Sistemes Operatius (SO)

Apunts de l'assignatura

Mario Vilar

30 de maig de 2023

Universitat de Barcelona

Índex

1

Introducció

1.1 Què és un sistema operatiu

Definició 1.1 (Sistema operatiu**).**

1. Conjunto de programas y herramientas que gestionan los recursos de hardware y software de un ordenador, y proporcionan una interfaz para que los usuarios (y sus aplicaciones) puedan interactuar con el ordenador.
2. Componente esencial para poder construir un sistema informático fiable, portable, eficiente y seguro.
3. Un SO es una capa de programa que gestiona los recursos de un ordenador para los usuarios y las aplicaciones que ejecutan.
4. Un SO actúa como un intermediario entre el hardware y el software, permitiendo que los usuarios (y sus aplicaciones) interactúen con la computadora de manera más fácil y eficiente.
5. Oculta al usuario los detalles complejos realizados internamente para acceder a los dispositivos.

Conocimiento de los SOs y sus servicios nos permiten la programación de aplicaciones.

Exemple 1.2.

1. *Servidor web*: ser capaz de gestionar 100+ o 1000+ de peticiones de conexiones de usuarios simultáneamente.
2. *Navegador web*: protegerse de posibles scripts maliciosos que se ejecutan al navegador.
3. *Nuevo hardware*: en caso de disponer de nuevos recursos, el servidor o navegador se debería beneficiar de éstos sin necesidad de instalar nuevas actualizaciones.

¿Por qué un SO puede ser considerado una MV?

1. *Árbitro*: El SO gestiona los recursos a los que acceden las aplicaciones. El SO debe aislar las aplicaciones entre sí para evitar que fallos o malware afecte a otras aplicaciones o al propio SO. Al mismo tiempo, el SO debe permitir la comunicación entre aplicaciones.
2. *Ilusionista*: El SO hace creer a las aplicaciones que utilizan todos los recursos para ellas solas, a pesar de que son compartidos con otras aplicaciones.
3. *Interfaz común*: Provee de interfaces a las aplicaciones para acceder y utilizar el hardware. Las aplicaciones son diseñadas independiente del dispositivo donde se ejecutan.

Observació 1.3 (Reptes no únics**).** Otros sistemas de programación complejos tienen requisitos similares:

1. Navegador web:
    - *Árbitro*: gestionan cargas de múltiples páginas a la vez. Debe ser seguros a software malicioso.

- *Ilusionista*: muchos servicios web están geográficamente distribuidos para ofrecer una mejor tolerancia a fallos. El navegador utilizará un servicio u otro de manera transparente para el usuario.
- *Interfaz*: el navegador debe asegurar que los scripts son portables y se pueden ejecutar en cualquier máquina y SO.

2. Cloud computing:
   - *Árbitro*: el sistema tiene que distribuir la computación entre todas las aplicaciones que se ejecutan en la nube.
   - *Ilusionista*: abstracción en el desarrollo de la aplicación y recursos web.
   - *Interfaz*: se ofrece servicios para poder distribuir las tareas de cada aplicación entre las diferentes máquinas de la nube.

1.2 Criteris d'avaluació de SOs

1. *Fiabilidad*: El sistema realiza la(s) tarea(s) para el cual ha sido diseñado. Debe ser capaz de recuperarse frente a errores o fallo en el suministro eléctrico, manteniendo la integridad de los datos.
2. *Seguridad*: El SO se suele ejecutar en un entorno hostil, donde virus y otros programas maliciosos intentan tomar el control del sistema. Los ataques se centran en vulnerabilidades específicas del software.
3. *Privacidad*: Los datos deber ser accesibles únicamente a usuarios autorizados.
4. *Portabilidad (aplicaciones)*: El SO provee a las aplicaciones de una interfaz virtual con restricciones para que pueden ejecutarse al ordenador y acceder a los dispositivos. Parte de esta interfaz (Application Programming Interface, API) son las llamadas a sistema, que utilizan las aplicaciones para acceder a los servicios del SO.
5. *Portabilidad (SO) / Escalabilidad*: Gran parte de los SOs se implementa de forma independiente al hardware mediante la llamada Hardware Abstraction Layer (HAL). Eso permite desarrollar y mejorar el SO de forma sencilla independientemente del hardware (computadoras, dispositivos móviles, grandes servidores).
6. *Rendimiento*: El SO debe ser rápido y eficiente en el uso de los recursos. Se puede medir de diversas formas.
   - ¿Cuánto tiempo se necesita para completar una operación? ¿Cuántas operaciones se pueden realizar por unidad de tiempo? (tiempo de respuesta, rendimiento)
   - ¿Cuántas operaciones necesita el SO para acceder a un dispositivo? (eficiencia)
   - ¿Cómo se reparten los recursos entre las diferentes aplicaciones o usuarios de una máquina? (sobrecarga)
   - ¿Varia el rendimiento a lo largo del tiempo? (predictibilidad)
7. *Usabilidad*: Un SO debe ser fácil de usar y tener una interfaz intuitiva para que los usuarios puedan

interactuar con él sin dificultad.

8. *Mantenibilidad*: el SO debe ser fácil de mantener y actualizar para solucionar problemas de seguridad o mejoras en la funcionalidad.

Definició 1.4 (Avaluació d'un SO**).** La evaluación de un SO se basa en una combinación de estos criterios de evaluación y otros factores específicos para el caso de uso, como la compatibilidad con un software o hardware en particular.

En general, el éxito y la adopción de un SO dependen de la capacidad del sistema para satisfacer las necesidades y deseos de los usuarios, así como de su capacidad para mantenerse actualizado y ofrecer características y funcionalidades nuevas y mejoradas.

2 Nucli

2.1 Nuclis i processos

Los SO se ejecutan en entornos hostiles donde existen muchas amenazas (virus, malware, recursos limitados,...) Los navegadores web pueden ejecutar código JavaScript. Si el navegador ofrece protección, un código JavaScript malicioso podría tomar el control del navegador, accediendo a historial de navegación, contraseñas, firmas, etc.

Definició 2.1 (Nucli**).** El núcleo (o kernel) es el componente central del SO. Es el encargado de gestionar los recursos del hardware y proporciona una interfaz para que las Apps interactúen con el hardware. La gestión del acceso de un proceso al hardware está controlado por el núcleo. Responsable de la protección del SO. Se encarga de:

1. *Fiabilidad*: Un error en la programación en una App (o malware) no debería afectar al resto de Apps. El SO debería funcionar correctamente independientemente de las Apps en ejecución.
2. *Seguridad*: Es necesario limitar las acciones que realizan las aplicaciones. P.ej. Impedir que una App escriba directamente datos a disco o modifique el propio código del SO.
3. *Privacidad*: En SO multiusuario, cada usuario (y sus Apps) solo podrá acceder a los datos a los cuales tiene permiso.
4. *Eficiencia*: Las Apps no deberían consumir todos los recursos de la máquina (CPU, memoria) en detrimento de otras Apps. El SO debería repartir los recursos.

¿Cómo lo consigue?

1. El núcleo:
    1. Es la pieza de confianza del software para comunicarse con el hardware.

    2. Tiene acceso total a las capacidades del hardware.
    3. Se ejecuta directamente en el procesador con permisos ilimitados.
2. Las procesos (Apps de usuarios, *obrir un navegador...*):
    1. no son de confianza para el SO,
    2. se ejecutan en un entorno restringido, con acceso limitado a la capacidad real del hardware,
    3. se ejecutan en el procesador con ciertas limitaciones en operaciones potencialmente peligrosas (p.ej. escritura de datos en disco) gracias a la colaboración del hardware.

Un proceso (App en ejecución) necesita autorización del núcleo para acceder a: (i) una determinada posición de memoria, (ii) para escribir a disco, (iii) modificar la configuración de una máquina, etc.

Procés 2.2 (Execució d'un programa**).**

1. *SO llama a la función main (el proceso se esta ejecutando).*
2. *SO copia las instrucciones máquina de disco a memoria.*
3. *Crea la pila.*
4. *Crea zona memoria dinámica.*

Definició 2.3 (Procés**).** Un proceso es una instancia de un programa (de la misma forma que un objeto es una instancia de una clase en POO).

Exemple 2.4. El compilador es un programa y al ejecutarlo, el SO crea el proceso correspondiente. Lo mismo pasa con el intérprete de Python, Java, navegador web, etc.

Observació 2.5.

- Podemos ejecutar múltiples instancias de un mismo programa. El SO se encarga de cargar a memoria múltiples copias de las instrucciones del programa, así como crear múltiples copias de la pila y zona de memoria dinámica.
- La creación de una pila y zona de memoria dinámica implica asignar los permisos necesarios a aquella zona para que el proceso pueda leer y/o escribir.

2.2 Mode dual

Definició 2.6 (Mode d'operació dual**).** El mecanismo actual se basa en que el hardware se puede ejecutar en dos modos, cada uno con diferentes niveles de privilegio. El modo dual de operación:

1. En modo núcleo (o kernel) el procesador puede ejecutar cualquier instrucción.
2. En modo usuario el procesador comprueba cada instrucción antes de ejecutarla.

El núcleo se ejecuta en modo núcleo, mientras que los procesos en modo usuario. Si el hardware detecta que un proceso hace una operación inválida, se avisa al núcleo de esta infracción.

¿Qué hardware es necesario para que el núcleo pueda proteger los procesos entre sí?

1. **Instrucciones privilegiadas**: Todas las instrucciones potencialmente peligrosas están prohibidas en modo usuario.
2. **Protección de memoria**: Todos los accesos a memoria física de un proceso fuera del espacio que se le ha asignado están prohibidos en modo usuario.
3. **Interrupciones de temporizador**: Independientemente de lo que haga un proceso, el núcleo del proceso debe tomar el control de forma periódica de la máquina para realizar "gestiones".

Además, el hardware debe proveer de un mecanismo para pasar de forma segura de modo núcleo a modo usuario ya la inversa.

Para ejecutar un proceso (de usuario), es necesario que

1. El proceso resida en memoria física para que pueda ser ejecutado.
2. El núcleo del SO debe residir también en memoria para que pueda gestionar las llamadas a sistema del proceso, interrupciones de temporizador así como las posibles excepciones.

Típicamente puede haber múltiples (decenas!) de procesos ejecutante:

- El sistema operativo debe configurar el hardware para que cada proceso sólo pueda acceder a su parcela de memoria.
- Si un proceso intenta acceder (por lectura o escritura) a una posición de memoria no permitida, se produce también una excepción.

¿Cómo puede evitar un SO (en particular, el núcleo) que un proceso acceda a partes no permitidas en memoria física? Un esquema sencillo utiliza dos registros hardware: base y límite. Hoy en día los ordenadores de sobremesa utilizan memoria virtual.

- Cada vez que se accede a la memoria, se suma la base a la dirección. Si se supera el límite (bound), se produce una excepción.
- **El SO opera directamente sobre la memoria real**.

Observació 2.7.

1. ¿Pila y memoria dinámica expansible? Con solo 2 registros se debe prever la memoria máxima que ocupará el proceso en el momento de cargarlo de disco a memoria.
2. ¿Compartición de memoria? Este esquema no permite que diversos procesos compartan una zona de memoria.
3. ¿Fragmentación de memoria? Con este esquema todos los procesos ocupan un espacio continuo de memoria que no se puede fragmentar en trozos.

El procesador genera direcciones virtuales que son traducidas por un hardware específico a una *dirección física*.

Propietat 2.8 (Característiques**).**

- *Los procesos viven al espacio virtual. Cada proceso cree tener disponible todo el espacio de memoria posible, aunque no exista físicamente.*
- ***El SO (núcleo) se encarga de gestionar las tablas de procesos.***
- *El mapeado de una dirección virtual a una dirección física se realiza a nivel de hardware.*
- *Posiciones contiguas a memoria virtual no tienen por qué ser continuas en el espacio físico.*

Un proceso que se ejecuta en un SO puede ejecutar cualquier instrucción no privilegiada, llamar cualquier función, hacer un bucle, etc. El SO hace creer a cada proceso que tiene toda la CPU disponible para si mismo. Para ello, el SO asigna de forma periódica el uso de la CPU a cada proceso, de tal manera que todos los procesos tienen su tiempo de ejecución. Para realizar todas estas gestiones, el *SO toma de forma periódica el control de la CPU*.

- El SO utiliza un temporizador a nivel de hardware que interrumpe la ejecución del proceso en determinados momentos (p.ej. 10 ms)
- Al llegar el temporizador a 0, se transfiere el control a una determinada función del SO y se pasa a modo núcleo. El SO puede entonces decidir que proceso se ejecuta a continuación en la CPU, accede a disco, red, etc.
- Se pueden producir otras interrupciones de ejecución de un proceso: el disco avisa al SO que tiene que leer datos, el ratón avisa que se ha movido, etc.

2.3 Transferència de control de seguretat

Todos los procesos se ejecutan en modo restringido (usuario) con la protección necesaria. Cada proceso realice varias transiciones del modo usuario al modo núcleo (y viceversa). Estas transiciones entre los diferentes modos son muy habituales (100s- 1000s). Por tanto deben ser rápidas y fiables. Los SOs modernos utilizan diversas técnicas de optimización para reducir el número de transiciones necesarias para realizar una tarea determinada:

1. *Memoria compartida*: Permite que varios procesos accedan a la misma zona de memoria sin tener que cambiar al modo núcleo.
2. *Mapeo de archivos*: Permite que los archivos se muestren directamente en la memoria de un proceso sin tener que leer o escribir explícitamente en el archivo.
3. *Temporizadores*: Permiten que los procesos realicen operaciones de espera sin tener que cambiar al modo núcleo.
4. *Protocolos de comunicación eficientes*: Protocolos como sockets y tuberías minimizan el número de transiciones entre el modo usuario y el modo núcleo.
5. *Planificación de procesos*: Optimiza la manera en que los procesos se ejecutan en el SO, lo que puede

reducir la necesidad de realizar cambios de modo para cambiar entre diferentes procesos.

¿Cuándo se realiza la transferencia de modo usuario a modo núcleo? Excepciones, Interrupciones, Llamadas a sistema (cridas a sistema).

Observació 2.9.

- Las excepciones y llamadas a sistema son síncronas (se producen por la ejecución del proceso).
- Las interrupciones son asíncronas.

Definició 2.10 (Excepció**).** Evento inesperado durante la ejecución de un proceso.

- El proceso intenta ejecutar una instrucción privilegiada.
- El proceso intenta acceder a una posición de memoria fuera del espacio permitido.
- Acceso de escritura a una posición de memoria con solo permisos de lectura.
- División por cero. "Breakpoint" de un debugger.

Al producirse una excepción, el hardware para la ejecución del proceso actual y pasa a ejecutar en modo núcleo una determinada función del núcleo del SO. Permite a los procesos detectar y gestionar errores en tiempo de ejecución.

Propietat 2.11.

- ***Las excepciones son útiles en la virtualización****: emulan hardware que no existe.*
- *Las excepciones también se utilizan para la* ***gestión de memoria virtual****. Se producen cuando:*
  - *Un proceso accede a una zona de memoria no permitida.*
  - *Un proceso accede a una posición válida pero en ese momento no es a la RAM.*

Exemple 2.12. Por ejemplo, los procesadores de bajo consumo pueden no soportar determinadas operaciones de coma flotante. El SO puede simular estas operaciones:

- Cuando un proceso realiza una operación de este tipo se produce una excepción.
- El SO emula la instrucción.
- El SO devuelve el valor al proceso y continua su ejecución.

Definició 2.13 (Interrupció**).** Señal asíncrona del hardware producida por un evento (no producido por la ejecución de un proceso). Funciona de manera similar a una excepción. El procesador para la ejecución del proceso actual y pasa a ejecutar una determinada función del SO.

Propietat 2.14.

1. *Cada tipo de interrupción tiene una función gestora asociada.*
   - *Interrupciones por temporizador.*
   - *Interrupciones sobre operaciones E/S de un dispositivo externo (movimiento de ratón, finalización de una operación a disco, llegada de datos por red, etc.).*

2. *Método de comunicación entre dispositivos externos y SO (solicita su atención).*

Definició 2.15 (Trucada a sistema (*syscall*)**).** Cualquier función que provee el SO y que puede ser llamada por el usuario. Es la interfaz que ofrece el SO a los procesos para acceder a los dispositivos o realizar otra operación (restringida) que solo tenga acceso el núcleo. A nivel de programación se parece a una llamada a una función (la llamada a sistema tiene parámetros y valores de retorno). *"Solicitud que un programa de usuario hace al núcleo del SO para que realice una tarea en su nombre".*

Las llamadas a sistema se implementan gracias a una instrucción específica (*trap instruction*).

Definició 2.16 (Trap instruction**).** Instrucción de procesador especiales que se utilizan para implementar las llamadas al sistema.

Procés 2.17.

1. *Cada llamada a sistema tiene un ID único.*
2. *Al invocar una llamada a sistema, el ID se indica en un registro de proceso específico y se ejecuta la trap instruction.*
3. *La CPU pasa de modo usuario a modo núcleo.*
4. *El SO ejecuta el código de la operación solicitada en el núcleo.*
5. *Una vez finalizada, se vuelve a modo usuario continuando con la ejecución justo después del trap instruction.*

Exemple 2.18. `count = write(fd,vector,nbytes)` es un ejemplo de llamada a sistema que permite escribir datos en un dispositivo. La función `write`, disponible en una librería de usuario, pone los parámetros en el lugar adecuado y realiza la llamada a sistema (con una trap instruction).

Procés 2.19 (Quan passem de mode nucli a mode usuari?**).**

1. ***Al restablecer la ejecución después de una excepción, interrupción o llamada a sistema**. El núcleo acaba de gestionar la operación, restablece la ejecución pasando a modo usuario.*
2. ***Al cambiar la ejecución a otro proceso**. El SO decide cambiar la ejecución a otro proceso. Para ello, el SO debe recuperar la información del estado del proceso "antiguo" y restablecerlo en el punto que se va quedar.*
3. ***Ejecutar un proceso nuevo**. Para crear un proceso nuevo, el SO carga en memoria las instrucciones a ejecutar, configura la pila, establece la primera instrucción a ejecutar del proceso, y pasa a modo usuario.*
4. ***Upcall** (llamada del núcleo a los procesos). Mecanismo por el cual los procesos puedan recibir notificaciones asíncronas de eventos, de manera similar a las interrupciones pero a nivel de usuario.*

Hemos visto cuando se transfiere de modo usuario (restringido) a modo núcleo (ilimitado), y viceversa. Desde el punto de vista técnico, *¿cómo se realiza esta transferencia de modos?* Esta transferencia se realiza mediante la estrecha colaboración entre el hardware y el núcleo del SO.

El número entero está asociado a una entrada en un vector de interrupciones. Esta entrada apunta a la función que gestionará el evento (interrupción, excepción o llamada a sistema). En el caso de las llamadas a sistema, el proceso es similar a una llamada a cualquier función (a nivel de usuario), pero se transfiere el control al SO.

Procés 2.20 (Trucada al sistema**).** *Los pasos a realizar en una llamada de sistema:*

1. *El usuario hace la llamada (`write`). Esta función almacena (en modo usuario) los parámetros de la función a la pila de usuario o registros.*
2. *Se ejecuta la trap instruction. A sistemas x86 también se llama "interrupción del programa". Se pasa a modo núcleo y se ejecuta una determinada función del SO.*
3. *Dentro del SO, cada llamada de sistema se implementa en funciones diferentes. Se copian los argumentos a la pila del núcleo y se comprueba que tenga valores correctos.*
4. *Se ejecuta la llamada y, al acabar, se sigue el camino inverso para continuar con la ejecución en el punto en el que se ha realizado la llamada. Se pasa a modo usuario.*

2.4 Estructura

Existen varios tipos de núcleos de SO: Núcleos monolíticos, Núcleos híbridos, Microkernels.

Propietat 2.21 (Nuclis monolítics**).**

1. *Centralizan la funcionalidad de todo el SO alrededor del núcleo, se aumenta el rendimiento y se facilita la integración total de los módulos. Hay menos flexibilidad para hacer modificaciones.*
2. *La mayoría de la funcionalidad del SO se centra en el núcleo (por eso normalmente no diferenciamos entre SO o núcleo).*
3. *Los módulos del núcleo llaman directamente a otros núcleos para realizar ciertas tareas.*
4. *Utilizados en la mayoría de los SO actuales (Windows, Linux, MacOS). La interfaz gráfica, librerías, o líneas de comando se ejecutan en modo usuario. No forman parte del SO.*

Los diseñadores de SOs pueden desarrollar APIs para comunicar módulos entre si. Existen dos puntos importantes con respecto a la portabilidad:

- *Hardware Abstraction Layer (HAL)*: interfaz portable que utiliza el núcleo a las operaciones específicas del dispositivo.
- Gestor de dispositivos (device driver) dinámico.

Hay una gran variedad de interfaces de hardware para gestionar dispositivos. A Linux, por ejemplo, un 70% del código del SO es software específico para acceder a los dispositivos. Es interesante desacoplar el SO de detalles específicos de cada dispositivo. Hoy en día se utilizan gestores de dispositivos dinámicos.

Definició 2.22 (Gestor de dispositiu dinàmic**).** Programa específico (programado por el fabricante) que se ejecuta una vez el núcleo ha comenzado a ejecutar y que se encarga de gestionar los dispositivos conectados a la máquina. Los gestores dinámicos tienen un defecto importante:

1. Se ejecutan en modo núcleo y por tanto pueden corromper el núcleo del SO y sus estructuras. Un programador malicioso puede utilizar los gestores de dispositivos para introducir un virus.
2. Estudios recientes han demostrado que alrededor del 90% de los motivos por los que cuelga el SO son debidos a errores del gestor del dispositivo y no debido al propio SO.

Definició 2.23 (Microkernel**).** En un núcleo microkernel, sólo se ejecutan las funciones esenciales del SO, mientras que las funciones no esenciales, como los controladores de dispositivos y los sistemas de archivos, se ejecutan en el espacio del usuario.

- El microkernel facilita la modularización, fiabilidad, y seguimiento de errores.
- Por otro lado, reduce el rendimiento ya que no se realizan llamadas directas (como en el monolítico).
- Para el programador no hay diferencia a la hora de programar para un SO monolítico o microkernel.

Observació 2.24. Al descentralizar la funcionalidad del núcleo, se augmenta la seguridad y el mantenimiento del sistema, pero puede reducir la eficiencia debido a la constante comunicación entre el núcleo y espacio de usuario.

Definició 2.25 (Nucli híbrid**).** El núcleo híbrido es una combinación de los núcleos monolíticos y microkernel.

- Algunas funciones esenciales del SO se ejecutan en el espacio del núcleo, mientras que otras se ejecutan en el espacio del usuario.
- Esto permite un equilibrio entre la eficiencia y la seguridad del núcleo monolítico y la flexibilidad del núcleo microkernel.
- En la actualidad, se intenta adoptar un modelo híbrido en el que parte de los servicios se ejecutan en modo usuario y parte en modo núcleo. Todo depende del compromiso entre complejidad y rendimiento.

2.5 Conclusions

1. El núcleo es el corazón del SO.
2. El hardware ofrece una serie de servicios para que el núcleo del SO pueda gestionar la ejecución de procesos. Todo ello gracias a una estrecha colaboración entre el hardware y el SO.
3. El hardware ofrece instrucciones privilegiadas, herramientas de protección de memoria e interrupciones de temporizador.
4. Se puede pasar de modo usuario a núcleo mediante las excepciones, interrupciones y llamadas a sistema.

5. Gracias a estas herramientas del hardware, el SO puede hacer el papel de árbitro, ilusionista y ofrecer una interfaz común a los procesos.

3

Els processos

3.1 Processos i trucades a sistema

Definició 3.1 (Bloc de processos**).** El Bloque de Control de Procesos (Block Control Process BCP) es una estructura de datos utilizada por el SO para representar y controlar cada uno de los procesos que se están ejecutando en el sistema. Contiene información detallada sobre cada proceso necesario para iniciar o reiniciar un proceso, etc.

1. Planificar el uso de la CPU
2. Cambiar de contexto
3. Gestión de procesos en espera
4. Asignación de recursos
5. Terminación de procesos

A medida que el proceso se ejecuta, su estado cambia. El estado de un proceso viene determinado por su actividad actual. *Es comporta com una màquina d'estats*.

1. Nuevo/New: ha sido creado, pero no admitido por el SO
2. Listo/Ready: en espera de que se le asigne tiempo CPU para ejecutarse
3. Ejecución/Running: esta siendo ejecutado por la CPU
4. En espera o bloqueado/Waiting: en espera de algun evento externo
5. Terminado o Salida/Terminated o Exit: finalización de ejecución.

3.2 Interfície de sistema

Observació 3.2 (Execució de processos**).**

- **Sistemas monoprocesadores**: *¿Cómo se ejecutan varios procesos a la vez?* No puede, solo 1 proceso se ejecuta en un momento dado ("running"). El resto de procesos pueden estar en modo "Ready" o "Waiting". Se asigna un "time slice"/"ciclo de tiempo"/"rebanada de tiempo" de CPU a cada proceso (unos pocos milisegundos).
- **Sistemas multiprocesadores**: Paralelismo real. Cada procesador únicamente ejecuta 1 proceso en un momento determinado. El SO decide (mediante el planificador) que proceso se ejecuta en cada momento.

Se require de llamadas al sistema para comunicar procesos entre sí. Proporcionan una interfaz para los servicios disponibles por un SO. Generalmente disponibles como funciones escritas en alto nivel (C y C++), aunque ciertas tareas de bajo nivel pueden estar escritas en lenguaje ensamblador. Principalmente utilizadas por programas a través de una interfaz de sistema.

Definició 3.3 (POSIX**).** POSIX (Portable Operating System Interface for Unix) es un conjunto de estándares para los SOs compatibles con POSIX (Linux, macOS, FreeBSD, Solaris,..). Aumenta la interoperabilidad y portabilidad de las aplicaciones y SOs.

Definició 3.4 (API**).** Una interfaz de sistema (API) es un conjunto de funciones, protocolos y herramientas que permiten que los procesos interactúen con el SOs o con otros procesos. Define los servicios que el SO proporciona a las aplicaciones, y también define cómo los procesos pueden acceder a esos servicios y utilizarlos:

1. Funciones para realizar operaciones de E/S.
2. Administrar el almacenamiento de archivos.
3. Gestión de procesos y recursos del sistema).

Toda la funcionalidad esta implementada a nivel de núcleo. Los procesos acceden a estos servicios mediante llamadas al sistema. Al utilizar una interfaz de sistema, los desarrolladores de software pueden escribir aplicaciones que sean compatibles con el SO y con otros programas, sin tener que conocer los detalles internos de cómo funciona el SO o los demás programas.

Propietat 3.5 (Serveis**).**

1. *Gestión de procesos. Administrar y controlar los procesos que se ejecutan en el sistema. ¿Puede un proceso crear otro proceso? ¿Puede un proceso esperar a que otro proceso acabe de ejecutarse? ¿Podemos parar o continuar la ejecución de un proceso?*
2. *Entrada-salida. Controlar el flujo de datos entre los dispositivos de E/S (discos duros, teclados, pantallas, etc.) y la CPU.*
3. *Gestión de memoria, gestión de red, etc.*

Antiguamente, el núcleo creaba los procesos. ***En la actualidad los mismos procesos crean otros procesos****.*

¿Qué beneficios tiene que un proceso cree otro proceso?

- Permite una mayor modularidad, flexibilidad, control, aislamiento y escalabilidad en el diseño de SOs y de software en general.
    - Si un proceso necesita realizar una tarea compleja o que requiere mucho tiempo, puede crear otro proceso para realizar una parte de esa tarea.
    - Modularidad: Cada proceso puede realizar una tarea específica y comunicarse con otros procesos para completar una tarea más compleja.

  - Control: Los programas de usuario pueden tener un mayor control sobre cómo se ejecutan las tareas en el SO.
  - Aislamiento: Los procesos hijos están aislados del proceso padre y otros procesos.
- Un navegador web puede utilizar Apps externas para dibujar una página por pantalla. Incrementa la funcionalidad del programa.
- Un servidor web puede invocar un proceso externo para dar formato a una página web antes de ser visualizada por el navegador de un dispositivo.

3.3 Gestió de processos

Definició 3.6 (Gestió de processos). Parte fundamental de un SO que se encarga de administrar y controlar los procesos que se ejecutan en el sistema. El proceso que crea un proceso se llama *padre*, mientras que el proceso que es creado se llama *hijo*. Las tareas que realiza el núcleo del SO al crear un nuevo proceso son:

1. Crea e inicializa el bloque de control de procesos.
2. Crea e inicializa el espacio de dirección de memoria.
3. Carga el programa a memoria.
4. Prepara el núcleo para comenzar a ejecutar el main.

Dado que los procesos pueden crear otros procesos, el SO almacena información sobre esta *jerarquía de procesos*.

Exemple 3.7. `fork()` crea un nuevo proceso hijo, independiente del padre. El hijo consiste en una copia del espacio de direcciones del proceso padre y permite que el padre se pueda comunicar fácilmente con su hijo. Devuelve dos veces, una para el padre (PID hijo) y otro para el hijo (0).

```
int main(void) {
    int ret;
    ret = fork();
    if (ret == 0) {
        printf(%d\n,getpid());
        return 0;
    } else {
        printf(%d\n,ret);
        return 0;
    }
}
```

Exemple 3.8. Después de crear un proceso hijo con `fork()`, una de las llamada a sistema más común en UNIX es `exec()` para ejecutar un nuevo programa (¡No crea un nuevo proceso!). Carga un archivo binario

a memoria e inicia su ejecución, i.e. reemplaza la imagen del proceso por la especificada en `exec()`. El padre se puede poner en `wait()` hasta que termine su hijo.

Al realizar el `fork()`, el proceso hijo hereda el contexto del padre (privilegios, ficheros abiertos, etc.). Después del `fork()`, y antes de hacer el `exec()`, se puede establecer un nuevo contexto para programa a ejecutar mediante llamadas al sistema.
¿Hace falta que el padre espere a que el hijo finalice? Es posible que el padre necesite un resultado de su hijo para continuar con su ejecución. ¿Cómo lo conseguimos? La función `wait()`.
¿Siempre el padre debe esperar a que su hijo finalice? No necesariamente. El hecho de no utilizar el wait() permiten que el padre e hijo se ejecutan "en paralelo".

Observació 3.9. Al crear un nuevo proceso mediante fork(), el nuevo proceso hijo es una copia exacta del proceso padre, incluyendo su contexto de ejecución (datos y recursos asociados a al proceso en el momento dado). Un proceso padre puede controlar el contexto del hijo:

1. Valores en los registros del procesador
2. Privilegios
3. Tiempo máximo de ejecución
4. Memoria máxima que puede ocupar
5. Prioridad para ejecutarse sobre la CPU respecto a otros procesos
6. Descriptores de ficheros abiertos
7. Lugar donde se envía lo que se imprime con un `printf` (salida)
8. Donde recibe lo que se captura con el `scanf` (entrada)

Exemple 3.10. `setrlimit`. Función del SO que establece límites de recursos para un proceso en ejecución:

- Tiempo CPU
- Uso de memoria dinámica
- Uso de la pila
- Número de ficheros que se pueden abrir

Estos límites se aplican solo al proceso actual y a sus procesos hijos, y no afectan a otros procesos.

Exemple 3.11. `ulimit`. Comando de shell que establece y consulta los limites de recursos para el shell actual y sus procesos hijos. Nos permite:

- Ejecutar la aplicación while-infinit.c limitando el tiempo máximo de ejecución.
- Podemos ejecutar otras aplicaciones limitando, por ejemplo, el tamaño máximo de la pila.

Además de establecer límites de privilegios sobre los procesos hijos, hay otras cosas que se pueden controlar desde el proceso padre:

1. Cambio de prioridad: Asignación de más o menos recursos.
2. Monitoreo de estado: Monitorear el estado del proceso hijo para saber si está funcionando correctamente, si ha finalizado o si ha ocurrido algún error.
3. Manejo de señales: El proceso padre puede enviar señales al proceso hijo para indicarle que realice ciertas acciones (terminar o detenerse temporalmente).
4. Asignación de recursos: Asignar recursos adicionales al proceso hijo según sea necesario, como memoria, tiempo de CPU o acceso a dispositivos de entrada y salida (E/S).

3.4 Comunicació entre processos

Hemos visto que el SO ofrece protección entre procesos, por lo que los procesos no pueden interferir entre si. Sin embargo, los procesos tienen que comunicarse entre si (con permiso del propio SO). *¿Cuáles son las técnicas básicas de comunicación entre procesos?* **Redirección de E/S estándar** y **Tuberías (pipes)**.

Exemple 3.12. `printf`. Esta instrucción genera, en el espacio del usuario, la cadena a imprimir. Una vez generada la cadena, se realiza una llamada al SO con un `write`:

```
count = write (fd, vector, nbytes);
```

La función `write` (una librería de usuario), realiza la llamada al sistema:

- `fd` corresponde con el descriptor del fichero, un entero que identifica el dispositivo donde se escribirán los datos.
- `vector` y `nbytes` hacen referencia al vector de bytes que contiene los datos que se escribirán y el número de bytes a escribir.

Descriptors de fitxers. Todo proceso creado por el SO (Unix o Windows) tiene creado 3 ficheros por defecto:

1. *Descriptor 0*. "Entrada estándar". Esta asociado por defecto al teclado (se captura con un scanf).
2. *Descriptor 1*. "Salida estándar". Asociado por defecto a la pantalla del terminar (se imprime con printf)
3. *Descriptor 2*. "Salida de error". Asociado por defecto a la terminal.

Por defecto, descriptor 1 (printf) esta asociado al dispositivo "pantalla". Ahora queremos asociar este descriptor 1 a un *fichero de disco*.

1. Hacemos un fork.
2. Se abre el fichero donde queremos volcar la información que se escribe al descriptor 1. Realizamos las llamadas al sistema que asocian el fd 1 al fichero.
3. Ejecutamos el programa. Ahora todo lo que se escribe al fd1, se escribe a disco.

Definició 3.13 (dup2**).** `dup2(int oldfd,int newfd)` Es una llamada al sistema que se utiliza para duplicar un descriptor de archivo. Duplica el descriptor de archivo `oldfd` en el descriptor de archivo `newfd`. Es útil para redirigir la entrada y salida estándar de un proceso.

Exemple 3.14. La línea de comandos también permite redirigir la salida estándar a un fichero: `ls -l > fichero.txt`. Probar también: `./stdstreams > fichero.txt`. Una cosa similar pasa con `var=$(ls)`. El intérprete de comandos redirige la salida estándar del comando `ls` para después leerlo y asignarlo a `var`. Al ejemplo anterior capturamos la salida estándar (`fd=1`), pero no la de error (`fd=2`), que es donde se imprimen típicamente los errores.

Igual que redirigimos la salida estándar (`fd=1`) a un fichero, podemos redirigir un fichero a la entrada estándar (`fd=0`). Asociamos el descriptor o `scanf` a un fichero de disco.

1. Hacemos un fork.
2. Se abre el fichero desde el cual el descriptor o coge los datos.
3. Ejecutamos el programa. scanf cogerá los datos de disco.

3.5 Tuberies

Definició 3.15 (Tuberies**).** La tubería (pipe) es un mecanismo de comunicación entre procesos que permite la transferencia de datos entre un proceso emisor y un receptor. *¿Cómo lo implementamos?* Llamada al sistema pipe() que crea:

- Un buffer al SO que gestiona la comunicación.
- Dos descriptores de ficheros: Uno de escritura, uno de lectura.

Las tuberías es canal de comunicación unidireccional entre dos procesos, siempre y cuando estos procesos tengan relación padre-hijo.

Exemple 3.16 (fork-pipe.c**).**

1. Un proceso puede introducir datos en la tubería y el otro las va leyendo.
2. No hace falta que el primer proceso acabe para que el segundo comience a leer. Es un flujo de comunicación continúa entre los dos procesos.
3. Al terminal, para realizar operaciones con una tubería se requiere de las funciones dup2, así como el cierre de los extremos de la tubería que no se utilizan.

Definició 3.17 (Búfer de la tuberia**).** El búfer es un buffer interno del SO. Todos los datos pasan por este búfer. El búfer es relativamente pequeño (64 KB). Si al leer del búfer no hay ningún dato, el proceso que realiza llamada se bloquea hasta que hay datos disponibles. Si al escribir en el búfer, éste se encuentra lleno, el proceso realizar una llamada se bloquea hasta que el otro proceso comienza a vaciar el búfer.

Observació 3.18 (Conclusions**).**

1. El SO ofrece una interfaz (llamadas al sistema) para crear procesos: `fork()`,`exec()`.
2. Los procesos pueden crear nuevos procesos con un contexto de ejecución limitado (setrlimit, ulimit).

3. Los procesos son independientes entre si.
4. El SO ofrece herramientas para que los procesos se puedan comunicar entre si.
5. Los mecanismos más habituales de comunicación: redirección y tuberías.

4

Comunicacions entre processos

Definició 4.1 (Comunicació entre processos**).** La comunicación entre procesos (IPC, del inglés Inter-Process Communication) es una función clave de los SO multiusuarios y multitarea que permite que los procesos se coordinen de manera eficiente y controlada.

¿Por qué es necesaria? Porque permite que los procesos en un SO interactúen y compartan información entre sí.

Exemple 4.2. Por ejemplo, un proceso puede necesitar información de otro proceso para completar una tarea o puede necesitar enviar información a otro proceso para que este pueda continuar con su ejecución.

1. *Coordinación y colaboración efectiva entre procesos*. Un proceso de edición de video puede enviar información a un proceso de renderizado para que este pueda generar una vista previa en tiempo real.
2. *Compartir información de manera eficiente y controlada*. Un proceso de base de datos puede enviar información a un proceso de visualización para que este pueda mostrar los datos en una tabla o gráfico.
3. *Permitir que el SO funcione de manera eficiente*. Un proceso de gestión de memoria puede enviar información a un proceso de ejecución para que este pueda liberar memoria cuando sea necesario.

Definició 4.3 (Fitxer**).** Para POSIX, un fichero es una secuencia de bytes organizados en un dispositivo de almacenamiento (red, disco, flash USB, tarjeta SD...). Los ficheros se tratan de manera uniforme, independientemente del tipo de dispositivo en el que estén almacenados. En UNIX, los ficheros son accedidos secuencialmente por defecto. Cuando un usuario abre el fichero, el núcleo inicializa el puntero de lectura/escritura a cero. Cada vez que el proceso lee o escribe datos, el núcleo avanza el puntero en la cantidad de bytes transferidos. Todos los ficheros abiertos por un proceso tienen asociado un descriptor de fichero `fd` que asigna el núcleo del SO.

El SO nos ofrece llamadas al sistema (que devuelven un `fd`) para realizar comunicaciones entre procesos.

1. Abrir una comunicación: `open()` (oara abrir un fichero de disco), `pipe()` (para crear una tubería anónima), `mkfifo()` (para crear una tubería con nombre),`socket()` (para crear una conexión remota vía red)...
2. Enviar y recibir información (tienen `fd` como primer argumento). `write()`,`read()`.
3. Cerrar una comunicación (único argumento: `fd`). `close()`.

Definició 4.4 (Senyals**).** Mecanismo de **comunicación asíncrona** entre procesos (necesita PID del proceso receptor) y/o entre el núcleo y los procesos. Es una notificación que indica a un proceso que ha ocurrido un evento (provocado por otro proceso o por el SO) en el sistema. Al enviar una señal, el núcleo interrumpe la ejecución del proceso y se llama una determinada función (previamente indicada por el proceso). El proceso puede:

1. Terminar su ejecución.
2. Ignorar la señal.
3. Capturarla y manejarla.
4. Detenerse y esperar interacción del usuario.

Exemple 4.5.

- Al producirse una excepción el SO envía una señal al proceso que ha provocado la excepción. El proceso puede enviar información a un fichero log del problema antes de "morir".
- En los Sistemas de Alimentación Ininterrumpida (SAI), el sistema puede realizar una notificación asíncrona a los procesos si le queda poca batería. Así los procesos pueden guardar datos a disco antes de que se apague el sistema.

Senyals POSIX més comuns.

1. `SIGKILL`: Señal para matar un proceso inmediatamente. El proceso no puede capturar o ignorar esta señal (No se puede ignorar). La señal SIGKILL es similar a SIGTERM pero con una diferencia: el SIGKILL no se puede capturar como lo hemos hecho con SIGTERM. Es un método a prueba de fuego para matar un proceso en caso de que no nos haga caso.
2. `SIGTERM`: Señal para terminar un proceso de manera controlada. El proceso puede capturar y manejar esta señal.
3. `SIGSTOP`: Señal para detener un proceso temporalmente. El proceso no puede capturar o ignorar esta señal (No se puede ignorar).
4. `SIGCONT`: Señal para reanudar un proceso detenido anteriormente con `SIGSTOP`.
5. `SIGINT`: Señal para interrumpir la ejecución de un proceso desde el teclado.
6. `SIGUSR1`,`SIGUSR2`: Señales definidas por el usuario para ser usada en programas de aplicación.

¿Qué acciones pueden producirse al recibir una señal? Cada proceso indica al SO que hacer en caso de recibir una señal determinada.

- **Acción por defecto** (en caso de que no se indique nada).
- **Capturar la señal**. El proceso indica al SO que función tiene que llamar en producirse un determinado evento.
- **Ignorar la señal**. El proceso puede indicar al SO que ignore la señal. Puede ser peligroso (para el

proceso) en caso de que, por ejemplo, se produzca una división por cero y se continue la ejecución normalmente.

`signal()` se utiliza en Unix para establecer el comportamiento que un proceso debe tomar en respuesta a una señal específica enviada por el SO o por otro proceso.

Las **tuberías anónimas** tienen una gran desventaja: Solo permiten la comunicación entre procesos con **relación padre-hijo**.

Definició 4.6 (FIFO**).** Las tuberías con nombre (también llamadas tuberías FIFO; First-In-First-Out) combinan características de fichero de disco y tubería.

1. Se crean con anticipación y tienen un nombre asociado en el sistema de archivos.
2. Los procesos pueden abrir la tubería con nombre como si fuera un archivo normal, y pueden enviar o recibir datos a través de ella.
3. Permiten que los procesos se comuniquen de manera eficiente, eliminando la necesidad de almacenar datos intermedios en archivos en el sistema de archivos, lo que reduce el uso de recursos del sistema.

Propietat 4.7 (Característiques de les FIFO**).**

1. *Al abrir una tubería con nombre, se le asocia un nombre de fichero de disco. Este fichero funciona como una tubería FIFO. El SO decide sonde se almacena este fichero.*
2. *Cuando se ejecuta el comando mkfifo, se crea un archivo especial FIFO en el sistema de archivos. Sin embargo, los datos escritos en un archivo FIFO no se almacenan en el disco como lo hacen los archivos regulares.*
3. *Los datos escritos en un archivo FIFO se almacenan temporalmente en la memoria y se transmiten directamente a cualquier proceso que esté leyendo del archivo FIFO.*
4. *Una vez que los datos se leen del archivo FIFO, se eliminan de la memoria.*
5. *Cualquier proceso puede abrir este fichero (lectura o escritura). Solo se requiere que los procesos utilicen el mismo nombre de la tubería.*

Definició 4.8 (Regular file**).** Un archivo regular (regular file) es tipo de archivo que contiene datos almacenados en una secuencia de bytes, que se pueden leer y escribir. Pueden ser utilizados como un mecanismo de comunicación simple y efectivo entre procesos.

1. Uno de los procesos debe crear el archivo regular y establecer los permisos de acceso adecuados para permitir que otros procesos escriban y lean del archivo.
2. El proceso emisor escribe datos en el archivo regular, y el proceso receptor lee los datos.

Observació 4.9. Esta técnica es especialmente útil para la comunicación entre procesos que no necesitan una alta tasa de transferencia de datos (rendimiento sujeto a la velocidad de acceso al disco).

Otros tipos de archivos en los SO incluyen **directorios**, archivos de dispositivo (**device files**) o enlaces simbólicos (**symbolic links**).

Pros	Cons
Disco accesible para todos	Manipular archivos más complicado
Cualquier posición del archivo	El fichero contienen datos consistentes
Múltiple escritura/lectura	

Llamadas al sistema para manipular ficheros:

- `open` y `create`: abre y crea un fichero en el disco. La función devuelve un entero (el fd del fichero abierto)
- `close`: Cierra un fichero. Hay que pasar por parámetro el fd del fichero a cerrar.
- `read` y `write`: lee y escribe datos (p. ej. Bytes) de/a cualquier fichero. Hay que pasarle el fd como parámetro.
- `lseek`: establece la posición actual dentro de un fichero en el disco (lectura/escritura). Necesita fd.

Observació 4.10.

- `open,read,write`... son funciones que llaman directamente al SO.
- El SO tiene un búfer interno (para cada fichero) que es accesible por todos los procesos.
- `write` escribe datos al búfer interno del SO y devuelve enseguida. La operación de escritura a disco se realiza más tarde.
- `read` comprueba primero si hay datos en el búfer interno del SO. Si no hay datos disponibles, se leen al disco. Mientras tanto, el proceso que hace el read queda en estado "bloqueado".
- Si un proceso escribe a fichero y otro lee después, se leerán los datos correctos aunque los datos no se hayan escrito todavía a disco.

Definició 4.11 (Librería estándar**).** La librería estándar "stdio" nos ofrece las siguientes funciones para manipular ficheros en disco.

1. `fopen`: abre fichero de disco (lectura, escritura,..).
2. `fclose`: cierra el fichero.
3. `fread` y `fwrite`: lee y escribe datos (p. ej. Bytes). Internamente se llama a la función read y write del SO.
4. `fscanf` y `fprintf`: Lee y escribe datos en formato ASCII. Internamente se llama a `read` y `write` del SO.
5. `fseek`: define la posición actual dentro de un archivo.

Definició 4.12 (Estructura FILE**).** Una estructura FILE es utilizada para representar un archivo abierto. La estructura FILE contiene información sobre el estado del archivo, como la posición actual del cursor de lectura/escritura y si el archivo se abrió para lectura, escritura o ambos.

La estructura FILE y las funciones fopen, fprintf, fscanf, fwrite, fread,... son funciones de la librería estándar "stdio":

- Utiliza búfer propio (no visible a otros procesos).
- Las operaciones `fprintf` o `fwrite` escriben datos al búfer `FILE`. Cuando el búfer se llena, se escriben los datos en el disco utilizando la llamada al sistema `write` (que escribe en un búfer interno del SO).
- Las operaciones `fscanf` o `fread` leen los datos del búfer `FILE`. Si no hay datos disponibles, se realiza la llamada al sistema `read` para llenar el búfer `FILE`. Mientrastanto, el proceso queda bloqueado.
- La librería estándar consigue así minimizar el uso de las funciones write y read (llamadas a sistema).
- Múltiples procesos (cada uno con su propio búfer). Un dato no se verá de forma inmediata por otro proceso.

Definició 4.13 (Archivos mapeados a memoria**).** Permite que dos (o más) procesos mapeen el contenido de un archivo en memoria, permitiendo que los procesos puedan acceder a él como si fuera una parte de la memoria del proceso. El archivo mapeado a memoria es un archivo que se asocia a una región contigua de memoria virtual. Los cambios realizados por un proceso en un archivo mapeado son inmediatamente visibles para todos los procesos que comparten el mismo archivo mapeado a memoria.

Definició 4.14 (mmap**).** Para utilizar archivos mapeados a memoria, los procesos primero deben crear o abrir el archivo en el sistema de archivos. A continuación, los procesos pueden utilizar la función `mmap()` para asignar una región de memoria virtual del proceso al archivo. `mmap()` devuelve un puntero a la región de memoria virtual asignada, que se puede utilizar para leer y escribir datos en el archivo.

Propietat 4.15 (Atributs**).**

1. *address: especifica la dirección en la que se desea que comience el mapeo. Si se establece en 0, el SO elige la dirección automáticamente.*
2. *length: especifica la longitud del mapeo en bytes.*
3. *protect: especifica los permisos de protección de la memoria mapeada. Se pueden establecer permisos de lectura, escritura y/o ejecución.*
4. *flags: especifica las opciones de configuración del mapeo, como si se desea que el mapeo sea compartido entre procesos o privado.*
5. *fd: especifica el descriptor de archivo del archivo que se desea mapear. Debe haberse abierto previamente con la llamada a open().*
6. *offset: especifica el desplazamiento en el archivo desde el cual se iniciará el mapeo.*

Pros	*Cons*
alta eficiencia	*tamaño del archivo limitado*
seguridad	*más complicado*
el SO se encarga de cargar y guardar memoria virtual	

4.1 Sockets

Definició 4.16 (Socket**).** Un socket es un objeto que permite la comunicación entre procesos (del mismo o diferentes ordenadores) a través de una red o un canal de comunicación local. Proporciona un mecanismo para enviar y recibir datos entre procesos, y se puede utilizar para implementar una amplia variedad de protocolos de comunicación, como TCP/IP, UDP, y UNIX domain sockets.

Procés 4.17.

1. *Los procesos primero deben crear un objeto de tipo socket utilizando la función socket(). Esta función devuelve un descriptor de archivo que se puede utilizar para enviar y recibir datos a través del socket.*
2. *Después de crear el socket, los procesos pueden utilizar las funciones bind() y listen() para asignar una dirección al socket y esperar conexiones entrantes.*
3. *Cuando un cliente desea conectarse al servidor, utiliza la función connect() para establecer una conexión con el servidor.*
4. *Una vez que se ha establecido la conexión entre el cliente y el servidor, se pueden utilizar las funciones send() y recv() para enviar y recibir datos a través del socket.*

- Un ordenador se identifica mediante una dirección IP (192.168.80.165).
- Un proceso A puede escuchar las peticiones entrantes y engancharse a un puerto (p.ex. 10243).
- Cualquier otro proceso B puede enviar un mensaje al proceso A: Primero, se abre un canal de comunicación bilateral con la dirección IP y el puerto. Después, si B escribe un mensaje, este llegará a A (y viceversa).

5 Planificació de processos

Definició 5.1 (Planificació de processos**).** La planificación de procesos (scheduling) es una de las funciones más importantes del SO, ya que manejamos recursos limitados. Herramienta mediante el cual el SO decide el orden en el que se ejecutarán los procesos activos. Asigna tiempo de CPU de manera "equitativa" a los procesos activos del sistema. Se pretende maximizar el rendimiento y garantizar que todos los procesos se ejecuten de manera "justa".

A medida que el proceso se ejecuta, su estado cambia. El estado de un proceso viene determinado por su actividad actual:

1. Nuevo/New: ha sido creado, pero no admitido por el SO.
2. Listo/Ready: en espera de que se le asigne tiempo CPU para ejecutarse (candidato del SO/planificador a ejecutarse).

3. Ejecución/Running: esta siendo ejecutado por la CPU.
4. En espera o bloqueado/Waiting: en espera de algun evento externo.
5. Terminado o Salida/Terminated o Exit: finalización de ejecución.

- **Listo ⇔ ejecución**. En general, el SO (a través del planificador) decide cuando un proceso deja de ejecutarse y pasa al estado Listo/Ready (y a la inversa). Un proceso también puede voluntariamente dejar de ejecutarse y pasar a un estado Listo/Ready (función `sched_yield` de C).
    1. Realiza tarea que no requiere toda la potencia de la CPU.
    2. Espera a que ocurra un evento externo.
- **Bloqueado**. Un proceso esta En espera/Bloqueado/Waiting si esta esperando 1 evento. Este proceso Bloqueado no es elegible de ser ejecutado por el SO hasta que no tenga lugar este evento. Entonces pasará a un estado Listo/Ready.
    1. Llamada a función de lectura para leer datos de red, teclado o disco. Hasta que los datos no estén disponibles, el proceso esta bloqueado.
    2. Se duerme voluntariamente mediante un sleep o wait (espera por 1 proceso).

Definició 5.2 (Canvi de context**).** Mecanismo mediante el cual el SO (núcleo) cambia la ejecución de un proceso por otro.

Definició 5.3 (Planificador**).** Algoritmo que utiliza el SO para gestionar los procesos y decidir cual es el siguiente proceso a ejecutar. Cara SO puede utilizar una técnica (e información) diferente para tomar esta decisión.

Definició 5.4 (BCP**).** El Bloque de Control de Procesos (Block Control Process BCP) es una estructura de datos utilizada por el SO para representar y controlar cada uno de los procesos que se están ejecutando en el sistema. Contiene información detallada sobre cada proceso necesario para iniciar o reiniciar un proceso, etc.

- Planificar el uso de la CPU.
- Cambiar de contexto.
- Gestión de procesos en espera.
- Asignación de recursos.
- Terminación de procesos.

Definició 5.5 (Cambio de contexto**).** Al pasar un proceso de modo Ejecución a Listo o Bloqueado, se produce un cambio de contexto.

1. Para dejar de ejecutar un proceso, el SO guarda toda la información del proceso que se está ejecutando en su BCP.

2. Una vez el SO elige un proceso a ejecutar, se restauran los valores de registros almacenados en su BCP. Estos valores se copian a la CPU y el proceso continua ejecutándose dónde lo había dejado anteriormente.

¿En qué situaciones se produce un cambio de contexto?

- Al producirse una *interrupción de temporizador* (p.ej. 100 ms). El planificador decide dejar de ejecutar el proceso actual y pasa a ejecutar otro.
- Cuando se *bloquea* un proceso. En ese momento, el planificador decide cual de los procesos con estado Listo pasa a ejecutarse.
- Con otros tipos de interrupción. Cuando llegan datos por la red, al haberse leído datos en disco, etc.. se interrumpe la ejecución de un proceso. El planificador puede decidir ejecutar el proceso que se solicitado datos o continuar con la ejecución del proceso que se estaba ejecutando en aquel momento.
- Otros casos.

Exemple 5.6 (Interrupción de temporizador**).** Cuando ocurre una interrupción de temporizador en un SO, se produce un cambio de contexto para permitir que el planificador de procesos seleccione el siguiente proceso que se ejecutará. Este proceso de cambio de contexto debido a una interrupción de temporizador se repite continuamente en el SO, permitiendo que los procesos se ejecuten de manera equitativa y que el sistema utilice los recursos de manera eficiente.

1. El controlador de interrupciones del SO se activa debido y detiene la ejecución del proceso.
2. Se invoca al planificador de procesos para determinar qué proceso debe ejecutarse a continuación.
3. El planificador de procesos selecciona el siguiente proceso que se ejecutará a partir de la lista de procesos Listos.
4. El estado del proceso seleccionado (PCB) se carga en la CPU y se restaura su contexto.
5. El proceso seleccionado se ejecuta en la CPU hasta que ocurre otro cambio de contexto o hasta que finaliza su ejecución.

¿Cómo elige el SO (planificador) el proceso a ejecutar a continuación? ¿Cuál es el mejor método? La solución más "justa" podría ser una cola FIFO circular pero, en ciertas ocasiones, puede ser la peor opción ya que el usuario no percibe visualmente una buena respuesta.
Aquellos algoritmos de planificación que funcionan bien en entornos de alta carga computacional (CPU bound) pueden no ser idóneos en entornos con carga alta de entrada-salida (I/O bound), y al contrario.

- CPU bound. Pocos requerimientos de E/S, pero utiliza la mayoría del tiempo realizado cálculos computacionales.
- I/O bound. Pasa más tiempo realizando tareas de E/S que computacional.

Observació 5.7 (Algorisme de planificació**).** No existe un planificador optimo general. Cada uno tiene sus ventajas e inconvenientes. El objetivo de este tema es ver algoritmos que funcionan bien en entornos con carga variada, que utilizan los ordenadores personales.

1. Planificación uniprocesador.
2. Planificación multiprocesador.
3. Planificación de ejecución en tiempo real.
4. Planificación en sistemas sobrecargados.

Definició 5.8 (Tasca**).** Un proceso puede ejecutar a lo largo de su ejecución múltiples tareas diferentes, cada una con características diferentes. Por eso, hablaremos de **tareas** y no de procesos. Además, un proceso puede estar formado por muchos hilos de ejecución, cada uno ejecutando una tarea.

5.1 Round robin

El planificador dispone de una lista de tareas y las ejecuta de forma ordenada (de manera **circular**). Cada una de las tareas se ejecutan durante un tiempo limitado (rebanada de tiempo o *time slice*). Cada vez que finaliza una rebanada el planificador hace un cambio de contexto a otra tarea. La tarea se puede ejecutar en menos tiempo. Al realizar un cambio de contexto, se pone la tarea actual al final de la lista de tareas (en el estado Listo) y se coge la primera tarea de la lista de tareas Listas (y se pone en Ejecución).
Al realizar un cambio de contexto, se necesita guardar registros del proceso actual en ejecución, seleccionar nuevo proceso y cargar sus registros a la CPU. Hay que tener en cuenta el impacto en la memoria caché: La tarea seleccionada para ejecutar puede haber tenido (o no) datos en la caché durante su rebanada de tiempo anterior. Todo dependen de lo que haya hecho las otras tareas en su "ausencia". Quizás la tarea en ejecución debe acceder a la RAM para que se vuelvan a copiar los datos a la caché y eso es muy lento.
Debe haber un *compromiso* entre el tiempo de respuesta y la sobrecarga a la hora de elegir valor de la rebanada de tiempo.

1. Si el valor de la rebanada de tiempo es muy pequeña, habrá una sobrecarga de planificación. Se perderá mucho tiempo en hacer cambios de contexto y las tareas no se podrán realizar (alta sobrecarga).
2. Si el valor de la rebanada de tiempo es muy grande, las tareas esperarán mucho tiempo para obtener su turno y poder ejecutarse (tiempo de respuesta bajo).

Exemple 5.9. La planificación round robin generalmente **no es adecuada en un entorno con carga variada**. Supongamos 3 tareas: 2 CPU bound y I/O bound que solo necesita la CPU entre 1 ms y 10 ms de disco. Si la tarea I/O bound se ejecuta sola, podría tener ocupado el disco y acabar la tareas rápidamente. Con las otras dos tareas CPU bound y una rebanada de tiempo de 100ms, la tarea I/O bound se ralentizaría en un factor de 20. Solo necesita 1ms y luego debe esperar 200ms. Nos interesa priorizar I/O bound para que no espere tanto.

Observació 5.10.

1. Round robin no es buena solución si hay tareas CPU bound y I/O bound.
2. Adecuado si todas las tareas son del mismo tipo, en especial CPU bound.
3. Las tareas con gran volumen de E/S generalmente necesitan poco tiempo de CPU para realizar la siguiente operación E/S. Una vez emitida la operación, se realiza un cambio de contexto a otra tarea.
4. En un editor de texto necesita pocos ms para mostrar por pantalla el valor de una tecla pulsada. Con una rebana de tiempo de 100ms, el editor de texto debería esperar múltiples rebanadas de tiempo para ejecutar y mostrar el resultado de sucesivas teclas.

5.2 Max-min fairness

El algoritmo max-min fairness se desarrollo inicialmente para la planificación de envío de paquetes por la red. Fue adoptado después para la planificación de tareas. La idea es asignar a cada instante de tiempo la CPU a una tarea Preparada que ha recibido menos tiempo de CPU.

- A las tareas "pequeñas" se les asigna el tiempo que solicitan y reparten el resto del tiempo entre las tareas "grandes".
- Los recursos son asignados en orden creciente de demanda y ninguna tarea tiene más recursos del que necesita.
- Si hay mezcla de tareas (CPU y I/O bound) resulta en una asignación más justa.
- Si solo hubieran tareas de computación seria sencillo, ya que todas las tareas obtienen el mismo tiempo de CPU (*round robin*).

En la práctica el max-min fairness se implementa (idealmente) cada vez que el planificador puede escoger una tarea a ejecutar, escoge aquella que tiene hasta ese momento el menos tiempo acumulado de CPU. Requiere de una cola de prioridad (¡alto coste computacional!).

5.3 Multi-level feedback queue

1. Las tareas de una cola del mismo nivel son planificadas de acuerdo a un round robin. Una tarea de prioridad alta se ejecuta antes que una de prioridad más baja.
2. Las tareas se mueven por el planificador entre colas de prioridad para obtener equitatividad.
3. Si una tarea consume su rebanada de tiempo (sin bloquearse), se mueve a una cola de prioridad más baja.
4. Si se bloquea se incrementa la prioridad.

5.4 Contextos de planificació

1. No existe un planificador optimo general.

2. Cada uno tiene sus ventajas e inconvenientes.
3. El objetivo de este tema es ver algoritmos que funcionan bien en entornos con carga variada, que utilizan los ordenadores personales: **Planificación uniprocesador**, **Planificación multiprocesador**, **Planificación de ejecución en tiempo real**, **Planificación en sistemas sobrecargados**.

Definició 5.11 (Planificació multiprocessador**).** En SOs Win, Mac y Linux, cada CPU tiene su propia estructura MFQ, y por tanto, sus propias tareas. Una tarea se ejecuta siempre en el mismo procesador para aprovechar los datos almacenados en la memoria caché. Si un procesador se queda sin nada a ejecutar, puede "robarle" una tarea a otro procesador.

Definició 5.12 (Planificació en temps real**).** En determinados sistemas, el planificador tiene que asegurarse que determinadas tareas tienen plazos para ejecutarse. Existen diversos formas para intentar asegurarnos que se cumplen los plazos (¡aunque no hay seguridad al 100%!):

1. **Priorización por plazos**: Las tareas de tiempo real con plazos más cortos tienen una prioridad más alta y se les asignan recursos de forma preferente. Esta técnica funciona si las tareas son computacionales (CPU bound).
2. **Preempción**: Las tareas pueden ser interrumpidos en cualquier momento por tareas de mayor prioridad. Necesario para garantizar que las tareas críticas se ejecuten en el momento adecuado.

Definició 5.13 (Planificació en sistemes sobrecarregats**).** Una solución sencilla seria rechazar las peticiones de conexión. Otra posibilidad para gestionar la sobrecarga es reducir el tiempo de respuesta para determinadas peticiones. Sin embargo, muchos sistemas hacen lo contrario: realizan más trabajo para tareas a medida que aumenta la carga.

6

Concurrència

Como programadores y programadoras, estamos acostumbrados a pensar de forma secuencial: Se ejecuta la función principal (main en C) y después se ejecutan las instrucciones del programa de manera secuencial.

Definició 6.1 (Concurrència**).** *Concurrencia* se refiere a la habilidad de distintas partes de un programa, algoritmo, o problema de ser ejecutado en desorden o en orden parcial, sin afectar el resultado final. Los cálculos (operaciones) pueden ser ejecutados en múltiples procesadores, o ejecutados en procesadores separados físicamente o virtualmente en distintos hilos de ejecución.

La concurrencia es una forma de estucturar una solución que puede ser paralelizable.

1. `fork()`: Procesos independientes se comunican entre sí para reparetirse el trabajo.

2. Los SOs actuales nos ofrecen herramientas para que un determinado proceso puede ejecutar diversas partes del código al mismo tiempo. De aquí sale el concepto de *hilo*.
3. El SO nos permite gestionar la concurrencia a través de multiples procesos o multiples hilos.

Definició 6.2 (Thread**).** Unidad básica de ejecución dentro de un proceso.

1. ID de hilo, contador de programa (PC), un conjunto de registros y una pila.
2. Los hilos del mismo proceso comparten su sección de código, datos y otros recursos del SO (ficheros abiertos y señales).
3. Un proceso tradicional tiene un único hilo de control. Si un proceso tiene varios hilos de control, puede realizar más de una tarea a la vez.

Hasta ahora hemos considerado que un proceso se ejecuta en un único hilo (thread). SOs modernos permiten a los procesos ser ejecutados utilizando multiples hilos (multithreading). La programación multihilo es muy popular hoy en dia donde nos encontramos con sistemas multicore que nos ofrencen las CPUs.

Definició 6.3 (Multicore**).** Sistemas con múltiples núcleos de computación en un solo chip de procesamiento, dónde cada núcleo aparece como una CPU separada para el SO.

Exemple 6.4 (Processos multithread**).**

- Navegador internet. Hilo 1: muestra imágenes/texto. Hilo 2: recoge datos de la red
- Navegador de ficheros. Cada hilo puede generar el thumbnail de cada imagen por separado.
- Procesador de textos. Hilo 1: Muestra gráficos. Hilo 2: Leer información de teclado. Hilo 3: Revisión ortográfica y gramatical.

Observació 6.5 (Concurrència amb processos**).** Cada proceso tiene un espacio de memoria independiente. Por defecto, no pueden escribir en el espacio de memoria de otro proceso. Para comunicarse entre sí hace falta utilizar los servicios del SO (fichero compartido, tuberia, red...).

Observació 6.6 (Concurrència amb threads**).** Los hilos de un proceso comparten el espacio de memoria del proceso. Cada hilo tiene su propia pila y registros de la CPU y puede ejecutar una parte diferente del código. Para comunicarse entre si pueden utilizar el espacio de memoria que comparten.

elements d'un procés	elements d'un fil
espai de direccions	comptador de programa
fitxers oberts	registres de la CPU
llista de fils	pila
altres	estat (ready, block, exec)

En la actualidad, el SO realiza la planificación con los hilos. Cada hilo tiene un estado diferente y el planificador realiza el **cambio de contexto a nivel de hilo**. Cada hilo puede estar ejecutándo una parte diferente del código del proceso. Incluso el mismo codigo puede ser ejecutado por diferentes hilos al mismo tiempo.

Observació 6.7 (Beneficis de la programació multithread**).**

1. Típicamente crear un hilo es menos costoso (tiempo, memoria) que un hijo.
2. Mejora el tiempo de respuesta de las aplicaciones. Cambio de contextos entre hilos es normalmente más rápido que entre procesos.
3. Mejor gestión de recursos. Se comparten códigos y datos, utilizando el mismo espacio de direcciones (más facil que comunicación de procesos).
4. Simplificación del código para tratar eventos asíncronos (ratón, teclado, red).
5. Grandes beneficios sobre todo en arquitecturas multiprocesos, donde hilos pueden ejecutarse en paralelos en diferentes cores.
6. Las ventajas de la programación multihilos son evidentes también con sistemas uniprocesadores.

processos	**threads**
La comunicación puede ser entre procesos, mismo o diferente PC	Todos los hilos de un proceso se ejecutan en el mismo ordenador.
La comunicación se realiza mediante servicios que ofrece el SO	A través del espacio de memoria del proceso
	La complejidad se centra en la sincronización y coordinación de los hilos

¿Procesos o hilos? ¿Qué opción es mejor? Si la decision depende de la *cantidad de datos a compartir*, utilizar **hilos** si hay muchos datos a compartir y utilizar **procesos** si hay pocos datos a compartir. Si depende de la **seguridad a implementar**, si un **hilo** realiza una operación inválida, el SO mata *todo el proceso* (con todos sus hilos). Si un **proceso** realiza una operación inválida en memoria, el SO mata el **proceso** que ha realizado la operación inválida, pero no el resto.
Es necesario sincronizar los hilos/procesos entre sí cuando se accede a recursos (variables) compartidas.

Definició 6.8 (Secció crítica**).** Parte del código de un programa que accede a un recurso compartido, como una variable o un archivo, que puede ser modificado por otros procesos o hilos concurrentes.

Definició 6.9 (Exclusió mútua**).** Hilos y/o procesos diferentes no pueden acceder a la misma sección crítica en el mismo instante de tiempo.

Observació 6.10. Un sistema concurrente admite más de una tarea al permitir que todas las tareas progresen. Por el contrario, un sistema paralelo puede realizar más de una tarea simultáneamente. Por lo tanto, **es posible tener concurrencia sin paralelismo**.

7

Memòria virtual

Definició 7.1 (Memòria RAM**).** Es un tipo de memoria de acceso aleatorio que se utiliza en las computadoras para almacenar temporalmente los datos y programas que están en uso. La RAM se utiliza para almacenar

los datos que el procesador necesita para ejecutar las aplicaciones y realizar las tareas del SO. Cuando se inicia un proceso, este se carga en la RAM para que el procesador pueda acceder rápidamentme a los datos que necesita para ejecutarlo.

Todos los procesos creen que tienen todos los recursos disponibles (p. ej. memoria) y que ésta se encuentra almacenada de forma contigua. *¿Las direcciones de memoria corresponden a direcciones localizadas en la memoria física de la máquina?*

- Un puntero que apunta a una dirección de memoria de la RAM donde se almacena una variable.
- El registro del contador de programa (Program Counter; PC) apunta a la dirección RAM que se está ejecutando.

Definició 7.2 (Entorn de memòria virtual**).** Un entorno de memoria virtual es una técnica utilizada por los SOs modernos para permitir que un proceso tenga acceso a una cantidad de memoria mayor que la cantidad física de RAM disponible en el sistema.

1. El SO divide la memoria en bloques más pequeños llamados páginas, y utiliza una colaboración entre el hardware y software para asignar y administrar estas páginas de memoria.
2. El hardware y el SO se encargan de gestionar un sistema que traduce cualquier dirección virtual que genera un proceso a una dirección física de la memoria RAM.

Observació 7.3 (Sistema de traducció de direccions**).**

1. Aislamiento de procesos/protección de procesos entre sí. Permite además crear “sandboxes” para ejecutar aplicaciones de terceros.
2. Autoaislamiento de procesos. Las diferentes partes de un código están protegidas entre si. Por ejemplo, la zona de código ejecutable no se puede ejecutar por el mismo proceso.
3. Ejecución de procesos. Un proceso puede ejecutarse sin tener todo el código ejecutable cargado en la memoria RAM.
4. Comunicación entre procesos. Permite tener una zona de memoria compartida entre procesos para que puedan compartir datos.
5. Gestión de memoria de la pila/memoria dinámica. El SO ubica memoria para estas zonas a medida que crecen.
6. Ficheros mapeados a memoria. Se puede acceder a un fichero accediendo a posiciones de memoria como si fuera un vector, sin utilizar las llamadas al sistema read o write.

Todas las direcciones de memoria generadas por los procesos son traducidas a una dirección física. Esta traducción (dirección virtual-física) se realiza a nivel de hardware: Memory Mapping Unit (MMU).

1. El SO se encarga de gestionar y configurar este hardware para realizar esta tarea de traducción de forma correcta.

2. Al traducir la dirección, se comprueba si la dirección virtual esta mapeada a una dirección física. En caso contrario se produce una excepción.
3. El SO comprueba si la dirección virtual pertenece al proceso.
    - Si no pertenece al proceso, el SO lo mata. Le envía una señal SIGSEGV: "segmentation fault" por acceso a memoria no asignada.
    - Si pertenece al proceso, el SO realiza las gestiones necesarias para que la dirección virtual se mapee a una dirección física para que el proceso pueda acceder al dato solicitado.

Definició 7.4 (MMU**).** La unidad de gestión de memoria (MMU) es un dispositivo de hardware responsable del manejo de los accesos a la memoria por parte de la CPU. Funciones (entre otras): traducción de las direcciones lógicas (o virtuales) a direcciones físicas (o reales), la protección de la memoria, el control de caché, etc. Cuando la CPU intenta acceder a una dirección de memoria lógica, la MMU realiza una búsqueda en una memoria caché especial llamada Buffer de Traducción Adelantada (TLB, Translation Lookaside Buffer), que mantiene la parte de la tabla de páginas usada hace menos tiempo.

7.1 Base i límit

Uno de los esquemas más básicos de traducción de direcciones utiliza dos registros: base y límite. Estos registros solo pueden ser modificados por instrucciones privilegiadas. Cada proceso tiene sus propios registros base y límite.
Cada vez que se accede a la memoria, se suma la base a la dirección. Si se supera el límite, se produce una excepción.

Observació 7.5 (Defectes del base i límit**).**

1. Pila y memoria dinámica no expansibles. Con solo 2 registros, hay que hacer una buena previsión de la memoria que ocupará un proceso a la hora de cargarlo en disco.
2. No fragmentación de la memoria. Este esquema solo permite utilizar espacios de direcciones contiguas y no se puede fragmentar.
3. No aislamiento de los procesos. Un proceso puede sobrescribir la zona de código ejecutable.
4. No memoria compartida. Este esquema no permite que varios procesos compartan zona de memoria (p.ej. Instrucciones máquina).
5. Dificulta la comunicación entre procesos. Varios procesos no se pueden comunicar entre sí a través de una zona de memoria.

7.2 Memòria segmentada

En vez de tener un par de registros (base y límite) por proceso, el hardware da soporte a múltiples pares de registros base-límite para cada proceso. A este esquema de gestión de memoria lo llamaremos *segmentación*.

Definició 7.6 (Segment**).** Cada par de registros base-límite tiene asociada una porción del espacio de direcciones llamado segmento. Cada segmento se almacena de forma contigua en la memoria física y puede tener un tamaño variable.

1. El número de segmento (la base y límite asociado).
2. Offset dentro del segmento.

En una dirección virtual, los bits altos están asociados al número de segmento y los bits bajos al offset. El número de segmentos total posible depende del número de bits altos asignados al representar el segmento. A nivel de hardware se pueden asignar diferentes permisos de acceso (lectura, escritura, ejecución) a cada segmento. El SO se encarga de gestionarlo. Si un proceso produce una dirección de memoria virtual inválida (que no pertenece al proceso), se produce una excepción y el SO genera una señal `SIGSEGV` que se envía al proceso.

Observació 7.7 (Avantatges**).**

- Autoaislamiento. Permite que un proceso se proteja a si mismo en diferentes partes del código (segmentos diferentes).
- Fragmentación de memoria. El espacio de memoria de un proceso puede estar fragmentado en diversas secciones. Estos segmentos están asociados a regiones grandes (segmento asociado al código, pila, o memoria dinámica.
- Compartición de segmentos. Varios procesos pueden compartir código (librerías, etc) al compartir los registros base y límite de un segmento.
- Comunicación entre procesos. Diversos procesos pueden comunicarse entre sí compartiendo zona de memoria (segmento).

Exemple 7.8 (Desavantatges de la memòria segmentada**).**

1. A medida que pasa el tiempo, la memoria física se divide en trozos ocupados y libres.
2. Al crear un nuevo proceso necesitamos crear nuevos segmentos. Es posible que no haya ninguna zona libre suficientemente grande para un segmento. Pero si sumamos las regiones libres si que podría haber una zona libre suficientemente grande para el segmento.
3. En el caso anterior, el SO puede compactar las regiones para unificar las regiones libres. Es necesario cambiar los registros base y límite para cada segmento y mover los datos a la memoria física. Las direcciones virtuales no cambian en ningún momento.
4. La operación de compactación es costosa. Un ordenador podría tardar un segundo en realizar este proceso.

7.3 Mmeòria paginada

Esquema de gestión de memoria utilizada en los SO modernos para traducir direcciones virtuales a direcciones físicas de memoria. La memoria física se divide en páginas de tamaño fijo (marcos de página) y la traducción de direcciones se realiza mediante el uso de tablas de páginas. Cada proceso tiene su propia tabla y la gestionan el SO.

- Los marcos de página son de tamaño fijo y potencia de 2.
- Utilizan los bits altos de la dirección como índice de la tabla y el offset indica la posición dentro del marco.
- La traducción se realiza haciendo una sustitución de los bits altos de la dirección virtual por los bits indicados en la tabla.

Exemple 7.9. Cada entrada en la tabla contiene la traducción al marco físico así como información sobre las propiedades asociadas (lectura, escritura, ejecución, si está disponible o no,...). Si un proceso realiza una operación inválida, se produce una excepción. Cada proceso cree que tiene disponible todo el espacio de memoria (virtual) disponible:

- En sistemas de 32 bits son 4GB.
- En arquitecturas de 64 bits actuales sólo se "ven" 48 bits por motivos electrónicos (También usan marcos de página de 4 kb).

El SO se encarga de gestionar las tablas de los procesos. Las tablas se almacenan en memoria RAM.

Definició 7.10 (Copy-On-Write (COW)**).** Técnica de optimización de la gestión de memoria que se utiliza en SOs para reducir la cantidad de memoria física utilizada por procesos (p.ej. padre-hijo) que comparten páginas de memoria idénticas (modo lectura). La idea es postergar la copia de páginas de memoria idénticas hasta que se intente modificar una de las páginas compartidas. Solo se copian aquellas páginas que se modifican. El resto son compartidas entre los procesos.

- El código máquina se comparte entre padre e hijo (ya que no se modifica).
- Por lo general, compartir zonas de memoria (librerías, código ejecutable, ...) es sencillo con memoria paginada.
- Si después del fork se hace un exec, la tabla de páginas se adapta al nuevo ejecutable.

Exemple 7.11 (Què passa al fer un *malloc*).** La función malloc solo inicializa el mapa de memoria virtual. La asignación de marcos de página física se realiza a medida que accedemos.

1. Si la dirección no existe al espacio físico se produce un fallo de página.
2. El SO busca una página disponible al espacio físico.
3. Se vuelve del fallo y se realiza el acceso.

En el código `exemple_malloc`, al llamar a la función `malloc` solo se realiza la reserva en el espacio virtual, no en el espacio físico. El SO asigna espacio físico al acceder a una determinada posición del vector.

Observació 7.12 (Avantatges memòria paginada**).**

1. La memoria paginada resuelve el principal problema de la memoria segmentada: encontrar zonas libres de memoria física.
2. El SO mantiene una tabla con los marcos de páginas libres.
3. La memoria física es limitada. Puede llegar un momento en el que todos los marcos de páginas físicas estén ocupados. Si un proceso requiere de nuevos marcos, el SO decide que marcos descarta del espacio de memoria física.
4. El SO utiliza un algoritmo para decidir que marcos descartar.
    - First-In-First-Out (FIFO)
    - Least Recently Used (LRU)
    - Least Frequently Used (LFU)
    - Most Recently Used (MRU)
5. Estos marcos pueden pertenecer a un proceso diferente del que se está ejecutando. Veremos los principios cuando hablemos de "archivos mapeados a memoria".

7.4 Traducció de direccions

En arquitecturas 16, 32 y 64 bits, la principal diferencia es la cantidad de bits que se utilizan para representar una dirección de memoria.

Observació 7.13. A tener en cuenta:

1. En un sistema x86 de 32 bits los marcos de página físicos tienen tamaño de $212 = 4096$ bytes. Eso quiere decir que la tabla tiene $232 - 12 = 220 = 1048576$ entradas. Suponemos que cada entrada ocupa 4 bytes (traducción y permisos), significa que una tabla ocupa 4 Mbytes para cada proceso. Hay una tabla para cada proceso.
2. ¿Qué ocurre en un sistema de 64 bits? Cada tabla ocuparía muchos Gbytes. Podríamos aumentar el tamaño del marco físico (en la actualidad continua siendo 4 Kbytes) pero malbarataríamos memoria RAM.

¿Cómo se almacena entonces la tabla de memoria? La solución en sistemas de 64 bits es que no utiliza una tabla!

- Utiliza esquemas de **traducción multinivel** (p. ej. árbol, tabla de hash, etc.). En vez de utilizar una tabla para la traducción, se utilizan varias tablas de páginas para dividir el espacio de direcciones virtuales en varios niveles.

- Cada tabla de páginas en un esquema multinivel contiene un subconjunto de las entradas de la tabla de páginas, y se utiliza para traducir una parte específica del espacio de direcciones virtuales.
- El SO utiliza los bits altos para selecciona tabla de páginas de nivel superior y un parte de la dirección virtual para seleccionar tabla nivel inferior.

Observació 7.14 (Avantatges traducció multinivell**).**

1. Reduce la cantidad de memoria necesaria para almacenar las tablas de páginas.
2. Mejora la eficiencia del proceso de gestión de memoria virtual al permitir la compartición de tablas de paginas entre procesos.

7.4.1 Esquemes d'arbre: Paginació segmentada

Árbol de dos niveles:

1. El segmento apunta a una tabla.
2. Marcos de páginas físicas.

Las propiedades de acceso (lectura, escritura, ejecución) se pueden estableces a nivel de segmento.

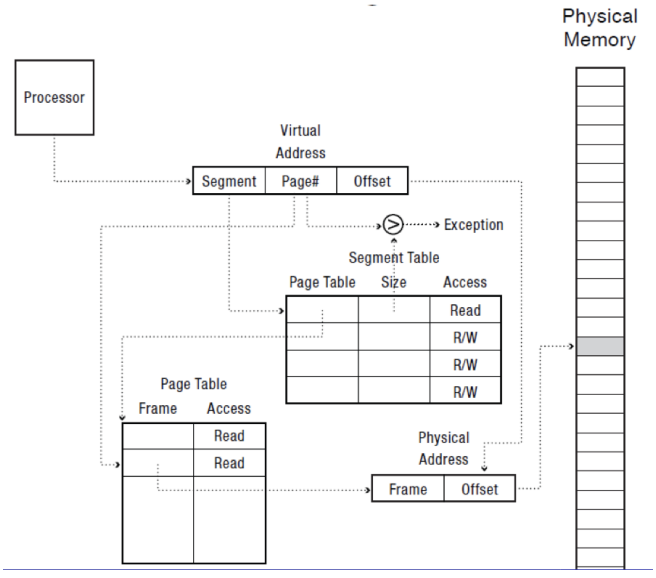

Figura 1: Paginació segmentada.

7.4.2 Esquemes d'arbre: Paginació multinivell

Árbol de muchos niveles. Solo se ubican las partes del árbol que realmente son utilizadas por el proceso.

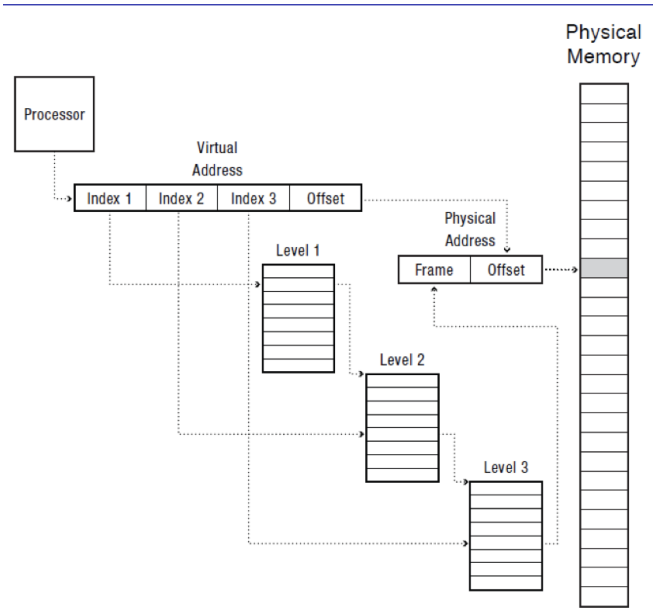

Figura 2: Paginació multinivell.

7.4.3 Esquemes d'arbre: Paginació multinivell segmentada

Se pueden combinar los dos esquemas anteriores para obtener un esquema de paginación multinivel segmentada: cada segmento contiene una tabla multinivel. Es el esquema utilizado por los sistemas x86, tanto por 32 como por 64 bits. Por cuestiones históricas, el número de "segmento" se almacena en un registro de CPU separado (actualmente no se utiliza).

- En sistemas de 32 bits, se utiliza un esquema multinivel de dos niveles: 10 bits en el primer nivel, 10 en el segundo y 12 por el offset.
- En sistemas de 64 bits, actualmente, sólo se utilizan 48 bits del espacio de direcciones virtual (Terabytes). Se utilizan 4 niveles de árbol (48 = 9+9+9+9 bits índices + 12 bits offset). Sólo se ubican las partes de el árbol que realmente son utilizadas por el proceso.

7.5 Arxius mapejats a memòria

El SO permite manipular archivos (como si fueran un vector) mapeándolos directamente a memoria. Cualquier archivo de disco puede ser manipulado utilizando el espacio de direcciones virtuales.

Definició 7.15 (Bloc o sector del disc**).** Mínima unidad que el disco puede leer o escribir. El SO guarda a memoria los bloques de disco si:

1. son referenciados.
2. si son guardados en disco automáticamente.
3. al salir de la aplicación o si la página mapeada a memoria física es liberada por el SO.

Definició 7.16 (Page fault**).** En un sistema de memoria virtual paginada, un fallo de página (page fault) es una excepción arrojada cuando un proceso cuando intenta acceder a una página que se encuentra en la memoria virtual pero no está actualmente cargada en la memoria física (RAM). Esto significa que la página solicitada se encuentra en el almacenamiento secundario (p.ej. disco duro) en lugar de la memoria física (RAM). Es importante tener en cuenta que los fallos de página son una parte normal del funcionamiento de la memoria virtual paginada.

Definició 7.17 (Demanding paging**).** Técnica utilizada en los SO como memoria virtual paginada para optimizar el uso de la memoria. Permite a los procesos **acceder a más memoria de la que físicamente está disponible** al cargar en memoria solo las páginas que se necesitan en un momento determinado. Las páginas de memoria física se pueden interpretar como una caché de los bloques de disco. Si una aplicación accede a una pagina no disponible de la RAM, el SO la lleva de forma trasparente (para el usuario) de disco a RAM.

Procés 7.18 (Com funciona**).**

1. *Al intentar acceder a una posición válida (pero que aún no esta cargada a memoria RAM) se produce un fallo (excepción) de página.*
2. *El SO captura la excepción de página, detecta la correspondiente dirección virtual válida y convierte la dirección virtual a una posición de bloque de disco.*
3. *El SO lee el bloque de disco. La lectura puede bloquear el proceso (si no esta en caché) y el SO realiza un cambio de contexto (podría ejecutar otro proceso).*
4. *Una vez leído el bloque de disco, se produce una interrupción de disco y avisa al SO (llamada a sistema).*
5. *El SO actualiza la tabla multinivel con una nueva página cargada. Es posible que el sistema haya expulsado (descartado) otra página cargada anteriormente.*
6. *El SO recupera la ejecución del proceso en el punto donde se produjo la excepción de página.*

Alguns detalls més:

- *Al expulsar una página de memoria, el SO podría tener que guardar esa página en disco. ¿Cómo sabe si lo ha de hacer? Eso lo sabe porqué el hardware guarda en la tabla un bit (dirty bit) que indica si la página ha sido modificada previamente.*
- *El SO decide qué página expulsar utilizando un algoritmo que utiliza el use bit, un bit de la tabla de páginas donde el hardware indica si el proceso ha accedido a la página (R,W). El SO recorre periódicamente las tablas para saber cuales han sido accedidas y hacer un reset de tabla.*
- *El SO puede descartar una página que no necesariamente pertenece al proceso que se está ejecutando en ese momento.*

7.6 Memòria virtual

Definició 7.19 (Memòria virtual**).** La memoria virtual es una generalización del concepto de archivos mapeados a memoria: todos los segmentos (o regiones) están mapeados a un fichero de disco. Eso es, **cada segmento del proceso es mapeado a disco**: ejecutable, librerías, variables globales, pila, memoria dinámica.

Exemple 7.20. Al cargarse un programa a disco, este se puede comenzar a ejecutar sin que este completamente cargado a memoria. El SO carga a RAM las páginas de disco a memoria a medida que se va ejecutando el programa.

1. El Sistema de memoria virtual nos ofrece flexibilidad: Podemos tener más procesos ejecutándose de los que caben en la memoria. El sistema de memoria virtual se encarga de gestionar las páginas asociadas a cada proceso.
2. El balanceo de cuantas páginas se asignan a cada proceso es crítico. Se debe reducir al máximo los fallos de página: un ordenador moderno puede gestionar unos 100 fallos de página por segundo y la CPU puede ejecutar miles de millones de instrucciones por segundo.
3. Al producirse un fallo de página, el sistema puede necesitar descartar una página. Lo hace basado en el use bit (se descartan las páginas que no han sido accedidas recientemente).

La memoria RAM es limitada y puede llegar a agotarse si hay demasiados procesos en ejecución o si un proceso consume mucha memoria.

Definició 7.21 (Swap**).** La Swap es un area del disco duro que utiliza el SO para almacenar temporalmente parte de la memoria RAM de un proceso cuando el sistema se queda sin memoria física diponible. Cuando un proceso necesita acceder a una página que se encuentra en la zona swap, el SO la recupera de la zona swap y la coloca de vuelta en la memoria RAM.

Observació 7.22.

- Los ejecutables o librerias dinamicas se mapean con el fichero original en modo lectura (y ejecución).
- Las zonas de la pila o memoria dinámica se mapean a la zona (partición) de swap para que se puedan almacenar aquellas páginas que no caben en la memoria. Se guardan las modificaciones de las páginas pero al salir del proceso se pierden estos datos.

Definició 7.23. La carpeta `/proc` no contiene ficheros convencionales (no representan datos almacenados en el disco), sino que son estructuras del SO que proporcionan información sobre el estado actual del sistema, los procesos en ejecución, los dispositivos y otros recursos del sistema. *En los SO basados en Linux, la carpeta* `/proc/<PID>/maps` contiene información sobre el espacio de direcciones de memoria de un proceso específico (PID).

Definició 7.24 (RSS). RSS (Resident Set Size). Cantidad de memoria física (RAM) utilizada actualmente por un proceso en particular (sin incluir el espacio de intercambio `swap`).

Definició 7.25 (VSZ). VSZ (Virtual Memory Size). Es el tamaño total de la memoria virtual utilizada por un proceso. Incluye tanto la memoria física (RSS) como la memoria paginada en el espacio de intercambio (`swap`) y cualquier otra memoria virtual asignada al proceso (bibliotecas compartidas, regiones de memoria mapeadas). El valor de VSZ suele ser mayor que el de RSS, ya que refleja tanto la memoria física como la memoria paginada en el sistema.

7.7 Conclusions

1. El SO gestiona una tabla (multinivel) para cada proceso que permite traducir una dirección virtual a una dirección física.
2. Esta tabla permite proteger las páginas de los procesos entre si.
3. La tabla permite también que diferentes procesos puedan compartir direcciones, p.ej. código, librerías, etc.
4. El sistema de memoria virtual gestiona que páginas se cargan a memoria física. Por ejemplo, de una librería muy grande solo se asigna memoria física a aquellas partes que se van necesitando.

A

Testos

Pregunta 1: ¿En qué situaciones se genera una interrupción en un sistema informático?

1. Cuando un programa accede a una dirección de memoria no válida.
2. Cuando un programa realiza una llamada a sistema.
3. Cuando un usuario ingresa un comando en la línea de comandos a la cual no tiene permisos para ejecutar.
4. **Cuando un dispositivo de hardware necesita atención del sistema operativo.**

Pregunta 2: ¿Cómo definirías una llamada a sistema en un sistema operativo?

1. Un proceso de cifrado de datos en la memoria del núcleo del sistema.
2. Una forma de actualizar el sistema operativo en tiempo real.
3. Un proceso de eliminación de procesos inactivos en el sistema.
4. **Un mecanismo por el cual un proceso solicita un servicio o recurso del sistema operativo.**

Pregunta 3: ¿Cuál es la diferencia principal entre el modo núcleo y el modo usuario en un sistema operativo?

1. El modo núcleo es más seguro que el modo núcleo porque no limita el acceso del sistema operativo a los recursos del hardware.
2. El modo usuario permite a los programas de usuario ejecutarse de forma limitada, mientras que el modo núcleo solo se ejecuta en un hardware virtualizado.
3. **El modo núcleo permite al sistema operativo acceder a todos los recursos del hardware, mientras que el modo usuario limita el acceso del sistema operativo a los recursos del hardware**.
4. El modo núcleo solo se utiliza para depurar errores del sistema operativo, mientras que el modo usuario es utilizado para la ejecución normal del sistema operativo y las aplicaciones.

Pregunta 4: ¿Cuál es la función principal del núcleo de un sistema operativo?

1. Ejecutar aplicaciones y programas guardados en la máquina.
2. Facilitar la conexión a internet de los procesos en ejecución.
3. **Controlar el acceso a los recursos del sistema**.
4. Almacenar archivos y datos del usuario en el disco duro.

Pregunta 5: ¿Qué ocurre cuando se produce una excepción en un sistema operativo?

1. El proceso en ejecución continúa sin interrupción, pero la excepción se registra para su posterior tratamiento.
2. El sistema operativo reinicia el equipo para evitar daños mayores.
3. El sistema operativo ignora la excepción y continúa con la ejecución normal del proceso.
4. **El sistema operativo interrumpe la ejecución normal del proceso y realiza una acción para manejar la excepción.**

Pregunta 6: ¿Cuál es la diferencia entre un núcleo monolítico y un núcleo microkernel?

1. Un núcleo microkernel es más rápido que un núcleo monolítico.
2. En un núcleo microkernel, todos los servicios del sistema se ejecutan en el mismo espacio de memoria, mientras que, en un núcleo monolítico, los servicios se ejecutan en espacios de memoria separados.
3. Un núcleo monolítico es más seguro que un núcleo microkernel.
4. **En un núcleo monolítico, todos los servicios del sistema se ejecutan en el mismo espacio de memoria, mientras que en un núcleo microkernel, los servicios se ejecutan en espacios de memoria separados**.

Pregunta 7: ¿Cuál es el número de descriptor de fichero para la entrada estándar (stdin), la salida estándar (stdout) y el flujo de error estándar (stderr) en sistemas operativos basados en Unix?

1. stdin: 1, stdout: 2, stderr: 0
2. stdin: 1, stdout: 2, stderr: 3
3. stdin: 1, stdout: 0, stderr: 2
4. **stdin: 0, stdout: 1, stderr: 2**

Pregunta 8: Las tuberías utilizan un búfer interno del sistema operativo para gestionar la comunicación entre procesos. ¿Qué ocurre cuando este búfer se encuentra vacío?

1. El proceso que realiza la llamada read espera a que el búfer este lleno.
2. El proceso que realiza la llamada write espera a que el búfer comience a llenarse.
3. **El proceso que realiza la llamada read espera a que el búfer comience a llenarse**.
4. El proceso que realiza la llamada write espera a que el búfer este lleno.

Pregunta 9: A diferencia de Windows que utiliza "CreateProcess", los sistemas UNIX genera nuevos procesos en dos pasos. ¿Cuál son las ventajas concepto fork y exec con respecto al "CreateProcess"?

1. **La creación de procesos en dos pasos permite aislar el proceso hijo del proceso padre, lo que significa que, si el proceso hijo falla, no afectará al proceso padre**.
2. El sistema fork y exec se diseñó en 1973 y actualmente se podría cambiar por un sistema más potente y útil.
3. La creación de procesos en dos pasos solo es útil para implementar tuberías y redirección.
4. El sistema fork y exec no permite comunicar procesos entre sí.

Pregunta 10: Por lo general, las tuberías son un tipo de sistema de comunicación que...

1. Permiten la ejecución de varios procesos de forma simultánea en un solo núcleo de CPU.
2. **Permiten la comunicación unidireccional entre procesos que tengan relación padre-hijo**.
3. Permiten la comunicación bidireccional entre procesos en diferentes sistemas operativos.
4. ~~Permiten la comunicación unidireccional entre procesos, aunque no tengan una relación padre-hijo~~.

Pregunta 11: ¿Cuáles son las características principales de las llamadas al sistema fork() i exec()?

1. Fork() reemplaza el espacio de direcciones del proceso actual con un nuevo programa, mientras que exec() crea un nuevo proceso a partir de un proceso padre.
2. Fork() crea un nuevo proceso a partir de un proceso padre mientras que exec() se utiliza únicamente para especificar los argumentos de ejecución que se le pasan al proceso hijo.
3. Exec() se utiliza siempre en combinación con fork().
4. **Fork() crea un nuevo proceso a partir de un proceso padre mientras que exec() reemplaza el espacio de direcciones del proceso actual con un nuevo programa**.

Pregunta 12: ¿Cuál es el estado de un proceso que está esperando a que se libere un recurso para continuar su ejecución?

1. Estado de ejecución (running).
2. Estado listo (ready).
3. Estado de finalización (terminated).
4. **Estado de espera (waiting)**.

Pregunta 13: ¿Qué diferencias existen entre escribir en un archivo regular con llamadas al sistema (write) y llamadas a la librería estándar (fwrite)?

1. La llamada a la librería estándar es particularmente útil si se desea realizar comunicación entre varios procesos.
2. **La librería estándar utiliza un búfer de usuario para almacenar los datos a escribir y hace la llamada al sistema en el momento en que este buffer se llena**.
3. La librería estándar no realiza llamadas al sistema y, por tanto, permite ser más eficiente que las llamadas al sistema.
4. No hay ninguna diferencia entre ambas funciones.

Pregunta 14: ¿Cuál de las siguientes afirmaciones es verdadera sobre los sockets como mecanismo de comunicación entre procesos?

1. Los sockets solo pueden utilizarse para la comunicación en el mismo equipo.
2. **Los sockets son un mecanismo de comunicación orientado a conexión.**
3. Los sockets son un mecanismo de comunicación únicamente sincrónico entre procesos.
4. Los sockets solo pueden ser utilizados por procesos en el mismo lenguaje de programación.

Pregunta 15: ¿Cuál de las siguientes afirmaciones es verdadera sobre las tuberías con nombre y las tuberías anónimas?

1. Las tuberías anónimas no permiten la comunicación entre procesos en diferentes.
2. **Las tuberías con nombre permiten la comunicación bidireccional entre procesos**.
3. Las tuberías anónimas son más eficientes que las tuberías con nombre.
4. Las tuberías con nombre son más seguras que las tuberías anónimas.

Pregunta 16: ¿Cuál es la sintaxis correcta para utilizar la llamada a sistema mmap y compartir un archivo con otro proceso?

1. `addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_FIXED, fd, 0)`
2. `addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);`
3. `addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, fd,`
4. `addr = mmap(NULL, size, PROT_NONE, MAP_SHARED | MAP_ANONYMOUS, fd, 0);`

Pregunta 17: ¿Qué es una señal en la comunicación entre procesos?

1. Un mecanismo mediante el cual un proceso puede solicitar al sistema operativo que realice una notificación asíncrona a otro proceso.
2. Un mecanismo de interrupciones de hardware.
3. **Todas las respuestas son correctas**.
4. Un mecanismo de interrupciones de software.

Pregunta 18: El mecanismo de archivos mapeados a memoria…

1. Permite comunicar procesos entre sí, siendo el sistema operativo quien gestiona qué partes deben estar en memoria RAM y cuáles en disco.
2. Permite que un proceso acceda al contenido de un archivo como si fuera un vector sin necesidad de utilizar explícitamente las funciones read y write.
3. Es el mecanismo que utiliza el SO para implementar el sistema de memoria virtual.
4. **Todas las opciones son correctas**.

Pregunta 19: ¿En cuál de los siguientes ejemplos decide el planificador de un sistema operativo pasar un proceso del estado de ejecución a bloqueado?

1. El proceso está en modo listo y realiza una llamada a la función sleep.
2. **El proceso está en modo de ejecución y realiza una operación de lectura de disco**.
3. El proceso que se está ejecutando ha agotado su rebanada de tiempo.
4. Un segundo proceso que está listo decide bloquear a otro proceso porque el segundo proceso tiene prioridad sobre el primero.

Pregunta 20: ¿Cuál de las siguientes afirmaciones es correcta sobre la planificación de tareas y procesos en un sistema operativo?

1. **Un proceso es una entidad activa que contiene información sobre su estado actual, mientras que una tarea es una unidad de trabajo que se puede asignar a un proceso**.
2. Una tarea es una unidad de trabajo que se puede asignar a un proceso, mientras que un proceso es una entidad pasiva que no puede ejecutarse de manera autónoma.
3. Un proceso es una unidad de trabajo que se puede asignar a una tarea, mientras que una tarea es una entidad pasiva que no puede ejecutarse de manera autónoma.
4. Una tarea es una entidad activa que contiene información sobre su estado actual, mientras que un proceso es una unidad de trabajo que se puede asignar a una tarea.

Pregunta 21: En el contexto del algoritmo de planificación multi-level feedback queue (MFQ), ¿cuál de las siguientes afirmaciones es correcta?

1. **MFQ permite que los procesos cambien de cola según su comportamiento de uso de la CPU**.
2. Los procesos que han utilizado más tiempo de CPU tienen un valor de prioridad más alto y se planifican antes que los procesos que han utilizado menos tiempo de CPU.
3. MFQ utiliza una única cola en la que todos los procesos son colocados y planificados según su tiempo de llegada.
4. En sistemas multiprocesadores, MFQ no permite la ejecución de procesos en paralelo.

Pregunta 22: ¿Cuál es el objetivo principal de la planificación de procesos en un sistema operativo?

1. Minimizar el número de procesos activos en el sistema.
2. **Minimizar el tiempo de espera de los procesos**.
3. ~~Maximizar la utilización de la CPU.~~
4. Maximizar el tiempo de ejecución de los procesos.

Pregunta 23: ¿En cuál de los siguientes ejemplos decide el planificador de un sistema operativo pasar un proceso del estado de ejecución a listo/preparado?

1. El proceso que se está ejecutando ha agotado su rebanada de tiempo.
2. El proceso está en modo de ejecución y realiza una operación de lectura de disco.
3. El proceso está en modo de ejecución y realiza una llamada a la función sleep.
4. Un segundo proceso que está listo decide bloquear a otro proceso porque el segundo proceso tiene prioridad sobre el primero.

Pregunta 24: ¿Cuál es el propósito del cambio de contexto en un sistema operativo?

1. Permitir que los procesos en espera sean ejecutados en la CPU.
2. Permitir que los procesos en ejecución sean detenidos en cualquier momento.
3. Permitir que el sistema operativo se reinicie después de un fallo.
4. **Permitir que el sistema operativo pueda cambiar de un proceso a otro de manera eficiente**.

Pregunta 25: Cuando se llama a la función fork() en un sistema operativo para crear un nuevo proceso hijo, ¿qué sucede con la memoria del proceso hijo en relación con el proceso padre?

1. El proceso hijo tiene una memoria completamente independiente del proceso padre desde el principio.
2. El proceso hijo no tiene acceso a la memoria reservada por el proceso padre.
3. **Inicialmente, la memoria del proceso hijo es una copia exacta de la memoria del proceso padre**.
4. La memoria del proceso hijo es compartida con el proceso padre y cualquier cambio realizado por uno de los procesos siempre se refleja en el otro.

Pregunta 26: ¿Cúal de las siguientes afirmaciones describe mejor la memoria virtual en un sistema operativo?

1. **Es un mecanismo que amplía la capacidad de la memoria física utilizando almacenamiento secundario**.
2. Es un componente de hardware utilizado para traducir direcciones de memoria.
3. Es una técnica que permite el acceso directo a la memoria física.
4. Es un mecanismo que amplía la capacidad de la memoria virtual utilizando almacenamiento secundario.

Pregunta 27: Cuando reservamos memoria mediante un malloc. ¿Qué ocurre?

1. Se reserva memoria en el espacio virtual y se asignan todos los marcos que pueda utilizar este vector.
2. Se reserva memoria en el espacio físico.
3. **Se reserva memoria en el espacio virtual y se asignan los marcos a medida que accedamos a este vector**.
4. Se reserva memoria en el espacio de memoria swap.

Pregunta 28: Supongamos que se produce una excepción por fallo de página y que la dirección virtual no pertenece al proceso. ¿Qué acción se debe tomar en este caso?

1. Realizar una copia de la página correspondiente desde el archivo de intercambio al marco de página libre.
2. El sistema operativo cambia el estado del proceso a bloqueado.
3. El sistema operativo (por defecto) mapea la dirección virtual a una nueva dirección física libre.
4. **El sistema operativo (por defecto) mata el proceso y realiza un cambio de contexto para ejecutar otro proceso**.

Pregunta 29: En un sistema de memoria virtual, ¿qué ventajas tiene un esquema multinivel con respecto a un esquema que no lo es (tabla plana)?

1. Un esquema multinivel permite una mayor flexibilidad en la asignación de memoria.
2. Un esquema multinivel mejora el rendimiento del sistema operativo al reducir el número de operaciones de E/S.
3. **Un esquema multinivel reduce el tamaño de la tabla de páginas y, por lo tanto, el uso de memoria**.
4. ~~Todas las opciones son correctas.~~

Pregunta 30: ¿Qué es la traducción de direcciones en memoria virtual?

1. **El proceso de convertir direcciones lógicas en direcciones físicas**.
2. La técnica utilizada para copiar datos de una ubicación de memoria a otra.
3. La técnica utilizada para administrar archivos de paginación en el almacenamiento secundario.
4. El proceso de convertir direcciones físicas en direcciones lógicas.