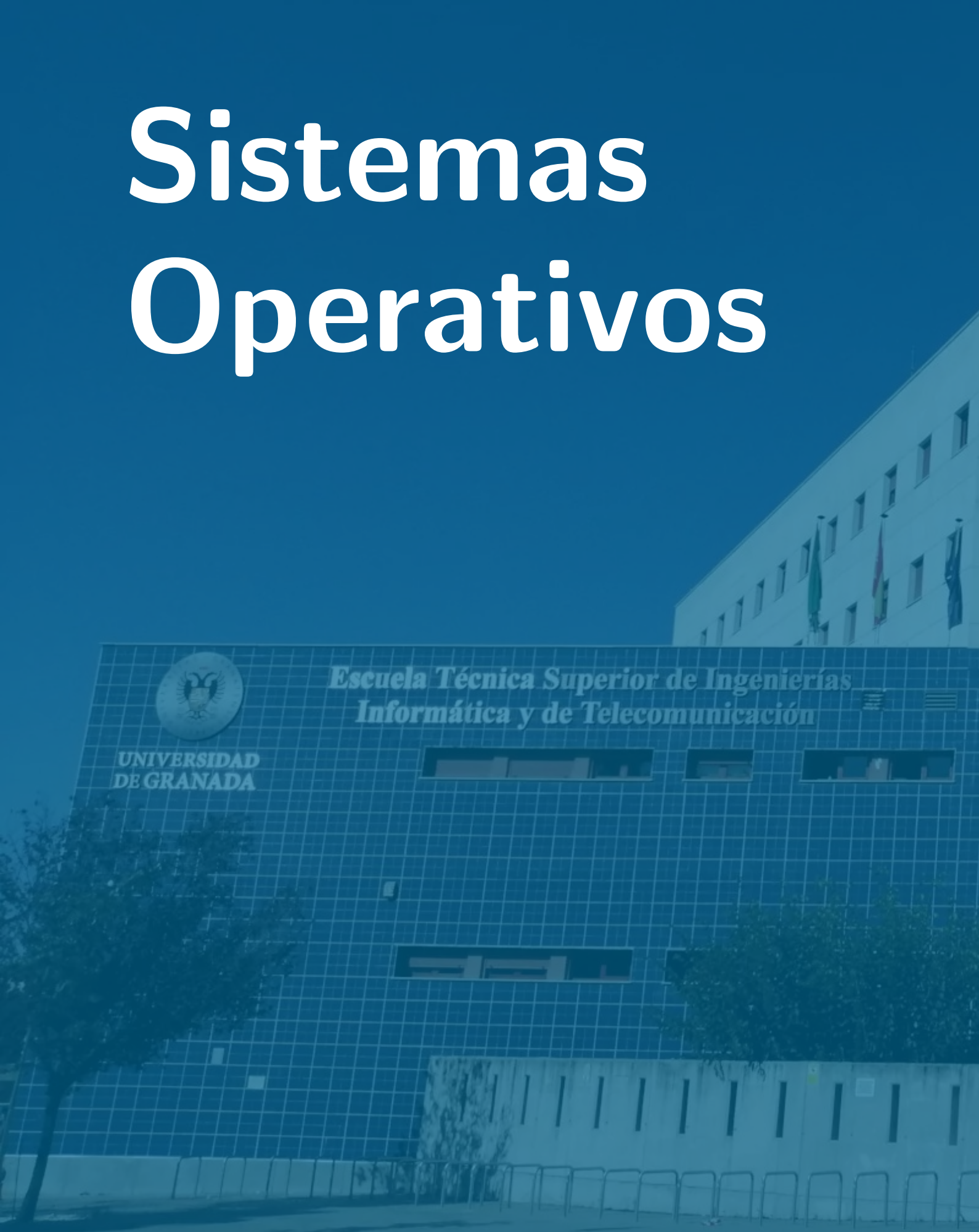
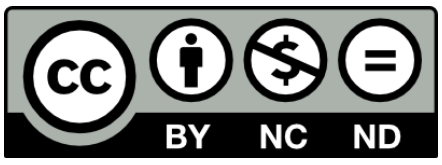
Sistemas Operativos



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

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	E													5	0	1
C		L	L	L	L	E	E	E	E									8	4	2
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 1.2: 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