# CONCEPTOS DE SISTEMAS OPERATIVOS

**P R I M E R A  E D I C I Ó N**

*( corregida... herrar es umano... )*


**Pablo Bossi**
**Rodrigo Castro**
**Luján Del Río**
**Ariel Eidelstein**
**Gustavo González**
**Pablo Hidalgo**
**Ernesto Muñóz**
**Juan Pablo Proazzi**
**Gustavo Schmidt**
**Augusto Vega**
**Pablo Wolfus**

# INDICE

# PREFACIO

El presente trabajo fue realizado por un grupo de alumnos de la materia *Sistemas Operativos*, Facultad de Ingeniería (U.B.A.), con el único objetivo de proporcionar una guía clara, de fácil lectura, y completa sobre una gran variedad de temas del campo de la tecnología de los sistemas operativos. Comenzó como una necesidad en la etapa preparatoria de exámenes finales, y culminó como una interesante recopilación, estructurada en una dinámica "pregunta-respuesta", de manera tal que la lectura y comprensión de los temas sea lo más rápida y directa posible. Si bien el presente material es lo suficientemente completo, requiere de una base previa de conocimientos, y además este trabajo no pretende ser reemplazo de los libros de texto, sino simplemente una guía. Recomendamos entonces que, para una total comprensión de la materia, la presente guía sea leída acompañada de por lo menos algún libro sobre conceptos básicos de sistemas operativos.

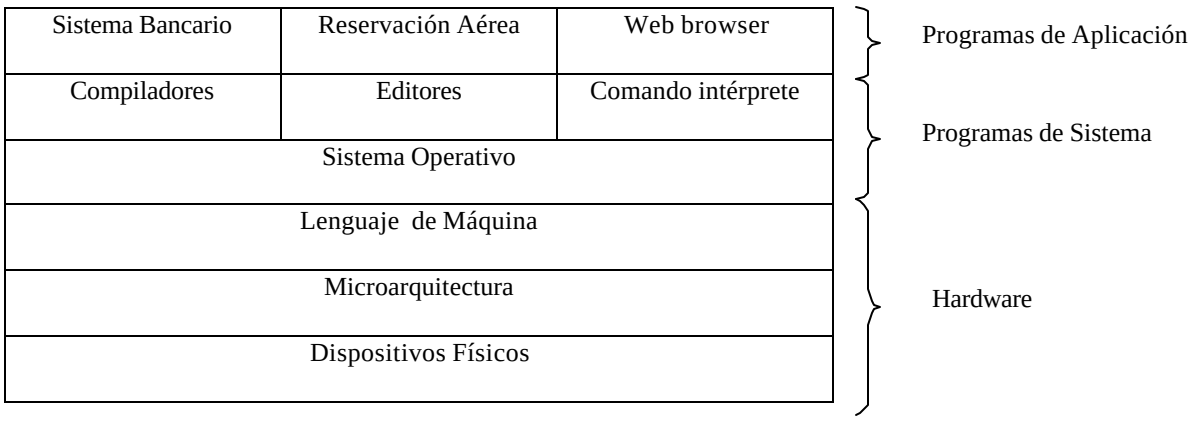
Deseamos que este trabajo sea lo más útil posible al interesado lector, y esperamos poder continuar enriqueciéndola con nuevos aportes de los autores presentes, de futuros alumnos de la materia *Sistemas Operativos*, o de quien se interese en la misma. No podemos cerrar este prefacio sin el explícito agradecimiento al Profesor Lic. Ing. Osvaldo Clúa, quien realizó un gran aporte para que este sueño se hiciera realidad.

*Los autores*
*Invierno de 2002*

# HISTORIA Y COMPARACIONES

**1) ¿En qué conceptos se basa el modelo de máquina multinivel?**

Un actual sistema de computadora consiste en uno o más procesadores, memoria, discos, impresoras, teclado, pantalla, etc. Escribir programas que mantengan un orden en el uso de estos dispositivos de manera de poder usarlos en forma correcta, es una tarea difícil. Por esta razón las computadoras están equipadas con un sistema operativo, cuyo trabajo es manejar todos estos dispositivos y permitirle al usuario una interfaz más simple que el hardware.
En el siguiente diagrama se representa un posible modelo de máquina multinivel, en él puede verse además el lugar que ocupa el sistema operativo.




**2) Describa las características de cada nivel, los elementos de interés y las actividades de quien diseña sobre ellos.**

Debajo de todo se encuentra el hardware, el cual está compuesto por tres niveles. El primero contiene dispositivos físicos, como circuitos integrados, chips, cables, suministro de electricidad, tubos de rayos catódicos y dispositivos similares. El siguiente es el nivel de microarquitectura, en el cual los dispositivos físicos se agrupan para formar unidades funcionales. Este nivel contiene algunos registros internos de la CPU y un data path que contiene una unidad aritmética lógica. En algunas máquinas, la operación de el data path es controlada por software, el cual recibe el nombre de microprograma. En otras máquinas, éste es controlado por circuitos de hardware. El propósito de el data path es ejecutar conjuntos de instrucciones, las cuales quizás usan registros u otras facilidades del hardware.
El tercer y último nivel que constituye el hardware recibe el nombre de lenguaje de máquina, el cual típicamente tiene entre 50 y 300 instrucciones, la mayoría para mover datos a través de la máquina, para hacer operaciones aritméticas y también para comparar valores. En este nivel los dispositivos de entrada salida se manejan cargando valores dentro de registros especiales para esos dispositivos.
Para ocultar la complejidad que presenta el hardware, existe el sistema operativo. El cual además de ocultar al hardware, le entrega al programador un conjunto de instrucciones más convenientes para poder trabajar con él.
Arriba del sistema operativo están el resto de programas del sistema, aquí encontramos el shell, sistemas de ventanas, compiladores, editores y similares aplicaciones independientes. Es importante darse cuenta que estos programas no son parte del sistema operativo.
Por último y para finalizar, en el nivel superior están los programas de aplicación, los cuales son escritos por los usuarios con la finalidad de resolver problemas particulares.

**3) Defina brevemente un Sistema Monotarea, un Sistema Batch, un Sistema de Time Sharing, una Workstation, un Sistema Personal, un Sistema Distribuido habilitado por Web y un Sistema Distribuido Orientado a Objetos.**

Sistema Monotarea

Como su nombre lo indica es un sistema capaz de realizar una única tarea por vez, por lo que el tiempo de desperdicio de los recursos es demasiado grande comparado con el de los sistemas de multiprogramación.

Sistema Batch
La característica principal de este tipo de sistemas es que permite agrupar distintos trabajos a procesar en forma consecutiva, para que así la máquina los pueda ejecutar en forma secuencial. Su traducción al castellano sería algo así como sistema de procesamiento por lotes.

Sistema de Time Sharing
En castellano sería sistema de tiempo compartido. Permite que más de un usuario tenga acceso a la misma máquina. Esto se debe a que la interacción del usuario con la CPU es bastante pequeña en tiempo, por eso mientras un usuario A está pensando, otro usuario B está procesando un conjunto de datos X.

Workstation
Es una máquina personal de gran tamaño como una Sun, Apollo o Vaxstation, cuyo hardware es más poderoso, rápido y complejo que las computadoras personales comunes.

Sistema Personal
Al reducirse los costos de hardware, cada vez de se ha hecho más factible contar con un sistema de computación para un solo usuario, a este tipo de computadoras se las conoce comúnmente como computadoras personales.

Sistema Distribuido habilitado por Web
Los usuarios están enterados de la multiplicidad de máquinas y para el acceso a estos recursos necesitan conectarse a la máquina remota apropiada o transferir datos de la máquina remota a la propia.

Sistema Distribuido Orientado a Objetos
La más reciente innovación en el diseño de un sistema operativo es el uso de tecnologías orientadas a objetos. Una estructura como esta basada en objetos permite a los programadores construir a gusto un sistema operativo sin desestabilizar la integridad del sistema. Una orientación a objetos también facilita el desarrollo de sistemas operativos distribuidos.

**4) Para cada elementos de los anteriores, indique cual de los elementos actuales de un Sistema Operativo aparece y como evoluciona a través de ellos.**

Sistemas Operativos para Mainframes
Los sistemas operativos para Mainframes están fuertemente orientados hacia el procesamiento de muchos trabajos al mismo tiempo, muchos de los cuales necesitan grandes cantidades de E/S.
En la actualidad un sistema de estas características es el batch, en el cual procesa en forma continua y secuencial un gran número de trabajos sin interacción de usuario, como por ejemplo hacer un reporte de ventas de una cadena de almacenes o tiendas.
Otro ejemplo es un sistema de timesharing, el cual permite a una gran cantidad de usuarios a correr trabajos en una computadora al mismo tiempo, como por ejemplo consultas a una gran base de datos.

Sistemas Operativos para un Servidor
Este tipo de sistemas corren en servidores que pueden ser grandes pc, workstatios, o mainframes. Ellos prestan servicio a múltiples usuarios a la vez a través de una red y le permite a los mismos, compartir recursos de hardware y software. Los servidores pueden proveer servicios de impresión, de archivos, o de Web. Los proveedores de internet ponen en funcionamiento muchas máquinas para soportar a sus clientes y a los sitios Web que los mismos solicitan.

Sistemas Operativos para una Computadora Personal
En esta categoría el sistema operativo tiene como tarea principal proveer una buena interfaz a un único usuario. Ellos son usados ampliamente para procesadores de textos, planillas de cálculo y acceso a internet

Sistema Operativos de Red

La principal función de un sistema operativo de red es ofrecer un mecanismo para transferir archivos de una máquina a otra. En este entorno, cada instalación mantiene su propio sistema de archivos local y si un usuario de instalación A quiere acceder a un archivo en la instalación B, hay que copiar explícitamente el archivo de una instalación a otra.
La idea básica de la Web, es hacer un sistema distribuido que parezca una gran colección de documentos enlazados. Una segunda aproximación es hacer un sistema distribuido que parezca un gran file system.
Usar un modelo de file system para un sistema distribuido significa que hay un único file system, con múltiples usuarios, de todas partes del mundo, capaces de leer y escribir archivos para los cuales tengan autorización.

Sistemas Operativos Orientados a Objetos
Windows NT y otros sistemas operativos recientes confían plenamente en los principios del diseño orientado a objetos. El cual tiene como concepto principal el objeto. Un objeto es una unidad de software que contiene una colección de datos y procedimientos. Generalmente esta colección de datos y procedimientos no son visibles fuera del objeto. Para ello hay definido interacciones que permiten a otro software tener acceso a los datos y a los procedimientos.

**5) ¿Cuáles son las tareas de administración de un Sistema Operativo? ¿Qué objetivos guían esta visión?**

El sistema operativo básicamente desempeña dos funciones que no presentan relación alguna entre si y que son, extensión de la máquina y administración de recursos.
El sistema operativo como una extensión de la máquina es una definición que surge debido a la arquitectura que presentan la gran mayoría de éstas, en el nivel de lenguaje de máquina. Esta arquitectura es primitiva y complicada de programar. El sistema operativo le oculta el hardware al usuario y le presenta un punto de vista mucho más simple y entendible para leer y escribir archivos, y realizar distintas operaciones con ellos. También se encarga de esconder un montón de operaciones desagradables que tienen que ver con las interrupciones, la administración de memoria, etc. Desde este punto de vista el sistema operativo se presenta al usuario como el equivalente de una máquina extendida que es más fácil de programar que el subyacente hardware.
Una alternativa a esta definición, es pensar al sistema operativo como un administrador de recursos. Considerar que su trabajo es proveer una asignación, en forma ordenada y controlada, de los procesadores, memorias, y dispositivos de E/S entre todos los programas que compitan por ellos. Este punto de vista del sistema operativo nos muestra que su principal tarea es mantener conocimiento de quien está usando que proceso, para otorgar respuestas a los pedidos de recursos, y para mediar entre conflictos que puedan surgir entre diferentes programas y usuarios por los recursos pedidos.

**6) Ensaye una justificación de la división en períodos de la historia de la Arquitectura de Computadoras como lo hace Tanenbaum en 1.2 o como se hizo en clase.**

Períodos de la computación:

- Primera Generación 1945-1954. Tubos de Vacío y Plugboards.
- Segunda Generación 1955-1965. Transistores y Sistemas Batch.
- Tercera Generación 1966-1980. Circuitos Integrados y Multiprogramación.
- Cuarta Generación 1981-presente. Computadoras Personales.

**7) ¿Qué grandes lineamientos de diseño aparecen en cada una de estas etapas?**

Primera Generación:

- Desarrollada durante la guerra.
- Tubos de vacío.
- Mucha gente desarrolló máquinas de calcular automáticas.
- Las máquinas ocupaban gran espacio, llenaban habitaciones enteras.
- No había sistemas operativos.
- No había lenguajes de programación.
- Los usuarios alquilaban las computadoras por tiempo. Mayormente cálculos científicos y matemáticos.

Segunda Generación:

- Aparecen los transistores.
- Las máquinas se vuelven más confiables.
- Más factible para los productores vender computadoras a clientes.
- Se originó el trabajo de operador de computadora.
- Aparecieron los trabajos batch, que se introducían en las máquinas, vía tape para mejorar el rendimiento.
- Surge el primer sistema operativo.

Tercera Generación:

- Los circuitos integrados reemplazan a los transistores.
- Sistemas Mainframe.
- IBM crea System/360 corriendo como sistema operativo el OS/360.
- El comienzo de la multiprogramación, la cual apunta hacia la necesidad de un sistema operativo más complejo.
- Spooling. El código de un job puede leerse desde disco y cargarse a memoria cuando se necesite. Ambos input y output.
- Se introduce el concepto de time sharing el cual permite trabajo interactivo.
- Los sistemas operativos comienzan a volverse más complejos debido a que tiene que tratar con todos estos conceptos nuevos.

Cuarta Generación:

- Integración a gran escala (LSI).
- Gran desarrollo de la PC (personal computer).
- Uno de los requerimientos para la original IBM PC fue un sistema operativo - Bill Gates provee el MS-DOS
- Comenzaron a utilizarse en procesadores no-Intel, el sistema operativo UNIX.

**8) ¿Cómo se organizaba el trabajo del Centro de Cómputos en la época de los procesadores periféricos y del Block Time?**

Durante el primer período de la historia de la computación después de las primeras máquinas mecánicas, surgieron aquellas que se constituían por tubos de vacío. Estas máquinas eran enormes, tanto que llenaban habitaciones enteras con decenas de miles de tubos de vacío, pero aún así eran millones de veces más lentas que las máquinas personales de hoy en día.
En esos días, un único grupo de personas diseñaban, construían, programaban, operaban y mantenían cada máquina. Toda la programación era hecha absolutamente en lenguaje de máquina, operando con el cableado de las plugboards, para controlar las funciones básicas de la máquina. Los lenguajes de máquina eran desconocidos y ni hablar de un sistema operativo. El modo usual de operación para el programador era firmar por un bloque de tiempo, en una especie de pizarrón que había en la pared, y entonces iba a la sala de computadoras, insertaba su plugboard dentro de la computadora y pasaba las pocas horas que tenía esperando que ninguno de los 20.000 tubos de vacío, o más, no se quemase durante la ejecución del programa. Generalmente los principales problemas eran los cálculos matemáticos de senos, cosenos y logaritmos.

**9) ¿Cómo se organizaba el trabajo en la época de un Mainframe de procesos Batch?**

La aparición de los transistores a mediados de los 50 produjo un cambio radical a la hora de la construcción de máquinas. Estas se volvieron más confiables, por eso pudieron ser construidas esperando que alguien pague por ellas. Pero solamente grandes corporaciones, como agencias del gobierno o universidades, podían afrontar un costo de millones de dólares.
Estas máquinas, llamadas mainframes, se encontraban encerradas en habitaciones especiales con aire acondicionado, y eran manejadas por un staff de operadores profesionales. Para hacer correr un job, el programador debía en primera instancia escribir el programa en papel ( en FORTRAN o en assembler ), después pasarlo a tarjetas especiales que se usaban como input. Una vez hecho esto se dirigía a la sala de input y entregaba a uno de los operadores de la máquina, luego se iba y no volvía hasta que la máquina terminase su tarea.

Cuando la computadora terminaba cualquier trabajo que estuviese corriendo, un operador iba hacia la impresora, retiraba los resultados obtenidos y los llevaba hacia la sala de output, para que el programador pueda retirarlos. Como podrá apreciarse, se perdía mucho tiempo en el ida y vuelta de los operadores.
La solución adoptada fue el sistema de procesamiento por lotes o sistema batch. La idea consistía en juntar una pila llena de jobs, en la sala de input y pasarlos a una cinta magnética a través de una máquina pequeña y barata, como la IBM 1401. Mientras que luego una máquina mucho más cara, como la IBM 7094, realizase los cómputos.
Después de juntar los jobs en un tape, este se rebobinaba y se llevaba a la sala de máquina, donde se montaba en un tape drive. El operador entonces cargaba un programa especial (el ancestro del sistema operativo), el cual leía los jobs en forma secuencial y los corría. Cada vez que un job terminaba de ejecutarse, el sistema operativo automáticamente leía el próximo job del tape. Las salidas de cada job eran almacenadas en otro tape, el cual una vez que se terminaba con todos los jobs, se llevaba a la impresora.

**10) ¿Qué papel juega Java en estas últimas etapas de la evolución?**

Los sistemas operativos más pequeños corren sobre tarjetas inteligentes (smart cards), que contienen su CPU en un pequeño chip. Estos sistemas tienen un gran poder de procesamiento como también restricciones de memoria. Algunos de ellos pueden manejar solamente una función, pero otros pueden manejar múltiples funciones en la misma tarjeta.
Algunas de estas tarjetas son orientadas a Java, esto significa que la ROM de la tarjeta contiene un intérprete de applets ( programas Java ), que pueden ser bajados y corridos en estos sistemas. Muchas de estas tarjetas pueden manejar más de uno de estos applets al mismo tiempo, por lo que la administración y protección de recursos también se convierten en un punto importante cuando dos o más applets están presentes al mismo tiempo.

**11) Defina un Sistema de Información Multimedial y dé algunos ejemplos de aplicaciones.**

El término ‘multimedia’ hace referencia a la capacidad de presentar simultáneamente tipos variados de información, como parte de un diseño común.
Películas digitales, video clips, y temas musicales se están convirtiendo rápidamente en una manera entretenida de presentar información a través de la computadora. Los archivos de audio y video pueden ser almacenados en disco y reproducidos en cualquier momento. Sin embargo sus características son muy diferentes a los tradicionales archivos de textos que fueron diseñados por los actuales file systems. Como consecuencia de esto, nuevos file systems son necesarios para manejar este tipo de archivos. Más aún todavía, almacenar y reproducir audio y video exige nuevas demandas al scheduler y a otras partes del sistema operativo.
Los archivos multimedia consisten en múltiples procesamientos en paralelo, generalmente un video, un audio y a veces también un subtítulo. Todo esto debe ser sincronizado durante la reproducción. Por eso es necesario un sistema operativo capaz de manejar este tipo de archivos.

# PROCESOS

**1) Describa las diferencias entre *scheduling*, *dispatching* y *conmutación de contexto*.**

En un sistema con *multiprogramación*, ***scheduling*** consiste en elegir un proceso entre varios en estado *ready*, para otorgarle tiempo de CPU, y que pueda realizar sus tareas. Para ello se aplica un algoritmo (***scheduling algorithm***), y el mecanismo que lo lleva a cabo se conoce como ***scheduler*** (Modern Operating Systems; A. Tanenbaum; pág. 132).
El *scheduler* tiene en cuenta los siguientes items:

- Cantidad requerida de recursos.
- Cantidad actualmente disponible de recursos.
- Prioridad del trabajo o proceso.
- La cantidad de tiempo de espera.

***Dispatching*** consiste en darle tiempo de CPU al proceso seleccionado por el *scheduler*, lo cual implica las siguientes operaciones:

- Conmutar el contexto (*context switching*).
- Cambiar a modo usuario (*user mode*).
- Saltar a la posición apropiada en el programa de usuario para ejecutar ese programa.

El tiempo es dividido en pequeños segmentos denominados *time slices* (que son variables en casi todos los sistemas). Cuando el *time slice* termina, el *dispatcher* le permite al *scheduler* actualizar el estado de cada proceso, entonces selecciona el siguiente proceso a ejecutar.

***Context switching*** (conmutación de contexto) consiste en almacenar el estado de un proceso (cuando este sale del estado *running*), para poder luego reanudar dicho proceso (cuando vuelva al estado *running*). Luego de almacenar el estado (*contexto*), el scheduler le da CPU a otro proceso. El estado o contexto de un proceso consiste de la siguiente información: registros, mapas de memoria, cache de memoria, etc. El tiempo que dura el *context switch* es desperdiciado (*overhead*) ya que el sistema no puede aprovecharlo para hacer otra cosa.

**2) Describa paso a paso el proceso de *conmutación de contexto*. Indique el rol de las interrupciones y en qué momentos el Sistema debe estar en modo protegido.**

El proceso de *context switching* (conmutación de contexto) consta de los siguientes pasos:

1. Se almacena el contexto del procesador, incluyendo el contador de programa (*program counter*) y otros registros.
2. Se actualiza el *bloque de control del proceso* (PCB) que está actualmente en el estado *running*. Esto implica cambiar el estado del proceso a alguno de los otros estados (*ready*, *blocked*, etc.).
3. Se mueve el PCB de este proceso a la cola apropiada (*ready*, *blocked*, etc.).
4. Se selecciona otro proceso para ejecución (esto implica llamar al *scheduler*).
5. Se actualiza el PCB del proceso seleccionado. Esto implica cambiar el estado de este proceso a *running*.
6. Se actualizan estructuras de datos para administración de la memoria.
7. Se restaura el contexto del procesador que existía en el momento en que el proceso seleccionado fue sacado del estado *running* por última vez. Esto implica cargar los valores del contador de programa y otros registros.

Aclaración: los pasos anteriores fueron extraídos del libro de Stallings. Dicho autor llama al conjunto de pasos anteriores con el nombre de *process switching*, y lo diferencia del *context switching*. Sin embargo, Tanenbaum y otros autores no hacen diferencia alguna, y tanto *process switching* como *context switching* son la misma cosa.

Dentro del ciclo de instrucción (*instruction cycle*) se encuentra el ciclo de interrupción (*interrupt cycle*), en el cual el procesador chequea si ocurrió alguna interrupción. Si no hay interrupciones pendientes, el procesador carga la siguiente instrucción del proceso actual. Si una interrupción está pendiente, el procesador:

1. Guarda el contexto del programa que está siendo ejecutado.
2. Setéa el contador de programa (*program counter*) a la dirección donde comienza un programa de manejo de interrupciones (*interrupt handler*).

El procesador ahora carga la primer instrucción del programa de manejo de interrupciones, el cual atenderá la interrupción. Una interrupción podría estar seguida de un cambio del proceso en ejecución, aunque también podría suceder que, luego de la interrupción, vuelva a reanudarse el mismo proceso que venía ejecutándose (Stallings; pág. 129-130).

**3) Indique qué información debe usar el Sistema Operativo para administrar procesos. ¿Cuáles de estos datos pueden estar en el área de usuario y cuáles en área del S.O.?**

Cada proceso es representado en el sistema operativo por su propio *Process Control Block* (PCB), el cual es un registro que contiene la siguiente información:

- El estado del proceso (*process state*) podría ser *new*, *ready*, *running*, *idle*, o *halted*.
- El contador de programa (*program counter*) indica la dirección de la siguiente instrucción a ser ejecutada por este proceso.
- Los registros de la CPU varían en cantidad y tipo, dependiendo de la arquitectura. Incluyen acumuladores, registros índices, y registros de propósito general. Junto con el contador de programa, esta información de estado debe ser almacenada cuando ocurre una interrupción.
- Información de manejo de memoria.
- Información de estado de I/O.
- Información de *scheduling* de CPU, incluyendo la prioridad del proceso, punteros a las colas de *scheduling* y otros parámetros de *scheduling*.

En UNIX, un proceso debe contener la siguientes información:

*Contexto a nivel de usuario*
- Texto del proceso: instrucciones de máquina ejecutables del proceso.
- Datos del proceso: datos accesibles por el programa.
- Pila del usuario: parámetros, variables locales y punteros para funciones ejecutándose en modo usuario.
- Memoria compartida: memoria compartida por otros procesos.

*Contexto de registro*
- Contador de programa: dirección de la siguiente instrucción a ser ejecutada.
- Registro de estado del procesador.
- Puntero a la pila: apunta a la cima de la pila del kernel o del usuario.
- Registros de propósito general.

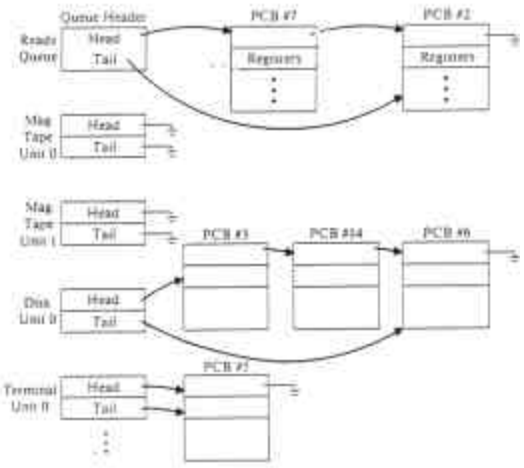
*Contexto a nivel de sistema*
- Entrada en la tabla de procesos: define el estado de un proceso.
- Área U: información de control del proceso.
- Tabla de región de preproceso: define el mapéo de direcciones virtuales a físicas.
- Pila del kernel.

(Stallings; pág. 143)

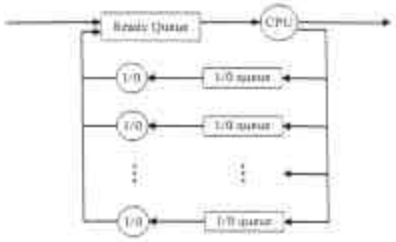
**4) ¿Cómo encadena el S.O. los procesos para permitir su despacho?**

Los procesos que están en estado *ready* y esperando a ser ejecutados son mantenidos en una lista llamada *ready queue*. El encabezado de la lista contendrá punteros al primer y último PCB en la lista. La *ready queue* podría ser implementada como una cola FIFO, una cola de prioridad, un árbol, una pila, o simplemente una lista desordenada. Sin embargo,

conceptualmente todos los procesos en la *ready queue* están listos y esperando por una chance para usar la CPU. La siguiente figura muestra una representación de una *ready queue* y varias *device queue*:




Si un proceso que se está ejecutando debe esperar que se complete una operación de I/O, entonces es colocado en otra cola, llamada *device queue*. Cada dispositivo tiene su propia *device queue*.




**5) Explique paso a paso el proceso de *dispatching*.**

El *dispatcher* le da control de la CPU al proceso seleccionado por el *scheduler*; esto implica las siguientes operaciones:

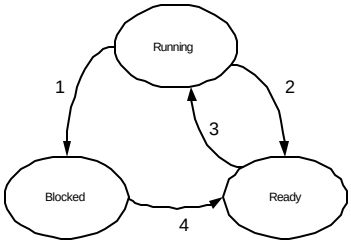
- Conmutar el contexto (*context switching*).
- Cambiar a modo usuario (*user mode*).
- Saltar a la posición apropiada en el programa de usuario para ejecutar ese programa.

**6) ¿Cómo nace un proceso en un sistema operativo *batch* y en uno de *time sharing* como UNIX?**

En un sistema *batch* los usuarios le presentan al sistema los trabajos a realizar (*jobs*). Cuando el sistema operativo decide que dispone de los recursos para ejecutar un trabajo, crea un nuevo proceso y ejecuta el trabajo (que se toma de la cola de trabajos). A veces los procesos se crean automáticamente al iniciarse el S.O., con el nombre de "particiones". Es importante recalcar que en un sistema *batch* perfectamente puede haber *multriprogramación* (es un error clásico relacionar *batch* con *monoprogramación*).

En un sistema *time sharing*, cada usuario de cada terminal solicita la ejecución de programas (tipeando un comando o *clickeando* un icono), con lo cual se ejecuta un nuevo proceso.
En líneas generales, un sistema *batch* no presenta interacción con el operador, mientras que un sistema *time sharing* sí la tiene.

**7) Describa un diagrama de estados de procesos e indique cómo se operan las transiciones entre ellos.**




1.- El proceso no puede seguir ejecutándose porque, por ejemplo, está esperando por datos de entrada que aún no fueron ingresados, y pasa al estado *blocked*.
2.- El sistema operativo decide dar CPU a otro proceso. Por lo tanto, el proceso que se está ejecutando en ese instante pasa al estado *ready* (esto significa que no se ejecuta, pero está listo para cuando el SO le vuelva a dar tiempo de CPU).
3.- Todos los procesos han tenido tiempo de CPU, y por lo tanto el *scheduler* le da tiempo de procesamiento al primer proceso nuevamente (de todos los que están en estado *ready*).
4.- Se produce un evento externo que el proceso bloqueado estaba esperando (por ejemplo, se dio entrada a la información que el proceso necesitaba para ejecutarse). Además, si no hay ningún proceso ejecutándose en ese instante, entonces se disparará automáticamente la transición 3.

**8) ¿Qué objetivos de diseño puede tener el *scheduler*?**

Los objetivos de diseño dependen de cada caso, como se muestra en la lista siguiente:

En todos los sistemas:
*Equidad*: dar a cada proceso equidad en cuanto al uso de la CPU (ningún proceso debería aplazarse en forma indefinida).
*Cumplimiento de una política*: controlar que la política establecida sea llevada a cabo.
*Balance*: mantener ocupadas todas las partes del sistema. Debe favorecerse a aquellos procesos que requieren recursos poco utilizados.

Sistemas batch:
*Rendimiento*: maximizar los trabajos (*jobs*) por hora.
*Tiempo de retorno (turnaround time)*: minimizar el tiempo promedio entre que el trabajo comienza hasta que es completado.
*Utilización de la CPU*: mantener la CPU ocupada todo el tiempo.

Sistemas interactivos:
*Tiempo de respuesta*: responder rápidamente a los requerimientos (minimizar el tiempo entre que se ejecuta un comando y se obtiene un resultado).
*Proporcionalidad*: cumplir las expectativas de los usuarios. Está relacionado con la idea (no siempre correcta) que tienen los usuarios de que *"las cosas que parecen más complejas requieren más tiempo, y cosas que parecen más simples deben requerir poco tiempo"*. Así, el scheduler se debe ajustar a esta premisa.

Sistemas de tiempo real:
*Cumplir con los plazos:* evitar la pérdida de información.
*Ser predecible:* una tarea debe ejecutarse aproximadamente en el mismo tiempo y casi al mismo costo sea cual sea la carga del sistema.

**9) Usando los simuladores, obtenga un diagrama de Gannt de a) *Shortest Job First* b) *First Come First Served* c) *Round Robin* d) *Round Robin con prioridades*.**

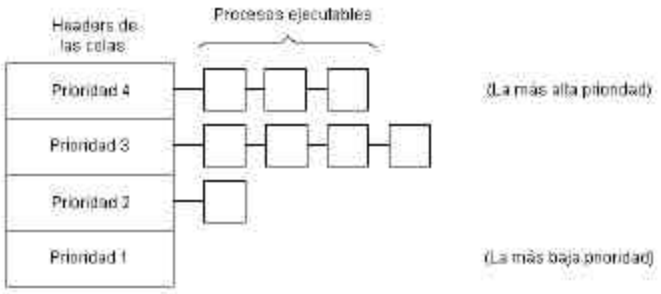
No desarrollada (implica el uso de simuladores, que deben bajarse de Internet).

**10) Para el ejercicio 9), explique y calcule valores que se relacionen con los objetivos de 8) (Ej: tiempo medio de la tarea en el sistema, para el objetivo de maximizar tiempo de respuesta)**.

No desarrollada (implica el uso de simuladores, que deben bajarse de Internet).

**11) Describa el funcionamiento de un *scheduler de colas múltiples* ¿Puede simularlo?**

En el modelo de scheduler de *colas múltiples*, existen "clases de prioridad". Los procesos que están en la primer clase de prioridad son ejecutados por un *quantum*. Los que están en la segunda clase de prioridad se ejecutan por dos *quantums*, los que están en la tercer clase de prioridad se ejecutan por cuatro *quantums*, y así sucesivamente en potencias de dos. Cuando un proceso consume todos los *quantums* que le fueron cedidos, entonces se lo mueve a la clase siguiente. Además, a medida que el proceso se "hunde" más profundamente en la jerarquía de clases de prioridad, entonces será ejecutado menos frecuentemente, ahorrando CPU para procesos interactivos cortos (Modern Operating Systems; A. Tanenbaum; pág. 144).




**12) ¿Qué son las *threads*? Describa sus variantes de implementación.**

Un ***thread*** es similar a un proceso (en cuanto que comparten varias características). Cada uno contiene un *program counter*, registros, un *stack*, y se pueden ver como "entidades" que se ejecutan en la CPU. Un proceso puede estar compuesto por uno o más *threads* (en este último caso hablamos de *multithreading*). Los *threads* permiten ejecuciones múltiples dentro del mismo proceso, con un alto grado de independencia unos de otros. Dentro del mismo ambiente de proceso, los *threads* comparten el espacio de direcciones de memoria (*address space*), archivos abiertos, y otros recursos.
Los *threads* pueden implementarse en el espacio de usuario (*user space*) o en el *kernel*.

Implementación de *threads* en el espacio de usuario:
El paquete de threads se coloca completamente en el espacio de usuario (el kernel no sabe nada acerca de su existencia). La principal ventaja es que un paquete de threads de usuario pueden ejecutarse en sistemas operativos que no soportan threads. Cada proceso a nivel usuario cuenta con un sistema de tiempo de ejecución (*run-time system*), que maneja los threads, y por consiguiente cada proceso necesita su propia tabla de threads (*threads table*). El manejo que hace el sistema de tiempo de ejecución con los threads a nivel usuario, es similar al manejo hecho por el *scheduler* con los procesos a nivel kernel. El cambio de threads hecho a nivel usuario es algunos órdenes de magnitud más rápido que el hecho a nivel kernel, ya que entre otras cosas, no se requiere *conmutación de contexto* (*context switching*). Otra ventaja consiste en que cada proceso puede tener su propio algoritmo de *scheduling*.
Uno de los problemas que presenta esta opción es respecto al bloquéo de *system calls*. Dejar que los threads ejecuten los *system calls* es inaceptable, ya que podrían parar a todos los threads. La solución consiste en agregar código a los *system calls* que determinen si dicha llamada se bloqueará (en cuyo caso no se ejecuta). Otro problema tiene que ver con los *page faults*. Si el programa necesita una instrucción que no está en memoria, el sistema operativo deberá obtener dicha instrucción desde disco. El proceso se bloquea completamente hasta que termine la operación de I/O, aunque haya algún thread en estado *ready* (ya que el kernel no conoce sobre la existencia de los threads). Otro problema consiste en que un thread que está ejecutándose podría no ceder nunca la CPU a otro thread que no lo está, ya que no existe ningún reloj que se encargue de esta tarea. Es el propio thread quien debe entrar al sistema de tiempo de ejecución (*run-time system*). El problema tal vez más importante es que los programadores generalmente necesitan threads en aplicaciones donde los estos se bloquéan muy frecuentemente (por ejemplo, en un servidor web con múltiples threads).

Implementación de *threads* en el *kernel*:
En este caso no se necesita el sistema de tiempo de ejecución (*run-time system*) ni la tabla de threads en cada proceso. El kernel cuenta con su propia tabla de threads, en donde mantiene la información de todos los threads del sistema. Cuando un *system call* bloquéa un thread, el kernel puede ejecutar otro thread del mismo proceso o de otro proceso. Debido al alto costo que presenta la creación y destrucción de threads a nivel de kernel, es posible que estos se reciclen. Cuando un thread es destruido, se lo marca como "no ejecutable", pero todas sus estructuras son mantenidas. Luego, cuando debe ser creado un nuevo thread, se reactiva uno de estos threads, reduciendo el costo. Por otra parte, si se produce un *page fault*, el kernel puede cambiar la ejecución a otro thread, mientras el primero espera que se complete la operación de I/O.
La principal desventaja de esta implementación es que el costo de los *system calls* es elevado.

Implementación híbrida de *threads*:
En este caso, el kernel solo conoce y maneja los threads a su nivel, algunos de los cuales podrían tener múltiples threads a nivel de usuario. Estos threads a nivel de usuario son manejados de la misma forma que en la implementación de threads en el espacio de usuario.
(Modern Operating Systems; A. Tanenbaum; pág. 90-94)

**13) ¿Qué campos de los mencionados en 3 pasan al *Thread Control Block*?**

En el *Thread Control Block* (TCB) encontramos la siguiente información:

- Estado de ejecución: registros de la CPU, contador de programa (*program counter*), puntero al *stack*.
- Información de *scheduling*: estado (*new*, *ready*, *running*, *waiting*, o *terminated*), prioridad, tiempo de CPU usado.
- Varios punteros (para implementar colas de *scheduling*).

**14) ¿Qué opciones hay para la interacción del *scheduler* con las *threads*?**

Cuando varios procesos tienen múltiples *threads* cada uno, tenemos dos niveles de paralelismo: procesos y *threads*. El *scheduling* en ambos sistemas difiere sustancialmente, dependiendo si son soportados los *threads* a nivel de usuario o los *threads* a nivel de kernel (o ambos).
En el primer caso (*threads* a nivel de usuario), el kernel no tiene conocimiento de la existencia de los *threads*, y por tanto toma cada proceso y lo ejecuta en un *quantum* (intervalo de tiempo que se le asigna a un proceso para que se ejecute); por ejemplo, ejecuta el proceso *A*. El *scheduler* de *threads* dentro del proceso *A* decide que *thread* ejecutar, por ejemplo *A1*. Este *thread* se ejecutará tanto como él quiera (ya que los *threads* no son interrumpidos), hasta que se consuma el *quantum*, y el kernel decida tomar otro proceso para ejecutar. Cuando el proceso *A* vuelva a ser ejecutado, el *thread A1* continuará hasta volver a consumir todo el *quantum* del proceso, o hasta que termine su trabajo. Se debe notar que el *scheduler* del kernel no sabe (ni tampoco le interesa) qué está ocurriendo dentro del proceso *A*.
En el segundo caso (*threads* a nivel de kernel), el kernel toma un *thread* y lo ejecuta. No tiene en cuenta a qué proceso pertenece dicho *thread*. Al *thread* se le da un *quantum*, y es forzado a suspenderse en caso de exeder dicho intervalo de tiempo. El kernel sabe que cambiar entre dos *threads* pertenecientes a distintos procesos es más costoso que si los *threads* pertenecen al mismo proceso (porque se requiere de un *context switch* completo). Por lo tanto, puede tener en cuenta esta información para decidir qué *thread* ejecutar en un determinado momento.

**15) ¿Cómo se inserta el Modelo de Objetos en esta evolución?**

En esta evolución, los objetos pueden verse como unidades de agrupamiento. En un caso concreto (Windows NT), tanto los procesos como los *threads* son objetos. Cada proceso se define por una cierta cantidad de atributos y encapsula una cierta cantidad de acciones (o servicios). Un proceso en Windows NT debe contener al menos un *thread* para ejecutar. Ese *thread* podría ejecutar otros *threads*. Algunos de los atributos del *thread* son derivados del proceso.

Sin embargo, los objetos de Windows NT son solamente unidades de encapsulamiento, y no tienen nada que ver con los objetos de OOP. No hay herencia ni polimorfismo ni *frameworks* de integración (conjunto de clases cooperantes). En un ambiente orientado a objetos "nativo" (no hay comerciales, pero la *Java Virtual Machine* provee parte de dicho ambiente)

cada objeto "puede" poseer al menos un *thread* propio. Si no lo hace es por razones de performance. En este ambiente, los procesos se "diluyen" ya que cada objeto tiene "vida propia".

**16) Ensaye una justificación tecnológica del uso de multiprogramación y multithreading.**

La *multiprogramación* es una manera de ejecutar procesos en un sistema, por la cual cada proceso hace uso de la CPU por un determinado intervalo de tiempo. Si el sistema operativo tiene la habilidad de intercambiar rápidamente la ejecución de los procesos, entonces dará la sensación de ejecución simultánea o paralela de los mismos. Algunas ventajas y desventajas de esta técnica son:

*Ventajas*
- Puede mantener la CPU ocupada.
- Puede superponer la ejecución de tareas, incrementando el rendimiento.

*Desventajas*
- Aumenta la complejidad del sistema.
- Es potencialmente propensa a sufrir daños en la seguridad.

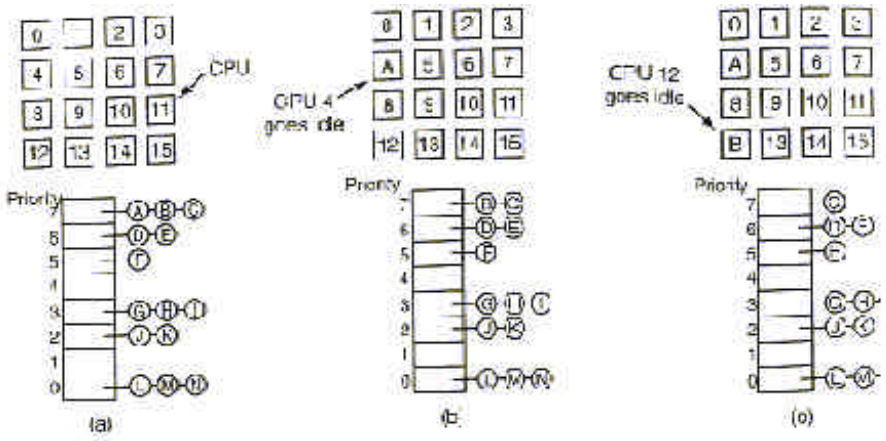
En una tarea *multithreading*, mientras un *thread* servidor está bloqueado y esperando, un segundo *thread* en la misma tarea puede ejecutarse. La cooperación de múltiples *threads* en el mismo trabajo brinda mayor rendimiento y mejora la performance. Además, el uso de *multithreading* permite separar las tareas minimizando los puntos de contacto ("cohesión" y "acople" dirían los seguidores de la programación estructurada). Por ejemplo, se maneja la *GUI* por un *thread* y el proceso por otro *thread*, sin "contaminar" la programación.

Aclaración: en esta pregunta, se pedía que se "ensaye" una justificación tecnológica del uso de *multiprogramación* y *multithreading*. Lo anterior son conceptos, con lo cual, el ensayo queda a criterio de cada alumno. Sin embargo, el Ing. Osvaldo Clúa me pidió que resaltara lo que está subrayado.

**17) ¿Cómo impacta el multiprocesamiento en las operaciones de *scheduling*, *dispatching* y *conmutación de contexto*?**

En multriprocesamiento, el *scheduling* tiene dos dimensiones: debe decidir qué proceso ejecutar y en qué CPU ejecutar dicho proceso. Veamos algunos casos:

Timesharing: en este caso, los procesos a ejecutar no mantienen ninguna relación entre ellos. El sistema mantiene un conjunto de listas de acuerdo a prioridades (es una lista de listas) con los procesos en estado *ready* (esta estructura es compartida por todas las CPU). Así, cada procesador que se vaya desocupando, tomará el proceso con mayor prioridad en ese conjunto de listas. Hacer el *scheduling* de esta forma es una opción razonable.



Space Sharing: en este caso los procesos están relacionados de alguna forma. Hablando a nivel de *threads*, todos aquellos que pertenecen a un mismo proceso están relacionados. El *scheduling* de múltiples *threads* al mismo tiempo a través de varias CPU es llamado *space sharing*. Cuando un conjunto de *threads* debe ser ejecutado, el *scheduler* chequéa si existen tantas CPU libres como *threads* a ejecutar. En caso afirmativo, a cada *thread* se le asigna su propia CPU (notar que no hay *multiprogramación* en cada CPU), y todos comienzan a ejecutarse. Si no hay suficientes CPU libres, entonces ningún *thread* comienza a ejecutarse. Cuando un *thread* termina de ejecutarse, la CPU es colocada en un pool de CPU disponibles.

Gang Scheduling: una clara ventaja de *Space Sharing* consiste en la eliminación de la *multiprogramación* (porque en una CPU se ejecuta un único proceso o thread), con lo cual se evita la carga provocada por el proceso de *conmutación de contexto*. Sin embargo, de igual forma, una clara desventaja de dicha técnica es el tiempo perdido cuando una CPU se bloquea y no tiene nada para hacer hasta que vuelva al estado *ready*. Así, se ha llegado a esta alternativa que junta las ventajas de *Timesharing* y *Space Sharing*, y que consta de tres partes:

- Los grupos de *threads* relacionados son tratados por el *scheduler* como una unidad (*gang*).
- Todos los miembros de un *gang* se ejecutan simultáneamente (en diferentes CPU cada miembro).
- Todos los miembros de un *gang* comienzan y terminan juntos sus intervalos de tiempo.

La idea de *gang scheduling* es tener todos los *threads* de un proceso ejecutándose juntos, así si uno de ellos envía un pedido a otro *thread*, este tendrá el mensaje inmediatamente y responderá inmediatamente. Además, el tiempo es discretizado en *quantums*, de manera tal que si un *thread* se bloquéa, su CPU estará en desuso hasta que termine el *quantum.*

**18) ¿Cómo es la técnica de programación *pseudoconversacional*, *transaccional* o usando *autómatas finitos* (son distintos nombres de la misma técnica)?**

Una *transacción conversacional* es aquella que implica más de una entrada desde la terminal, de manera tal que la transacción y el usuario establecen una especie de "conversación". Una *transacción no-conversacional* tiene solo una entrada (la que invoca a la transacción). Veamos un ejemplo de *transacción conversacional*:

**Programa**: Pedir DNI, esperar respuesta.
**Usuario**: Ingresar DNI.
**Programa**: Pedir password, esperar respuesta.
**Usuario**: Ingresar password.
**Programa**: Enviar menú, esperar respuesta.
**Usuario**: Enviar ítem de menú.
**Programa**: Enviar resultados, terminar.

El programa ha "saltado" de estado a estado. Estuvo ocupando recursos (sobre todo espacio) durante toda la "conversación", y si el usuario desaparece, el programa queda "colgado".

La *programación pseudoconversacional* se basa en el hecho de que, en una *transacción conversacional*, la cantidad de tiempo gastado procesando cada respuesta al usuario es extremadamente corto comparado con la cantidad de tiempo esperando la entrada del usuario. Una secuencia en una *transacción pseudoconversacional* contiene una serie de *transacciones no-conversacionales* que ven al usuario como una simple *transacción conversacional* con varias pantallas de entrada. Cada transacción en la secuencia maneja una entrada, responde, y termina.
Antes de que una *transacción pseudoconversacional* termine puede pasar datos a la siguiente transacción, iniciada desde la misma terminal. Veamos un ejemplo de *transacción pseudoconversacional*:

**Programa 1**: Pide DNI, y termina.
**Usuario**: Ingresa DNI.
**Programa 2**: Recibe DNI, pide password, envía DNI (que el usuario puede o no ver), y termina.
**Usuario**: Ingresa password.
**Programa 3**: Recibe DNI y password, envía menú, DNI y password (posiblemente oculto, hay problemas de seguridad pero es un ejemplo) y termina.
**Usuario**: Envía ítem de menú.
**Programa 4**: Recibe DNI, password e ítem de menú, verifica y envía resultados, y termina.

El programa se abrió en 4 programas "invertidos" o "Data Driven". En cada interacción se llama a un programa distinto. Además hay transferencia de datos ocultos al usuario (el tema de seguridad se trata de otra forma, no viajan varias veces los pares usuario-password, sino un identificador de corta vida llamado *ticket*).
Un sistema así admite más cantidad de usuarios simultáneos, y los problemas de espacio y de “cuelgues” se desvanecieron.