Sistemas Operativos

Victor Franco Matzkin - Darién Julián Ramírez

Índice

1. Introducción a los Sistemas Operativos

1.1. Sistema Operativo

Capa de software que se encarga de los recursos de la computadora para sus usuarios y aplicaciones. En sistemas de propósito general, los usuarios interactúan con las aplicaciones, las aplicaciones se ejecutan en un ambiente proporcionado por el SO y el SO controla el acceso al hardware.

¿Qué se necesita para que un SO pueda correr programas?

1.2. Roles de un SO

- *Árbitro:* manejan recursos compartidos entre diferentes aplicaciones corriendo en la misma máquina (físicamente). Los SO aíslan las aplicaciones entre si de manera que un error en una no afecte a la otra, deben protegerse a si mismos y a las aplicaciones de virus, y al compartir recursos, deben decidir qué aplicaciones los obtienen.

  Compartir recursos: (objetivos para SO)

  - *Alocar recursos:* el SO debe mantener las aplicaciones en simultáneo por separado, asignando recursos cuando sea necesario. Se debe proveer un multiplexado de recursos de manera de que la ejecución en un programa (que eventualmente requiera tiempo de procesamiento infinito) no impida el curso de las demás. Adicionalmente el usuario podría detener la ejecución de un programa que no responde.
  - *Aislamiento:* un error en una aplicación no puede interrumpir la ejecución de otros o del sistema operativo. Esto se denomina *fault insolation* y requiere restringir el comportamiento de las aplicaciones de manera que no tengan control total del hardware.
  - *Comunicación:* proporcionar comunicación entre diferentes aplicaciones y usuarios. Se deben establecer límites que podrán ser cruzados de forma controlada por el SO.

- *Ilusionista:* los SO proporcionan una abstracción física para simplificar el diseño de las aplicaciones (por ejemplo, haciendo creer que la memoria es infinita, o que el microprocesador está dedicado enteramente). Esto permite que las aplicaciones sean creadas con un nivel de abstracción mayor y sea el SO el encargado de asignar o gestionar los recursos.

  Enmascarar limitaciones de hardware: el hardware está limitado físicamente por lo que el SO debe decidir como dividir los recursos entre las diferentes aplicaciones en simultaneo. Un ejemplo es el uso del microprocesador, debiéndose proveer un método de virtualización (hacer creer a las aplicaciones que se dispone de recursos que no están presentes) y atomicidad (asegurar que las aplicaciones se realizan o no, y no quedar a medias).

- *Glue:* conjunto de servicios entre aplicaciones para facilitar una apariencia o experiencia de usuario común (como copiar y pegar). Además los SO proporcionan una capa que separa las aplicaciones del hardware (para independizar aplicaciones de hardware en particular).

  Servicios en común: el SO debe proporcionar servicios para facilitar el intercambio entre aplicaciones (por ejemplo, un servidor web debe ser capaz de leer un archivo escrito por el editor de texto) dependiendo de éste cuáles implementar (por ejemplo, las PC's pueden venir configuradas para funcionar con varios dispositivos diferentes -interfaces de red, discos, sensores, etc- y se usa una interfaz común pero en otras aplicaciones se podrá necesitar un nivel de abstracción menor).

  Otro servicio estándar es la librería de la interfaz de usuario. Esto facilita a tener una apariencia similar en común, de manera que operaciones comunes (copiar y pegar) se manejen de forma consistente.

1.3. Patrones de diseño del SO

Existen sistemas que si bien poseen otro dominio, deben cumplir tareas similares a los sistemas operativos (junto a las abstracciones):

- *Cloud computing:* modelo a gran escala donde las aplicaciones se ejecutan en la nube (computadoras compartidas).

- *Navegadores:* muchas páginas web poseen *scripts* (programas) que deben ser ejecutados por el navegador, por lo que éste debe aislar al usuario, a otros sitios y hasta al mismo navegador de actividad maliciosa (esto aumenta en el uso de extensiones).
- *Reproductores multimedia:* proveen un entorno de ejecución para programas en *scripts*.
- *Sistemas de bases de datos multiusuario:* deben aislar los datos que no pueden ser vistos por determinados usuarios, que la información sea consistente, etc.
- *Aplicaciones paralelas:* éstas imponen un sistema en tiempo de ejecución sobre el SO para proporcionar paralelismo.
- *Internet.*

Todos estos sistemas deben proveer los tres roles de una forma en particular para funcionar.

1.4. Desafíos de los SO

- *Confiabilidad:* el sistema hace lo que tiene que hacer, si hay errores el SO puede romperse o detenerse y perder la información, o gracias al aislamiento, la aplicación puede fallar ella sola y el SO manejar su reinicio. Este objetivo es difícil de lograr porque hay aplicaciones que se aprovechan de errores de implementación para tomar el control de sistema, por lo que se consigue mediante *testing*.
- *Disponibilidad:* es el porcentaje de tiempo en el que el SO es usable. Está afectado por dos factores: frecuencia de fallas (MTTF), y el tiempo que tarda el sistema en reiniciarse luego de una falla (MTTR).
- *Seguridad:* es la propiedad de que el funcionamiento de la PC no pueda ser comprometido por un atacante malintencionado. Un SO debe minimizar su vulnerabilidad de ser atacado, pero también las aplicaciones deben seguir criterios de seguridad. El SO se asegura de que ciertas acciones estén permitidas a través del *enforcement* (aplicación).
- *Privacidad:* es una parte de la seguridad, que establece que los datos almacenados sean accesibles por usuarios autorizados. Un política de seguridad define qué está permitido (quién puede acceder a cuáles datos y quién puede realizar cuáles operaciones).
- *Portabilidad:* una abstracción portable de un SO es una que no cambia cuando cambia el hardware (no volver a reinventar las aplicaciones o el SO), es decir, los SO's deben soportar aplicaciones que aún no han sido escritas para correr en hardaware todavía no desarrollado. Consiste en una forma simple y estándar de que las aplicaciones interactúen con el SO mediante una interfaz abstracta. La *interfaz de máquina abstracta* (AMI) es la interfaz entre el sistema operativo y las aplicaciones. La *interfaz de programación de aplicaciones* (API) es una parte de la AMI que es una lista de funciones que el SO da a las aplicaciones AMI, además incluye un modelo de acceso a la memoria y cuáles instrucciones pueden ser ejecutadas.
  El concepto de portabilidad se extiende al mismo SO: éste puede ser implementado independientemente de los detalles de hardware. Esta interfaz se denomina *capa de abstracción de hardware* (HAL).
- *Rendimiento:* el SO tiene un gran impacto en el rendimiento percibido de las aplicaciones porque éste decide qué y cuántos recursos va a utilizar. Una forma de medir el rendimiento es mediante la eficiencia (o inversamente el *overhead* -el costo de recursos agregado por implementar una abstracción-), es ver el grado con el cual la abstracción impide el rendimiento de la aplicación.
- *Equidad (fairness):* es establecer una división equitativa de los recursos.
- *Tiempo de respuesta:* cuanto tarda una tarea desde que inicia hasta que se completa. Un concepto relacionado es el *throughput*, que es la tasa en la cual un grupo de tareas se ejecuta.
- *Predictibilidad:* el tiempo de respuesta (u otra métrica) es consistente a través del tiempo.

1.5. Sistemas Operativos MultiUsuarios

Hay varias formas de compartir los recursos con muchos usuarios. Cuando se tiene un SO *batch*, un programa se carga en memoria y en ese tiempo se ejecuta otro. Este SO se instala en la memoria y ejecuta un ciclo: *cargar-ejecutar-descargar* para cada trabajo, y mientras se ejecuta uno, el SO puede comenzar a transferir mediante los dispositivos *input-output* para el siguiente trabajo, a través del DMA (permite lear y escribir en la memoria independientemente del microprocesador).

Los sistemas *batch* luego se extendieron para ejecutar varias aplicaciones al mismo tiempo (*multitasking*). Se dice que el *multitasking* mejora la eficiencia del microprocesador porque se tienen muchos programas en memoria al mismo tiempo, cada uno esperando a ser ejecutado. Para aumentar el aislamiento se agregó protección de memoria por hardware.

1.6. Sistemas Operativos de tiempo compartido

Son optimizados para los usuarios (y no el microprocesador) o mejor dicho, están diseñados para soportar el uso interactivo de la PC. El usuario usa los periféricos que envían interrupciones al microprocesador, interpreta los eventos y los manda a una cola del SO de manera que cuando la aplicación de usuario se ejecute, tome los eventos del SO, los procese, altere la salida visual y siga al evento siguiente. Los SO deben, por lo tanto, estar diseñados para ejecutar muchas actividades cortas.

1.7. Sistemas Operativos Modernos

- *Escritorio:* poseen un usuario a la vez, varias aplicaciones, varios periféricos.
- *SO móviles:* un usuario, muchas aplicaciones, necesidad de aislamiento.
- *Sistemas embebidos:* los dispositivos embebidos ejecutan un SO a medida, con el software necesario para controlar el dispositivo.
- *Máquinas virtuales:* SO's que son ejecutados por otros SO's como si fueran una aplicación.
- *Servidores:* versiones industriales de los SO's de escritorio que operan en ambientes de red hostiles o donde hay mucho tráfico de red.
- *Clusters:* permiten descentralizar las tareas, ya que si un servidor falla, otro puede tomar el control, o también puede ejecutar partes de una misma tarea en simultaneo. Operan con una interfaz abstracta que aísla el software del hardware.

2. Procesos

2.1. Protección

Aislar las aplicaciones y el SO de las aplicaciones y usuarios potencialmente dañinos. Es esencial para lograr los objetivos de *confiabilidad, seguridad, privacidad* y *eficiencia*. Es implementada por el *kernel* del sistema (nivel del software corriendo en el sistema, con acceso a todos los recursos de hardware). Todo lo que pueda no ser confiable deberá ejecutarse en un ambiente restringido con menores niveles de acceso (nivel de usuario).

2.2. Proceso

Es la abstracción de protección proporcionada por el *kernel*, es decir, la ejecución de un programa con derechos restringidos. Este proceso necesita permiso del *kernel* antes de acceder a la memoria de otro proceso, antes de leer o escribir en el disco, antes de cambiar la configuración de hardware, etc.

2.3. Concepto de proceso

En el caso general, un programador escribe código en un lenguaje de alto nivel. Luego un compilador convierte ese código en una secuencia de instrucciones de máquina, que se guardan en un archivo llamado *imagen ejecutable*. Para iniciar el programa, el SO copia las instrucciones y datos de la imagen ejecutable a la memoria física. El SO separa también memoria para la pila de ejecución para almacenar el estado de las variables locales durante la llamada a procedimientos. También separa memoria para el *heap* para cualquier estructura de datos u objetos dinámicos que el programa pueda necesitar. Al momento de cargar un programa a la memoria, el SO debe estar cargado en la misma con su pila y su *heap*. Una vez que el programa está en la memoria, el SO lo puede ejecutar estableciendo el puntero de la pila y saltando a la primera instrucción del programa.

Si un usuario quiere ejecutar múltiples copias de un programa, el SO puede hacerlo copiando las instrucciones, datos estáticos, *heap* y pila en memoria. Por lo tanto, *la diferencia entre proceso y programa es que un proceso es una instancia de un programa con un estado*.

El SO sigue los procesos en ejecución usando una estructura de datos llamadas *bloque de control de procesos*, que almacena toda la información que requiere de un proceso en particular (ubicación en memoria, en qué parte del disco se encuentra su imagen, usuario que lo ejecutó, privilegios, etc).

2.4. Dual-Mode Operation

Trabajar con dos modos permite que los programas puedan ser ejecutados y las instrucciones que son seguras de ejecutar corran directamente en el hardware, mientras que las demás se ejecuten en un modo restringido. Este modo se presenta mediante un bit en el registro de estado del microprocesador que indica el estado actual. En el *modo usuario* el microprocesador se fija si la instrucción tiene permiso para ser ejecutada cuando es leída, mientras que en el *modo kernel* no se realiza un control de seguridad.

Para que el *kernel* pueda proteger las aplicaciones y usuarios mediante este mecanismo, el hardware debe soportar:

- *Instrucciones privilegiadas:* instrucciones potencialmente peligrosas que no pueden ejecutarse en modo usuario.

  El aislamiento de los procesos es posible si existe una forma de evitar que los programas en modo usuario cambien su nivel de privilegio. Las instrucciones con privilegios son aquellas que están sólo disponibles en *modo kernel*. De esta forma el SO se ejecuta en *modo kernel* mientras que las aplicaciones poseen un conjunto de instrucciones reducido. Si una aplicación intenta acceder a una posición de memoria que no le corresponde, o quiere cambiar su nivel de privilegio, ocurre una excepción en el microprocesador, que causa que el control quede en manos del *exception handler* en el *kernel*. Generalmente el SO interpreta esto como un bug y detiene la ejecución del programa.

- *Protección de memoria:* en modo usuario, los accesos a memoria por afuera del proceso están prohibidos.

  ¿Cómo el SO previene que una aplicación acceda a partes de la memoria física (que no corresponden)?

  Mediante mecanismos como *base and bounds*. *Base* es el inicio de la región de memoria del proceso y *bounds* su longitud. Estos registros sólo pueden ser cambiados por instrucciones privilegiadas. Cada vez que el procesador toma una instrucción, se fija en la dirección del *program counter* para ver si yace en los registros *base and bounds*, si es así, la instrucción continúa, de lo contrario se lanza una excepción.

  De todas formas, este sistema no permite las características:

  - *Heap* y pilas extensibles luego de cargar el programa en memoria.
  - Compartir memoria (al ejecutar varias instancias del mismo programa).
  - Direcciones de memoria absolutas (ya que todo es relativo a estas variables).
  - Fragmentación de memoria al no poder realocar los programas una vez iniciados.

  El SO se ejecuta sin *base and bounds*.

- *Interrupciones por tiempo:* sin importar lo que haga el proceso, el *kernel* debe tener una forma de tomar el control del proceso periódicamente.

Direcciones virtuales:
Nivel de abstracción introducido por los microprocesadores en los que la memoria de los procesos comienzan en el mismo lugar y cada proceso cree que dispone de toda la memoria. El hardware se encarga de traducir las direcciones virtuales a físicas, de forma de solucionar los problemas anteriores.

Interrupciones por tiempo (timer interrupts): es una forma de que el sistema pueda terminar con la ejecución de un proceso, o ganar el control para cambiar a otra tarea. Se realiza mediante un *hardware timer*, que interrumpe el microprocesador cada cierto tiempo o cantidad de instrucciones. Cuando ocurre una interrupción del *timer* el control es transferido del proceso en *modo usuario* al SO en *modo kernel*. En la mayoría de los casos el SO resumirá la ejecución sin cambiar nada (modo, *program counter*, registros).
Reiniciar el *timer* es una instrucción privilegiada.

2.5. Transferir a *modo seguro*

Hasta aquí se tienen los procesos de usuario en un marco seguro, debiéndose establecer ahora cómo cambiar de modo, requiriendo un mecanismo seguro para no corromper el sistema.

De *modo usuario* a *modo kernel*

Hay *tres* formas de que el *kernel* tome el control desde un proceso:

- *Excepciones:* condiciones inesperadas causadas por el comportamiento del programa. Cuando ocurren, el hardware detiene el proceso y deja el control en la *exception handler*.
  - Ejemplos: acceso a memoria no autorizados, división por cero, *debugging* (break points).
- *Interrupciones:* señales asíncronas que se envían al microprocesador indicando que ocurrió un evento externo y puede requerir su atención. Actúa de manera similar que en las excepciones, el microprocesador detiene el proceso en ejecución y comienza a ejecutarse en el *kernel* mediante un *interruption handler*. Cada tipo de interrupción tiene su *handler*.
  - Ejemplos: *timer*, mouse/teclado, ethernet/wifi, discos.

  Una alternativa a las interrupciones es el *polling*, que consiste en un ciclo de *kernel* que verifica si ocurrió algún evento en los periféricos.
- *Llamadas al sistema:* son procedimientos del *kernel* que pueden ser llamados a nivel de usuario (interrupciones *trap*). Pueden ser ejecutados automáticamente luego de una excepción.
  - Ejemplos: establecer conexión, intercambiar archivos en red, crear o modificar archivos, crear nuevos procesos.

De *modo kernel* a *modo usuario*

En el caso inverso hay causas diferentes para efectuar la transición:

- *Nuevo proceso:* luego de que el *kernel* copia el programa en memoria, cambia el *program counter* y el puntero de la pila, cambia a *modo usuario*.
- *Continuar luego de una excepción, interrupción o llamada al sistema:* restaura el *program counter* y sus registros, y cambia a *modo usuario*.
- *Cambiar a otro proceso:* en algún caso el *kernel* querrá cambiar el proceso actual por otro luego de una excepción, interrupción, etc. Para ello, el *kernel* guarda el estado del proceso (*program counter*, registros, etc) en el bloque de control de procesos. El *kernel* carga estos datos del otro proceso y cambia a *modo usuario*.
- *Upcall a nivel de usuario:* análogo a las interrupciones, pero al revés. Son mecanismos para que los programas de usuario reciban notificaciones o eventos asíncronos del *kernel*.

2.6. Ejecutar procesos

Para ejecutar aplicaciones a nivel de usuario el *kernel* debe:

- Alocar e inicializar el bloque de control de proceso.
- Alocar memoria para el proceso.
- Copiar el programa del disco a la memoria recién alocada.
- Alocar una pila para el nivel de usuario.
- Alocar una pila para el nivel de *kernel* para manejar *llamadas al sistema*, *interrupciones* y *excepciones*.

Para ejecutar un programa, el *kernel* también debe:

- Copiar argumentos a la memoria del usuario.
- Transferir control al modo usuario.

2.7. La interfaz de programación

Falta definir cómo usar la abstracción del proceso, esto es, qué funcionalidades deben ponerse en el *kernel*, cuáles debe brindar a las aplicaciones, qué debe ponerse en las bibliotecas de usuario, y cómo debe organizarse el SO. *Esto es lo que distingue un SO de otro*.

2.8. Funciones que debe proporcionar un SO

a. *Gestión de procesos:* ¿Un programa puede crear una instancia de otro? ¿Puede detener a otro programa? ¿Enviar un evento asíncrono?

b. *Input/Output:* ¿Cómo se comunican los procesos con los periféricos? ¿Y entre procesos?

c. *Manejo de hilos:* ¿Pueden crearse hilos que compartan memoria dado un proceso?

d. *Manejo de memoria:* ¿Puede un proceso pedir memoria? ¿Puede compartirla con otros procesos?

e. *Sistema de archivos y almacenamiento:* ¿Cómo un proceso guarda los datos de forma persistente?

f. *Redes y sistemas distribuidos:* ¿Cómo un proceso controla pixeles? ¿Cómo se usan los aceleradores?

g. *Autenticación y seguridad:* ¿Qué permisos tiene un programa y cómo se actualizan? ¿Cómo sabemos si el usuario es quién dice ser?

2.9. Implementación de las funciones del SO

Las funciones antes descritas se pueden ubicar de diferentes formas:

a. Poner la funcionalidad en un programa en modo usuario.
   - Ejemplo: programa para el login, para administrar procesos.

b. En una librería a nivel de usuario que se enlaza con cada aplicación.
   - Ejemplo: widgets de la interfaz de usuario (Win/Mac).

c. En el *kernel* que se accede con una llamada al sistema.
   - Ejemplo: manejo de procesos, *filesystem*/pila de red (Unix).

d. Acceder a la función mediante una llamada al sistema e implementarla en un servidor invocado por el *kernel*.
   - Ejemplo: gestor de ventanas.

2.10. Criterios para la implementación de las funciones

I. *Flexibilidad:* es más fácil cambiar el código del SO que se encuentra fuera del *kernel*. Un cambio en la interfaz de las *llamadas al sistema* implica cambios en el *kernel* y las aplicaciones.

II. *Seguridad:* la seguridad debe estar implementada en el *kernel* (ya que si se hace en *modo usuario* puede ser salteada).

III. *Confiabilidad:* los módulos de *kernel* no están protegidos entre sí, por lo que un error en el *kernel* puede comprometer los datos de usuario y del *kernel*, por lo que la confiabilidad se logra manteniendo un *kernel* pequeño. En un diseño *microkernel* las partes menos críticas del SO (sistema de archivos, ventanas) se aíslan del resto del *kernel* resultando en un *kernel* pequeño. Se pueden agregar nuevas características sin recompilar todo el *kernel*. Por otro lado, los sistemas monolíticos son más rápidos pero son más grandes (deben almacenar los drivers en el espacio del *kernel*), son más inseguros frente a los fallos de drivers (ya que pueden comprometer al sistema).

IV. *Rendimiento:* transferir en control al *kernel* es más costoso que una llamada a una librería, por eso no prosperó el *microkernel* de WinNT.

2.11. Administración de procesos

En los primeros sistemas de procesamiento, el *kernel* tenía el control por necesidad, siendo éste quien creaba los procesos a partir de las tareas que enviaba el usuario. Un enfoque diferente es permitir que los programas creen y administren sus propios procesos.

En Windows, se puede usar la función `createProcess()`, que en síntesis crea en inicializa el bloque de control de proceso en el *kernel*, crea e inicializa un nuevo espacio de direcciones, carga el programa al espacio de direcciones, copia los argumentos en memoria al espacio de direcciones, inicializa el contexto de hardware para comenzar la ejecución y por último informa al planificador que el proceso está listo.
El proceso padre puede controlar los privilegios del hijo, donde enviar su *Input*/*Output*, dónde almacenar los archivos, prioridad del planificador, etc.

En UNIX, se utiliza un enfoque diferente, dividiendo la tarea en dos pasos:

- `fork()`: crea una copia completa del proceso padre (se le da el control al ser una copia del padre), y el proceso hijo establece sus privilegios, prioridades y I/O para el programa que se iniciará. Devuelve un entero.
- `exec()`: carga la imagen ejecutable creada anteriormente y la ejecuta. Requiere como argumentos el nombre del ejecutable a correr y un arreglo de argumentos para pasarle al programa.

Para implementar `fork` en el *kernel* hay que crear en inicializar el *bloque de control de proceso* (PCB) en el *kernel*, crear un espacio de direcciones, inicializar el espacio de direcciones con una copia de los contenidos del espacio de direcciones del padre, heredar el contexto de ejecución del padre, e informar al planificador que hay un proceso listo para ejecutarse.
El `fork()` retorna valores dos veces: en el padre, devuelve el *process ID* del hijo, y en el hijo retorna cero (éxito), de manera de saber quién es el hijo. Esta relación no establece que el hijo se ejecutará después del padre, ya que ambos procesos están listos y el padre puede quedar en espera por el *timer*, por lo que el orden depende del planificador.
El `exec()` completa los pasos necesarios para correr el programa: cargar el programa al espacio de direcciones actual, copiar los argumentos a la memoria e inicializar el contexto de hardware para iniciar la ejecución.
Notar que no crea un nuevo proceso. En el caso de que el proceso padre requiera que el proceso hijo termine, se usa la *llamada al sistema* `wait()`.
Para liberar recursos asociados a un proceso se usa la *llamada al sistema* `exit`, mientras que para que un proceso le envíe a otro un modificación (*upcall*) se usa `signal()`

.....

Comandos:
Pipes. Permiten conectar varios comandos. Ejemplo: `ls -l -R | less` permite recorrer la salida.

Scheduling:

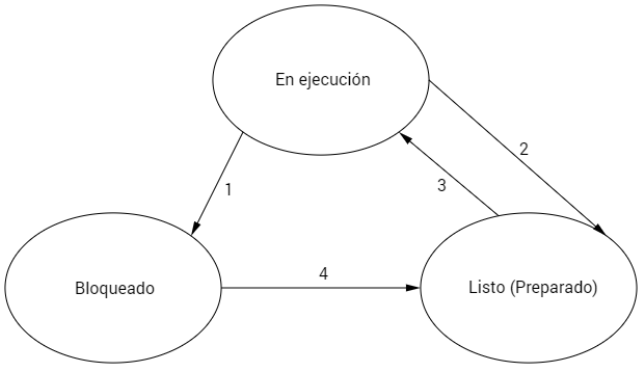
Un planificador de procesos es un componente cuya función es repartir el tiempo disponible de un microprocesador entre todos los procesos que estén disponibles para su ejecución.

Estados de los procesos:

- *En ejecución:* el proceso está usando la CPU en este momento.
- *Listo:* el proceso se detuvo temporalmente para que se ejecute otro.
- *Bloqueado:* el proceso no puede ejecutarse hasta que ocurra un determinado evento.

Hay *cuatro* posibles transiciones:

1. Se da cuando el proceso no puede continuar.
2. Se da cuando ocurre una *interrupción* por el *timer* y hay que dejar el CPU para otro proceso.
3. Se da cuando ya se ejecutaron todos los demás procesos y se continúa trabajando en el primero.
4. Se da cuando se produce el evento que el proceso estaba esperando (cuando llegue el turno).



Tipos de recursos:

- *Apropiativo:* el microprocesador puede ser compartido, ya sea por una *interrupción* del *timer* o porque llegó un proceso con mayor prioridad.
- *No apropiativo:* uso exclusivo de los recursos (proceso evoluciona a *en espera* $\implies$ *interrupción* $\implies$ se crea o termina un proceso).

Métricas:

a. *Throughput:* procesos completados por unidad de tiempo.

b. *Latency:* tiempo entre ocurrencia y respuesta de un evento.

c. *Turnaround:* tiempo que una tarea necesita para completarse.

d. *Waiting time:* tiempo de un proceso en cola estando *listo*.

e. *Fairness:* tiempo de espera/tiempo procesador.

Los objetivos de los planificadores son minimizar el tiempo de respuesta, minimizar el *throughput* (minimizar el *overhead* y usar los recursos eficientemente), minimizar el tiempo de espera (equidad: que cada proceso espere la misma cantidad de tiempo).

Políticas de clasificación:
Lo habitual es usar políticas mixtas. Generalmente a corto plazo se usa *round-robin*, mientras que a largo plazo varias colas FIFO.

- *FIFO o FCFS:* los procesos se ejecutan según el orden de llegada, ejecutándose hasta que termine o ceda la CPU. Minimiza el *overhead*, aunque es ineficiente cuando hay tareas que requieren poco tiempo de procesamiento detrás de una tarea que toma mucho tiempo.
- *Round-Robin:* los procesos toman turnos de un período limitado de tiempo para usar el microprocesador, ejecutando la primera tarea lista, estableciendo una *interrupción* por *timer* con una duración denominada *quantum*. Al final del *quantum*, si la tarea no se completó, la misma es apropiada y se le da la siguiente tarea lista al microprocesador. La tarea apropiada se pone en espera (estado *listo*).

  Se puede además establecer prioridades a las tareas, de manera de tener colas por prioridad y procesar primero la mayor prioridad. Esto presenta el potencial problema de inanición (proceso con prioridad baja que jamás se puede ejecutar), que se puede solucionar cambiando las prioridades en base al tiempo de procesamiento (bajar) o el tiempo de espera (subir).

- *Shortest Job First (SJF):* se ejecuta el trabajo con menor tiempo de procesamiento (estimado).
- *SRTF:* es una versión del SJF, pero apropiativo. Consiste en elegir siempre el proceso al que le reste menos tiempo de ejecución (CPU *burst*). Minimiza el tiempo de respuesta promedio. Si todos los procesos tienen la misma duración SRTF es FIFO.

.....

Disco:
Contiene uno o más platos que giran de 5400 a 10800 rpm. Posee un brazo mecánico (cabeza de lectura-escritura). Con un punto de giro colocado en una esquina. La información se escribe en el disco en una serie de círculos concéntricos. En cualquier posición del brazo, cada una de las cabezas puede leer una región llamada pista, y en conjunto todas las pistas dada una posición del brazo se denominan cilindro. Cada pista se divide en sectores (generalmente de 512 *bytes*). Cuando se desplazan los brazos al encontrar la pista correcta, se espera a que el sector necesario se ubique debajo de ésta. Un *cluster* es un conjunto de sectores.

Memoria virtual:
En los SO actuales, una parte del espacio de direcciones se mantiene en la memoria principal, y la otra parte en el disco. La memoria virtual sirve para ejecutar programas más extensos que la memoria física, llevando y trayendo pedazos entre la RAM y el HDD.

Archivo:
Colección de información relacionada y almacenada en un dispositivo de almacenamiento secundario de direcciones lógicas contiguas. Las estructura interna puede ser una secuencia de *bytes* (Windows/Linux) o una secuencia de registros de longitud fija o variable.

Tipos de archivos:

- *Regulares:* contienen información del usuario.
- *Directorios:* estructuras que permiten llevar el registro de los archivos. Pueden ser también archivos.
- *De dispositivo:* modelan dispositivos *Input*/*Output* en serie. Ejemplo: impresoras, redes, etc.

Forma de acceso a los archivos:

- *Secuencial:* un proceso puede leer todos los *bytes* o registros en un orden, empezando del comienzo sin poder saltear pero sí rebobinar. Es el modo de acceso con cintas magnéticas.
- *Aleatorio:* leer *bytes* o registros fuera de orden.

Atributos de los archivos:

- *Nombre*.
- *Tipo*.
- *Localización (puntero)*.
- *Tamaño actual*.
- *Permisos (lectura, escritura,ejecución)*.
- *Fecha y hora*.

Semánticas de consistencia:
El SO administra la seguridad del sistema de manera que sea accesible sólo por usuarios autorizados. Específicamente las modificaciones de datos por un usuario se observan por otros usuarios.

Funciones básicas de un sistema de archivos:
Tener conocimiento de todos los archivos del sistema, controlar la compartición y forzar la protección de los archivos, gestionar el espacio de los sistemas de archivos y traducir direcciones lógicas de archivo a direcciones físicas.

Implementación de un sistema de archivos:

Aquí se define la forma en que se almacenan archivos y directorios, cómo se administra el espacio en disco y como hacer para que todo funcione con eficiencia y confiabilidad. Los SO poseen sus propios sistemas de archivos.

Los sistemas de archivos poseen dos problemas de diseño:

- Definir cómo el usuario ve el sistema de archivos.
- Definir algoritmos y estructuras de datos que deben crearse para establecer una correspondencia entre el sistema de archivos (lógico) y los dispositivos físicos que los almacenan.

Las técnicas de implementación más usadas son:

FAT:

Este sistema de archivos usa una lista enlazada como estructura. La versión más reciente (FAT32) usa un arreglo de 32 bits en un área reservada del volumen. Cada archivo en el sistema se corresponde con una lista enlaza de entradas FAT. Cada entrada FAT contiene un puntero a la siguiente entrada FAT (o el valor `eof` para la última). La FAT posee una entrada para cada bloque en el volumen, y los bloques de archivo se corresponden con entradas de la FAT.

Limitaciones de la FAT:

a. Mala localidad (fragmentación).

b. Acceso aleatorio lento (ya que es una lista).

c. Metadatos limitados (no hay información sobre permisos, cualquier usuario puede manipular cualquier archivo).

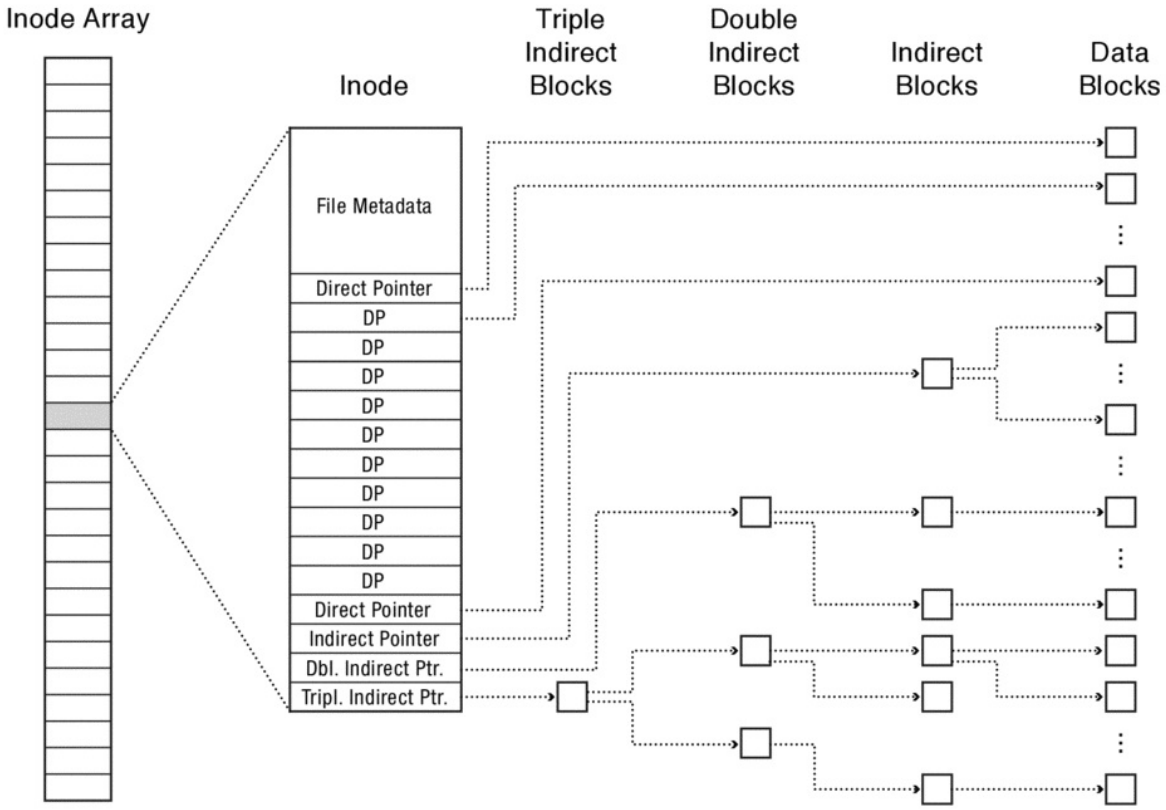
d. Tamaño de archivo limitado (los tamaños de los archivos se codifican en 32 bits por lo que ningún archivo puede ser mayor a $2^{32} - 1$ bytes $= 4$ GB)

FFS:

Usa índices basados en árboles multinivel, mejorando el acceso aleatorio. Además utiliza heurísticas de localidad para obtener buena localidad espacial. EXT2 y EXT3 están basados en FFS.

Para seguir el rastro de los bloques de datos que pertenecen a cada archivo, FFS usa un árbol asimétrico llamado *multi-level index*. Cada archivo es un árbol con bloques de datos de tamaño fijo (por ejemplo, 4 KB) como hojas. Cada árbol (para cada archivo) comienza en un i-nodo, que posee los metadatos del archivo (dueño, permisos, fechas, si es una carpeta). El tamaño máximo de un archivo estará dado por la capacidad de direccionamiento del i-nodo.

Estructura de un i-nodo:



Un archivo contiene un arreglo de punteros directos e indirectos:

- *Punteros directos:* apuntan a los bloques de datos (hojas).
- *Punteros indirectos:*
  - *Simples:* apunta a un bloque de almacenamiento que posee un arreglo de punteros directos. Con bloques de 4 KB y punteros de 4 bytes, un bloque indirecto puede contener 1024 punteros directos (4 MB).
  - *Dobles:* apunta a un nodo interno del árbol llamado bloque doble indirecto (un arreglo de punteros indirectos que apuntan a un bloque indirecto). Con bloques de 4 KB, un puntero indirecto doble puede apuntar a 1024^2 bloques de datos.
  - *Triples:* apuntan a bloques triples indirectos que contienen un arreglo de punteros indirectos dobles. Con bloques de 4 KB puede indexar 1024^3 bloques de datos que contienen $4[KB] \cdot 1024^3 = 2^{12} \cdot 2^{30} = 2^{42} = 4[TB]$

Todos los i-nodos del sistema se ubican en un arreglo de i-nodos que se almacena en una ubicación fija del disco. Cada archivo posee un número (*i-number*) y ese número en FFS es un índice al arreglo de i-nodos. Para abrir un archivo se busca su i-nodo en un directorio y luego se busca la entrada apropiada en el arreglo de i-nodos para encontrar sus metadatos.

Fragmentación:
Memoria desperdiciada como consecuencia de las políticas de ajuste de bloques que tiene un sistema de archivos o la estructura del mismo.

Tipos de fragmentación:

- *Interna:* se da cuando el tamaño de un archivo (o la diferencia entre el tamaño de una cantidad entera de *clusters*) es menor que al tamaño del *cluster* (el *cluster* entero se reserva para ese archivo por más que no esté completamente ocupado).
- *Externa:* ocurre luego de la sucesiva creación y eliminación de archivos de distintos tamaños, aislando bloques libres, y cuando la política de ajuste del disco no elige a estos bloques para nuevos archivos.

Planificación de discos:
Los algoritmos de planificación intentan minimizar el tiempo de búsqueda. La efectividad se mide sumando la diferencia de pistas entre cada solicitud y dividiendo por la cantidad de solicitudes.

- *FCFS (FIFO):* las solicitudes se procesan en un orden secuencial. Es justa para todos los procesos. Si hay muchos procesos es similar a una técnica aleatoria.
- *SSTF:* primero se procesan las peticiones que minimizan el movimiento de cabezas desde la posición actual. Puede causar inanición.
- *SCAN:* el movimiento del brazo empieza en un extremo y continúa hasta el otro. Allí se cambia el sentido y se vuelve al otro extremo atendiendo tareas a la vuelta.
- *C-SCAN:* las cabezas se mueven de un extremo al otro del disco y luego vuelven al principio. No se atienden peticiones mientras las cabezas vuelven a la posición inicial (ir del último al primero es más rápido que ir parando).

3. Preguntas

1. ¿Es posible que un proceso a nivel de *usuario* logre privilegios del *kernel*? Justifique.

   No, ya que al estar a nivel de *usuario* el microprocesador no permite que se ejecuten instrucciones del *kernel*. Lo que se puede hacer es avisar al sistema operativo de alguna forma que se quiere ejecutar una instrucción privilegiada, ya que este si posee privilegios de *kernel*. Esto se puede hacer mediante *llamadas al sistema (trap)*.

2. ¿Qué diferencia existe entre una *llamada al sistema (trap)* y una invocación a un procedimiento convencional? Justifique.

   Una invocación a un procedimiento (función) se realiza en *modo usuario* desde el inicio hasta el fin, mientras que en una *llamada al sistema (trap)* esta instrucción cambia el modo del microprocesador a *modo kernel* y ejecuta una serie de instrucciones siguiendo un *handler* que le pone límites.

3. Asuma que un sistema tiene que planificar la asignación de la CPU en forma *justa* a un conjunto de tareas, algunas de las cuales tienen un alto requerimiento de CPU y otras, una alta demanda de E/S ¿a qué tipo de tareas daría más prioridad? El mecanismo SRTF consiste en asignar la CPU al trabajo que tenga un duración menor. Dado que es imposible conocer la duración exacta de una tarea, ¿de qué manera se resuelve esto?

   La asignación de la CPU es *justa* cuando los procesos son atendidos en orden de llegada.

   La prioridad de las tareas dependerá del planificador de procesos, en este caso FCFS (FIFO). Mientras un programa está siendo ejecutado por el microprocesador otro se carga en memoria a través de los dispositivos *input*/*output* mediante el DMA (*acceso directo a memoria*).

   El DMA permite leer y escribir en la memoria independientemente del microprocesador.

   El mecanismo SRTF determina cuáles son los tiempos de las tareas mediante estimaciones estadísticas.

4. ¿Qué tipo de evento(s) provoca la evolución de un proceso del estado *en ejecución* a *listo*? ¿Es posible que un proceso evolucione desde un estado bloqueado a un estado en ejecución? Justifique.

   Esta evolución se da cuando ocurre una *interrupción* por el *timer*. Esta es una forma de que el sistema operativo tome el control del proceso para hacer algo con éste o cambiar a otra tarea. El hardware interrumpe el microprocesador cada cierto tiempo o cantidad de instrucciones. Si hay un mecanismo de planificación activo que use *quantos* de tiempo (por ejemplo, *Round Robin*), el control pasará al siguiente proceso de igual prioridad cambiando el estado del proceso actual de *en ejecución* a *listo*.

No, no es posible. Para llegar del estado *bloqueado* al estado *en ejecución*, los procesos primero deben pasar al estado *listo*. Los procesos bloqueados están a la espera de un *evento* que de llegar cambia el estado de *bloqueado* a *listo*, para ser ejecutado según el planificador.

5. Cuando se invoca una *llamada al sistema* (system call), el control evoluciona, en algún momento, de la *capa usuario* a la *capa kernel*. Detalle en forma de PSEUDOCÓDIGO la manera en que se concretan tales cambios de privilegios considerando las estructuras de datos involucradas.

   Se puede proveer una librería que encapsule las *llamadas al sistema* mediante un *stub* ubicado en la región de memoria del *kernel*, por lo que este *stub* hace la llamada al sistema con privilegios. Por ejemplo, para abrir un archivo:

   1. El programa de usuario llama al *stub* como a cualquier función (sin saber que esa función realiza instrucciones en *modo kernel*).
   2. El *stub* llena el código necesario para la *llamada al sistema* y ejecuta la
      instrucción *trap*.
   3. El harware transfiere el control al *kernel* a través del *handler* para llamadas externas. El *handler* actúa como un *stub* pero del lado del *kernel*, copiando y verificando los argumentos y luego llamando a la implementación de la *llamada al sistema* en el *kernel*.
   4. Luego de que se completa la *llamada al sistema* se vuelve con el resultado al *handler*.
   5. El *handler* vuelve al *modo usuario* pero a la siguiente instrucción del *stub*.
   6. El *stub* devuelve el resultado a la función original.

6. ¿Por qué entiende las variables locales se almacenan en la pila en lugar de en un segmento de memoria convencional?

   Las variables locales al poder referenciarse sólo en la función en las que se las han declarado, deben existir sólo durante la ejecución de la función que la declara. Por esto, la mejor estructura para modelar este comportamiento es un *pila* ya que primero se insertan las variables de las funciones externas y cuando se procesa la última función interna, en el tope de la pila quedan las variables de esta función que posteriormente serán eliminadas

7. Detalle las ESTRUCTURAS DE DATOS que provee el *kernel* en un sistema de i-nodos para gestionar los archivos y el rol de cada una. Explique a través de un PSEUDOCÓDIGO el uso de las mismas cuando se realice una operación de escritura de un archivo. Considere que el i-nodo tiene dos enlaces directos y uno indirecto simple con 2 enlaces.

   Los i-nodos constan de una estructura de árbol multinivel.

   - El *super bloque* contiene información referente al estado del disco y a las particiones de los datos.
   - El *bitmap i-nodo* se utiliza para indicar el estado de los i-nodos y ver si están ocupados o libres.
   - El *bitmap bloques* se utiliza para indicar el estado de los bloques y ver si están ocupados o libres.
   - Todos los i-nodos de los archivos del sistema se ubican en un *arreglo* de i-nodos que se almacena en una ubicación fija del disco. Un i-nodo contiene enlaces directos, enlaces indirectos simples, enlaces indirectos dobles y enlaces indirectos tripes. Los enlaces directos apuntan a bloques de datos. Los enlaces indirectos simples apuntan a bloques de puntero que apuntan a bloques de datos. Los enlaces indirectos dobles apuntan a bloques de enlaces indirectos simples. Los enlaces indirectos triples apuntan a bloques de enlaces indirectos dobles.
   - Todos los bloques de datos de los archivos del sistema se ubican en un *arreglo* de bloques que se almacena en una ubicación fija del disco.

   1. Primero se intentan ocupar los bloques de datos pertenecientes a los bloques directos.

2. Si los bloques de los enlaces directos no son suficientes se pasan a utilizar los bloques de datos de los enlaces indirectos simples hasta almacenar el archivo completo.

8. ¿Qué objetivos intentan satisfacer las arquitecturas de sistemas operativos monolítica y *microkernel*?

   Microkernel: las partes menos críticas del sistema operativo (por ejemplo, el sistema de archivos o las ventanas) se aíslan del resto del *kernel*, resultando en un *kernel* pequeño. Esto permite tener un *kernel* más dinámico, en el cual se pueden agregar características sin tener que recompilar todo el *kernel*. Al tener esta división entre *kernel* y librerías se obtiene un rendimiento menor que en los monolíticos, pero la división misma otorga confiabilidad ya que una falla de una de las librerías no afecta a todo el sistema.

   Monilítico: en éste, todas las partes del sistema operativo se almacenan en el espacio del *kernel*, resultando más rápidos que los anteriores pero más vulnerables a fallas del sistema.