilidad de modificación (añadir/borrar /modificar), cuando los datos son:

1. Modificados infrecuentemente, y accedidos frecuentemente de forma aleatoria.

   En este caso al ser accedidos de forma aleatoria, descartamos el no contiguo enlazado, ya que este es más eficiente cuando los datos son accedidos de forma secuencial. Por tanto, nos quedamos con el contiguo y el no contiguo indexado.

   Entre estos, es cierto que el contiguo puede producir fragmentación externa y los archivos no pueden crecer, pero como se modifican infrecuentemente, se supone que no habrá problemas, ya que el tamaño no variará prácticamente. Por tanto, se opta por el contiguo, ya que es más eficiente en términos de velocidad de acceso.

2. Modificados con frecuencia, y accedidos en su totalidad con cierta frecuencia.

   En este caso, como es modificado con frecuencia, descartamos el contiguo, ya que este no permite crecer el archivo y, al ser modificado con frecuencia, se supone que crecerá. Así, al evitar este método, se evita la fragmentación externa también.

   Entre los otros dos, depende de si los accesos se realizarán de forma secuencial o aleatoria o directa. Como se menciona que se accede en su totalidad, se supone que se accede de forma secuencial, por lo que se opta por el no contiguo enlazado, ya que requerirá menos accesos a memoria (no se emplea el bloque índice).

3. Modificados frecuentemente y accedidos aleatoriamente y frecuentemente.

   En este caso, descartamos el contiguo, ya que no permite crecer el archivo y, al ser modificado con frecuencia, se supone que crecerá. Así, al evitar este método, se evita la fragmentación externa también.

   Entre los otros dos, como se accede de forma aleatoria, se opta por el no contiguo indexado, ya que permite acceder de forma aleatoria, mientras que el no contiguo enlazado no.

Ejercicio 1.4.12. ¿Cuál es el tamaño que ocupan todas las entradas de una tabla FAT32 que son necesarias para referenciar los cluster de datos, cuyo tamaño es de 16 KB, de una partición de 20 GB ocupada exclusivamente por la propia FAT32 y dichos cluster de datos?

Observación. Entendemos que "ocupada exclusivamente por la propia FAT32 y dichos cluster de datos" se refiere a que no hay más datos en la partición, pero sobreentendemos que el tamaño de la FAT32 no se incluye en los 20 GB, sino que se contabiliza aparte. De esta forma, los 20 GB son ocupados exclusivamente por los cluster de datos.

Veamos en primer lugar cuántos clusters de datos hay en la partición:

$$20\text{ GB} \cdot \frac{2^{30}\text{ B}}{1\text{ GiB}} \cdot \frac{1\text{ cluster}}{2^{14}\text{ B}} = 1310720\text{ clusters}$$

Por tanto, hay 1310720 entradas en la tabla FAT32. Como cada entrada es de 4 bytes (32 bits), entonces el tamaño total de la tabla FAT32 es:

$$1310720\text{ entradas} \cdot \frac{4\text{ B}}{1\text{ entrada}} = 5242880\text{ B} = 5\text{ MiB}$$

Ejercicio 1.4.13. Cuando en un sistema Unix/Linux se abre el archivo `/usr/ast/work/f`, se necesitan varios accesos a disco. Calcule el número de accesos a disco requeridos (como máximo) bajo la suposición de que el i-nodo raíz ya se encuentra en memoria y que todos los directorios necesitan como máximo 1 bloque para almacenar los datos de sus archivos.

En primer lugar, como el i-nodo raíz ya se encuentra en memoria, entonces no es necesario acceder a disco para obtenerlo. Este i-nodo nos informará de la dirección

del bloque de datos del directorio raíz. Por tanto, el primer (∗) acceso a disco será para obtener el bloque de datos del directorio raíz.

En este bloque de datos, se encuentra la entrada del directorio `usr`, que contiene el número de i-nodo del directorio `usr`. Por tanto, el segundo (∗) acceso a disco será para obtener el i-nodo del directorio `usr`. Este i-nodo nos informará de la dirección del bloque de datos del directorio `usr`, y el tercer (∗) acceso será para obtener dicho bloque de datos. Es decir, para cada carpeta es necesario dos accesos a disco: uno para obtener el i-nodo y otro para obtener el bloque de datos.

Por tanto, el cuarto (∗) y quinto (∗) acceso a disco serán para obtener el i-nodo y el bloque de datos del directorio `ast`, respectivamente. El sexto (∗) y séptimo (∗) acceso a disco serán para obtener el i-nodo y el bloque de datos del directorio `work`, respectivamente.

Por última vez, una vez ya se tenga el bloque de datos del directorio `work`, se buscará la entrada del archivo `f`, que contiene el número de i-nodo del archivo `f`. Por tanto, el octavo (∗) acceso a disco será para obtener el i-nodo del archivo `f`. Este i-nodo nos informará de las direcciones de los bloques de datos del archivo `f`.

En total, se necesitan 8 accesos a disco para obtener el i-nodo del archivo `f`. Para abrir dicho archivo, además se necesitará acceder tantas veces como bloques de datos tenga el archivo `f`, para obtener cada uno de los bloques de datos del archivo `f`, pero estos no los contabilizamos, ya que no conocemos dicho número.

Ejercicio 1.4.14. Suponiendo una ejecución correcta de las siguientes órdenes en el sistema operativo Linux:

```
/home/jgarcia/prog
$ ls -i  //lista los archivos y sus números de i-nodos del directorio prog
18020 fich1.c
18071 fich2.c
18001 pract1.c

/home/jgarcia/prog > cd ../tmp
/home/jgarcia/tmp > ln -s ../prog/pract1.c p1.c  //crea enlace simbólico
/home/jgarcia/tmp > ln ../prog/pract1.c p2.c     //crea enlace absoluto
```

represente gráficamente cómo y dónde quedaría reflejada y almacenada toda la información referente a la creación anterior de un enlace simbólico y absoluto ("hard") a un mismo archivo, `pract1.c`.

Al crear el enlace simbólico, se crea un nuevo i-nodo, que contiene la información del enlace simbólico, y se crea una nueva entrada en el directorio `/home/jgarcia/tmp`, que contiene el nombre del enlace simbólico y el número de i-nodo del enlace simbólico. Ese i-nodo contendrá la información del enlace simbólico, que es el nombre del archivo al que apunta.

Al crear el enlace absoluto, se crea una nueva entrada en el directorio `/home/jgarcia/tmp`, que contiene el nombre del enlace absoluto y el número de i-nodo del archivo al que apunta, es decir, el número de i-nodo 18001. Por tanto, el archivo `p2.c` y el archivo

`pract1.c` apuntan al mismo i-nodo, y por tanto al mismo archivo. No se crea ningún nuevo i-nodo.

Ejercicio 1.4.15. En un sistema de archivos ext2 (Linux), ¿qué espacio total (en bytes) se requiere para almacenar la información sobre la localización física de un archivo que ocupa 3 Mbytes? Suponga que el tamaño de un bloque lógico es de 1 Kbytes y se utilizan direcciones de 4 bytes. Justifique la solución detalladamente.

Veamos cuántos bloques lógicos ocupa el archivo:

$$3 \cdot 2^{20} \text{ B} \cdot \frac{1 \text{ bloque}}{2^{10} \text{ B}} = 3072 \text{ bloques}$$

Por tanto, ya sabemos que las 12 primeras direcciones directas a bloque de datos se emplean. Veamos además cuántos bloques se pueden direccionar con un bloque índice. Como cada bloque índice tiene 1 KiB, y cada dirección de bloque es de 4 bytes, entonces un bloque índice contiene:

$$2^{10} \text{ B} \cdot \frac{1 \text{ dirección}}{4 \text{ B}} = 2^8 = 256 \text{ direcciones}$$

Por tanto, en el primer nivel de indexación se pueden direccionar 256 bloques de datos, en el segundo nivel 256^2 bloques de datos, y en el tercer nivel 256^3 bloques de datos.

Como $12 + 256 < 3072$ pero $12 + 256 + 256^2 > 3072$, entonces sabemos que el archivo ocupa todos los bloques que se direccionan de forma directa y el primer nivel de indexación, mientras que el segundo nivel no se emplea entero. Por tanto, para almacenar la información sobre la localización física del archivo se necesitan:

- 12 direcciones de bloque directas.
- 1 dirección de bloque de índice, que contiene 256 direcciones de bloque.
- 1 dirección de bloque de índice, que contiene 256 direcciones de bloque índice, y cada uno de estos contiene 256 direcciones de bloque.

Por tanto, el número total de direcciones de bloque necesarias es:

$$12 + 1 + 256 + 1 + 256 + 256^2 = 66062 \text{ direcciones}$$

Como cada dirección de bloque es de 4 bytes, entonces el espacio total requerido es:

$$66062 \text{ direcciones} \cdot \frac{4 \text{ B}}{1 \text{ dirección}} = 264248 \text{ B} \approx 258{,}05 \text{ KiB}$$

Ejercicio 1.4.16. En la mayoría de los sistemas operativos, el modelo para manejar un archivo es el siguiente:

- Abrir el archivo, que nos devuelve un descriptor de archivo asociado a él.
- Acceder al archivo a través de ese descriptor devuelto por el sistema.

¿Cuáles son las razones de hacerlo así? ¿Por qué no, por ejemplo, se especifica el archivo a manipular en cada operación que se realice sobre él?

Ejercicio 1.4.17. Sea un directorio cualquiera en un sistema de archivos ext2 de Linux, por ejemplo, DirB. De él cuelgan algunos archivos que están en uso por uno o más procesos. ¿Es posible usar este directorio como punto de montaje? Justifíquelo.