Primera clase
Introducción a los Sistemas Operativos

Objetivo conceptual: que el alumno aprenda conceptos fundamentales involucrados en el estudio de los sistemas operativos.
Objetivo procedimental: que el alumno adquiera las habilidades necesarias para manejarse en un sistema operativo multiusuario; cargar, compilar y ejecutar sencillos programas provistos para experimentear las funciones básicas de los sistemas operativos.
Desarrollo: 1 (una) semana de 14 previstas.

La presente es una breve **Guía de estudios** para orientarlo en la lectura de los temas. Notará que está basada en varios libros sobre Sistemas Operativos y aunque los temas le parezcan recurrentes, es bueno que lea distintos enfoques del mismo tema. Es **muy importante** que no intente memorizar los temas, sino que lea con espíritu crítico, **analizando** lo que lee y discutiendo los temas con sus compañeros y en clase. Los sistemas operativos están en constante evolución, como también lo está el hardware; de manera que tal vez algunos conceptos desarrollados ahora caerán en desuso dentro de un tiempo. Si usted memoriza los conceptos actuales como palabra sagrada, dentro de unos años, cuando esté recibido, lo que aprendió le parecerá **obsoleto**, pero si aprendió razonando sabrá que tiene las bases y verá en esos sistemas operativos una **evolución** de lo que aprendió.

Aproximación a la definición de sistemas operativos

La evolución del hardware ha sido constante: del monoprocesador a los multiprocesadores, memorias ultrarápidas, incorporación de nuevos dispositivos, interconexión con otros sistemas ... pero ¿qué ha pasado con los sistemas operativos? Los sistemas operativos debieron acompañar la evolución.

Conceptos comunes de lo que es un sistema operativo es que es un manejador de recursos, un programa de control, el programa que se ejecuta constantemente en una computadora, etc.

Tanenbaum lo expresa desde dos puntos de vista (complementarios): como **máquina extendida o virtual** más fácil de entender y programar que el hardware puro[1] y como **administrador de recursos**, entendiendo por "recursos" a los componentes tanto físicos como lógicos: el procesador, memoria, discos, ratones o archivos. (Ver Tanenbaum & Woodhull "Sistemas Operativos, diseño e implementación".

Un sistema operativo es un **programa** que actúa como intermediario entre el usuario (en su sentido amplio) de un computador y el hardware del computador. El propósito de un sistema operativo es crear un entorno en el que el usuario pueda ejecutar programas de forma *cómoda* y *eficiente* (ver Silberschatz "Sistemas Operativos" quinta edición p. 3).

Los sistemas operativos deben acompañar la evolución de los sistemas de cómputo, brindando: portabilidad, interoperabilidad, interconectividad, ambiente multitareas,

[1] Un enfoque desde el punto de vista del **programador** de sistemas.

multiusuario, seguridad, protección (entre usuarios y desde el exterior), fácil administración, independencia de dispositivo, abstracción del hardware.

Tipos de sistemas

Hay distintos tipos de sistemas: batch, interactivo, monousuario, de tiempo compartido (*time sharing*) multiusuario, paralelo, distribuido, de red, de tiempo real, cliente servidor.

Los sistemas *batch*: es una terminología que proviene de los viejos sistemas por lotes, con tarjetas perforadas, en las que no hay interacción con el usuario, se usa para largos procesos, con entrada desde archivos y salida a archivos o impresión.

Los sistemas de tiempo compartido, son sistemas interactivos, multiusuarios (la CPU se reparte entre los distintos usuarios, cada uno de ellos en su PC/terminal). Para estos sistemas, se debe proveer **multiprogramación**.

Multiprogramación: es la posibilidad de tener varios programas en memoria. El **grado de multiprogramación** es la cantidad de programas que se tienen en memoria. En un sistema uniprocesador, habrá varios programas en memoria pero sólo uno en ejecución, en un momento específico. Los sistemas de tiempo compartido son una consecuencia lógica de la multiprogramación.

Los sistemas de tiempo real tienen restricciones de tiempo bien definidas, se usan para una aplicación dedicada, tienen una memoria primaria amplia y el almacenamiento secundario es limitado.

Los sistemas paralelos son sistemas multiprocesador, los procesadores comparten el bus y el reloj (*clock*). Si comparten memoria y periféricos son **fuertemente acoplados**. El multiprocesamiento puede ser simétrico o asimétrico. Las ventajas de los sistemas paralelos es que mejoran el *throughput* (resolución de mayor cantidad de procesos en un momento dado), permiten compartir periféricos (puede haber varios procesadores accediendo al mismo disco), suelen tener la capacidad para seguir dando un servicio proporcional al nivel de hardware que sobrevive, esto se denomina **degradación gradual** o *graceful degradation*. Los sistemas diseñados para degradarse gradualmente también se conocen como tolerantes a fallas o *fault tolerant* (ante la caida de un procesador, continua trabajando el resto, absorviendo su trabajo).

Hay distintos tipos de sistemas paralelos, por ejemplo los simétricos (SMP) en el que cada procesador tiene una copia idéntica del SO y esas copias se comunican entre sí, si es necesario. Asimétricos: uno de los procesadores distribuye y dirige la actividad de los otros. Equipos TANDEM, que duplican el hardware y el software para asegurar continuidad ante fallas.

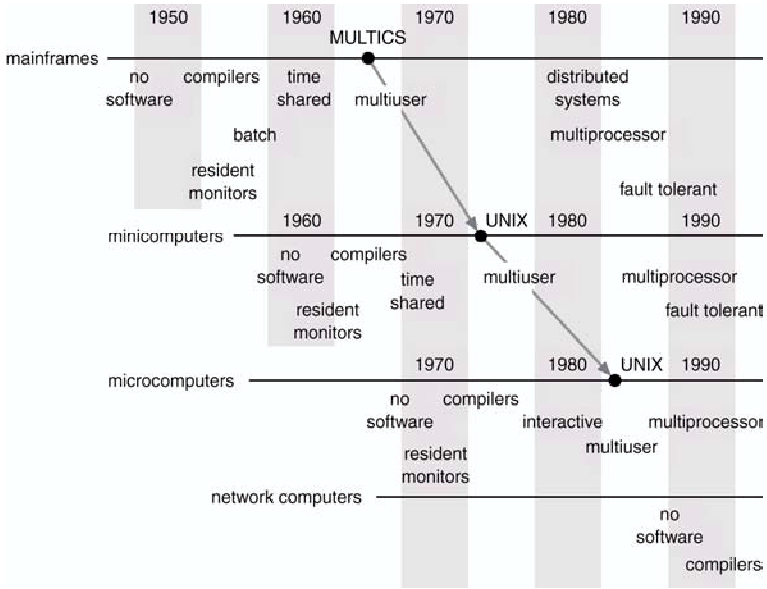
Los sistemas distribuidos: se distribuye el trabajo entre varios procesadores, pero estos no comparten memoria ni reloj. Ventajas: compartir recursos, mejorar el procesamiento por división de un proceso en subrprocesos, confiabilidad, comunicación.

Sistemas operativos de red: Los usuarios saben que hay varias máquinas, y necesitan acceder a los recursos iniciando una sesión en la máquina remota apropiada o bien transfiriendo datos de la máquina remota a su propia máquina. El sistema operativo de red es una capa que se agrega a cada sistema operativo para interactuar con la máquina de servicios. Por ejemplo Windows 95 ó 98: usted accede a "Entorno de red" para ver a sus vecinos, y va haciendo click para acceder a la computadora y luego al recurso compartido (por ejemplo el disco). Es decir, el acceso a los recursos es **explícito**. En los sistemas operativos distribuidos (por ejemplo Amoeba) los usuarios no tienen que saber que hay varias máquinas; acceden a los recursos remotos de la misma manera que a los locales. Hay **transparencia** de acceso a los recursos, no saben si el recurso al cual están accediendo es local o remoto.

Sistemas Cliente-Servidor. **Servicio**: entidad de software en una o mas máquinas que prevee una función particular. **Servidor**: software de servicio que corre en una máquina. **Cliente**: proceso que puede solicitar un servicio a través de un conjunto de operaciones bien definidas que forman la *interfaz cliente*. Ejemplos: servidor de impresión. El servicio es la impresión, el servidor es el programa que brinda el servicio y los clientes son los programas que solicitan el servicio. Servidor de archivos: el servicio es la administración de archivos, el servidor es un sistema de archivos (*filesystem*) y los clientes solicitan el servicio.

Un error muy común es llamar servidor a la máquina donde reside el software servidor. El concepto de servidor es un concepto de sotware.

Migración de conceptos y funciones de sistemas operativos



En Tanenbaum está contada la historia de los sistemas operativos y su evolución a través de las llamadas "generaciones". Esos conceptos están muy asociados a los tipos de sistemas que acabamos de ver. También puede ver los distintos tipos de sistemas en Silberschatz.

Componentes del sistema

Un sistema operativo crea el entorno en el que se ejecutan los programas. Podemos crear un sistema tan grande y complejo como un sistema operativo sólo si lo dividimos en porciones más pequeñas. Cada una de estas partes deberá ser un componente bien delineado del sistema, con entradas, salidas y funciones cuidadosamente definidas.

Gestión de procesos

Un programa no puede hacer nada si la CPU no ejecuta sus instrucciones. Podemos pensar en un *proceso* como una porción de un programa en ejecución o todo el programa, pero su definición se ampliará a medida que avancemos en el estudio.

Un proceso necesita ciertos recursos, incluidos tiempo de CPU, memoria, archivos y dispositivos de E/S, para llevar a cabo su tarea. Estos recursos se otorgan al proceso en el momento en que se crea, o bien se le asignan durante su ejecución.

Un proceso es la unidad de trabajo de un sistema. El "sistema" consiste en una colección de procesos, algunos de los cuales son procesos del sistema operativo (los que ejecutan código del sistema), siendo el resto procesos de usuario (los que ejecutan código de algún usuario).

Gestión de la memoria principal

La memoria principal es crucial para el funcionamiento de un sistema de computación moderno. La memoria principal es una matriz grande de palabras o bytes, cuyo tamaño va desde cientos de miles hasta cientos de millones. Cada palabra o byte tiene su propia dirección. La memoria principal es un depósito de datos a los que se puede acceder rápidamente y que son compartidos por la CPU y los dispositivos de E/S. El procesador central lee instrucciones de la memoria principal durante el ciclo de obtención de instrucciones, y lee y escribe datos de la memoria principal durante el ciclo de obtención de datos.

Gestión de archivos

Un archivo es una colección de información relacionada definida por su creador. Por lo regular, los archivos representan programas (en forma tanto fuente como objeto) y datos. Los archivos de datos pueden ser numéricos, alfabéticos o alfanuméricos. Los archivos pueden ser de forma libre, como los de texto, o tener un formato rígido. Un archivo consiste en una secuencia de bits, líneas o registros, cuyos significados han sido definidos por su creador. El concepto de archivo es muy general.

Gestión del sistema de E/S

Uno de los objetivos de un sistema operativo es ocultar las peculiaridades de dispositivos de hardware específicos de modo que el usuario no las perciba. Por ejemplo, en Unix, el *subsistema de E/S* oculta las peculiaridades de los dispositivos de E/S del resto del sistema operativo mismo.

Gestión de almacenamiento secundario

El propósito principal de un sistema de computador es ejecutar programas. Estos programas, junto con los datos a los que acceden, deben estar alojados en la memoria principal (*almacenamiento primario*) durante la ejecución. Dado que la memoria principal es demasiado pequeña para dar cabida a todos los datos y programas, y que pierde su información cuando deja de recibir corriente eléctrica, el sistema de computación debe contar con algún *almacenamiento secundario* para respaldar la memoria principal. La mayor parte de los sistemas de computador modernos utiliza discos como principal medio de almacenamiento. El sistema operativo se encarga de las siguientes actividades relacionadas con la gestión de discos:

- ✓Administración del espacio libre
- ✓Asignación del almacenamiento
- ✓Planificación del disco

Trabajo con redes

Un sistema distribuido es una colección de procesadores que no comparten memoria, dispositivos periféricos ni el reloj. Más bien, cada procesador tiene su propia memoria local y su propio reloj, y se comunica con los otros procesadores a través de distintas líneas de comunicación, como buses de alta velocidad o líneas telefónicas.

Sistema de protección

Es preciso proteger cada proceso de las actividades de los demás. El hardware de direccionamiento de memoria asegura que un proceso sólo pueda ejecutarse dentro de su propio espacio de direcciones. El temporizador cuida que ningún proceso ueda controlar y monopolizar la CPU indefinidamente. Los registros que controlan los dispositivos no están accesibles a los usuarios, a fin de proteger la integridad de los diferentes periféricos.

Sistema de interpretación de órdenes

Uno de los programas del sistema más importantes de un sistema operativo es el intérprete de órdenes o de comandos, que es la interfaz entre el usuario y el sistema operativo. Algunos sistemas operativos incluyen el intérprete de órdenes en el núcleo; otros, como MS-DOS y Unix, tratan el intérprete de órdenes como un programa especial que se está ejecutando cuando se inicia un trabajo, o cuando un usuario ingresa en un sistema de tiempo compartido. La función del intérprete de línea de comandos o *shell* es muy sencilla: obtener la siguiente orden y ejecutarla.

(**Amplíe** este tema por Silberschatz).

Servicios del sistema operativo

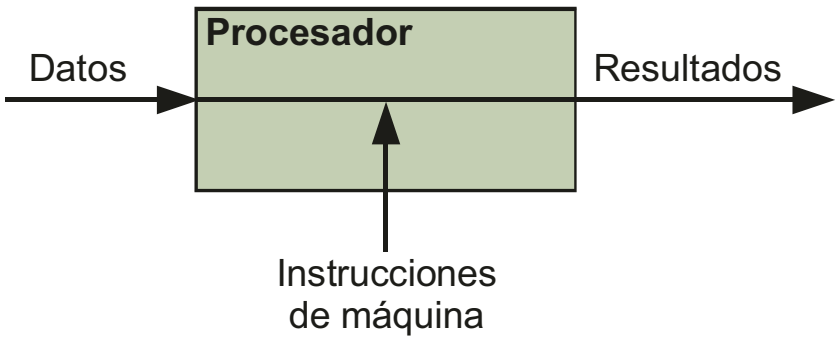
Un sistema operativo crea un entorno para la ejecución de programas. El sistema operativo proporciona ciertos servicios a los programas y a los usuarios de dichos programas. Desde luego, los servicios específicos varían de un sistema operativo a otro, pero podemos identificar algunas clases comunes. Podemos describir a un sistema operativo a partir de los servicios que presta.

- ✓Controlar la ejecución de procesos (creación, terminación, suspensión y comunicación).
- ✓Planificar el uso de la CPU
- ✓Asignar memoria principal en ejecución.
- ✓Asignar memoria secundaria.
- ✓Permitir acceso controlado desde los procesos a los dispositivos.
- ✓**Todo esto de manera eficiente y "transparente" al usuario.**

Estructura y funcionamiento de la computadora

La computadora es una máquina destinada a procesar datos. En una visión esquemática como la que muestra la figura este procesamiento involucra dos flujos de información: el de datos y el de instrucciones. Se parte del flujo de datos que ha de ser procesado. Este flujo de datos es tratado mediante un flujo de instrucciones de máquina, generado por la ejecución de un programa, y produce el flujo de datos resultado.

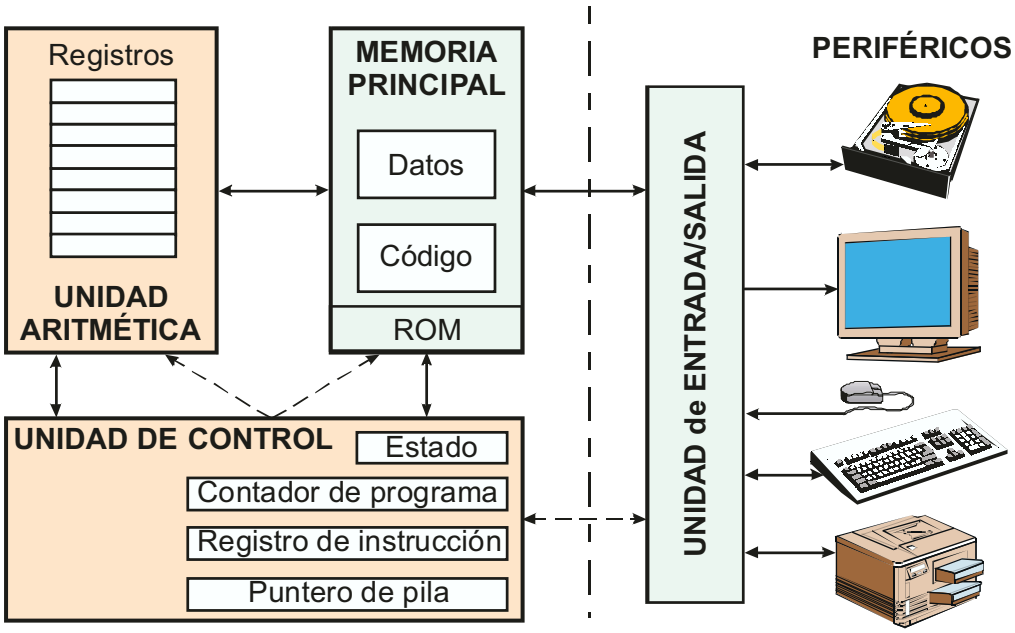
Figura 1.1 de Carretero Pérez – Estructura de funcionamiento de la computadora



Componentes de una computadora

Para llevar a cabo la función de procesamiento, una computadora con arquitectura de von Neuman está compuesta por los cuatro componentes básicos representados en la figura:

Figura 1.2 de Carretero Pérez. Componentes básicos de la computadora



Para lograr sincronización entre CPU y dispositivos de E/S hay dos métodos:

✓Transferencia de datos por interrupciones
✓Transferencia de datos por acceso directo a memoria (DMA)

Antes se realizaba por espera activa (la CPU esperaba que se realizara la E/S).

En un sistema basado en interrupciones hay un controlador por cada tipo de dispositivo, que cuenta con almacenamiento local en buffer y registros de propósito general.

El controlador del dispositivo es el responsable de transferir los datos entre el dispositivo que controla y su buffer local

Dinámica de una operación de E/S:

✓La CPU carga los registros apropiados del controlador del dispositivo con la acción a tomar y continúa su operación normal.
✓El controlador analiza los registros para ver qué acción realizar
✓Concluida la transferencia el controlador avisa a la CPU que la transferencia ya terminó. Esto se realiza causando una **interrupción**.
✓La CPU suspende lo que estaba haciendo y transfiere la ejecución a un lugar fijo que tiene la dirección de inicio de la rutina de atención de esa interrupción.
✓Dicha rutina transfiere los datos desde el buffer local del controlador a la memoria principal.
✓Una vez realizada la transferencia, la CPU puede continuar con los procesos interrupidos.

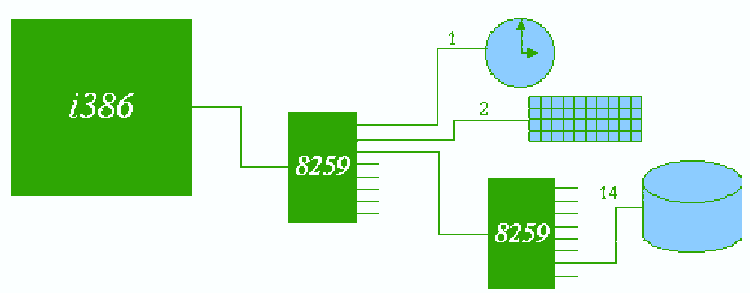
Cada diseño de computadora tiene su esquema de interrupciones.

Distinguir las interrupciones de las excepciones. Las excepciones o *traps* son las llamadas interrupciones por software. Las interrupciones son externas al proceso; las excepciones son internas a él.

El vector de interrupciones

Es un conjunto de posiciones ubicada normalmente en la parte baja de la memoria donde se almacenan las direcciones de las rutinas de atención de las interrupciones. Se indexa a través de un número único que se asocia a cada dispositivo. La estructura de interrupciones debe guardar la dirección de la instrucción interrumpida. Normalmente se guarda en la pila del sistema. Generalmente cuando se está atendiendo una interrupción se desactivan las otras. Actualmente existen arquitecturas que permiten procesar una interrupción mientras se atiende otra, asignando un esquema de prioridades. Se aceptará una interrupción de mayor prioridad a la que se está atendiendo; las de prioridad menor o igual se enmascaran o desactivan.

Hardware de las interrupciones



La circuitería externa relacionada con las interrupciones de las actuales placas base emula el funcionamiento de los originales PC-AT de IBM.

- ✓Llega petición a un 8259A. Cada vez que se pulsa una tecla, por ejemplo, el controlador de teclado activa la línea de interrupción conectada al primer chip 8259A.
- ✓El 8259A maestro envía al procesador la petición. Cuando el procesador puede atenderla (inmediatamente si las interrupciones están habilitadas) el chip 8259A envía por el bus de datos el número de interrupción. En nuestro caso, el número enviado es el 0x21. 0x20 es valor programado como base en el 8259A durante el arranque; más 1 que es la línea por la que llegó la interrupción.
- ✓El procesador utiliza este valor como índice en la tabla IDT (*Interrupt Descriptor Table*) para localizar la rutina que sirve la interrupción.

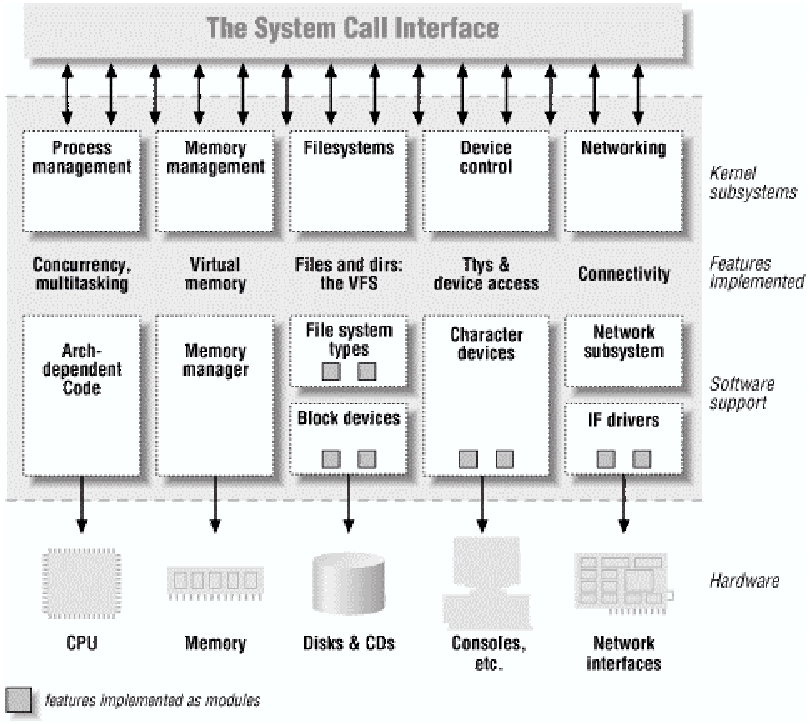
Observe que tanto la tabla de interrupciones como la rutina de tratamiento de la interrupción se han considerado parte del sistema operativo. Esto suele ser así por razones de seguridad; para evitar que los programas que ejecuta un usuario puedan perjudicar a los datos o programas de otros usuarios.

Repase lo aprendido en "Arquitectura de computadoras" para todos los temas cercanos al hardware. También puede repasarlos a partir del primer capítulo de Carretero Pérez "Conceptos arquitectónicos de la computadora".

Llamadas al sistema

Son la interfaz entre el sistema operativo y un programa en ejecución. Pueden ser instrucciones en lenguaje ensamblador (MSDOS) o pueden hacerse desde lenguajes de alto nivel como C (Unix, Minix, Linux, Windows NT). El control pasa al vector de interrupciones para que una rutina la atienda. El bit de modo se pone en modo monitor. El monitor analiza la instrucción que provocó la interrupción. Así se ejecuta la solicitud y vuelve el control a la instrucción siguiente a la llamada al sistema. Los parámetros asociados a las llamadas pueden pasarse de varias maneras: por registros, bloques o tablas en memoria o pilas. Hay varias categorías de llamadas al sistema:

- ✓Control de procesos: finalizar, abortar, cargar, ejecutar, crear, terminar, establecer y obtener atributos del proceso, esperar un tiempo, señalar y esperar evento, asignar y liberar memoria.
- ✓Manipulación de archivos: crear, aliminar, abri, cerrar, leer, escribir, reposicionar, obtener y establecer atributos de archivo.
- ✓Manipulación de dispositivos: solicitar, liberar, leer, escribir, reposicionar, obtener y establecer atributos de dispositivo.
- ✓Mantenimiento de información: obtener fecha y hora, datos del sistema, atributos.
- ✓Comunicaciones: crear, eliminar conexión de comunicación, enviar y recibir mensajes, transferir información de estado, etc.



Interfaz de llamadas al sistema en Linux

Trabajo Práctico Nº 1

Módulo de la Guía de Actividades en Clase: Ingreso al sistema. Comandos básicos de Linux. Tutorial del editor de textos. Invocación del compilador gcc.

Para las prácticas en Linux hemos elegido a la distribución ADIOS[2] (*Automated Download and Installation of Operating System*) desarrollada por "*Queensland University of Technology*". Se trata de un sistema GNU/Linux[3] instalado en el CD-ROM. Es decir que no se instala en el disco rígido de la computadora. Esto facilita enormemente las prácticas ya que instalarlo en el disco rígido requiere mayor conocimiento acerca del uso del sistema Linux y esto se encuentra fuera del alcance de la materia.

No obstante si usted ya tiene una computadora con sistema Linux instalado, úselo sin dudarlo, ya que la desventaja[4] que presenta esta distribución es que utiliza la memoria RAM como si fuera un disco rígido por lo tanto los cambios a los archivos de configuración y la creación de nuevos archivos es volátil, es decir que no permanecerán después de un reinicio de la computadora.

Por otra parte si no tiene experiencia en el uso y administración de un sistema Linux, con esta distribución corre pocos riesgos de arruinar su disco rígido o el sistema operativo que tenga en él (supuestamente Windows).

Este es un CD-ROM con arranque (*boot*) es decir que para cargarlo debe asegurarse que su computadora es capaz de arrancar un sistema operativo desde la unidad de CD-ROM. Verifique esto en el *setup* de la BIOS, establezca como primer dispositivo de arranque a la unidad de CD-ROM.

Al arrancar ADIOS detecta su disco rígido; si éste tiene una partición de intercambio (*swap*) de Linux intentará usarla, preguntándole previamente, por ejemplo:

"Use partition /dev/discs/disc0/part6 for Linux SWAP <y/n>?: "

A continuación aparece un menú de opciones de las cuales sólo tomaremos la opción "1) Ejecutar Linux desde CD y RAM solamente" y si no quisieramos ambiente gráfico (lo cual es recomendable porque ahorra mucha memoria RAM) deberíamos utilizar previamente la opción "r) Cambiar el nivel de ejecución de Linux (nivel de ejecución actual=5)" cambiandolo por nivel de ejecución=3.

Las opciones 2 al 5 acceden al disco rígido, sobretodo la 5 que instala ADIOS en el disco rígido. **NO** use estas opciones en las máquinas del laboratorio.

Tenga en cuenta que al ejecutarse desde el CD-ROM y que estos son mucho mas lentos que un disco rígido muchas veces tendrá la impresión que el sistema está "colgado" y no es así,

[2] http://dc.qut.edu.au/adios

[3] Visite http://www.gnu.org/ y http://www.linux.org/

[4] Su gran ventaja es en cierta manera su desventaja.

por lo tanto, tenga paciencia y no comience a sacudir el mouse frenéticamente ni apretar teclas tratando de obtener alguna respuesta.

Una vez finalizada la inicialización del sistema aparecerá el *prompt* para iniciar sesión (*login:*). El usuario predefinido es "adios" y la clave es "12qwaszx". Es fácil recordarla porque son las teclas del extremo izquierdo del teclado de arriba abajo.

Una vez iniciada la sesión, el sistema lo posiciona en el directorio "/home/adios", en el cual tiene permiso para creación de archivos y directorios.

El teclado predeterminado es el estadounidense, por lo tanto para cambiarlo ingrese el comando:

loadkeys es

Utilizaremos el editor de archivos de texto "vi" (en realidad es el "vi mejorado" *Vi iMproved* o vim) para crear y modificar los programas. Para invocarlo es conveniente indicar el nombre del archivo a crear (o modificar), por ejemplo "vi prog1.c". Si "prog1.c" no existe intentará crear un archivo nuevo, vea en la parte baja de la pantalla el cartel "[New File]". Para salir sin guardar los cambios coloque ":q!", es decir "dos puntos", luego "q" minúscula y signo de admiración. Con esto volvemos al *prompt*.

A continuación veremos una breve introducción para aprender a usar este programa, colocando el comando "vimtutor es", ingresamos a un tutorial autoasistido de unos 30 minutos de duración. Tome apuntes de todos los comandos aprendidos.

Por tratarse Linux de un sistema operativo multiusuario es posible que más de un usuario esté usando el sistema. Se provee de varias "consolas" desde las cuales puede iniciar sesión. Presionando Ctrl+F1 ... F6 puede permutar entre las seis consolas e ingresar con distintos usuarios (o el mismo si lo desea pero las sesiones serán tratadas como distintas).

El usuario "adios" con el que hemos estado trabajando es uno sin privilegios administrativos sobre la computadora. El usuario con privilegios administrativos o "superusuario" *root* es la cuenta de administración del sistema, la clave es la misma.

Atención: Siga estrictamente las indicaciones del profesor cuando trabaje con la cuenta del superusuario o puede echar a perder el sistema y deberá hacerse cargo. Si accidentalmente hace algo no previsto llame inmediatamente al profesor y pida ayuda.

A continuación cargaremos el siguiente programa, que es una versión modificada del famoso "Hola mundo!", el primer programa escrito en lenguaje C por Brian Kernighan y Dennis Ritchie, adaptado por Ori Pomerantz.

El núcleo de los sistemas operativos Unix es tradicionalmente monolítico. Linux también lo es, pero incorpora el concepto de "módulos", que son piezas de software con la capacidad de ser cargadas y descargadas dinámicamente del núcleo (*kernel*) en forma dinámica, es decir "en caliente", sin necesidad de reiniciar el equipo.

La función init_module nos va a permitir inicializar el módulo al insertarlo en el núcleo (equivaldría a la función main de un programa en **C**). Complementariamente, cleanup_module se usará para liberar los recursos utilizados cuando se vaya a extraer.

```
/* hello.c
 * Copyright (C) 1998 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version.
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Initialize the module */
int init_module()
{
  printk("Hello, world - this is the kernel speaking\n");

  /* If we return a non zero value, it means that
   * init_module failed and the kernel module
   * can't be loaded */
  return 0;
}

/* Cleanup - undid whatever init_module did */
void cleanup_module()
{
  printk("Short is the life of a kernel module\n");
}
/*Hasta aca el programa original, la siguiente linea
 es una modificacion del profesor para que no
 arroje el error de "tainted", es decir, declaramos
 este modulo con licencia GPL */
MODULE_LICENSE("GPL");
```

Para compilar:

```
$ gcc -Wall -DMODULE -D__KERNEL__ -DLINUX -c hello.c
```

Se inserta como superusuario:

```
# insmod hello.o
```

Y se extrae:
rmmod hello

Usualmente los sistemas Unix proveen un comando utilitario denominado "make" que simplifica enormemente la tarea de compilación y enlazado (*link*), sobretodo cuando son varios los programas "*.c" que hay que compilar y enlazar con bibliotecas del sistema. Este programa lee un archivo de datos denominado "Makefile" en el que encuentra todas las directivas de compilado.

Por ejemplo el archivo "Makefile" para este módulo sería el siguiente:

```
# Makefile for a basic kernel module

CC=gcc
MODCFLAGS := -O2 -Wall -DMODULE -D__KERNEL__ -DLINUX -Dlinux

hello.o: hello.c /usr/include/linux/version.h
		$(CC) $(MODCFLAGS) -c hello.c
		echo insmod hello.o to turn it on
		echo rmmod hello to turn if off
		echo
		echo X and kernel programming do not mix.
		echo Do the insmod and rmmod from outside X.
```

Si al intentar insertar el módulo obtenemos el error "kernel-module version mismatch" es debido a que compilamos el módulo con las cabeceras (archivos .h) de una versión distinta a la del núcleo que estamos ejecutando. Una solución poco elegante pero eficaz es modificar la primera línea de código del archivo "/usr/include/linux/version.h" y hacerla coincidir con la versión del núcleo que estamos ejecutando. Para averiguar cual es, como superusuario colocamos el comando "uname –a".

Otra solución es agregar al comienzo del archivo la directiva
#define __NO_VERSION__
que es un símbolo de preprocesador y previene la declaración de "kernel_version" en "<linux/module.h>"

El núcleo no dispone de salida estándar, por lo que no podemos utilizar la función printf(). A cambio, el núcleo ofrece una versión de ésta, llamada printk(), que funciona casi igual. Veremos la salida por consola de texto. Esta función puede no funcionar bien en modo gráfico, por lo tanto es recomendable hacerlo desde una consola de texto. Además podemos ver los mensajes almacenados en el archivo de registros del núcleo, con el comando "dmesg".

Si tenemos un diskette con los programas de los ejercicios, para accederlo habrá que "montarlo" con:

mount /mnt/floppy

y luego copiamos todos los programas "c" a nuestro directorio actual:

$cp /mnt/floppy/*.c .

Note que hay un punto "." indicando como "destino" nuestro directorio actual. De la misma manera si quisieramos guardar todos nuestros archivos "c" al diskette:

$cp *.c /mnt/floppy

Luego, al dejar de utilizar el diskette[5] es necesario "desmontar" la unidad:

umount /mnt/floppy

Esto es muy necesario para ir "salvando" nuestro trabajo ya que, como dijimos, al estar trabajando sobre un disco en RAM no permanecerá al apagar/reiniciar el equipo.
Para averiguar por qué es necesario montar los dispositivos en Unix (MINIX, Linux, etc) para acceder al sistema de archivos, lea el punto "1.3.2 Archivos" de Tanenbaum & Woodhull "Sistemas Operativos, diseño e implementación"

Como vemos programar a nivel de kernel no es tarea sencilla. Sin embargo podemos ver cómo **funciona** el núcleo de un sistema operativo gracias a la interfaz provista por las *llamadas a sistema*:

[5] Incluso antes de reemplazarlo por otro

Autoevaluación

1.Mencione los tres propósitos principales de un sistema operativo.
2.¿Qué es la multiprogramación?
3.¿Qué es el grado de multiprogramación?
4.Mencione la principal ventaja de la multiprogramación
5.Los procesadores modernos tienen (al menos) dos modos de operación; menciónelos.
6.¿Cuáles de las siguientes instrucciones sólo deben permitirse en modo kernel?
- ❑Inhabilitar todas las interrupciones
- ❑Leer el reloj de hora del día.
- ❑Establecer el reloj de hora del día
- ❑Cambiar el mapa de memoria.
- ❑Establecer el valor de un temporizador.

7.El modelo cliente-servidor es popular en los sistemas distribuidos ¿Puede usarse también en los sistemas de una sola computadora?
8.¿Por qué se necesita la tabla de procesos en un sistema de tiempo compartido? ¿Se necesita también en los sistemas de computadora personal, en los que sólo existe un proceso, el cual se apodera de toda la máquina hasta terminar?
9.Defina las propiedades esenciales de los siguientes tipos de sistemas operativos:
- ❑Por lotes:
- ❑Interactivos:
- ❑De tiempo compartido:
- ❑De tiempo real:
- ❑Distribuidos:

10.Describa las diferencias entre multiprocesamiento simétrico y asimétrico. Mencione tres ventajas y una desventaja de los sistemas multiprocesador.
11.¿Qué diferencias hay entre una trampa (o excepción) y una interrupción? ¿Para que sirve cada función?
12.¿Qué contiene una entrada de la tabla de vectores de interrupción?
- ❑El nombre de la rutina de tratamiento.
- ❑La dirección de la rutina de tratamiento.
- ❑El número de la interrupción.
- ❑El nombre de la tarea del S.O. que trata la interrupción.

13.¿Qué diferencia existe entre un mandato (o comando) y una llamada al sistema?
14.¿Cómo se solicita una llamada al sistema operativo?
15.¿Qué tipo de sistema operativo es mas fácil de modificar, uno monolítico o uno por capas? ¿Cuál es mas eficiente?
16.¿Qué ventajas considera que tiene escribir un sistema operativo utilizando un lenguaje de alto nivel?

Segunda clase
Procesos (1ª parte)

Objetivo conceptual: que el alumno aprenda conceptos fundamentales de la gestión de procesos, métodos de planificación de procesos, comunicación entre procesos, sincronización de procesos y manejo de bloqueos mutuos. El tema también incluye un tratamiento de hilos.
Objetivo procedimental: que el alumno complemente los conocimientos teóricos con ejemplos en código POSIX y WIN32.
Desarrollo: 2 (dos) semanas de 14 previstas (2ª y 3ª semana)

Una primera definición sencilla: un proceso es un programa en ejecución
Diferencias entre un programa y un proceso

Programa	Proceso
Es estático	Es dinámico
No tiene contador de programa	Tiene un contador de programa
Existe desde que se edita hasta que se borra	Su ciclo de vida comprende desde que se lo "dispara" hasta que termina

Los tipo de llamadas al sistema referidas a los procesos, que un sistema operativo normalmente ofrece son:

- Finalizar, abortar
- Cargar, ejecutar.
- Crear proceso, terminar proceso
- Obtener atributos de proceso, establecer atributos de proceso.
- Esperar un lapso de tiempo.
- Esperar suceso, indicar la ocurrencia del suceso.
- Asignar y liberar memoria.

Estados de un proceso: En su ciclo de vida el proceso pasa por diferentes estados. Básicamente:

- Nuevo (new).
- Ejecutándose (running)
- En espera (waiting)
- Listo para ejecutar (ready)
- Terminado (terminated)

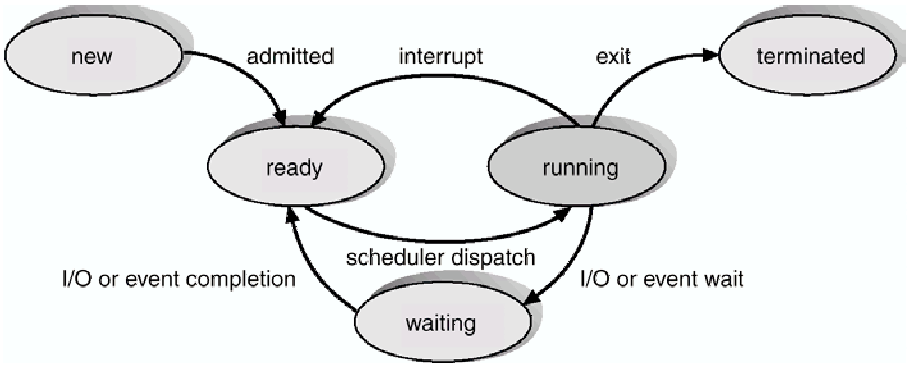


Diagrama de estados de un proceso (Silberschatz)

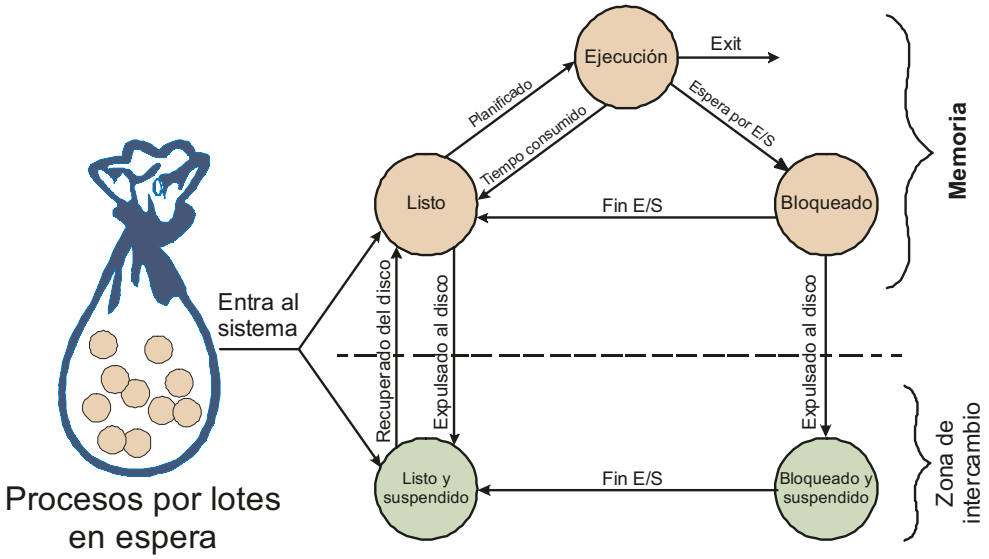


Diagrama de estados de un proceso (Carretero)

Bloque de control de procesos: Cada proceso se representa en el sistema operativo con un bloque de control de proceso (PCB, *process control block*) también llamado bloque de control de tarea.

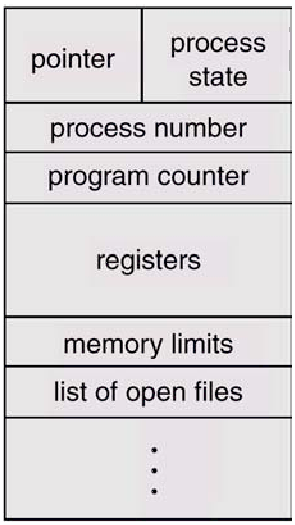


Fig. BCP (Silberschatz)

- ✓Estado del proceso: El estado puede ser: nuevo, listo, en ejecución, en espera, detenido, etcétera.
- ✓Contador de programa: El contador indica la dirección de la siguiente instrucción que se ejecutará para este proceso.
- ✓Registros de CPU: El número y el tipo de los registros varía dependiendo de la arquitectura del computador. Los registros incluyen acumuladores, registros índice, punteros de pila y registros de propósito general, así como cualquier información de códigos de condición que haya. Junto con el contador de program, esta información de estado se debe guardar cuando ocurre una interrupción, para que el proceso pueda continuar correctamente después.
- ✓Información de planificación de CPU: Esta información incluye una prioridad del proceso, punteros a colas de planificación y cualquier otro parámetro de planificación que haya.
- ✓Información de gestión de memoria: Esta información puede incluir datos tales como el valor de los registros de base y límite, las tablas de páginas o las tablas de segmentos, dependiendo del sistema de memoria empleado por el sistema operativo.
- ✓Información contable: Esta información incluye la cantidad de tiempo de CPU y tiempo real consumida, límites de tiempo, números de cuenta, números de trabajo o proceso, y demás.
- ✓Información de estado de E/S: La información incluye la lista de dispositivos de E/S asignadas a este proceso, una lista de archivos abiertos, etcétera.

El PCB sirve como depósito de cualquier información que pueda variar de un proceso a otro.

Jerarquía de procesos: La secuencia de creación de procesos genera un **árbol de procesos**. Para referirse a las relaciones entre los procesos de la jerarquía se emplean los términos de padre, hermano o abuelo. Cuando el proceso A solicita al sistema operativo que cree el proceso B, se dice que A es padre de B y que B es hijo de A. Bajo esta óptica, la jerarquía de procesos puede considerarse como un árbol genealógico. Algunos sistemas operativos, como Unix, mantienen de forma explícita esta estructura jerárquica de procesos – un proceso sabe quién es su padre -, mientras que otros sistemas operativos como el Windows NT no la mantienen.

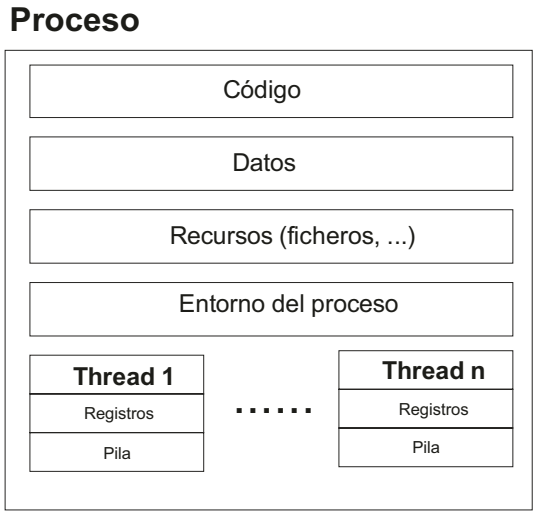
Planificador y activador: El planificador (*scheduler*) forma parte del núcleo del sistema operativo. Entra en ejecución cada vez que se activa el sistema operativo y su misión es seleccionar el proceso que se ha de ejecutar a continuación. El activador (*dispatcher*) también forma parte del sistema operativo y su función es poner en ejecución el proceso seleccionado por el planificador.

Cambio de contexto: La activación del sistema operativo se realiza mediante el mecanismo de las interrupciones. Cuando se produce una interrupción se realizan las dos operaciones siguientes:
Se salva el estado del procesador en el correspondiente PCB.
Se pasa a ejecutar la rutina de tratamiento de interrupción del sistema operativo.
Llamaremos **cambio de contexto** (*context switch*)al conjunto de estas operaciones. El tiempo de conmutación de contexto es exclusivamente gasto extra (*overhead*), porque el sistema no realiza trabajo útil durante la conmutación. Por ser un cuello de botella tan

importante para el desempeño del sistema operativo se están empleando estructuras nuevas (hilos) para evitarla hasta donde sea posible.

Procesos ligeros, hilos o *threads*: Un proceso ligero es un programa en ejecución (flujo de ejecución) que comparte la imagen de memoria y otras informaciones con otros procesos ligeros. Un proceso puede contener un solo fljo de ejecución, como ocurre en los procesos clásicos, o más de un flujo de ejecución (procesos ligeros). Desde el punto de vista de la programación, un proceso ligero se define como una función cuya ejecución se puede lanzar en paralelo con otras. El hilo de ejecución primario, o proceso ligero primario, corresponde a la función *main*. Todos los procesos ligeros de un mismo proceso comparten el mismo espacio de direcciones de memoria, que incluye el código, los datos y las pilas de los diferentes procesos ligeros.

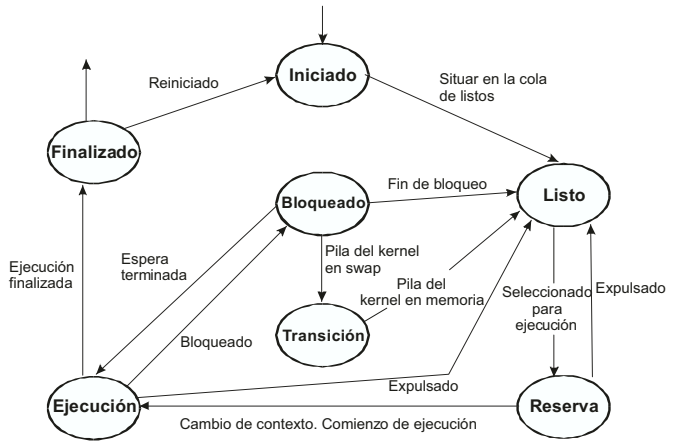
El proceso ligero constituye la unidad ejecutable en Windows NT.



Estructura de un proceso en Windows NT

Planificación en POSIX (Linux): POSIX especifica una serie de políticas de planificación, aplicables a procesos y procesos ligeros, que debe implementar cualquier sistema operativo que ofrezca esta interfaz.

Planificación en Windows NT/2000: En Windows NT la unidad básica de ejecución es el proceso ligero y, por tanto, la planificación se realiza sobre este tipo de procesos.



Estados de los procesos ligeros en Windows NT

Trabajo Práctico Nº 2

Unix provee el comando "ps" (*process status*) para ver el estado de los procesos. Si coloca este comando en Linux verá básicamente dos procesos, el del intérprete de comandos *bash* y el del propio proceso *ps*. **Fíjese** en las columnas, la primera indica el número identificador del proceso (PID, *process identification*). Con la flecha de cursor puede recuperar este comando para colocarlo de nuevo, repita esta operación varias veces, **notará** que el PID del "bash" no cambia, pero sí lo hace el de "ps". Esto es así porque "bash" es un proceso que sigue ejecutando, en cambio "ps" cumple todo su ciclo de vida y cada invocación que hace es en realidad un nuevo proceso, y por lo tanto, el sistema operativo le asigna un número nuevo. Cada vez que ejecuta "ps" el intérprete de comandos genera un hijo con la llamada al sistema **fork()** y **exec()**. Las cuales utilizaremos en un programa en lenguaje C.
El comando "ps" sin opciones nos muestra lo que ya hemos visto. Si queremos ver todos los procesos que se están ejecutando en el sistema coloquemos el comando "ps aux". La opción "a" significa "*all*", es decir, todos los procesos; la opción "u" indica "formato de usuario"; la opción "x" es para que muestre aún a los procesos no asociados a terminal.

Si quiere aprender más acerca de las opciones del comando "ps", consulte las páginas del manual, con el comando "man ps".

Podemos utilizar además el comando "top" que nos muestra en forma decreciente los procesos que más recursos consumen, y va actualizando la lista periodicamente. Para salir de este programa presione la tecla "q".

Observe que el orden se va alterando. Es decir, que de la cola de procesos listos para ejecutar, el planificador los va elijiendo por su prioridad, pero ésta no es constante en Linux, sino que se ven alteradas de acuerdo con un sistema de "créditos", que veremos más adelante.

Ahora a programar:

Descripción de los principales servicios que ofrece POSIX y Win32 para la gestión de procesos, procesos ligeros y planificación.

Obtención del identificador del proceso mediante la llamada (*system call*) al sistema cuyo prototipo en lenguaje C es:

pid_t getpid(void);

Carga y ejecución del siguiente programa

```
#include <sys/types.h>
#include <stdio.h>
main()
{
pid_t id_proceso;
```

```
pid_t id_padre;
id_proceso=getpid();
id_padre=getppid();
printf("Identificador de proceso: %d\n", id_proceso);
printf("Identificador del proceso padre: %d\n", id_padre);
}
```

Este programa se compila simplemente así:

```
gcc     -o      ejemplo         ejemplo.c
```

Suponiendo que así ha nombrado al programa. Y se ejecuta así:

```
./ejemplo
```

Es decir "punto" "barra" "ejemplo" sin espacios.

Ejemplo en POSIX de utilización de servicio para la creación de procesos.

Carga y ejecución del siguiente programa:

```
#include <sys/types.h>
#include <stdio.h>
main()
{
pid_t pid;
pid=fork();
switch(pid){
        case –1:        /* error en el fork() */
                perror("fork");
                break;
        case 0:         /* proceso hijo */
                printf("Proceso %d; padre = %d \n", getpid(), getppid() );
                break;
        default:        /* proceso padre */
                printf("Proceso %d; padre = %d \n", getpid(), getppid() );
        }
}
```

Observe, que el proceso hijo es una copia del proceso padre en el instante en que éste solicita el servicio *fork()*. Esto significa que los datos y la pila del proceso hijo son los que tiene el padre en ese instante de ejecución.
Piense y responda: ¿El proceso hijo empieza la ejecución del código en su punto de inicio, o en la sentencia que está después del *fork()*?
Observe, que el hijo no es totalmente idéntico al padre, algunos de los valores del BCP han de ser distintos.
Piense y luego responda: ¿cuáles deberían ser las diferencias más importantes?

	V	F
El proceso hijo tiene su propio identificador de proceso, distinto al del padre.		
El proceso hijo tiene una nueva descripción de la memoria. Aunque el hijo tenga los mismos segmentos con el mismo contenido, no tienen por qué estar en la misma zona de memoria (esto es especialmente cierto en el caso de sistemas sin memoria virtual)		
El tiempo de ejecución del proceso hijo se iguala a cero.		
Todas las alarmas pendientes se desactivan en el proceso hijo		
El conjunto de señales pendientes se pone a vacío		
El valor que retorna el sistema operativo como resultado del fork es distinto en el hijo y el padre		

Observe que las modificaciones que realice el proceso padre sobre sus registros e imagen de memoria después del *fork()* no afectan al hijo y, viceversa, las del hijo no afectan al padre,

Sin embargo, el proceso hijo tiene su propia copia de los descriptores del proceso padre. Este hace que el hijo tenga acceso a los archivos abiertos por el proceso padre. El padre y el hijo comparten el puntero de posición de los archivos abiertos en el padre.

Piense y luego responda si esto podría afectar y de que manera a lo que acaba de observar.

Segunda clase
Procesos (2ª parte)

Planificación de CPU

Conceptos

La planificación es una función fundamental del sistema operativo. Siendo la CPU uno de los principales recursos, la planificación de su uso es sumamente importante.

En un ambiente de multiprogramación, el sistema operativo debe decidir a qué proceso le da la CPU entre los que están listos para ejecutarse. ¿Cuál es el criterio por el cual se prefiere la ejecución de uno sobre los otros?

Existen diferentes algoritmos de selección, pero para escoger cuál es el que corresponde a nuestro sistema debemos tener claro bajo qué criterios nos interesa fijar las pautas. Por ejemplo:

- ✓ *Maximizar el uso de la CPU: que se mantenga ocupada el mayor tiempo posible.*
- ✓ *Maximizar la Productividad (**throughput**)*, considerando que *productividad (o rendimiento)* es el número de procesos que se ejecutan por unidad de tiempo.
- ✓ *Minimizar el Tiempo de retorno (**turnaround time**),* que es el tiempo transcurrido desde que el proceso ingresa hasta que termina, sumando espera para entrar en memoria, en cola de procesos listos, ejecutándose en CPU y haciendo E/S.
- ✓ *Minimizar el Tiempo de espera (**waiting time**),* que es el tiempo que el proceso está en la cola de listos.
- ✓ *Minimizar el Tiempo de respuesta (**response time**),* que es el tiempo que transcurre desde que se presenta una solicitud hasta que se tiene respuesta. Es el tiempo que tarda en comenzar a responder, no incluyendo el tiempo de la respuesta en sí.

Lo ideal es maximizar el uso de la CPU y la productividad y minimizar los tiempos de retorno, respuesta y espera.

Son otros criterios:

- ✓ La *equidad*: que todos los procesos tienen que ser tratados de igual forma y a todos se les debe dar la oportunidad de ejecutarse, para que ninguno sufra *inanición*.
- ✓ *Recursos equilibrados*: la política de planificación que se elija debe mantener ocupados los recursos del sistema, favoreciendo a aquellos procesos que no abusen de los recursos asignados, sobrecargando el sistema y bajando la performance general.

Organización. Colas y schedulers

Colas

En un sistema operativo los recursos compartidos exigen la organización en colas de las solicitudes pendientes. Tenemos colas para la planificación del uso de la cpu como también colas para el uso de dispositivos, ***device queues***, que son utilizadas para ordenar los requerimientos que varios procesos pueden tener sobre un dispositivo específico.

Al ser creados, los procesos se colocan en ***la cola de trabajos***, formada por los procesos que aún no residen en la memoria principal pero listos para su ejecución.

La residencia en memoria es imprescindible para la ejecución de un proceso. La cola de trabajos listos (***ready queue***) es aquélla formada por los procesos que están listos para la ejecución, en memoria principal. Son los procesos que compiten directamente por CPU.

Un proceso que está en la cola de listos no puede estar en las colas de dispositivo, pues en este último caso está esperando por E/S, por lo tanto no puede estar compitiendo por cpu.

Los elementos de cualquiera de estas colas no son los procesos en sí, sino sus PCB's, que están en memoria.

La cola de procesos listos (*ready queue*) se almacena como una lista enlazada donde la cabecera (***header***) de la ready queue tiene punteros al primer y al último PCB de los procesos de la lista. Cada PCB apunta al próximo en la lista.

Schedulers, dispatchers – Planificadores, activadores

Hay módulos del sistema operativo para elegir un proceso para su ejecución de entre los que están compitiendo por CPU. Esa elección dependerá del criterio adoptado (según lo explicado en el primer punto). La elección estará asociada al carácter del sistema. Por ejemplo, en un sistema batch no hay tanta exigencia en cuanto al tiempo de respuesta. En un sistema interactivo minimizar este tiempo es imprescindible.

El algoritmo que se eligirá para la elección de procesos será diferentes en cada caso. Recordemos que los sistemas operativos modernos permiten la convivencia de las dos modalidades (batch e interactivo): ¿cómo lo solucionamos?

Necesitamos que el sistema operativo, a través de sus módulos, elija un proceso de acuerdo al criterio elegido, haga el cambio de contexto, lo cargue en memoria y le dé la CPU. Cuando un proceso termina, normal ó anormalmente, debe elegirse otro proceso para cargar en memoria.

¿Qué pasa cuando un proceso de mayor prioridad debe ejecutarse y no hay memoria suficiente? Un módulo debe encargarse de seleccionar uno de los procesos que está en memoria y sacarlo temporariamente a memoria secundaria para dar lugar (*swapping)*, pero teniendo en cuenta que sigue compitiendo por CPU y que se debe resguardar su contexto.

Estos módulos del sistema operativo son rutinas especiales y dependerá de la implementación cómo las funciones que citamos se realizan en cada sistema.

En un sistema hay procesos con mayor proporción de ráfagas E/S (***I/O bound process***) y otros con mayor necesidad de cpu que de E/S (***CPU bound***). Si cuando se seleccionan procesos para ejecutar se eligieran todos I/O bound, tendríamos las colas de dispositivo llenas de solicitudes y, probablemente, la CPU ociosa. Si se eligen muchos procesos limitados por CPU la cola de listos estará llena y las colas de dispositivo, vacías.

Por eso es muy importante de qué manera se realiza la selección de trabajos para mantener el sistema en equilibrio.

Esta selección la realizan los planificadores o ***schedulers***.

Hay distintos planificadores que participan en diferentes momentos:

- ✓El de largo plazo o ***long term***, que elige entre la cola de listos, para cargarlo en memoria;
- ✓El de corto plazo o ***short term*** (también llamado *CPU scheduler*), que elige entre los que están listos en memoria para darle la CPU;
- ✓El de medio plazo o ***medium term***, que selecciona un proceso de entre los que están en memoria para llevarlo *temporariamente* a memoria secundaria por necesidades de espacio en memoria.

Es importante aclarar que en las colas los registros son normalmente las PCB's de los procesos.

Otro modulo de importante participación en la planificación es el activador o ***dispatcher***. Es el que da el control de la CPU al proceso elegido por el planificador de corto plazo.

Debe ser muy rápido pues una de sus funciones es encargarse del cambio de contexto (context switch). Al tiempo entre detener un proceso y comenzar a correr otro se le llama ***dispatch latency***.

El dispatcher es también el que se encarga de pasar a modo usuario el proceso que esta activando y "saltar" a la dirección de la instrucción que comienza la ejecución del programa.

Análisis sobre la productividad

Hemos hablado que en un ambiente de multiprogramación cuando un proceso se bloquea a la espera de algún evento, la CPU atiende al primer proceso que se halle en la cola de procesos listos (ready) para su ejecución.

Si un proceso P0 dura en total 11 segundos, intercalándose las ráfagas de cpu y de E/s, si no existiera la posibilidad de atender otro proceso P1 en los momentos que P0 hace E/S, P1 debería esperar hasta la culminación de P0 para ejecutarse.

Si vamos en cambio intercalando el proceso de CPU de P0 con el de P1 (en el momento que P0 hace E/S) aumentaremos la productividad pues en un período menor de tiempo aumentamos la cantidad de trabajos realizados

Incidencia del Cambio de contexto(context switching)

El tiempo del cambio de contexto varía según el hardware. La velocidad de memoria, número de registros y si hay o no instrucciones especiales, inciden en ese tiempo.

En el análisis que haremos aquí no consideramos el tiempo para el context switching pues este debería ser despreciable en comparación con el quantum que se elija. Véase que un sistema donde este tiempo es considerable, el sistema operativo tendrá mucho tiempo dedicado a esta operación, en vez de dedicarlo a procesar.

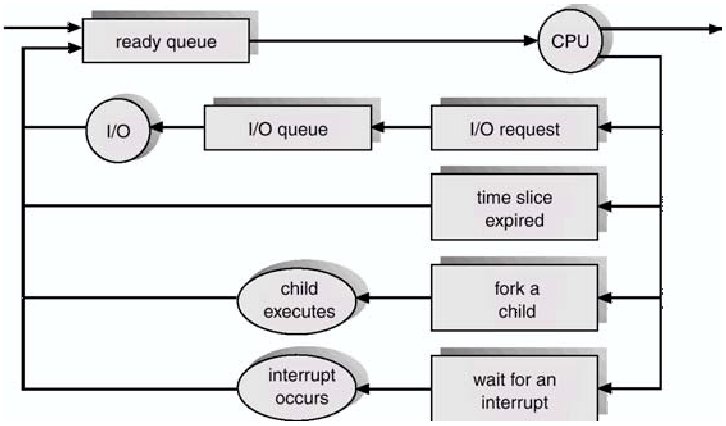
Procesos orientados a la E/S y procesos orientados a la CPU

Hay procesos *orientados a la E/S* y procesos *orientados a la CPU*.

Los procesos orientados a la CPU tienen proporcionalmente ráfagas de CPU mayores a las de E/S, siendo estas últimas además, pocas.

Normalmente los procesos de este tipo utilizan toda su porción de tiempo o *quantum* al no tener que abandonar la CPU por E/S.

Los procesos orientados a la E/S tienen proporcionalmente ráfagas de E/S mayores a las de CPU y muy seguidas. En un ambiente de multiprogramación donde haya preponderancia de estos procesos, habrá una intervención muy frecuente del planificador de corto plazo, y mucho cambio de contexto.



Representación con diagrama de colas de la planificación de procesos

Algoritmos de planificación

Estos algoritmos tratan sobre la decisión de elegir a cual de los procesos que están en la cola de listos se le asignara la CPU. Veamos algunos de ellos.

FCFS (first-come, first-serve) Primero en llegar, primero en ser atendido

Se le asigna la CPU al proceso que la requirió primero. Se maneja a través de una cola FIFO (*First In First Out*), tal que cuando un proceso requiere CPU, su PCB se coloca la final de la cola.

Cuando se debe elegir un proceso para asignarle CPU se elige el proceso cuya PCB esta primera en la cola.

El código para implementar este algoritmos es simple y comprensible. Pero el tiempo promedio de espera puede ser largo.

El efecto convoy del FCFS

Una situación indeseable que puede ocurrir cuando se mezclan procesos CPU bound con proceso I/O bound es el ***efecto convoy***.

EL proceso CPU bound toma la CPU y la mantiene pues el resto de los procesos al tener tanto I/O se la pasa circulando por las diferentes colas (de ready, de dispositivo) o en espera. El resultado es un bajo rendimiento de la CPU, que esta prácticamente atendiendo un solo proceso.

Este efecto podría evitarse atendiendo procesos mas cortos primero, aumentando la productividad.

Shortest-job-first Scheduling (SJF). Planificación "primero el trabajo mas corto"

Este algoritmo elige entre los procesos de la cola de listos, aquel que tenga la próxima ráfaga de CPU mas corta. Para ello este dato debe ser conocido, obviamente.

Es un algoritmo que permite que el tiempo de espera promedio sea bajo.

Se puede utilizar en planificadores de largo plazo, pero no en los de corto plazo pues no hay manera de conocer la medida de la próxima ráfaga. Se podría aproximar considerando el valor de la previa.

Por prioridad

Se le asocia un valor a cada proceso que representa la prioridad, y se le asigna la CPU al proceso de la cola de ready que tenga la mayor prioridad.

Los valores asociados a la prioridad son un rango fijo, de 0 a n, según cada sistema.

También lo determina cada sistema si el numero mas bajo es la mas alta o la mas baja prioridad.

La prioridad puede ser fija o variable, externa o interna.

Si es fija, ese valor no varia en el ciclo de vida del proceso. Si es variable, significa que ese valor puede cambiar dinámicamente de manera tal que haya factores, por ejemplo, el tiempo que lleva esperando en colas, que puedan "ayudar" a que haya un mejor nivel de competencia elevando la prioridad de procesos postergados y evitar una situación indeseable llamada *starvation* (inanición). A la técnica de elevar la prioridad a un proceso de acuerdo al tiempo que hace que esta en el sistema se le llama *aging.*

Si la prioridad es interna, es determinada en función del uso de los recursos (memoria, archivos abiertos, tiempos). Si es externa, puede decidirse darle alta prioridad a un proceso de importancia, a consideración del operador.

POSIX proporciona planificación basada en prioridades.

Round-Robin Scheduling (RR). Planificación por turno circular.

Se la asigna a cada proceso de la cola de listos un intervalo de tiempo de CPU. Ese intervalo es llamado *time slice* o *quantum.*

Para implementar este algoritmo la cola de listos se mantiene como una cola FIFO. Y la CPU se va asignando dándole un máximo de 1 quantum a cada proceso.

Si el proceso no va a usar todo ese tiempo, usa lo necesario y libera la CPU. Se elige entonces otro proceso de la cola. Si excede el quantum, se produce una interrupción.

En ambos casos al dejar la CPU hay un cambio de contexto.

El tiempo promedio de espera puede ser largo. Su performance depende fuertemente de la elección del quantum. Y el tiempo gastado en context switch es una variable que se debe tener muy en cuenta. Se debe determinar un quantum que sea mucho mayor que el tiempo de context switch.

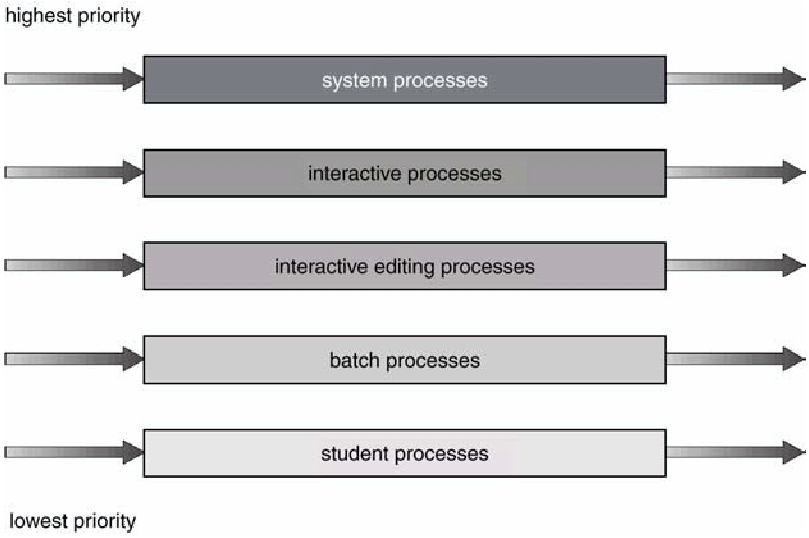
Se utiliza en los sistemas time-sharing.

Colas multinivel

En un sistema conviven procesos mixtos. Puede haber requerimientos de tiempo de respuesta distintos si conviven proceso interactivos con procesos batch (o para aquellos que provienen del Unix, procesos en foreground y procesos en background).

Existen algoritmos que contemplan esta situación dividiendo la cola de ready en distintas colas según el tipo de proceso, estableciendo una competencia entre las colas y entre los procesos del mismo tipo entre si.

Cada cola puede tener su algoritmo de scheduling: los procesos batch pueden ser "planificados" por un algoritmo FCFS y la cola de procesos interactivos, por un RR.



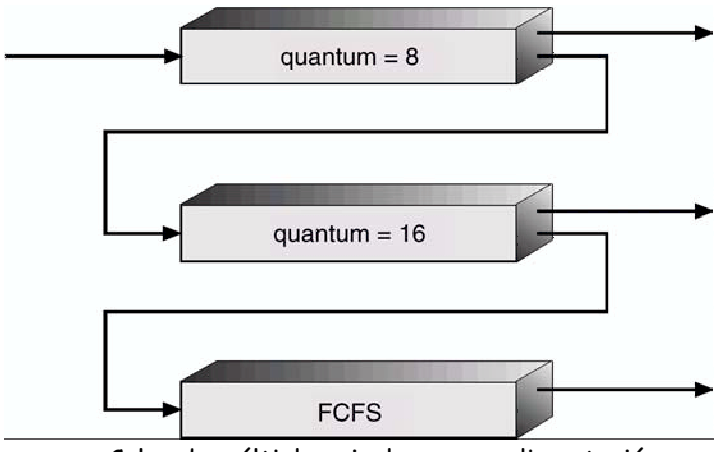
Planificación con colas de múltiples niveles

Colas multinivel con retroalimentacion

En el caso planteado anteriormente, un proceso permanecía en la cola asignada hasta su atención. En el modelo con retroalimentacion, un proceso puede "moverse" entre colas, es decir, según convenga puede llegar a asignarse a otra cola.

Los procesos se separan de acuerdo a la ráfaga de CPU que usan. Si esta usando mucha CPU se lo baja a una cola de menor prioridad. Se trata de mantener los procesos interactivos y de mucha I/O en las colas de mayor prioridad.

Si además un proceso estuvo demasiado tiempo en una cola de baja prioridad puede moverse a una cola de mayor prioridad para prevenir inanición (starvation).



Colas de múltiples niveles con realimentación

Scheduling apropiativo, expropiativo o expulsivo (preemptive)

Hay que considerar en el esquema de planificación en que momento se realiza la selección.

Un algoritmo ***no apropiativo*** es aquel que una vez que le da la CPU a un proceso se ejecuta hasta que termina o hasta que se bloquea hasta la ocurrencia de determinado evento.

En cambio, un algoritmo ***apropiativo*** es aquel en que un proceso que se esta ejecutando puede ser interrumpido y pasado a cola de listos (ready queue) por decisión del sistema operativo. Esta decisión puede ser porque llego un proceso de mayor prioridad, por ejemplo.

Si bien los algoritmos apropiativos tienen un mayor costo que los no apropiativos, el sistema se ve beneficiado con un mejor servicio pues se evita que algún proceso pueda apropiarse de la CPU.

No obstante si se utilizaran algoritmos apropiativos debe considerarse si el sistema operativo esta preparado para desplazar en cualquier momento un proceso. Por ejemplo, si ese proceso en ese momento, a causa de una llamada al sistema (*system call*) esta modificando estructuras de kernel y se lo interrumpe, esas estructuras pueden quedar en un estado inconsistente. O puede ocurrir con procesos que comparten datos y uno de ellos esta modificándolos al ser interrumpido.

En Unix, por ejemplo se espera que se termine de ejecutar el system call para permitir apropiación.

Relacionemos la idea de apropiación con los diferentes algoritmos vistos.

- ✓FCFS es no apropiativo. Cuando se le da la CPU a un proceso, este la mantiene hasta que decide liberarla, porque termino, o porque requiere I/O.
- ✓SJF puede ser apropiativo o no. Si mientras se esta ejecutando un proceso llega uno cuya próxima ráfaga de CPU es mas corta que lo que queda de ejecutar del activo, puede existir la decisión de interrumpir el que se esta ejecutando o no. Al SJF apropiativo, se le llama *shortest-remaining-time-first*. (primero el de tiempo restante más corto).
- ✓Prioridades puede ser apropiativo o no. Si mientras se esta ejecutando un proceso llega a la cola de ready un proceso de mayor prioridad que el que se esta ejecutando puede existir la decisión de interrumpir el que se esta ejecutando o no. Si es no apropiativo, en lugar de darle la CPU al nuevo proceso, lo pondrá primero en la cola de ready.
- ✓RR es apropiativo pues si el proceso excede el tiempo asignado, hay una interrupción para asignarla la CPU a otro proceso.

POSIX (ver Carretero p. 137) especifica tres tipos de políticas de planificación:

- ✓FIFO. Pero pueden ser expulsados por procesos de mayor prioridad y
- ✓RR (*round robin*) (turno circular). Los procesos dentro del mismo nivel de prioridad ejecutan según una política de planificación RR. Son expulsados de la CPU cuando acaba su rodaja de tiempo o cuando son expulsados por un proceso de mayor prioridad. El tercer tipo se denomina:
- ✓OTHER, que queda libre a la implementación y se debe documentar adecuadamente.

La planificación de procesos en ambientes multiprocesador

En el caso en que los *procesadores son idénticos*, cualquier procesador disponible puede usarse para atender cualquier proceso en la cola. No obstante, debemos tener en cuenta que puede haber dispositivos asignados de manera privada o única a un procesador, y si un proceso quiere hacer uso de ese dispositivo deberá ejecutarse en ese procesador. Otra ventaja es implementar ***load sharing* (carga compartida)**, permitiendo que si un procesador esta ocioso puedan pasarse procesos de la cola de otro procesador a este, o hacer una cola común, y la atención se realiza de acuerdo a la actividad de cada uno.

En ambientes de *procesadores heterogéneos* no olvidemos que un procesador podría ejecutar solo aquellos procesos que fueron compilados de acuerdo a su conjunto de instrucciones.

En otros sistemas se asume un estructura de procesadores master-slave que asigna a un procesador la toma de decisiones en cuanto a scheduling y otras actividades, dedicándose los otros procesadores a ejecutar código de usuario. La ventaja es que solo el procesador master accede a estructuras del kernel, evitando inconsistencias.

Un ejemplo: Windows NT

El sistema operativo Windows NT es un ejemplo de diseño moderno que utiliza modularidad para incrementar la funcionalidad y reducir el tiempo necesario para implementar nuevas características. NT apoya múltiples entornos operativos o *subsistemas* con los que los programas de aplicación se comunican empleando un sistema de transferencia de mensajes. Los programas de aplicación se pueden considerar como clientes del servidor de subsistemas NT.

El recurso de transferencia de mensajes en Windows NT se llama Recurso de Llamada a Procedimiento Remoto (LPC, Local Procedure Call), y sirve para que se comuniquen entre sí dos procesos que están en la misma máquina. El mecanismo es similar al de llamada a procedimientos remotos estándar de uso general pero optimado y específico para NT. Windows NT, al igual que Mach, utiliza un objeto puerto para establecer y mantener una conexión entre dos procesos. Cada cliente que invoca un subsistema necesita un canal de comunicación, mismo que el objeto puerto proporciona y que nunca se hereda. NT maneja dos tipos de puertos, puertos de conexión y puertos de comunicación, que en realidad son iguales pero reciben diferentes nombres según la forma en que se usa. Los puertos de conexión son objetos con nombre, visibles para todos los procesos, y ofrecen a las aplicaciones una forma de establecer un canal de comunicación. La comunicación funciona como sigue:

- ✓El cliente abre un mando para el objeto puerto de conexión del subsistema.
- ✓El cliente envía una solicitud de conexión.
- ✓El servidor crea dos puertos de comunicación privados y devuelve el mando de uno de ellos al cliente.
- ✓El cliente y el servidor usan el mando de puerto correspondiente para enviar mensajes o devoluciones de llamada y esperar respuestas.

NT emplea tres tipos de técnicas de transferencia de mensajes por un puerto que el cliente especifica cuando establece el canal. La más sencilla, que se usa si el mensaje es pequeño, consiste en utilizar la cola de mensajes del puerto como almacenamiento intermedio y copiar el mensaje de un proceso al otro. Es posble enviar mensajes de hasta 256 bytes con este método.

Si un cliente necesita enviar un mensaje más grande, lo transfiere a través de un objeto sección (memoria compartida). El cliente tiene que decidir, cuando establece el canal, si necesitará o no enviar un mensaje grande. Si el cliente determina que desea enviar mensajes grandes, pide que se cree un objeto sección. Así mismo, si el servidor decide que las respuestas serán grandes, también creará un objeto sección.

Para usar este objeto, se envía un mensaje pequeño que contiene un puntero e información de tamaño referente al objeto sección. Este método es un poco más complicado que el primero, pero evita el copiado de datos. En ambos casos se puede utilizar un mecanimsmo de devolución de llamada si el cliente o el servidor no puede responder de inmediato a una solicitud. El mecanismo de devolución de llamda hace posible el manejo asincrónico de mensajes.

Una de las desventajas del uso de transferencia de mensajes para desempeñar funciones rudimentarias, como la de gráficos, es que el desempeño podría no ser tan bueno como el de un sistema de memoria compartida. Con objeto de mejorar el desempeño, el entorno NT nativo (Win32) utiliza un tercer método de transferecia de mensajes llamado LPC rápida. Un cliente envía una solicitud de conexión al puerto de conexión del servidor e indica que usará LPC rápida. El servidor crea un hilo dedicado para atender las solicitudes, un objeo sección de 64 KB, y un objeto par de sucesos. A partir de ese momento, los mensajes se pasan a través del objeto sección y el objeto par de sucesos se encarga de la sincronización. Esto elimina el copiado de mensajes, el gasto extra de usar el objeto puerto, y el gasto extra que implica determinar cuál hilo cliente llamó, ya que sólo hay un hilo servidor para cada hilo clientes. Además, el núcleo concede al hilo dedicado preferencia de planificación. Desde luego, la desventaja de este método es que consume más recursos que los otros dos. Puesto que la transferencia de mensajes implica cierto gasto extra, NT podría juntar varios mensajes en uno solo para reducir ese gasto extra.

Trabajo Práctico Nº 3

Como decíamos anteriormente POSIX proporciona planificación basada en prioridades. Estas prioridades pueden ser modificadas dinámicamente mediante servicios específicos.

La planificación en Linux se lleva a cabo en la función "schedule" del archivo kernel/sched.c

Linux implementa el esquema de planificación definido en las extensiones de tiempo real de POSIX. Existen tres formas de planificar un proceso:

- ✓SCHED_FIFO: Un proceso con este tipo de planificación tiene asociada una prioridad estática. Son más prioritarios que los procesos "normales". Una vez que han conseguido el procesador, sólo pueden ser expulsados por otro proceso más prioritario. Esto significa que no tienen un quantum máximo de ejecución.
- ✓SCHED_RR: Igual que SCHED_FIFO, pero con una política de Round Robin para los procesos de igual prioridad.
- ✓SCHED_OTHER: Es la política de planificación "clásica" de Unix: un Round Robin con mayor quantum a los procesos que menos CPU han consumido recientemente.
- ✓Existe un bit adicional, SCHED_YIELD, que afecta a las tres políticas de planificación, haciendo que un proceso ceda la CPU a cualquier otro que se encuentre preparado, únicamente par el próximo ciclo de planificación, no indefinidamente.

Podemos analizar paso a paso el código de "schedule", que presenta un aspecto poco ortodoxo tratándose de una función en C, ya que contiene numerosos saltos y etiquetas. La idea es que la línea de ejecución más frecuente no conlleve ni un solo salto.

Los fragmentos de código correspondientes a cuerpos de sentencias if se han "quitado de en medio" a modo de subrutinas implementadas con sentencias goto a una etiqueta. Al final de la mayoría de los fragmentos hay un salto que vuelve a la instrucción siguiente al goto, a través de una etiqueta de vuelta con el mismo nombre más el sufijo _back.

Linux implementa el "sistema de archivos proc" que es como una especie de "mapa" de estructuras de datos en memoria, expresados como si fueran archivos.

Podemos con el comando:

cat /proc/stat

El número de cambios de contexto.

Creación y terminación de procesos

Un proceso crea a otro a través de una llamada al sistema. El creador es el proceso padre y el creado es el proceso hijo.

Estos hijos podrían crear otros, generándose una estructura de árbol con vértice en el padre: es una estructura jerárquica.

Un proceso usa recursos como tiempo de cpu, archivos, memoria, dispositivos para e/s.

Cuando un proceso crea un proceso hijo (o subproceso) éste necesita recursos también. Puede ser que el sistema operativo le asigne esos recursos o pueda usar un subconjunto de los de su padre. Este último método previene la saturación de un sistema por la cantidad de subprocesos que pueda crear un proceso. En la ejecución ante la creación de un subproceso hay dos alternativas:

✓el padre continúa ejecutandose concurrentemente con sus hijos
✓el padre espera que sus hijos hayan terminado para seguir la ejecución.

Un proceso termina cuando ejecuta su última instrucción y solicita al s.o. que lo elimine. Puede devolver información a su padre.

Un proceso puede "matar" a otro a través de un system call. Normalmente sólo puede hacerse desde el padre a un hijo. Puede ser por que:

1.el hijo se excedió en el uso de los recursos
2.porque su tarea ya no es necesaria.

En el caso 1 el padre debe inspeccionar los recursos consumidos por el hijo.
En algunos sistema se produce *la terminación encascada*: cuando termina un padre, deben terminar todos los hijos. En otras palabras: no puede existir un proceso sin que exista su padre ó un proceso no puede sobrevivir a su padre. Observemos que para las tareas de padres e hijos es preciso que los padres puedan identificarlos. Esto se hace a través de un PID (process identification description) que identifica univocamente a un proceso.

La destrucción de un proceso (normal o provocada) implica la devolución de los recursos que tenía asignados (incluso el espacio de memoria que ocupaba el PCB).

Suspensión y reanudación de procesos

Un proceso suspendido sólo puede ser reanudado por otro. La suspensión temporaria puede ser necesaria cuando el sistema está muy cargado. Si la suspensión fuera por largo tiempo, deberían liberarse los recursos apropiados: por ejemplo la memoria debería liberarse inmediatamente; una unidad de cinta puede retenerse un tiempo. Al reanudar un proceso se lo activa desde el punto que se lo suspendió. Los procesos pueden suspenderse por:

✓mal funcionamiento del sistema,
✓el usuario sospecha de los resultados del proceso y lo suspende sin abortarlo, para analizar resultados parciales,
✓carga elevada del sistema, se suspende hasta que los niveles sean más aptos.

Relación entre procesos: independiente y cooperativo
Independientes

No lo afectan y no puede afectar a otros procesos que se estén ejecutando en el sistema.

✓Su estado no es compartido por otro proceso.
✓El resultado de la ejecución depende de la entrada.
✓Su ejecución es reproducible: el mismo resultado para la misma entrada.
✓Se puede detener y reanudar la ejecución sin efectos adversos.

Cooperativo

Lo afectan y puede afectar a otros procesos que se estén ejecutando en el sistema.

✓Su estado es compartido por otro proceso.
✓No se puede predecir el resultado de la ejecución pues depende de una secuencia de ejecución relativa.
✓Su ejecución no es reproducible: el resultado no depende de la entrada.
✓Los procesos cooperativos pueden compartir datos y código o datos sólo a través de archivos. Para el primer caso se utilizan los hilos.

Procesos concurrentes

La ejecución concurrente de procesos existe por varias razones:

✓para compartir recursos físicos (por ejemplo, impresoras)
✓para compartir recursos lógicos (por ejemplo, archivos)
✓para acelerar cálculos (división de una tarea en subtareas, para ejecución paralela)
✓por construcción modularizada (dividir funciones)
✓por comodidad (para poder ejecutar varias cosas a la vez: editar, compilar e imprimir en paralelo)

Implementar concurrencia requiere que se utilicen mecanismos de sincronización y comunicación entre procesos.

Luego de esta introducción teórica, estamos en condiciones de encarar la práctica: Comenzaremos por introducir el comando "top".

El comando "top" provee una vista de la actividad del procesador en "tiempo real". Muestra un listado de los procesos (tareas) mas intensivos en CPU en el sistema, y puede proveer una interfaz interactiva para la manipulación de procesos. Las columnas son:

✓PID: el Identificador de Proceso de cada tarea.
✓USER: el nombre de usuario del propietario del proceso.
✓PRI: la prioridad del proceso.
✓NI: El valor "nice" (mejorado) del proceso. Los valores negativos tienen prioridad más alta.
✓SIZE: El tamaño del código del proceso mas los datos mas el espacio de pila, en kilobytes.
✓RSS: Cantidad total de memoria física usada por el proceso, en kilobytes.

- ✓SHARE: La cantidad de memoria compartida usada por el proceso.
- ✓STAT: El estado del proceso: S (Sleeping) durmiendo, D sueño ininterrumpible, R (running) corriendo, Z zombie, T (traced) trazado o detenido, "<" procesos con valor "nice" negativo, N para procesos con valores "nice" positivos, W para procesos en área de swap.
- ✓%CPU: La porción del proceso del tiempo de CPU desde la última actualización de la pantalla, expresada como porcentaje del tiempo total de CPU por procesador.
- ✓%MEM: La porción del proceso de la memoria física.
- ✓COMMAND: El nombre del comando que generó el proceso.

El comando "nice" corre un programa con la prioridad de planificación modificada. Los rangos van desde –20 (la prioridad más alta) hasta 19 (la más baja).

El comando "kill" envía una señal a un proceso, generalmente la de "matar" o terminar un proceso, pero no necesariamente.

El comando "yes" imprime por pantalla una serie de "y"es en forma interminable. Para finalizar (abortar) el comando/proceso debe presionar simultáneamente CTRL-C, esto envía una "señal" desde el teclado que le indica al comando/proceso que debe finalizar.

Podemos redirigir la salida de este comando a un archivo utilizando a la derecha del comando el carácter ">" (mayor). Existe un archivo/dispositivo especial que funciona como un "agujero negro" o una papelera de reciclaje, todo lo que va a parar ahí se pierde inmediatamente; este archivo/dispositivo especial es "/dev/null".

Todo comando/proceso hasta tanto no finalice, no nos entrega el "prompt" del intérprete de comandos, si está ejecutando desde una consola/terminal. A esto el Unix lo denomina "foreground". Y en contraposición existe el "background", lo que está ejecutando "por detrás" en forma no asociada a una consola/terminal. Esto permite que un proceso disparado desde una consola corra permanentemente, desafectando a la consola/terminal para que el usuario pueda correr otros comandos/procesos.

Para enviar un comando al "background" debe colocar al final de la línea el carácter "&".

Abra una sesión como usuario "adios" y ejecute el comando "top".
Abra otra sesión como usuario "root" (recuerde que la clave es la misma) en otra consola/terminal y ejecute el comando:

```
yes  >  /dev/null  &
```

Verá a continuación una respuesta del sistema como ésta:

```
[1] 829
```

El "829" es el número PID otorgado al proceso en ejecución, el "[1]" indica el número de tarea otorgado por el intérprete de comandos *bash* a la tarea que está ejecutando en *background*. El comando ha quedado ejecutando enviando las "y"es a "/dev/null" en donde

se pierden sin remedio. A pesar de que el proceso aún no finaliza nos ha devuelto el *prompt* para seguir colocando comandos (es decir, quedó corriendo en *background*)

Vuelva a la sesión del usuario "adios" y verá corriendo el proceso "yes" más o menos entre los primeros lugares.

Observe que el orden NO es permanente ¿Puede explicar por qué?

Vuelva a la sesión del usuario "root" y coloque ahora el comando

nice -n -15 yes > /dev/null &

Es decir, el mismo comando que recién pero ahora "mejorado" con una prioridad "-15". El sistema le contestará por ejemplo:

[2] 830

El significado es similar al primer caso. **Vuelva** a la sesión del usuario "adios" y observe ahora el comportamiento de "top". **Observe y compare** los valores de las columnas PID, PRI, NI, %CPU, STAT.

¿Qué conclusiones saca? Tome notas de todo lo observado

Vuelva ahora a la sesión del usuario "root" y coloque el comando:

kill %2

Apretando la tecla "enter" dos veces, el sistema le comunicará que el proceso ha sido terminado. Puede recuperar los anteriores comandos con las flechas de dirección hacia arriba (o hacia abajo) y modificar su contenido con las flechas de dirección a izquierda y derecha.

Experimente colocando distintos valores de "nice", en vez de "-15" pruebe con "15" por ejemplo. **Experimente** sin trabajar con el superusuario "root", sólo con el usuario no-privilegiado "adios". ¿Pueden enviarse señales entre procesos con el mismo UID o es necesario que los envíe el superusuario "root" con UID=0 ?

Observe y compare los resultados

Autoevaluación

1.En un sistema con hilos, ¿hay una pila por hilo o una pila por proceso? Explique.
2.Los planificadores *round robin* normalmente mantienen una lista de todos los procesos ejecutables, y cada proceso aparece una y sólo una vez en la lista. ¿Qué sucedería si un proceso ocurriera dos veces en la lista? ¿Puede usted pensar en alguna razón para permitir esto?
3.Describa las diferencias entre la planificación a corto plazo, mediano plazo y largo plazo.
4.Cite dos ventajas que tienen los hilos sobre los procesos múltiples. ¿Qué desventaja importante tienen? Sugiera una aplicación que se beneficiaría del uso de hilos, y una que no se beneficiaría.
5.¿Qué diferencias hay entre los hilos en el nivel de usuario y los apoyados por el núcleo? ¿En qué circunstancias es un tipo "mejor" que el otro?
6.Defina la diferencia entre planificación expropiativa y no expropiativa. Explique por qué es poco probable que se use una planificación no expropiativa estricta en un sistema.