Apunte 1

Nicolás Gómez Morgado

Sistemas Operativos

4 de noviembre de 2024

Índice

1. Importante

- Se sugiere tener instalada una distribución de Linux en el computador, en mayor medida Debian, ya que el Profesor trabaja con esta distribución.
- Linux y Unix están orientados al multiuser, a diferencia de Windows que esta orientado al single user.
- El S.O. es el intermediario entre el hardware y el usuario.
- Linux es el núcleo del S.O. y las distribuciones son como se adorna/agrupa este mismo.
- Para la tarea 1 se debe crear un programa con GNU GCC++ en Linux.
- Para la tarea 2 se debe crear un programa.
- **API:** Interfaz de programación de aplicaciones que permite a los programas comunicarse con el sistema operativo.

2. Introducción

En principios de la computación no existía el software que comunicara el hardware con el usuario, por lo que se debía programar directa y físicamente en lenguaje maquina (tarjetas perforadas). La operación de estas computadoras era completamente manual.Tiempo de uso de la computadora demasiado alto, involucrando colocar tarjetas, imprimir respuestas, etc.

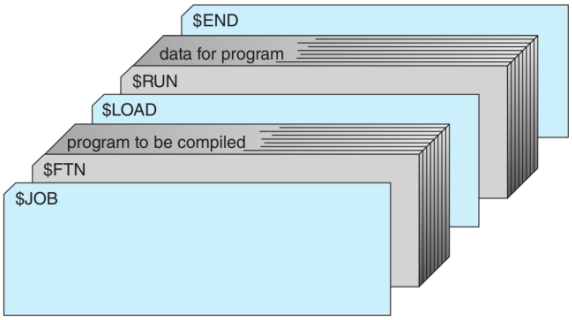
Sistemas informáticos compartidos

Con el avance de la tecnología se separo el trabajo para operar la maquina, el operador se encargaba de insertar las tarjetas y el programador a crearlas/ordenarlas. A pesar de esta optimización la CPU seguía sin utilizarse cuando el trabajo se detenía, por lo cual se desarrollan mecanismos de secuenciamiento automático de trabajos (primeros SO).

Se implementa un monitor residente para transferir el control al trabajo siguiente. A pesar de estas mejoras aun existía tiempo muerto en la CPU debido a la diferencia de velocidades en los dispositivos de entrada/salida.

Sistema Batch (Lote) simple

Los trabajos se agrupan en lotes y se ejecutan en secuencia, el monitor residente se encarga de cargar el siguiente trabajo en la memoria principal.



E/S solapada

Solución para reducir el tiempo muerto en la CPU, se implementa un mecanismo de E/S solapada, donde la CPU puede ejecutar otro trabajo mientras se realiza una operación de E/S ya que se guarda la información (instrucciones de las tarjetas) en las cintas magnéticas para procesarse posteriormente.

Spooling

Sistema de colas de trabajos, donde se almacenan los trabajos en una cola de entrada y se envían a la cola de salida para ser procesados por la CPU.

Que es un SO?

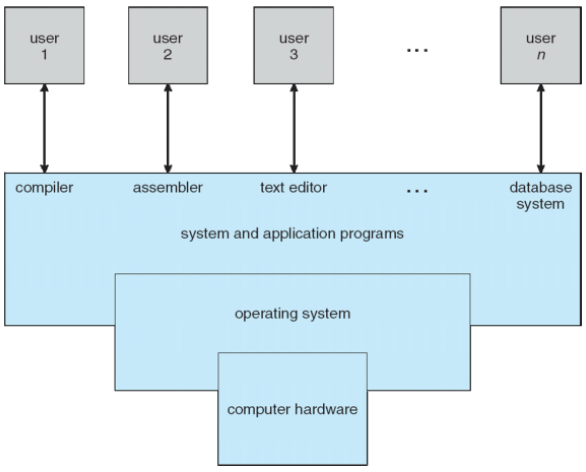
- Programa que actúa como intermediario entre el usuario y el hardware de la maquina cuyo propósito es proporcionar un entorno en el cual se puedan ejecutar programas de forma eficiente y controlada.
- Software que gestiona el hardware.

Estructura de un Sistema informático

El SO es parte del sistema general.

- Hardware: Conjunto de componentes físicos que componen la computadora.
- Sistema Operativo
- Programas de aplicación
- Usuario: Persona, maquinas u otras computadoras que utiliza la computadora.

En ningún caso las aplicaciones que utiliza el usuario pasan directamente al hardware, por temas de seguridad y optimización.
Actualmente cualquier cosa que se haga en una computadoras se hace a traves del sistema operativo.



Papel de un SO

- Punto de vista del usuario
  - SO diseñado para maximizar trabajo
  - SO diseñado para maximizar recursos
- Punto de vista del sistema
  - SO es un asignador de recursos
  - SO es un Programa de control

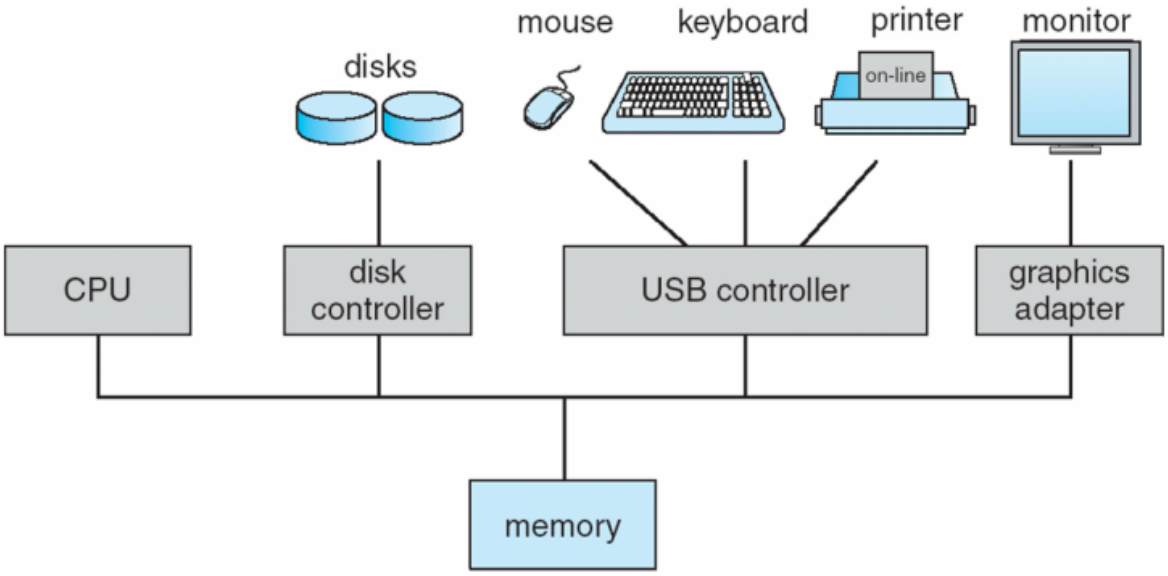


Figura 1: Organización de un SO

Interrupción: Forma que tiene el SO para enterarse de que algo ha pasado en el sistema y como reaccionar ante esto.

Iniciando una Computadora

Bootstrap: Programa de arranque que se encuentra en la memoria ROM de la computadora, este programa se encarga de cargar el SO en la memoria RAM. El programa de arranque se localiza y se carga el kernel a través del **GRUB** (Grand Unified Boatload) que es un gestor de arranque que permite elegir entre varios sistemas operativos. Ante cualquier suceso inesperado, este se indica a traves de una interrupción cual puede ser de 2 formas:

- Por HW: Envía señal al procesador por el BUS del sistema.
- Por SW: Mediante una llamada a sistema.

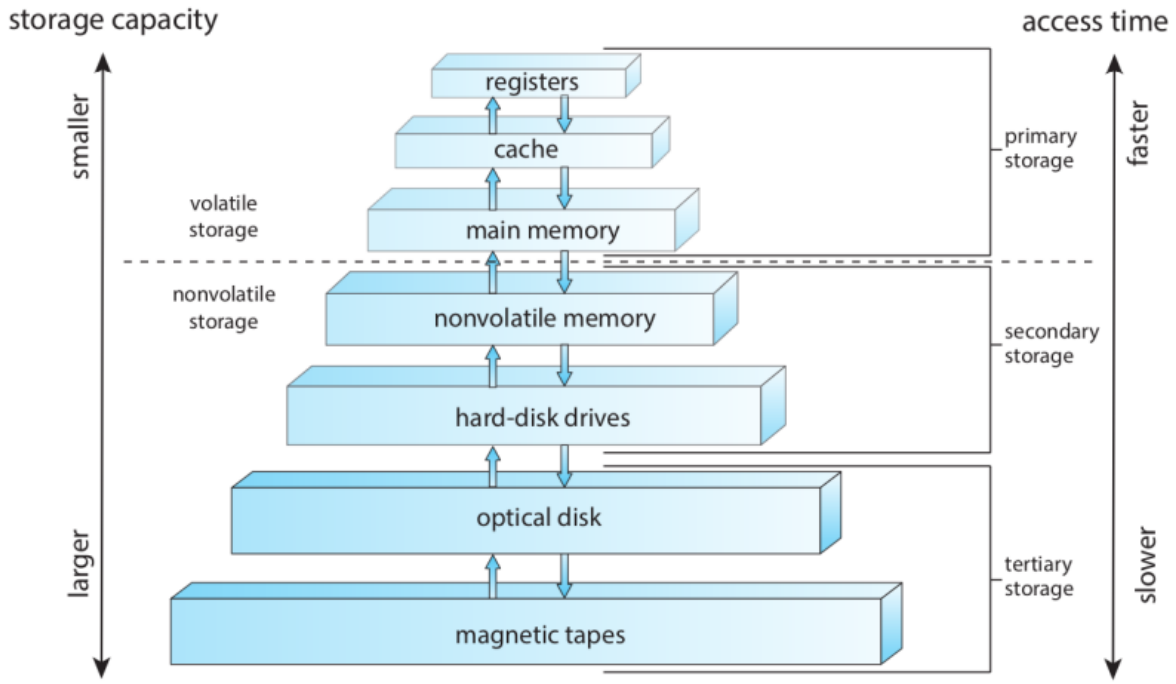
Llamada a sistema: Puente entre las aplicaciones y el SO, permite a las aplicaciones solicitar servicios al SO.

Controladores de dispositivos: El hardware sabe como comunicarse con estos dispositivos de e/s gracias a estos controladores.
El sistema guarda los tipos de interrupciones y sus significados por lo que esa es su forma de saber como reaccionar ante estas mismas.

DMA: Permite a los dispositivos de e/s acceder a la memoria sin pasar por la CPU.

Almacenamiento

1. Memoria principal (RAM)
2. Almacenamiento secundario:
   a) HDD
   b) NVM
      1) SSD



Caching

Información es copiada desde almacenamientos lentos a almacenamientos rápidos para mejorar la velocidad de acceso a la información.

Sistema multiprocessor

Dos o mas procesadores que comparten el bus del computador, estos procesadores pueden compartir la memoria y los dispositivos de e/s.

Sistemas NUMA (Non-Uniform Memory Access)

Division no uniforme de acceso a la memoria. Rápido cuando CPU necesita acceder a su memoria. Latencia aumenta si necesita acceder a memoria remota.

Cluster

Varios nodos donde cada uno puede tener uno o varios procesadores. Comparten almacenamiento a través de areas de almacenamiento (SAN - Storage Area Network).

- Asimétrico: Sistema en modo de espera y monitorea a la activa.
- Simétrico: Todos los sistemas activos y se monitorear entre significados

Multiprogramación

Organiza los trabajos para que la CPU no este ociosa, se ejecutan varios trabajos a la vez. CUando el trabajo en ejecución tiene que esperar el sistema cambia a otro trabajo.

Tiempo compartido

Asigna tiempos (menos de 1 seg) entre tareas de manera equitativa, conmutando entre programas. Si los procesos no encajan en la memoria, con el swapping se pueden mover a la memoria secundaria.

Swapping: Mecanismo que permite mover procesos entre la memoria principal y la secundaria.

Operación en Modo Dual

Se necesitan 2 modos de operación para proteger el SO y los programas de usuario. Estos son el modo **usuario** y el modo **kernel**.

Temporizador

Se asegura el control del SO sobre la CPU, se impide que los programas entren en ciclos infinitos, se impide acapararon de recursos, se usan contadores que después de un tiempo determinado generan una interrupción.

Gestión de procesos

Proceso es programa en ejecución, unidad de trabajo de un SO. Programa = entidad pasiva, proceso = entidad activa. Estos procesos necesitan recursos para completar su tarea:

- CPU
- Memoria
- E/S
- Datos de entrada

3. Estructuras de un SO

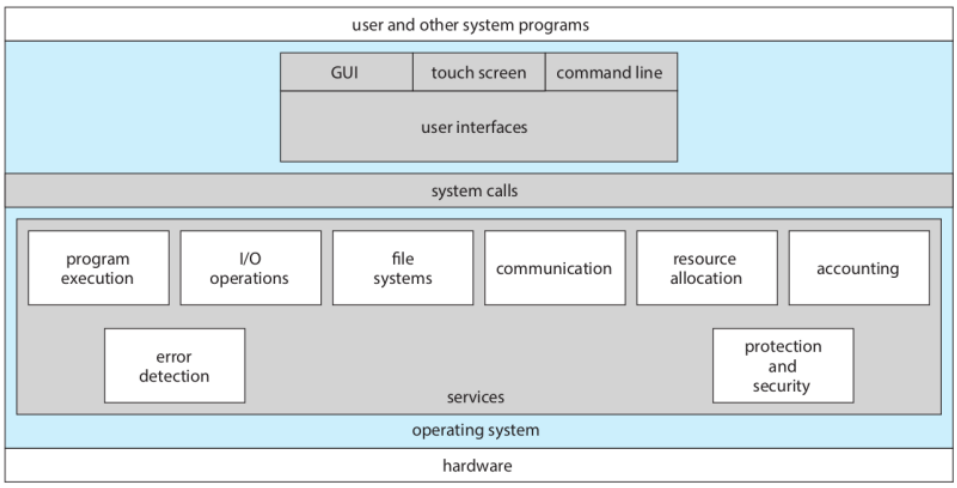
Servicios del SO

Un SO proporciona servicios a los programas de aplicación y a los usuarios, estos servicios pueden ser:

- Interfaz de usuario: Interfaz gráfica o de línea de comandos.
- Ejecución de programas: Carga de programas en memoria y ejecución.
- Operaciones de E/S: Operaciones de lectura y escritura.
- Manipulación de sistema de archivos: Creación, eliminación, copia, etc.
- Comunicaciones: Comunicación entre procesos. Mediante:
  - Memoria compartida.
  - Paso de mensajes.
- Detección de errores: Errores de hardware y software.
  - Detección en HW de CPU y memoria por fallos de alimentación, en dispositivos de E/S por fallos de conexión, etc.
  - Por cada error el SO debe decidir que hacer.
  - La depuración ayuda a encontrar errores en el software.

Mas funciones centradas en garantizar la eficiencia del sistema pueden ser:

- Asignación de recursos: CPU, memoria, dispositivos de E/S.
- Contabilidad: Uso de recursos.
- Protección y seguridad: Control sobre el uso de la información.
  - Protección: Asegurar que todos los accesos a los recursos sean legales.
  - Seguridad: Proteger el sistema de accesos no autorizados.



CLI

Interfaz de línea de comandos, permite al usuario comunicarse con el SO mediante comandos. La función principal es obtener y ejecutar los comandos del usuario.

GUI

Interfaz gráfica de usuario, permite al usuario comunicarse con el SO mediante ventanas, iconos, menús, etc. La función principal es obtener y ejecutar los comandos del usuario.

Touchscreen

Interfaz táctil, permite al usuario comunicarse con el SO mediante la pantalla táctil. La función principal es obtener y ejecutar los comandos del usuario.

Llamadas a sistema

Proporciona una interfaz entre el SO y los programas de aplicación, permite a los programas solicitar servicios al SO. Se utiliza la API para tener el conjunto de funciones que se pueden utilizar.

Implementación de llamadas a sistema

Cada llamada a sistema tiene un número asociado, el cual se almacena en un registro y se ejecuta una instrucción de interrupción. El SO tiene una tabla de interrupciones que contiene las direcciones de memoria de las rutinas de servicio.

Tipos de llamadas a sistema

- Control de procesos: Creación, terminación, suspensión, etc.
- Manipulación de archivos: Creación, eliminación, apertura, cierre, etc.
- Manipulación de dispositivos: Lectura, escritura, apertura, cierre, etc.
- Mantenimiento de información: Obtener información del sistema, configuración, etc.
- Comunicaciones: Creación, eliminación, envío, recepción, etc.

Programas del sistema

Proporcionan un entorno cómodo para desarrollar y ejecutar programas:

- Administrador de archivos: Creación, eliminación, copia, etc.
- Información de estado: Información sobre cosas sencillas (hora) y complejas (rendimiento).
- Modificación de archivos: Edición de archivos.

- Soportes de lenguaje: Compiladores, ensambladores, etc.
- Carga y ejecución de programas: Carga de programas en memoria y ejecución.
- Comunicaciones: Mecanismos para crear conexiones virtuales entre procesos, usuarios y computadoras.

Diseño e implementación de un SO

Métodos adecuados para el diseño e implementación de un SO:

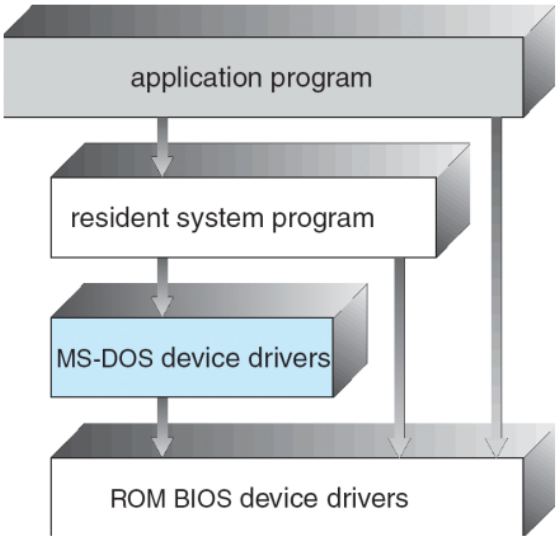
- Objetivos de diseño: definir los objetivos del sistema.
  - Elegir HW.
  - Tipo de sistema (por lotes, tiempo compartido, etc).
  - Objetivos del usuario (comodidad, facilidad, etc) y de sistema (facilidad de diseñar, implementar, mantener, etc).
- Mecanismos (como hacer algo) e implementación de políticas (que hacer).
  - Ejemplo: El temporizador es un mecanismo para controlar la CPU, la política es que cada proceso tiene un tiempo limitado de ejecución.
  - Separación de mecanismos y políticas: Los mecanismos deben ser generales y las políticas específicas. Permite la flexibilidad para el cambio futuro de las políticas.
- Implementación
  - Completado el diseño debe implementarse, tradicionalmente en ensamblador, pero hoy en dia se utiliza C y C++.

Estructura de un SO

La ingeniería de un SO debe hacerse de tal forma que sea fácil de modificar y funcione de la manera esperada. Un método habitual para lograr estos puntos es dividir en componentes mas pequeños en lugar de un sistema monolítico(estructura modular). Cada uno de los componentes deben definirse bien con E/S y sus funciones especificadas.

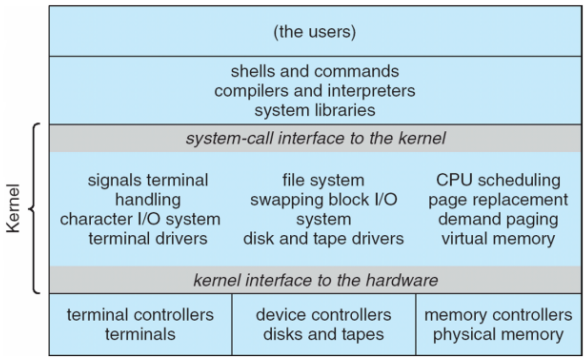
Estructura simple

- No existe protección ni multiprogramación.
- Falla en la programación puede encadenar una caída en el sistema.
- MS-DOS (Microsoft Disk Operating System) diseñado para maxima funcionalidad en el menor espacio posible.
- MS-DOS diseñado en HW no daba soporte protección ni modo dual.



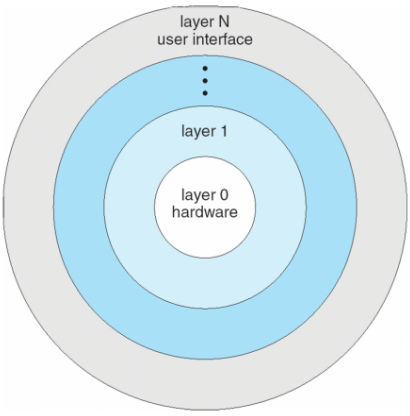
Sistema monolíticos

- Permiten multiprogramación y multiples usuarios.
- SO es conjunto de procedimientos que se agrupan en el núcleo.
- Núcleo protegido (modo dual).
- Núcleo de gran tamaño, no modular.
- Ante cualquier cambio se compila el núcleo por completo



Estructura en niveles

- Mejor modularización y protección de los componentes del sistema.
- Oculta detalles a niveles superiores, dando a los programadores mas libertad para la implementación de rutinas de bajo nivel.
- Comunicación entre niveles significa mucho costo de operación (Overhead).
- Se hace difícil definir bien los niveles ya que cada nivel usa servicios del nivel inferior.

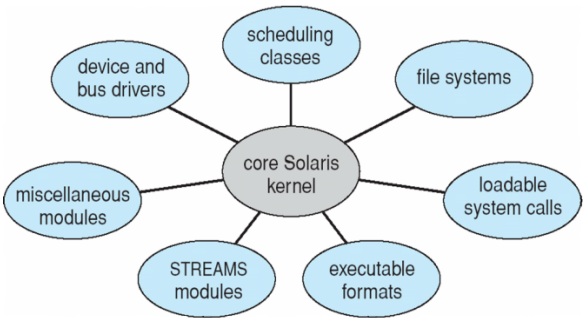


Microkernel

- Quita componentes no esenciales del kernel y los implementa como programas de sistema y de nivel usuario.
- Kernel mas pequeño y fácil de mantener.
- Facilidad ampliación SO (se añaden los servicios nuevos al espacio del usuario sin tener que modificar el kernel).
- Pueden presentar peor rendimiento en comparación a otras estructuras debido a la carga adicional de procesamiento.
- Primeras versiones: Windows NT, QNX, Mach, Chorus, etc.

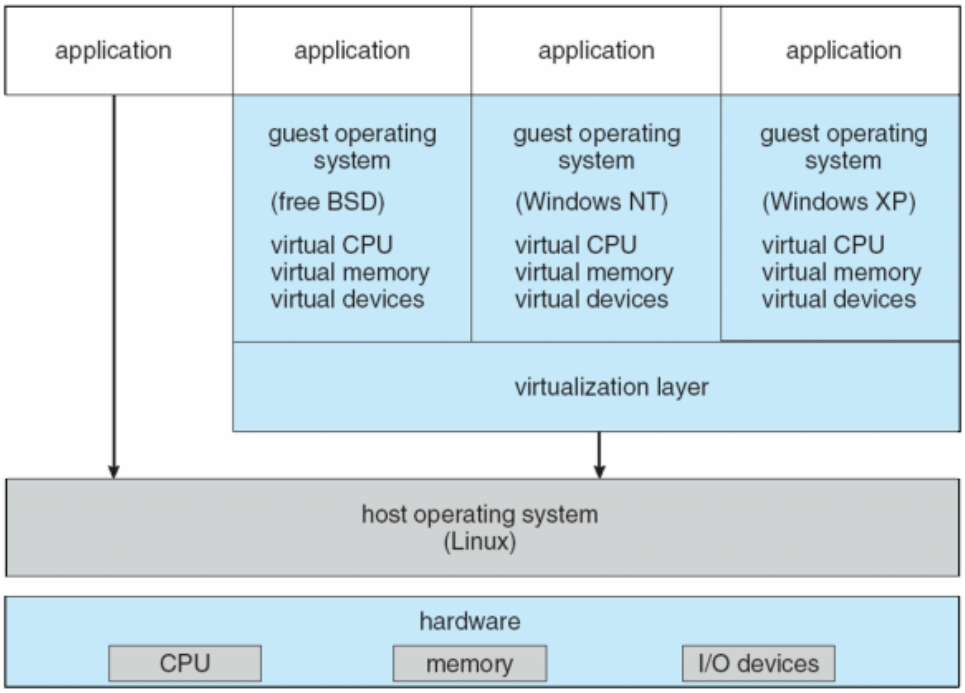
Módulos

- Implementados en los SSOO mas modernos.
- Kernel dispone componentes fundamentales.
- Enlaza de forma dinámica los módulos (servicios adicionales) en tiempo de ejecución.
- Versiones mas conocidas: Solaris, Linus, Mac OS X, etc.



Máquinas virtuales

- Emula una computadora física.
- Permite ejecutar varios sistemas operativos en una misma maquina.
- Se implementa en el nivel de usuario.
- Permite compartir un mismo HW en diferentes sistemas operativos de forma concurrente.
- Ejemplos: Java, .NET, etc.



Procesos

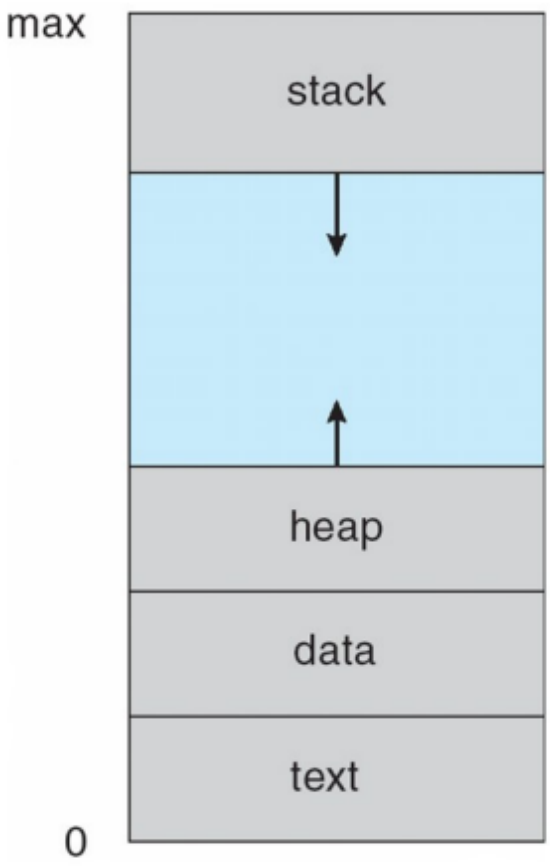
Concepto de proceso

- Programa en ejecución.
- Unidad de trabajo del SO.
- Entidad activa.
- Proceso = Programa + Estado del proceso.

Ademas del código un proceso también tiene:

1. **Contador de programa**: Indica la siguiente actividad actual.
2. **Pila-Stack**: Almacena las llamadas a subrutinas y las variables locales (datos temporales).
3. **Sección de datos**: Contiene variables globales.

Estructura de un proceso en memoria



Text = Código programa
Heap = Memoria global y reservada

Estado del proceso

1. **Nuevo**: Proceso esta siendo creado.
2. **En ejecución**: Instrucciones se están ejecutando.
3. **En espera**: Proceso espera por un evento (terminación operación E/S, recepción de una señal, etc.).
4. **Preparado (Listo)**: Proceso espera por la CPU.
5. **Terminado**: Proceso ha terminado de ejecutarse.

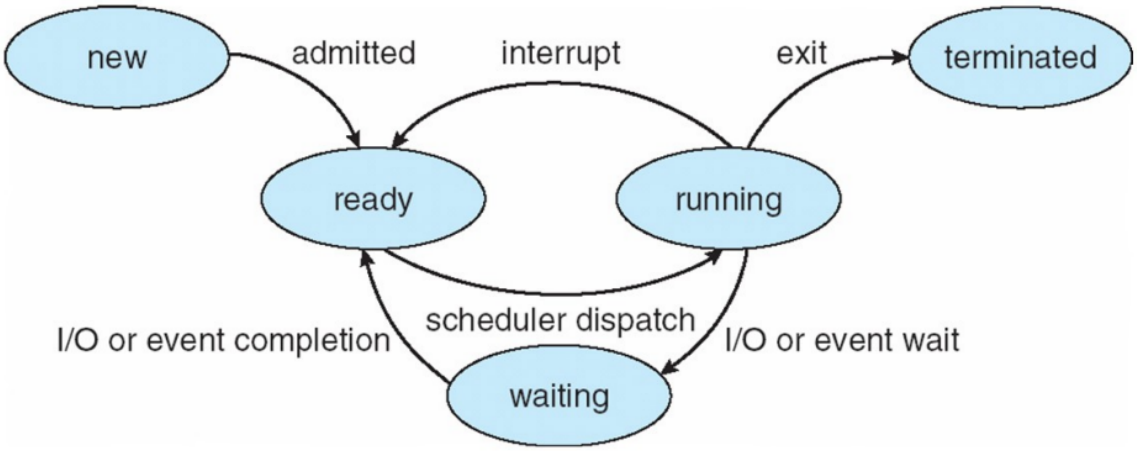


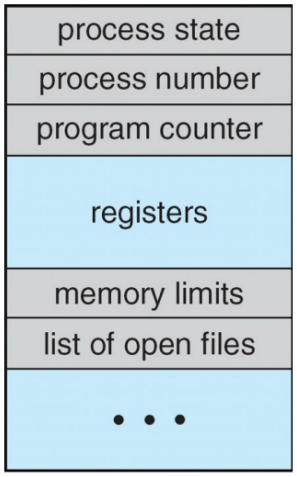
Figura 2: Hormiga de estados de un proceso

Bloque de control de procesos

Cada proceso se representa en el SO mediante un bloque de control de procesos (**PCB: process control block**), este contiene toda la información necesaria para la gestión del proceso. La información contenida en el PCB es:

- Estado del proceso: Nuevo, listo, en ejecución, en espera, terminado, etc.
- Contador de programa: Dirección de la próxima instrucción a ejecutar.
- Registros de la CPU: Contenido de los registros de la CPU (acumulador, contador de instrucciones, etc).
- Información de planificación de la CPU: Prioridad del proceso, tiempo de ejecución, etc.
- Información de gestión de memoria: Información sobre la memoria asignada al proceso (valores de los registros base y límite, tablas de segmentos o paginas, etc.).
- Información contable: Tiempo de CPU utilizado, numero del proceso, tiempo de reloj real, tiempo de CPU en espera, etc.

- Información del estado E/S: Lista de dispositivos E/S asignados al proceso, estado de los dispositivos, etc.



Planificación de procesos

El objetivo de la **multiprogramación** el lograr maximizar la utilización de la CPU ejecutando varios procesos simultáneamente.

El objetivo del **tiempo compartido** es conmutar (cambiar) la CPU entre varios procesos con una frecuencia tan alta que los usuarios puedan interactuar con cada programa mientras se ejecuta.

Para lograr el cumplimiento de estos objetivos la planificación de procesos se encarga de seleccionar el proceso que se ejecutara en la CPU en un momento dado.

Colas de planificación

Apenas los procesos llegan al sistema se colocan en Cola de trabajos la cual se encarga de almacenar todos los procesos que llegan al sistema. Dentro de esta cola se pueden encontrar varias colas de planificación:

- **Cola de procesos preparados (listos)**: Procesos almacenados en memoria principal y listos para ser ejecutados.
- **Cola de espera**: Procesos que están esperando por un evento (E/S, señal, etc).

Estas colas se almacenan en forma de listas enlazadas donde la cabeza de la cola apunta al primer y al ultimo PCB.

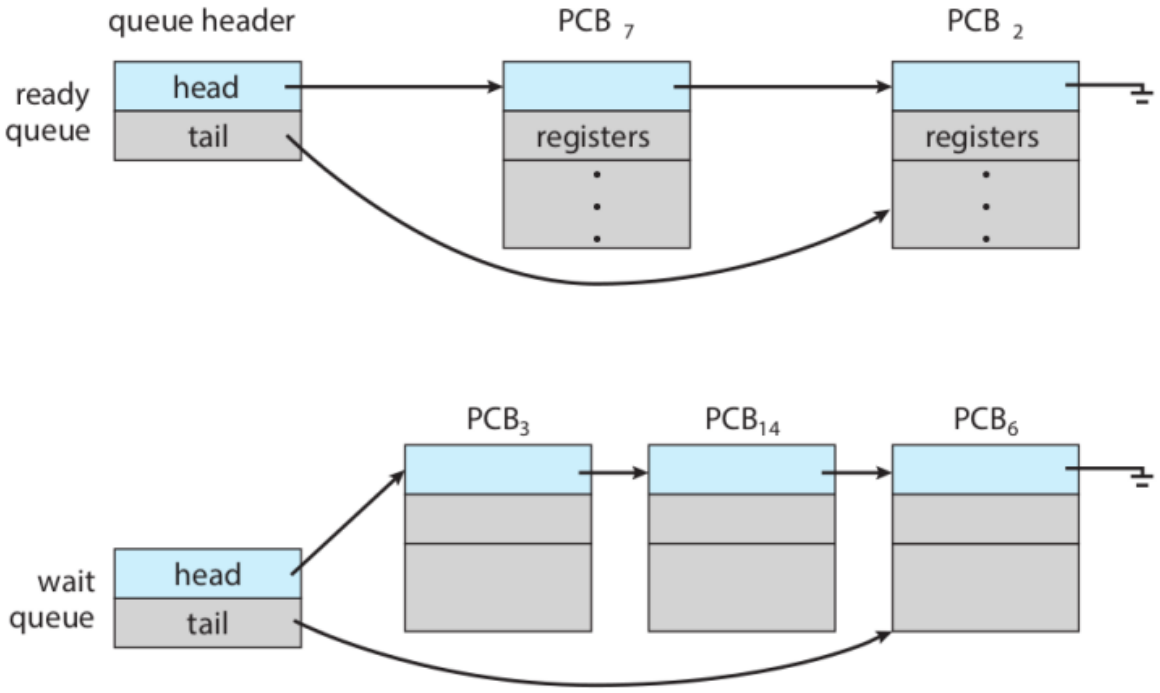
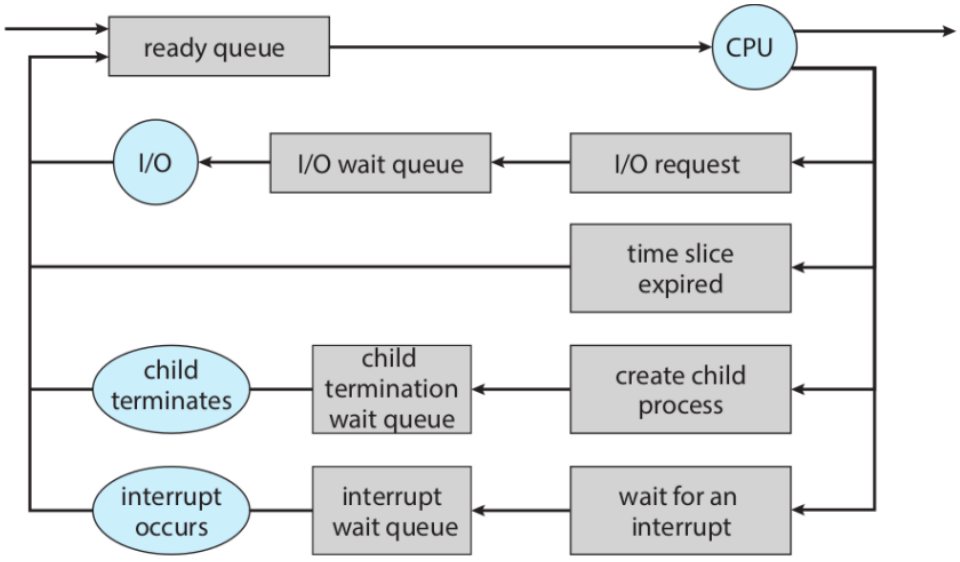


Figura 3: Colas de procesos listos y en espera

Diagrama de colas

A continuación se presentara una representación que explica la planificación de procesos en un SO. Dentro de esta representación se van a encontrar 2 tipos de colas, la cola de procesos listos y la cola de procesos en espera. Los círculos representan los recursos que le dan servicios a las colas y las flechas el flujo que sigue el proceso en el sistema.

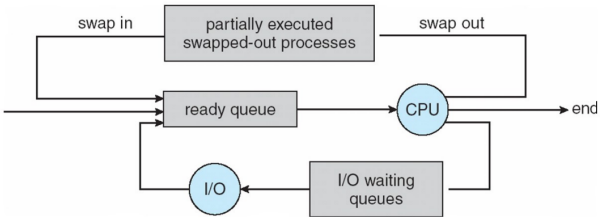


Planificación de CPU

Para entender lo que hace la planificación de CPU se debe entender que los procesos se mueven entre diversas colas durante su ciclo de vida, por lo que es imperante una planificación. El planificador de CPU selecciona un proceso de la cola de procesos listos y lo asigna a la CPU. Esta selección de proceso es muy frecuente ya que normalmente se hace cada 100 ms o menos.

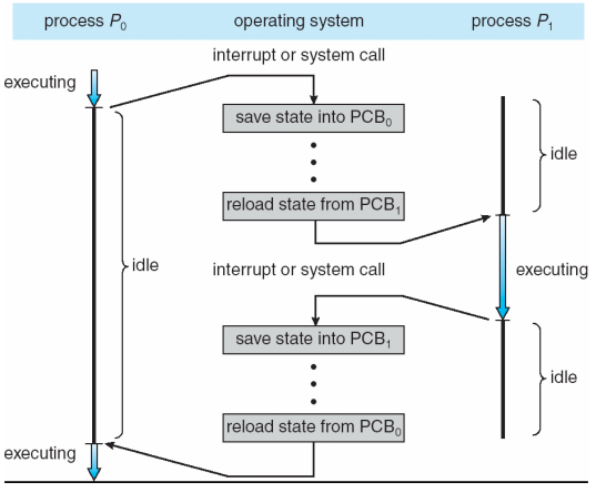
Planificación intermedia - Swapping

Algunos SSOO presentan este tipo de planificación, el cual se basa en un **intercambio**, donde el planificador descarga un proceso (swap out) de la memoria principal a la secundaria y carga otro proceso (swap in) de la memoria secundaria a la principal. Esta planificación logra reducir el grado de la multiprogramación, mejora la mezcla de procesos, libera memoria, etc.



Cambios de contexto

Es una interrupción que provoca que el SO obligue a la CPU a abandonar su tarea actual para ejecutar una rutina del kernel. Cuando se realiza este cambio de contexto el mismo evento debe encargarse de guardar el estado del proceso que va a ser interrumpido y cargar el estado del proceso que va a ser ejecutado. Estos contextos se almacenan en el PCB de cada proceso. Se debe tener en consideración que el tiempo utilizado en el cambio de contexto es tiempo muerto.



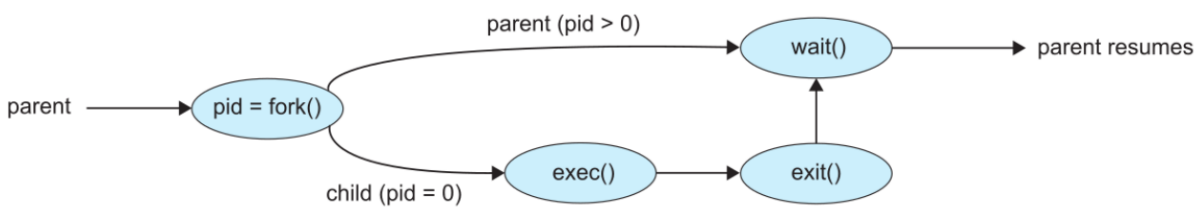
Operaciones sobre los procesos

La **creación** y **eliminación** de procesos debe estar respaldada por mecanismos adecuados, ya que, en la mayoría de los sistemas, los procesos se ejecutan de forma concurrente y se gestionan de manera dinámica.

Creación de procesos

Mientras se ejecuta un proceso, este mismo puede crear uno o varios procesos adicionales a traves de las llamadas a sistema **fork()** o **exec()**. Algunas características de la creación de procesos son:

- Proceso creador se conoce como **proceso padre**.
- Procesos creados se conocen como **procesos hijos**.
- Procesos hijos pueden a su vez crear otros procesos (árbol de procesos).
- Cada proceso, ya sea hijo o padre, presenta un identificador único (pid, process identifier).
- Ya se sabe que un proceso necesita de recursos para funcionar, por lo que cuando un proceso crea otro, este nuevo puede:
  - Obtener recursos propios.
  - Obtener subconjunto de recursos del proceso padre.
- Al crear un proceso hijo, existen 2 posibilidades de ejecución:
  - El padre se ejecuta concurrentemente con el hijo.
  - El padre espera a que el/los hijo/s terminen de ejecutarse a traves de la llamada a sistema `wait()`.
- Existen 2 posibilidades en función del espacio de direcciones del proceso hijo:
  - Hijo es una copia exacta del padre (mismos datos, variables, etc).
  - Hijo es un programa completamente nuevo.
- Para UNIX:
  - Llamada a sistema `fork()` crea un nuevo proceso.
  - Llamada a sistema `exec()` se utiliza después del `fork()` y se encarga de reemplazar el espacio de direcciones del proceso actual con un nuevo programa.



Eliminación de procesos

Un proceso se elimina cuando ha terminado de ejecutarse, ya sea por finalización normal o por error, y pide al SO que lo elimine a traves de la llamada a sistema `exit()`. Algunas características de la eliminación de procesos son:

- Proceso hijo puede devolver su estado al padre mediante `wait(&status)`.
- SO libera los recursos de un proceso eliminado.
- Proceso padre puede eliminar a un proceso hijo por las siguientes razones:
  - Hijo excede los recursos permitidos.
  - Tarea del hijo ya no es necesaria.
  - Padre abandona el sistema, el SO no permite la ejecución de procesos huérfanos.

Comunicaron interprocesos

Los procesos que se ejecutan concurrentemente pueden ser:

- **Independientes**: No comparten datos.
- **Cooperativos**: Comparten datos.

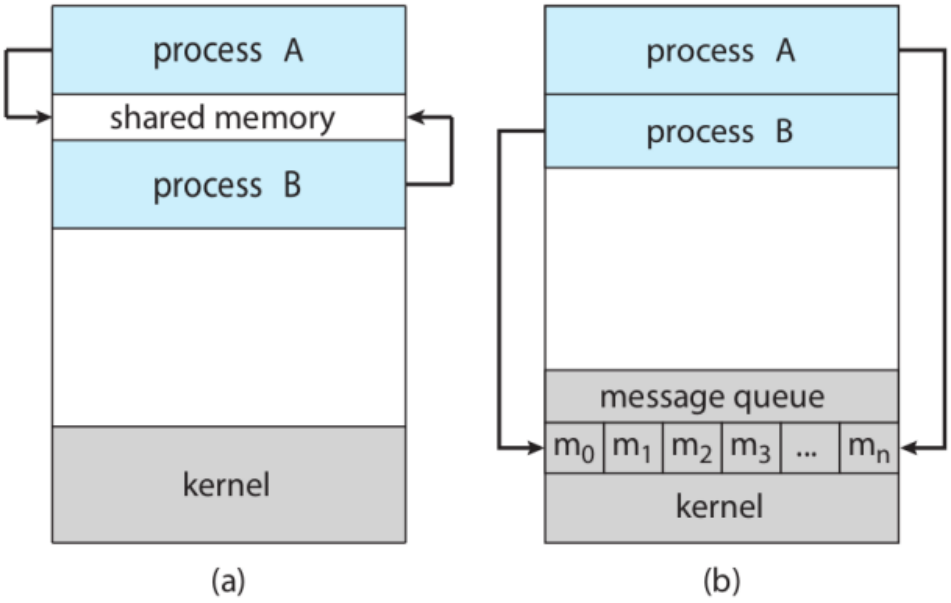
El permitir la cooperación entre procesos puede ser beneficioso, ya que:

- Permite a los procesos cooperar en la realización de una tarea.
- Permite a los procesos compartir datos (varios usuarios pueden acceder a la misma data).
- Permite a los procesos comunicarse.
- Acelera cálculos y divide tareas.
- Ejecución en paralelo (varias CPU's).
- Permite la construcción de sistemas modulares (modularidad), dividiendo las funciones del sistema en diferentes procesos.

La cooperación entre procesos necesita de mecanismos de comunicación (IPC, Interprocess Communication) que permitan a los procesos compartir datos y sincronizar sus actividades. Existen 2 tipos de modelos de IPC:

- **Modelo de memoria compartida**:
  - Region en memoria la cual es compartida por varios procesos cooperativos.
  - Intercambio de información entre procesos escribiendo/leyendo en la región de memoria compartida.
  - Permite velocidades máximas (velocidad de memoria).

- **Modelo de paso de mensajes**:
  - Comunicación mediante intercambio de mensajes.
  - Mas complejidad en la implementación.
  - Menor velocidad ya que usa llamadas al sistema, teniendo que intervenir el kernel.



(a) Memoria compartida (b) Paso de mensajes

Memoria compartida -> API's

Para hacer uso de la memoria compartida se pueden/deben utilizar las siguientes API's:

- IEEE POSIX Threads (PThreads):
  - Standard Unix threading API (Windows)
  - Más de 60 funciones: pthread create, pthread join, pthread exit, etc.
  - Bajo nivel.
  - Manejo explicito de hebras (creación, terminación, sincronización, etc).
- OpenMP:
  - API de programación en paralelo.
  - Directivas de compilador.
  - API de funciones.
  - Alto nivel de abstracción.
  - Manejo implícito de hebras.
  - Disponible para C, C++, Fortran.
  - Énfasis HPC (High Performance Computing).

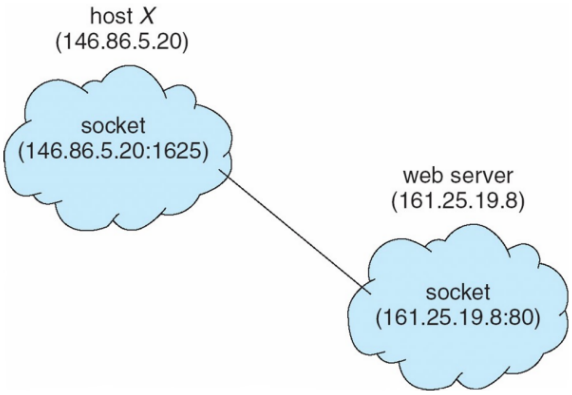
Paso de mensajes -> API's

Se observan varios procesos ejecutándose en diferentes nodos, de los cuales cada proceso tiene su espacio de direcciones privado. El acceso de los datos se realiza de forma remota mediante envió/recepción de mensajes, donde MPI (Message Passing Interface) es la API más utilizada para la programación de paso de mensajes.

Sockets

Los mecanismos de comunicación vistos anteriormente son utilizados para la comunicación entre procesos en la misma máquina. Para la comunicación entre procesos en diferentes máquinas se utilizan los sockets. Los sockets son un mecanismo de comunicación que permite la comunicación entre procesos en diferentes máquinas a través de la red. Los sockets son utilizados en la programación de redes y permiten la comunicación entre procesos en diferentes máquinas. La forma de utilización de sockets se puede resumir en los siguientes pasos:

- Par de procesos emplean una pareja de sockets para establecer comunicación.
- Se identifican por una dirección IP y un número de puerto.
- Servidores implementan servicios que **escuchan** en un puerto conocido (80->HTTP, 21->FTP, etc).
- Cliente X `146.86.5.20:1625` tiene establecida una conexión con el servidor web: `161.25.19.8:80`



4. Hebras (Threads)

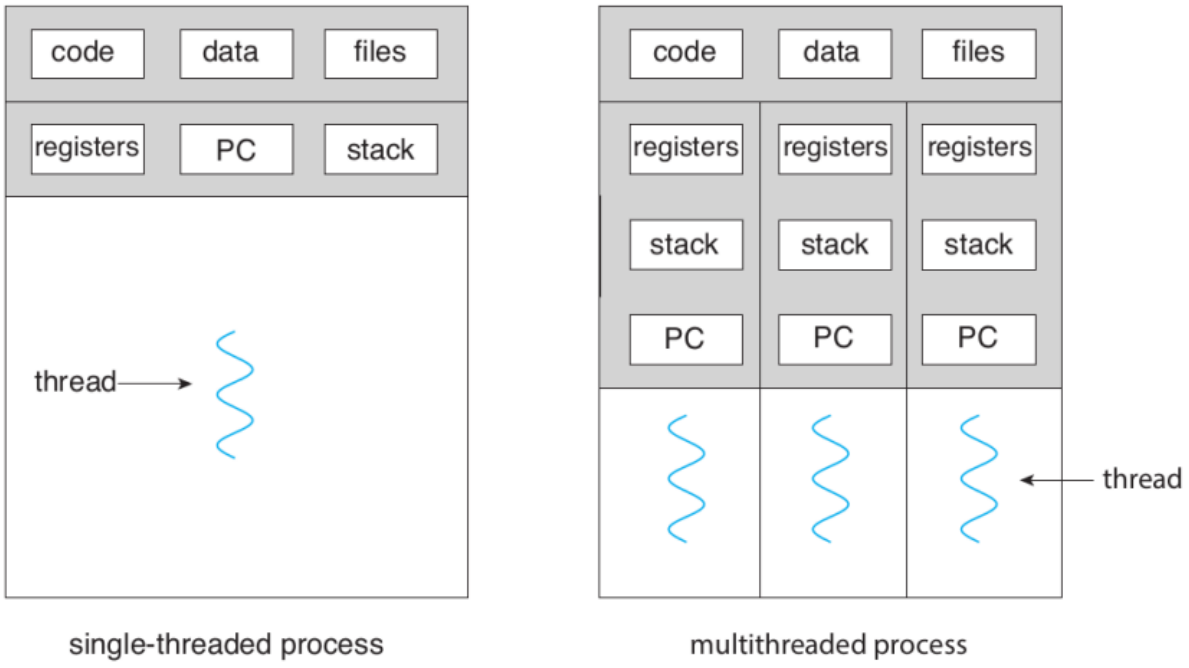
Las hebras son la unidad básica de utilización de la CPU. Estas comprenden:

- ID de la hebra.
- Contador de programa.
- Conjuntos y registros de pila.

Y comparte con otras hebras:

- Sección de código.
- Sección de datos.
- Otros recursos del SO.

Un proceso tradicional comprende solo una hebra de control, osea **monohebra (single-threaded)**. De igual manera un proceso con múltiples hebras se conoce como proceso **multihebras (multithreaded)**.

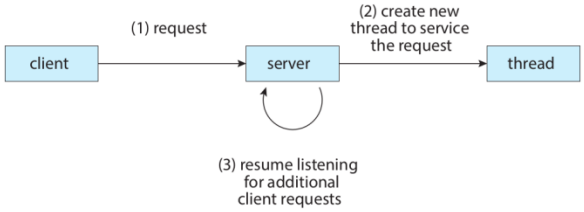


Hoy en dia, muchos procesos comprenden las multihebras, algunos ejemplos son:

- Aplicación para crear imágenes miniatura (thumbnail) desde una colección de imágenes.
- Un navegador web puede tener una hebra para la interfaz gráfica y otra para la descarga de archivos.
- Un procesador de texto puede tener una hebra enfocada en mostrar gráficos mientras otra se encarga en recibir lo digitado por el usuario y otra para verificar la ortografía.

Con esto en cuenta es posible requerir una sola aplicación para realizar varias tareas similares de manera simultanea.

- Un servidor web acepta solicitudes de clientes por paginas web, imágenes, sonido, etc.
- Sometido a gran carga puede tener miles de solicitudes concurrentes.
- Si proceso tiene sola una hebra, atiende una solicitud por vez.
- Si proceso crea otros procesos, fork(), lleva tiempo y hace uso intensivo de recursos.
- Si proceso es multihebra, por cada solicitud crea una nueva hebra.



Los SSOO también pueden ser multihebras, por ejemplo Linux al iniciarse crea varias hebras (kernel threads) para tareas especificas como la administración de dispositivos, de memoria, manejo de interrupciones, etc. Las aplicaciones también se pueden diseñar con este formato para aprovechar las capacidades de los procesadores multihebras.

Algunas de las ventajas de las hebras son:

- **Capacidad de respuesta**: Si una hebra se bloquea, las demás pueden continuar.
- **Compartición de recursos**: Las hebras comparten recursos del proceso al que pertenecen. Permite a una aplicación tener varias hebras diferentes en un mismo espacio de direcciones.
- **Economía**: Crear una hebra es más económico que crear un proceso tanto en la asignación de recursos como de memoria.
- **Escalabilidad**: Las hebras pueden ser distribuidas en diferentes procesadores, siempre y cuando exista una arquitectura multiprocesador.

Programación multinúcleo

Antes los sistemas operativos eran mononúcleo (single-core), pero hoy en día la mayoría de los sistemas son multinúcleo, por lo que la programacion multihebra proporciona mecanismos para el uso eficiente de estos multiples núcleos. Si pensamos en una aplicación con cuatro hebras:

- En un sistema con un solo núcleo de computación, la concurrencia significa que la ejecución de las hebras se entrelazara en el tiempo.

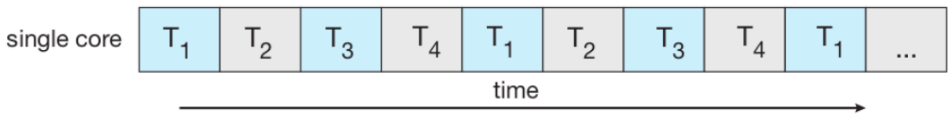


Figura 4: Ejecución single-core

- En un sistema con multiples núcleos, sin embargo, las hebras pueden ejecutarse en paralelo. El sistema puede asignar una hebra a cada núcleo

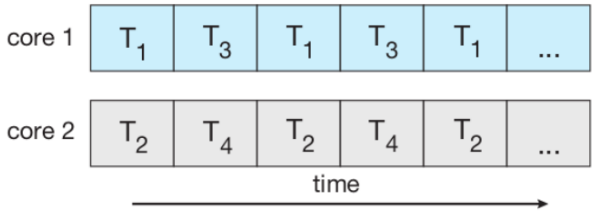


Figura 5: Ejecución paralela multi-core

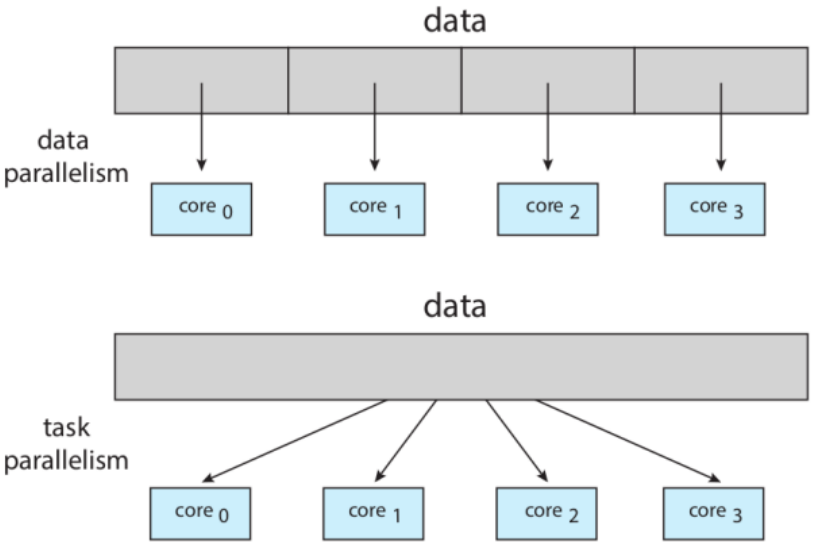
Desafíos de la programación multinúcleo

Al diseñar o modificar programas existentes es necesario tener en cuenta los siguientes desafíos:

- Identificación de tareas que puedan ejecutarse idealmente de manera independiente.
- Lograr un balance en las cargas de trabajo entre las hebras.
- Division de datos para ser usados y manipulados en los diferentes núcleos.
- Identificar dependencia de datos que pueden ser accedidos por una o más tareas.
- Sincronización de accesos en los datos compartidos.
- Pruebas y depuración aumentan en complejidad.

Tipos de paralelismo

- **Paralelismo de datos**: Se refiere a la distribución de subconjuntos de datos en los multiples núcleos y repetir esto para cada núcleo.
- **Paralelismo de tareas**: Se refiere a la distribución de tareas en los diferentes núcleos.
- Se pueden utilizar las 2 técnicas en conjunto ya que no son excluyentes.



Modelos multihebra

En la practica el soporte de hebras puede proporcionarse como:

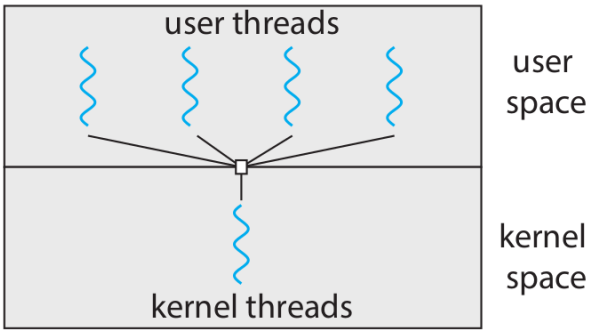
1. **Hebras de usuario**: Soporte por sobre el kernel, gestión de hebras sin soporte del kernel (POSIX Pthreads, Win32 threads, Java threads).
2. **Hebras de kernel**: Kernel soporta y gestiona las hebras (Windows, Linux, macOS).

Debe existir una relación entre las hebras de usuario y las de kernel:

- Modelo muchos a uno
- Modelo uno a uno
- Modelo muchos a muchos

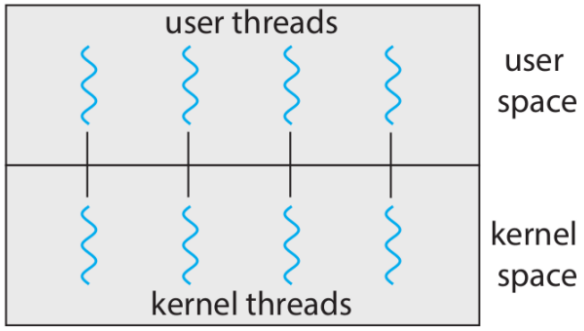
Modelo muchos a uno

- Cada hebra de usuario es mapeada a una hebra de kernel.
- Gestión de hebras en espacio de usuario, mediante una librería.
- Proceso completo se bloque si una hebra hace una llamada bloqueante.
- Solo una hebra puede acceder al kernel a la vez, no se puede aprovechar el paralelismo.



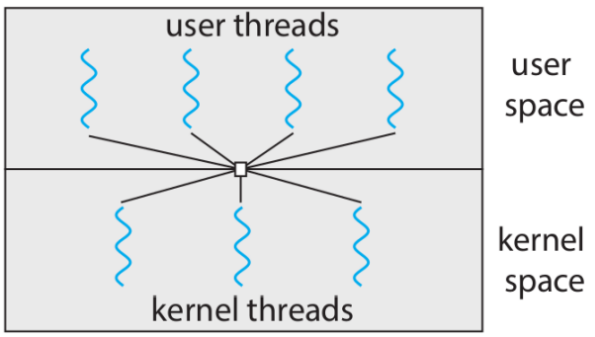
Modelo uno a uno

- Cada hebra de usuario es mapeada a una hebra de kernel.
- Mayor concurrencia que el modelo muchos a uno.
- Permite a una hebra bloquearse sin afectar a las demás.
- Permite ejecución en varios procesadores.
- Se restringe el numero de hebras soportadas por el sistema para no afectar al rendimiento.
- Windows, Linux.



Modelo muchos a muchos

- Asigna muchas hebras de usuario a un numero igual o menor de hebras de kernel.
- Se pueden crear tantas hebras de usuario como se desee.
- Las hebras de kernel correspondientes se ejecutan en paralelo en un sistema multiprocesador



Bibliotecas de hebras

Son necesarias ya que proporcionan al programador una API para gestionar y crear hebras, existen 2 formas de implementar estas bibliotecas:

- Por completo en el espacio de usuario, sin soporte del kernel.
  - Estructuras de datos y código de biblioteca en espacio de usuario.
  - Funciones producen llamadas locales y no al sistema.
- Soporte directo del SO
  - Estructuras de datos y código de biblioteca en espacio del kernel.
  - Funciones producen llamadas al sistema.

Las 3 principales bibliotecas son:

- **POSIX Pthreads**: API de hebras POSIX, soportada por la mayoría de los sistemas UNIX. Nivel de usuario/kernel.
- **Windows threads**: API de hebras de Windows, soportada por Windows. Nivel de kernel.
- **Java threads API**: API de hebras de Java, soportada por Java. Utiliza librería de hebras existente en el kernel.

Pthreads

- Basado en el estándar POSIX (IEEE 1003.1c) que define una API para la creación y sincronización de hebras.
- Standard UNIX threading API.
- Más de 60 funciones: `pthread_create`, `pthread_join`, `pthread_exit`, etc.
- Relativamente de bajo nivel.
- Manejo explícito de las hebras (creación, eliminación de hebras, sincronización).
- Linux, macOS, implementan la especificación Pthreads.

OpenMP

- Extensión de lenguaje basado en directivas de compilación y biblioteca de funciones.
- Disponible para C/C++ y Fortran.
- Énfasis en high-performance computing (HPC).
- Alto nivel de abstracción.

5. Planificación del CPU

En sistemas con un solo procesador, solo es posible ejecutar un proceso a la vez. Dado que el objetivo de la multiprogramación es maximizar el uso de la CPU, es necesario planificar cuidadosamente la ejecución de los procesos.

En sistemas simples, la CPU queda inactiva mientras espera las operaciones de E/S. Sin embargo, en sistemas más complejos, otros procesos pueden ejecutarse durante ese tiempo de espera.

La ejecución de procesos incluye:

- **Ráfaga de CPU**: Ciclo de ejecución de instrucciones de un proceso.
- **Ráfaga de E/S**: Ciclo de ejecución de instrucciones de un proceso que espera por una operación de E/S.

La ejecución de un proceso comienza con una ráfaga de CPU, E/S, etc. La CPU concluye con la solicitud al sistema para finalizar la ejecución.

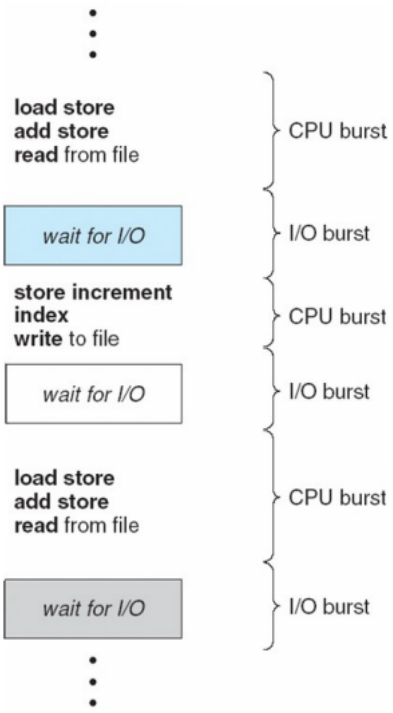


Figura 6: Ciclos de ráfagas de CPU y E/S

Planificador de la CPU

Recordando los estados de un proceso (ver Hormiga de estados), observamos las diferentes etapas por las que atraviesa un proceso. El planificador se encarga de seleccionar un proceso de la cola de procesos listos y asignarlo a la CPU.

Esta asignación no necesariamente sigue un formato FIFO, ya que se pueden emplear diferentes algoritmos de planificación, como planificación por prioridad, planificación en árboles, entre otros. Es importante destacar que los elementos en las colas son PCBs (Process Control Blocks), que contienen la información necesaria para gestionar cada proceso (registros).

Momento de elección planificación

La decision de cuando utilizar la planificación es esencial en estas situaciones:

1. Cambio de estado de ejecución (running) a espera (waiting): Solicitud de E/S, espera de un evento, invocación a wait, etc.
2. Cambia de estado de ejecución (running) a listo (ready): Interrupción de temporizador, fin de la ráfaga de CPU.
3. Cambia de estado de espera (waiting) a listo (ready): Se completa la operación de E/S.
4. Termina (terminated)

En planificación **sin expropiación** el proceso en ejecución no se interrumpe hasta que ocurra 1 o 4 en los cuales entrega la CPU por propia voluntad. En el resto de los casos se entiende que sucede una planificación **con expropiación**.

Despachador (Scheduler Dispatch)

El despachador es un componente que participa en la planificación de la CPU. Su función es seleccionar un proceso de la cola de procesos listos y asignarlo a la CPU. Dentro de esta función se incluye:

- Cambio de contexto: Guardar el estado del proceso que se interrumpe y cargar el estado del proceso que se va a ejecutar.
- Cambio de modo usuario: Cambiar de modo kernel a modo usuario.
- Salto a posición correcta dentro del programa de usuario para reiniciar la ejecución.

Este componente debe ser lo mas rápido posible, ya que el tiempo de cambio de contexto es tiempo muerto y se invoca en cada conmutación de proceso. La **latencia de despacho** es el tiempo que tarda en detener un proceso e iniciar otro.

Criterios de planificación

Dentro de los diferentes algoritmos de planificación existen propiedades que favorecen a una clase de procesos sobre otra. Las propiedades más comunes son:

- **Utilización de la CPU**: Porcentaje de tiempo que la CPU se encuentra ocupada.
- **Tasa de procesamiento - Throughput**: Número de procesos que se completan por unidad de tiempo.
- **Tiempo de ejecución - Turnaround time**: Tiempo transcurrido desde que un proceso entra al sistema hasta que termina (carga de memoria + tiempo de espera + tiempo de ejecución + operaciones de E/S).
- **Tiempo de espera - Waiting time**: Suma de los tiempos que los procesos pasan en la cola de listos.
- **Tiempo de respuesta - Response time**: Tiempo transcurrido desde que se envía una solicitud hasta que se obtiene la primera respuesta.

Criterios de optimización

Ya conociendo estas propiedades, se pueden establecer criterios de optimización para los diferentes algoritmos de planificación:

- Maximizar la utilización de la CPU.
- Maximizar tasa de procesamiento (throughput): Procesar la mayor cantidad de procesos por unidad de tiempo.
- Minimizar el tiempo de ejecución (turnaround time): Procesos deben ser completados lo más rápido posible.
- Minimizar el tiempo de espera (waiting time): Procesos deben ser atendidos lo más rápido posible.
- Minimizar el tiempo de respuesta (response time): Procesos deben ser atendidos lo más rápido posible, especialmente en sistemas de tiempo compartido.

Algoritmos de planificación

Estos algoritmos son la respuesta a la necesidad de optimizar los criterios de planificación. Estos algoritmos puede que resuelvan solo algunos de los criterios de optimización, pero no todos. Algunos de los algoritmos más comunes son:

FCFS (First-Come, First-Served)

Se basa en que el primer proceso que llega es el primero en ser atendido. Es un algoritmo no expropiativo, por lo que el proceso en ejecución no se interrumpe hasta que llega a un estado de espera o termina.

SJF (Shortest-Job-First)

Dentro de los procesos en la lista de listos, se selecciona el proceso con el tiempo de ráfaga de CPU más corto. En el caso de que existan procesos con el mismo tiempo de ráfaga, se selecciona el que llegó primero. A pesar de parecer un buen modelo existe una complicación ya que no siempre se esta seguro del tiempo de ráfaga de CPU.

Propiedades:

- A cada proceso se le asigna una **prioridad**.
- Proceso con mayor prioridad se le asigna la CPU.
- Procesos con la misma prioridad se manejan en **FCFS**.
- Las propiedades pueden ser:
  - Apropiativa: Con un procesos en ejecución, se puede cambiar a otro proceso con mayor prioridad.
  - Cooperativa: Proceso con mayor prioridad se coloca al inicio de la cola.

- Problema: **Inanición** (starvation) - Procesos con baja prioridad pueden no ser atendidos.
- Solución: **Envejecimiento** (aging) - Incrementar la prioridad de los procesos que esperan mucho tiempo.

Round Robin

Este algoritmo es un planificador por tiempo, donde cada proceso recibe un tiempo de ejecución fijo, llamado **quantum** (q = 10 - 100 ms). Si el proceso no ha terminado su ejecución al final del quantum, se coloca al final de la cola de listos y se selecciona el siguiente proceso. Este quantum debe asignarse de manera que no sea demasiado grande, ya que el algoritmo se convierte en FCFS, y no sea demasiado pequeño, ya que si es menor que el tiempo de cambio de contexto los procesos puede que no se ejecuten.

Colas multinivel

Este enfoque clasifica los procesos en grupos distintos, cada uno de los cuales utiliza un algoritmo de planificación específico. Los procesos se organizan en estas colas según sus características o prioridades, permitiendo la aplicación de diferentes estrategias de planificación en cada grupo.

- De primer plano (**foreground**): Se utiliza el algoritmo Round Robin.
- De fondo (**background**): Se utiliza el algoritmo FCFS.

Esta diferenciación es aplicada sobre la cola de procesos listos, y como se menciona anteriormente, cada grupo tiene un algoritmo de planificación diferente.

Colas multinivel retroalimentadas

Este enfoque es similar al de las colas multinivel, pero con la particularidad de que los procesos pueden moverse entre las distintas colas. Si un proceso consume mucho tiempo de CPU, se transfiere a una cola con un quantum más largo. En cambio, si un proceso utiliza poco tiempo de CPU, se desplaza a una cola con un quantum más corto.

La capacidad de moverse entre colas es posible gracias a la retroalimentación que caracteriza a este tipo de planificación. Este mecanismo ayuda a evitar el bloqueo indefinido y la inanición, ya que los procesos que no reciben suficiente tiempo de CPU pueden ser transferidos a una cola con un quantum más adecuado.

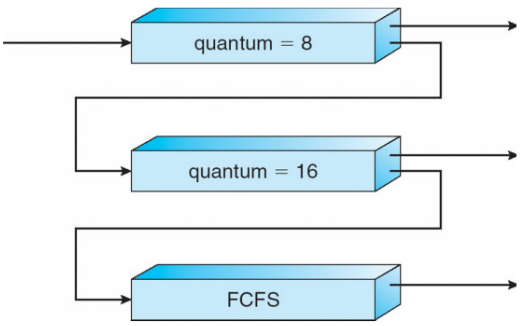


Figura 7: Ejemplo de colas multinivel retroalimentadas

6. Sincronización de procesos

El acceder concurrentemente a los datos compartidos puede resultar en datos inconsistentes por lo que mantener datos que sean consistentes requiere de mecanismos para asegurar una ejecución ordenada de procesos cooperativos.

Problema consumidor-productor

El problema consumidor-productor se basa en que en algún punto de un sistema existen procesos que producen datos y otros que los consumen. El problema radica en que los consumidores no pueden consumir datos que no han sido producidos y los productores no pueden producir datos si no hay espacio para almacenarlos.

Condición de carrera (Race Condition)

Sucede cuando dos rutinas funcionan correctamente por separado, pero al ejecutarse concurrentemente, el resultado no es el esperado. Esto se debe a que las rutinas comparten recursos y no se han tomado medidas para evitar la interferencia.

Problema de la sección crítica

La sección crítica es una parte del código donde los procesos acceden a los recursos compartidos (acceso a datos, actualización tablas, etc). Por lo tanto cuando un proceso esta ejecutando su sección crítica, ningún otro proceso debe ser capaz de ejecutar su propia sección crítica. En un proceso la estructura general es:

- Sección de entrada (solicitud para entrar en la sección crítica).
- Sección crítica (acceso a los recursos compartidos).
- Sección de salida (liberación de los recursos).
- Sección restante.

El problema de la sección crítica consiste en encontrar un mecanismo que permita a los procesos cooperar sin interferir entre ellos.

Cualquier solución que se proponga al problema de la sección crítica debe cumplir con los siguientes requisitos:

1. Exclusion mutua: Solo un proceso puede ejecutar su sección crítica a la vez.
2. Progreso: Si ningún proceso está ejecutando su sección crítica y hay procesos que desean entrar en su sección crítica, solo los procesos que no están en su sección crítica pueden decidir quién entra. Esta decisión no puede posponerse indefinidamente.
3. Espera limitada: Si un proceso desea entrar en su sección crítica, debe ser capaz de hacerlo eventualmente.

Los procesos en modo kernel pueden estar sujetos a posibles condiciones de carrera.

Métodos para gestionar la sección critica en SSOO

- **Kernel apropiativos**
  - Proceso puede desalojarse cuando se ejecuta en modo kernel.
  - Difícil de implementar en arquitecturas SMP.
  - Versiones comerciales de UNIX (Solaris, IRIX).
  - Linux V2.6.
  - Windows 7, 10.
- **Kernel no apropiativos**
  - Proceso en modo kernel no se desaloja, se ejecuta hasta que salga del modo kernel, termine, se bloquee o ceda la CPU voluntariamente.
  - Libre de condiciones de carrera, solo permite un proceso en modo kernel.
  - Anteriores a Windows 95, kernel tradicional de UNIX y Linux hasta V2.6.

Soluciones al problema de la sección crítica

Dentro de la gama de soluciones, el elemento principal que han de tener es la implementación de un **cerrojo**, el cual evita que las condiciones de carrera se produzcan protegiendo las secciones críticas de los procesos. Este cerrojo es un hardware de sincronización el cual se caracteriza por:

- Ser un soporte de HW para las secciones criticas.
- En las maquinas monoprocesadores deshabilitan las interrupciones.
- La ejecución del código debe ser sin apropiación.
- La instrucción `test_and_set()` es la más utilizada. Sirve para leer y modificar atomicamente una variable. Es una instrucción que no puede ser interrumpida.
- Implementarlo en un sistema multiprocesador es más complicado.

Sincronización en Linux con Pthreads

Esta API proporciona cerrojos **mutex**, variables de condición y cerrojos de lectura/escritura para sincronizar las hebras.

- `#include <pthread.h>`
- `pthread_mutex_t mutex;`
- `pthread_mutex_init(&mutex, NULL);`
- `pthread_mutex_lock(&mutex);`
- `pthread_mutex_unlock(&mutex);`

Sincronización en Linux con clase threads (C++11)

Esta API no proporciona cerrojos pero si variables de condición para sincronizar las hebras. Los cerrojos se pueden utilizar con otra API.

- `#include <mutex>`
- `#include <thread>`
- `std::mutex mutex;`
- `mutex.lock();`
- `mutex.unlock();`

7. Bloqueos mutuos

En entornos multiprogramación varios procesos compiten por una cantidad limitada de recursos, por lo tanto el orden en que se van consumiendo los recursos puede afectar el rendimiento del sistema. Un bloqueo mutuo (deadlock) es una situación en la que dos o más procesos no pueden continuar ejecutándose porque cada uno de ellos espera un recurso que posee otro proceso.

Modelo del sistema

Como ya se menciono en estos entornos los recursos escasos son distribuidos entre los **procesos competitivos**. Estos recursos se clasifican por tipos pudiendo ser ciclos de CPU, espacio de memoria, impresoras, etc. El protocolo de acceso a los recursos para los procesos se basa en la siguiente secuencia:

1. ***Solicitar recurso***: Si el recurso esta disponible, se asigna al proceso, si no se bloquea.
2. ***Usar el recurso***: El proceso puede utilizar el recurso asignado.
3. ***Liberar recurso***: Una vez que el proceso ha terminado de utilizar el recurso, lo libera.

Condiciones para el bloqueo mutuo

Una situación de deadlock se produce cuando se cumplen las siguientes condiciones **simultáneamente**:

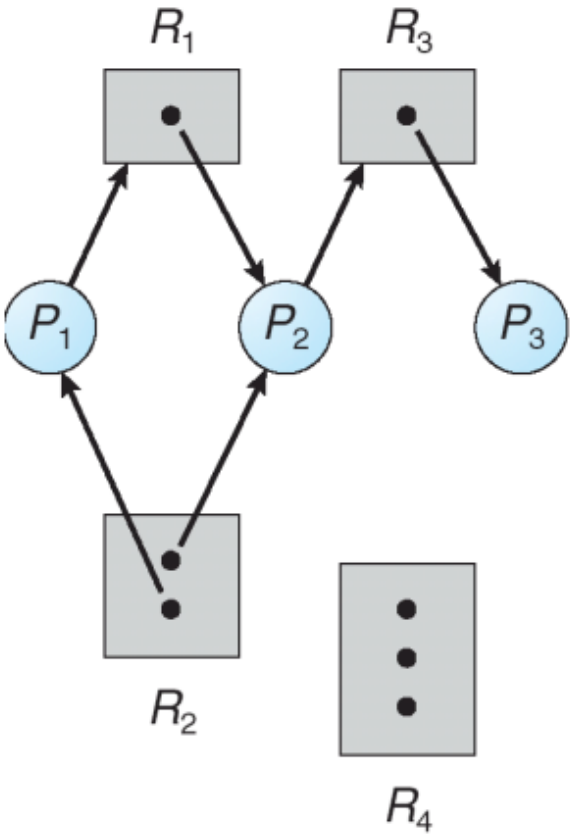
- **Exclusion mutua**: Debe haber a lo menos un recurso que solo pueda ser utilizado por un proceso a la vez (modo exclusivo).
- **Retención y espera**: Un proceso que ya tiene un recurso puede solicitar otro recurso y esperar a que se libere.
- **Sin desalojo**: Los recursos asignados a un proceso no pueden ser desalojados.
- **Esperar circular**: Dos o más procesos están esperando a que se libere un recurso que posee otro proceso.

Grafos de asignación de recursos

Para describir una situación de deadlock se puede utilizar un **grafo de asignación de recursos**, el cual se compone de:

- Vertices (2 tipos):
  - Procesos (P): $P_1, P_2, ..., P_n$ Representados por un círculo.
  - Recursos (R): $R_1, R_2, ..., R_m$ Representados por un cuadrado.
- Aristas (2 tipos):

- $P_i \to R_j$: Asignación de recurso R_j al proceso P_i, arista de solicitud.
- $R_j \to P_i$: Proceso P_i esta esperando el recurso R_j, arista de asignación.



$$P = \{P_1, P_2, P_3\}$$
$$R = \{R_1, R_2, R_3, R_4\}$$
$$\{P_1 \to R_1, P_2 \to R_3, R_1 \to P_2, R_2 \to P_1, R_2 \to P_2, R_3 \to P_3\}$$
Instancias: $\#R_1 = 1$, $\#R_2 = 2$, $\#R_3 = 1$, $\#R_4 = 3$

Condiciones de deadlock

- **Ciclo en el grafo**(condición necesaria): Si se encuentra un ciclo en el grafo, entonces es posible que exista un deadlock.
- **Ciclo y Recursos**(condición necesaria y suficiente): Si existe un ciclo en el grafo y todos los recursos tienen solo una instancia, entonces es seguro que existe un deadlock.

Para los casos en los que no exista ciclos en los grafos, **no existe deadlock**.

Métodos de manejo de deadlock

- Ignorar el problema y esperar que no ocurra.
- Protocolos que aseguran la no ocurrencia de deadlock.

  - **Prevención de deadlock**: Evitar que se cumplan las condiciones necesarias para que ocurra un deadlock.
  - **Evitación de deadlock**: Conociendo las necesidades de recurso de los procesos, se analiza la situación para saber si conduce a un deadlock.

- Protocolos que aceptan la ocurrencia de deadlock.

  - **Detección y recuperación de deadlock**: Detectar la ocurrencia de un deadlock y recuperarse de él.

Prevención de deadlock

Este método se basa en la prevención de la ocurrencia de deadlock, evitando que se cumplan las condiciones necesarias para que ocurra. Estas negaciones serian:

- **Negación de exclusión mutua**: Normalmente es una propiedad del recurso por lo que no es posible obligar a compartir un recurso que no puede ser utilizado de manera simultanea

  - **Evaluación**: La exclusión mutua es una propiedad del recurso por lo tanto **no se puede negar**.

- **Negación de retención y espera**: Un proceso que solicita un recurso no pueda mantener los que ya posee. Lo que se busca es que un proceso solicite todos los recursos que necesita al inicio.

  - **Evaluación**: Se pueden asignar recursos pero que no se utilize por periodos de tiempo prolongados. Puede producir postergación indefinida de asignación de recursos.

- **Negación de NO desalojo**: Cuando un recurso que retiene recursos solicita otros, se le pueden desalojar los recursos que ya posee, sin embargo esto provoca que el proceso recomience cuando estén disponibles todos los recursos que necesita.

  - **Evaluación**: Esto produce una perdida de trabajo, ademas de causar posiblemente una postergación indefinida y existen recursos que no se pueden expropiar.

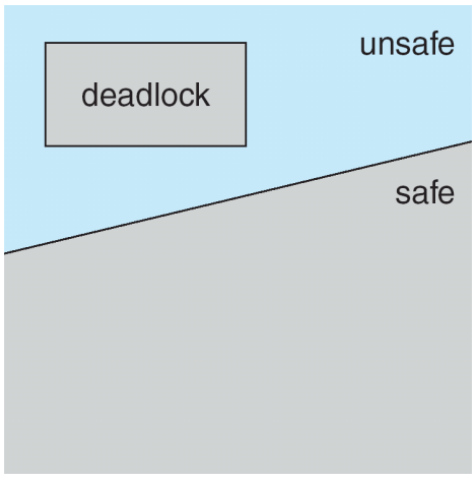
- **Negación de espera circular**: Asignación de números a recursos para saber si uno precede a otro, por lo que si un proceso con un recurso de menor número solicita uno de mayor número, se le niega el acceso ya que primero debe liberar el recurso de menor número.

  - **Evaluación**: Es difícil asignar un orden a los recurso en un ambiente dinámico, es bastante restrictivo y poco flexible.

Evitación de deadlock

Para evitar la ocurrencia de deadlock se necesita conocer las necesidades de recursos de los procesos, con estos datos se puede realizar asignaciones cuidadosamente. Esto lograría una mejor utilización de los recursos que la estrategia de prevención, sin embargo, estos métodos mas simples requieren que los procesos conozcan de antemano los recursos máximos que necesitaran.

- **Estado seguro**: Es un estado en el que es posible asignar recursos a los procesos de manera que no se produzca un deadlock.
  - Un sistema esta en un estado seguro si existe una secuencia de asignaciones de recursos que permiten a todos los procesos terminar.
  - Si no existe una secuencia de asignaciones de recursos que permitan a todos los procesos terminar, el sistema esta en un estado inseguro.
  - Un estado seguro evita la ocurrencia de deadlock.
  - Un estado inseguro no garantiza un deadlock pero puede conducir a uno.



Evitación de deadlock - Algoritmo del banquero (Dijkstra 1965)

Se basa en que un proceso solicita recursos y el sistema verifica si la asignación de recursos puede llevar a un estado seguro. Si es así, se asignan los recursos, si no, el proceso debe esperar. Para ello se requiere que cuando una nueva hebra entre al sistema declare el máximo número de recursos que necesitará.

Detección y recuperación de deadlock

Mediante un algoritmo se detecta si existe un deadlock, este algoritmo permite la existencia de las 3 primeras condiciones para la ocurrencia de un deadlock y detecta la cuarta.

- **Idea**
  - El SO chequea periódicamente si hay un ciclo en el grafo de asignación de recursos y si lo hay, si detecta un deadlock.
  - En caso de detectar un deadlock, el SO puede inicia un procedimiento de recuperación.

Cuando el algoritmo detecta la existencia de una deadlock puede tomar una de las siguientes acciones:

- **Recuperación manual**: Avisar al usuario de la existencia de un deadlock y que este tome las medidas necesarias.
- **Recuperación automática**: El SO toma las medidas necesarias para recuperarse del deadlock mediante:
  - Abortar uno o más procesos.
  - Expropiación de uno o más recursos.

Abortar uno o más procesos

- **Técnicas**
  - Abortar todos los procesos bloqueados.
  - Abortar los procesos bloqueados de a uno e ir viendo si se termino el deadlock.
- **Criterios**
  - Prioridad del proceso.
  - Tiempo de ejecución que lleva el proceso.
  - Cantidad y tipo de recursos que posee el proceso.
  - etc.

Expropiación de uno o más recursos

Consiste en quitar sucesivamente los recursos a los procesos y asignárselos a otros hasta romper el ciclo de bloqueo. Se debe considerar:

- Selección de victima: Intentar elegir el proceso que minimice el costo de la expropiación.
- Rollback (vuelta atrás): Si se expropia un recurso a un proceso, este debe volver a un punto de seguridad (estado anterior).
- Inanición: Asegurarse que el proceso al que se le expropian recursos no sea el mismo siempre.

8. Memoria principal (MP)

Fundamentos

Para entender la gestión de la memoria principal es necesario conocer los siguientes conceptos:

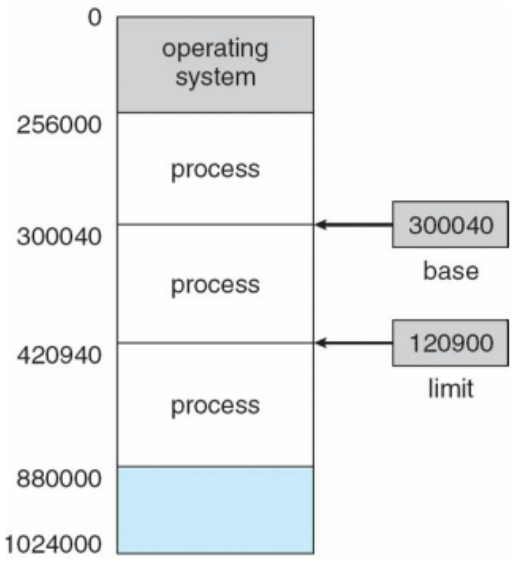
- Los programas deben estar cargados en MP para ser ejecutados.
- La MP y los registros integrados del procesador son las únicas áreas de almacenamiento al que la CPU puede acceder directamente.
- La MP es un arreglo de bytes identificados por direcciones únicas.
- La interacción con la MP se logra mediante secuencias de lecturas o escrituras de direcciones especificas de la memoria.
- El caché es la memoria más rápida entre la MP y la CPU y se utiliza para resolver problemas de velocidades.
- Se requiere garantizar la correcta operación que proteja al SO y los procesos de usuarios de accesos indebidos (registros base y limite).

Registros base y límite

El par de registros base y limite **definen el espacio de direcciones para cada proceso**. De manera mas especifica el registro base **almacena la dirección de memoria física legal mas pequeña**, mientras que el registro limite **especifica el tamaño del rango**.

La protección de espacio realizada por estos registros se logra haciendo que el HW de la CPU compare las direcciones generadas en modo usuario con el contenido de estos registros. SI el proceso busca ver una dirección fuera de sus limites se gatilla la protección.

Cualquier acceso ilegal (fuera del rango) provoca una interrupción hacia el SO.



Reasignación de direcciones

Es necesario que los programas de usuario recorran varias etapas antes de ser ejecutados, donde las direcciones pueden representarse de diferentes formas.

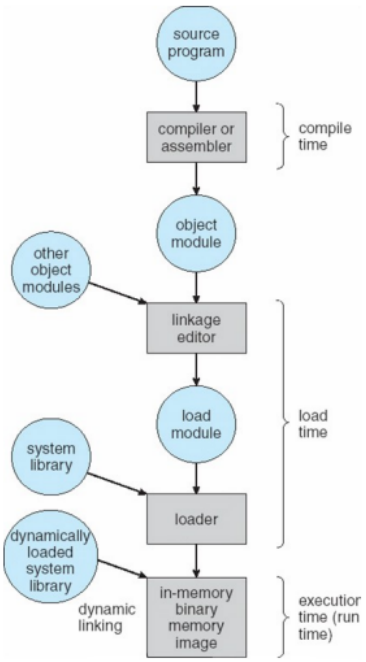
1. Programa fuente, las direcciones son simbólicas (la variable "suma", çontador", etc.).
2. Luego, el compilador reasignará tal dirección simbólica a reubicables (14 bytes a partir de este modulo). (simbólica → reubicable)
3. El cargador reasignará la dirección reubicable a direcciones absolutas (74014). (reubicable → absoluta)

Cada operación de Reasignación constituye una relación de un espacio de direcciones a otro.

La Reasignación de direcciones **se puede realizar en cualquiera de estas etapas**:

- **Tiempo de compilación**: Si se conoce donde va a residir el proceso al momento de compilar, el compilador puede generar un **código absoluto**. Si se cambia la ubicación, se hace necesario recompilar el programa (programas .COM de MS-DOS).
- **Tiempo de carga**: Si no se conoce la ubicación del proceso al momento de compilar, se genera un **código reubicable**. La Reasignación final se ralentiza hasta el momento de la carga. Luego si cambia la ubicación, solo es necesario cargar el código del usuario.
- **Tiempo de ejecución**: Si el proceso puede moverse durante su ejecución desde un segmento de memoria a otro, se retarda la Reasignación final hasta el instante de la ejecución. Requiere de un HW especial (MMU - Memory Management Unit). La memoria de los SSOO de propósito general utilizan este método.

Etapas para la ejecución de un programa



Define:

- Tiempo de carga: Donde se aloja el proceso en la MP. (Obsoleto)
- Tiempo de ejecución: Una vez en memoria no se mueve. (No muy común)
- Tiempo de compilación: Se mueve entre segmentos de memoria. (Más común)

Espacio de direcciones lógicas y físicas

- **Dirección lógica**: Dirección generada por la CPU, vista por el proceso.
- **Dirección física**: Dirección recibida por la unidad de memoria y que puede ser eventualmente distinta a la dirección lógica. (puede que exista la dirección lógica o no).
- **Espacio de direcciones lógicas**: Conjunto de direcciones generadas por un programa.
- **Espacio de direcciones físicas**: Conjunto de direcciones físicas correspondientes a un conjunto de direcciones lógicas.
- Cuando la Reasignación se hace en:
  - Tiempo de ejecución - carga : direcciones lógicas = direcciones físicas.

MMU

Registro base → registro de dirección.
Apunta a las verdaderas direcciones físicas. La MMU se encarga de la traducción de direcciones lógicas a físicas.

Carga dinámica

Tamaño del proceso limitado a la cantidad de memoria física disponible.
Carga dinámica: No es necesario tener en memoria todo el código, sino que solo una parte del el. A medida que se van necesitando las rutinas se van cargando.

- Rutinas de control de errores no siempre se usan asi que solo se cargan en memoria cuando son necesarias.
- Depende de como se diseñen los programas, no tanto del HW.

Montaje dinámico y bibliotecas compartidas

Algunos SO solo permiten **montaje estático**, es decir las al momento de la compilación se incluyen todas las librerías necesarias. Dando pie a programas muy grandes y poco eficientes.

- **Montaje dinámico**: incluye un stub en la imagen binaria para las rutinas que se necesitan de la librería.

- **Stub**: pequeño fragmento de código que indica donde se encuentra la rutina necesaria. Es reemplazado por la dirección de la rutina y se ejecuta (acceso directo).

Este mecanismo también es conocido como **Bibliotecas compartidas**.

Swapping-Intercambio

Dependiendo de la orientación de procesos (CPU, E/S) se hacen mixes para que la memoria sea utilizada de manera eficiente.

Asignación memoria contigua

Este proceso se realiza a traves de la division de la memoria en particiones, una para almacenar el SO y otra para los procesos de usuario.

Cada proceso esta en una única sección de memoria contigua, cada proceso en un rango limitado.

Mapeo y protección

Valida los limites del registros y suma los registros de reubicación a la dirección física.

MFT

Division de memoria en particiones fijas, cada partición puede contener un solo proceso. Ya no se usa ya que para procesos muy pequeños se desperdicia mucha memoria.

MVT

Tamaño de particiones variables, se asigna la partición mas pequeña que se ajuste al proceso. Existen bloques de memoria para asignar los procesos.

Asignación dinámica

Puede provocar fragmentaciones:

- **Fragmentación externa**: Espacio desperdiciado fuera de los bloques asignados. Existe espacio total para una solicitud pero no es contiguo.
    - Solución: Mover los espacios libres existentes para crear un bloque contiguo. A esto se le llama **compactación**.
- **Fragmentación interna**: Espacio desperdiciado dentro de un bloque asignado. Existe espacio que no utiliza el proceso, le sobra.

Paginación

- Asignación no contigua.
- Proceso esta alojado en varias partes de la memoria, a estas partes se les llama **páginas**.

- La memoria se divide en bloques de tamaño fijo llamados **marcos de página**.
- Puede provocar Fragmentación interna.
- La fragmentación externa no existe.
- Memoria caché también esta presente aquí para mejorar las velocidades.

Segmentación

- Considera la lógica del procesos por lo tanto divide en segmentos lógicos (código, datos, pila).
- Puede provocar fragmentaciones.

Segmentación paginada

- Aprovecha lo mejor de la segmentación y la Paginación.
- Programa se divide en segmentos y cada segmento en paginas.