

Módulo 2

DE PROGRAMA A PROCESOS

CONTENIDO:

- Conceptos básicos sobre Trabajos y Procesos.
- Conceptos introductorios a los Procesos.
- Estados de procesos
- El control y la ejecución de Procesos
- Concepto de Procesos livianos y de fibras.

OBJETIVOS DEL MODULO: Dar los conceptos básicos sobre las funciones; composición, arquitecturas, módulos, etc., de los Sistemas Operativos Computacionales en general.

OBJETIVOS DEL APRENDIZAJE: Después de la lectura y del estudio del presente módulo el alumno deberá conocer:

- (1) Los conceptos y elementos componentes de los Sistemas Operativos.
- (2) Los distintos tipos y estructuras de los Sistemas Operativos.
- (3) Concepto de ambiente de ejecución y de cambio de contexto.
- (4) Conceptos básicos sobre los Sistemas Operativos, sus propósitos, los servicios que brinda y su ejecución.
- (5) Conocer la terminología y sus significados utilizados en éste módulo.

Metas del Módulo 2:

Programas, Procesos y Procesadores.

Consideremos un Sistema Operativo como un conjunto de actividades, cada una de las cuales lleva a cabo una cierta función, como por ejemplo la administración de las Entradas/Salidas. Cada actividad consiste en que la ejecución de uno o más programas toda vez que se requiera tal función.

Utilizaremos la palabra **proceso** para referirnos a una actividad de este tipo. Consideremos a un programa como una entidad pasiva y a un proceso como una entidad activa, un proceso consiste entonces, en una secuencia de acciones llevadas a cabo a través de la ejecución de una serie de instrucciones (un programa en ejecución), cuyo resultado consiste en proveer alguna función en el sistema.

Las funciones del usuario también pueden asimilarse a este concepto, es decir que la ejecución de un programa de un usuario también será un proceso. Un proceso puede involucrar la ejecución de más de un programa. Recíprocamente, un determinado programa o rutina pueden estar involucrados en más de un proceso. De ahí que el conocimiento del programa en particular que está siendo ejecutado no nos diga mucho acerca de la actividad que se está llevando a cabo o de la función que está siendo implementada. Es fundamentalmente por esta razón por lo que es más útil el concepto de proceso que el de programa.

Un proceso es llevado a cabo por acción de un agente (unidad funcional) que ejecuta el programa asociado. Se conoce a esta unidad funcional con el nombre de **procesador**. Los conceptos de proceso y de procesador pueden emplearse con el fin de interpretar tanto la idea de concurrencia como la de no determinismo. La **concurrencia** puede verse como la activación de varios procesos a la vez. Suponiendo que haya tantos procesadores como procesos esto no reviste inconveniente alguno. Pero, si sucede que los procesadores son menos que los procesos se puede lograr una concurrencia aparente conmutando los procesadores de uno a otro proceso. Si esta conmutación se lleva a cabo en intervalos lo suficientemente pequeños, el sistema aparentará un comportamiento concurrente al ser analizado desde la perspectiva de una escala mayor de tiempo.

Resumiendo, un proceso es una secuencia de acciones y es, en consecuencia, dinámico, mientras que un programa es una secuencia de instrucciones y es estático. Un procesador es el agente que lleva a cabo un proceso. El no determinismo y la concurrencia pueden describirse en términos de interrupciones de procesos entre acciones (imposibilidad de prever el momento en el que se produce la interrupción) y de conmutación de procesadores entre procesos (se reparte el tiempo de CPU en porciones asignadas a cada proceso). Con el fin de que pueda llevarse a cabo esta conmutación, debe guardarse suficiente información acerca del proceso con el fin de que pueda reemprenderse su ejecución más tarde. Un proceso es un conjunto de instrucciones más su contexto de ejecución descrito en una estructura de control (Process Control Block - PCB).

2.0. Introducción

A continuación se definirán algunos conceptos importantes para poder explicar como se lleva un programa a procesos:

Definimos como **Programa** al conjunto ordenado de operaciones sobre un espacio de nombres de objetos (variables, estructuras de datos, archivos, etc.) creados por el programador y que representan un conjunto ordenado de instrucciones que pretenden resolver un problema.

Una **Instrucción** es una unidad de ejecución que dura un tiempo finito y se ejecuta sobre un procesador (es indivisible, no se descompone ni se interrumpe (salvo excepciones) y se dice que ejecuta atómicamente).

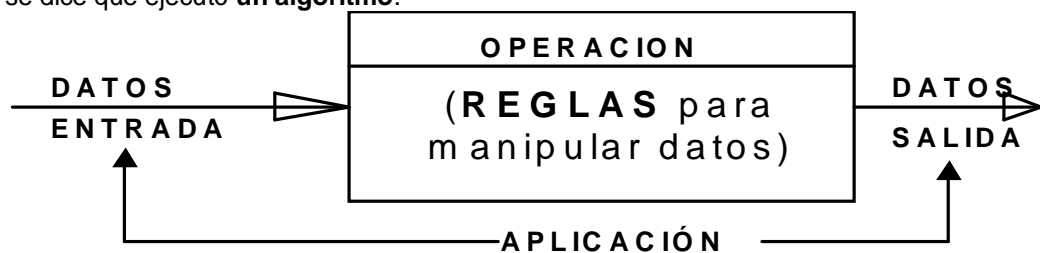
Cada instrucción determina:

- Un conjunto de Operaciones sucesivas
- Un conjunto de vías de datos involucrados en las operaciones

Las **OPERACIONES** son las Reglas para manipular los datos, por ejemplo:

- 'Una vez iniciada una operación debe terminar en un tiempo finito' o
- " La salida de una operación es una función independiente del tiempo (siempre que se apliquen los mismos datos a la entrada dará la misma salida)"

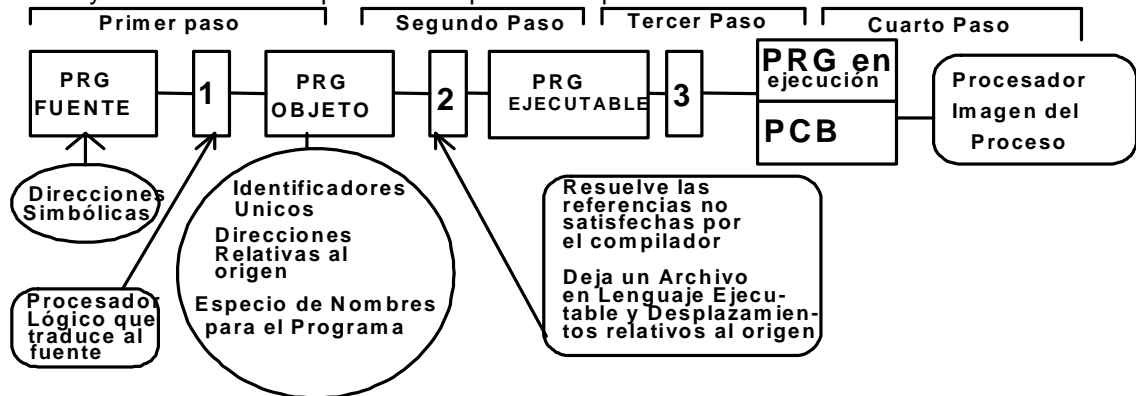
También definimos **una computación** como un conjunto finito de operaciones aplicados a un conjunto finito de datos con el fin de resolver un problema y si la computación logra el objetivo (resolver el problema) se dice que ejecutó **un algoritmo**.



DATOS Y OPERACIONES SON LAS COMPONENTES PRIMITIVAS DE LAS COMPUTACIONES

Fig. 2.01 operación Computacional

Un programador escribe su programa para una aplicación en un lenguaje de alto nivel (por ejemplo Pascal, C, C++, etc.). A este programa se lo llama **programa fuente** y tiene como característica principal que todas las variables y objetos tienen nombres simbólicos, o sea que sus direccionamientos serán simbólicos. Para poder ejecutarlo hay dos métodos: compilarlo o interpretarlo. El primero se lo debe someter a una traducción



- (1) Compilador
- (2) Link-editor
- (3) Loader + S.O.

Fig. 2.02 Pasos para llevar un programa a ejecución

mediante un **compilador** y enlazar las referencias mediante un **editor de enlaces** (linkeditor). Este proceso produce un archivo con un **programa ejecutable** cuyas direcciones son relativas al origen del programa y los nombres de objetos son únicos, sin ambigüedades. Para ejecutar un programa debe ser llevado a Memoria

Central mediante un **programa cargador** (loader) del S.O.. El segundo método se interpreta cada instrucción en lenguaje de alto nivel traduciéndolo a una serie de instrucciones de máquina que son inmediatamente cargados en memoria central.

Cada vez que un programa entra en ejecución, se crea un Bloque de Control para dicho programa llamado **Job Control Block (JCB)** que contiene todos los datos necesarios para su funcionamiento y sirve para que el Sistema Operativo pueda llevar el control de la ejecución de dicho programa.

Los programas se dividen en procesos en el momento de su ejecución.

Un **PROCESO** es una porción de un programa cargado en Memoria Central al cual se le asocia su contexto de ejecución (run time environment) mediante una estructura de datos llamada vector de estado o **Bloque de Control del Proceso (Process Control Block - PCB)**.

Para que sea posible la ejecución de varios procesos en forma concurrente, debemos guardar información acerca de dónde está evolucionando cada proceso, por lo que se almacenan los valores de los registros, archivos en uso, etc., así cuando el procesador interrumpa la ejecución del mismo sepa en qué estado había quedado para volver a restituirlo en el uso de los recursos. Estos datos sobre el proceso se guardan en una estructura de datos llamado vector de estado o **PCB (Process Control Block)**, y se almacenan en el **STACK** (pila) del proceso (el Stack es un área de memoria utilizada para la ejecución del proceso). Del Stack se copia una imagen del proceso que se carga sobre el procesador y se ejecuta.

El vector de estado o PCB no es visible mientras el proceso se ejecuta. El vector de estado contiene el contexto formado por el conjunto de recursos que necesita para su ejecución. Para eliminar un proceso simplemente se borra su vector de estado.

Entonces definimos:

- **Espacio de nombres de un programa:** Conjunto de nombres sobre el cual el programa puede actuar directamente (relación directa). Por Ej., un Archivo debe ser trasladado a Memoria Central para que el Programa pueda actuar sobre él. Para ello solicita el servicio del traslado al S.O.
- **Espacio de nombres de un proceso:** Conjunto de objetos que pueden ser usados por el proceso.
- **Espacio de nombres del procreador:** Conjunto de objetos que pueden ser usados por todos los procesos.
- **Espacio de memoria:** Conjunto de direcciones de memoria usadas para implementar el espacio de nombres del procesador.
- **Poder de un proceso (Modo Master-Slave):** Conjunto de información que define los recursos accesibles por dicho proceso, así como su modo de acceso. Puede evolucionar dinámicamente durante su ejecución.
- El espacio de nombres del proceso y del procesador no coinciden, en general el primero es un subconjunto del segundo.

Hasta ahora hemos hablado de programas y procesos un poco vagamente, y como si fueran sinónimos, pero son conceptos distintos. Un proceso es un programa en ejecución, incluyendo el valor actual del program counter (PC), registros y variables. Un programa es pasivo (es sólo código o texto y sus datos) y un proceso es activo y dinámico (varía en el tiempo).

Varios procesos pueden estar ejecutando el mismo programa, por ejemplo, si dos o más usuarios están usando simultáneamente el mismo editor de texto (Ejemplo el VI). El programa es el mismo, pero cada usuario tiene un proceso distinto, Stack distinto y con distintos datos producidos por cada ejecución.

Conceptualmente cada proceso tiene su propia CPU virtual. En la práctica, hay una sola CPU real cuando es monoprocesador, que cambia conmutándose periódicamente la ejecución de un proceso a otro, pero para entender el sistema es más fácil modelarlo como una colección de procesos secuenciales que ejecutan concurrentemente (pseudoparalelismo).

Un proceso dentro de un programa significa lo siguiente: Dados dos conjuntos de instrucciones que están dentro de un programa, si esos conjuntos son completamente independientes, de manera tal que el resultado de su ejecución es independiente, o sea no se afectan entre sí, se puede decir que cada conjunto es un proceso. Por ejemplo, si se tiene el siguiente conjunto de instrucciones:

$C = A + B$	y por otro lado :	$G = E + F$
$D = A + 1$		$H = D + 1$

Estos dos conjuntos de instrucciones son completamente independientes entre sí y si se ejecuten en forma independiente el resultado no depende de la ejecución al finalizar el programa, ya que cada uno maneja datos distintos. Para verlo más claro, supongamos que existe el siguiente conjunto de instrucciones en el cual se podría dividir un programa en un conjunto de tareas:

READ (A)

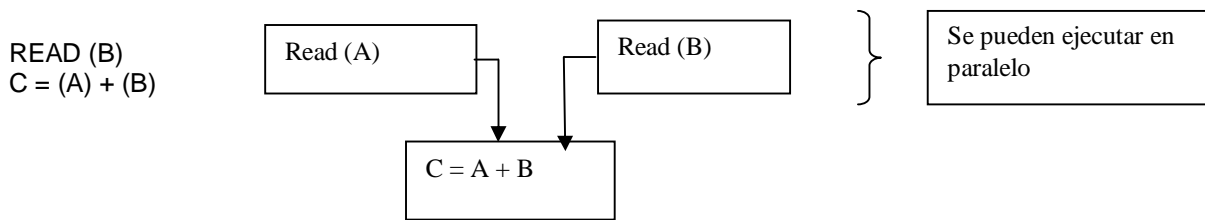


Fig. 2.03. Operaciones Read() y suma +

Cada instrucción READ, que desencadena una serie de operaciones, es completamente independiente de la otra que desencadena otro conjunto de instrucciones completamente independientes de la primera. En cambio, la operación:

$C = (A) + (B)$ es dependiente de las otras dos, desde el momento que se está usando el resultados de las dos anteriores. O sea que este fragmento de programa podría ser descompuesto en un grafo de la manera como se ve en la Fig. 2.03.

Pudiendo descomponerse de esta manera a un programa, es realmente posible que las dos operaciones, más tarde conjunto de instrucciones, pueden ser ejecutadas en forma simultánea, desde el punto de vista de la lógica del programa, es decir no afecta hasta aquí que cualquiera de las dos se ejecute antes, a partir de la asignación en sí.

Existe toda una serie de condiciones para establecer cuando es posible hacer esto que se verán más adelante cuando se estudien los procesos concurrentes (en el Módulo 4), pero hay que tener bien en claro que es posible dividir un proceso en subconjuntos de procesos que llamaremos Hilos o Threads, que son secuenciales si están compuestos por instrucciones que dependen unas de otras en una secuencia. Si el ejemplo anterior se diera en un ambiente de multiprocesamiento sería posible efectuar las dos operaciones READ en forma simultánea. Esto va a requerir algún elemento de sincronización, ya que no se podrá realizar la asignación hasta que no se hayan completado ambos READ.

Dado un programa es posible generar, por situaciones especiales, que se divida en distintas tareas que son independientes. Y estas también compiten por el recurso procesador, luego se tendrá que administrar el procesador también para distintas tareas que componen un programa y que van a utilizar ese recurso. Con lo cual también es posible que desde un bloque de control de un programa exista un puntero X al bloque de control de procesos correspondientes a ese programa, y que dentro de ese bloque de control de procesos, cada entrada corresponda a un proceso o tarea independiente que lo conforman.

Se puede hablar de programas o de procesos en forma indistinta cuando se trata de un procesamiento por Lotes (Batch) ya que dado un programa se puede descomponer en tareas disjuntas. En definitiva se están manejando bloques de control de procesos que van a requerir del uso del procesador en forma secuencial y concuerdan ambos bloques de control (Job Control Block y Process Control Block).

Entonces, un programa completo puede dividirse en procesos. Esto es típico de lenguajes que permiten multitareas, como el "C", y Pascal concurrente, ya que, o bien lo indica el programador o porque el compilador es lo suficientemente inteligente como para darse cuenta de segmentar el programa en distintos procesos, y esos procesos pueden después competir por el recurso como si fueran procesos independientes.

Utilizar este esquema dentro de una computadora que tiene un solo procesador es hacer **multiprogramación**. En los sistemas que tienen más de un procesador, y con la posibilidad de ejecutar, por ejemplo, cada READ en cada uno de ellos, se está en presencia de un sistema de **multiprocesamiento**. Más aún, si se inserta este esquema en computadoras distintas con algún mecanismo de comunicación, considerando inclusive que pueden estar ubicadas en sitios remotos, se puede decir que se está frente a un sistema de computación distribuido.

Dividir a los programas en tareas requiere luego, obviamente, de un sistema de comunicación entre los procesos que permita que una vez que estén divididos en tareas, se los pueda insertar en cualquier tipo de arquitectura y aprovechar entonces sus facilidades específicas.

2.1. El concepto de Trabajo, paso de trabajo, tarea y operaciones.

Trabajos, Procesos y Thread. Estos tres conceptos van definiendo el grado de granularidad en que el Sistema Operativo trata al conjunto de operaciones que se tienen que realizar cuando se ejecuta un programa.

Un **trabajo (Job)**: *Un Job es Sometido a un Sistema Computacional* y se conceptualiza como un conjunto de uno o más *procesos*. Decíamos que se puede definir a un programa como el conjunto ordenado de operaciones sobre un espacio de nombres de objetos (variables, estructuras de datos, archivos, etc.) creados por el programador y que representan un conjunto ordenado de instrucciones que pretenden resolver un problema.

Una **instrucción** es una unidad de ejecución que dura un tiempo finito y se ejecuta sobre un procesador (se procesa atómicamente). Cada instrucción determina un conjunto de operaciones sucesivas y un conjunto de vías de datos involucrados en las operaciones.

Un **proceso (Process)** se define como un programa en ejecución, es decir, en memoria y usando un procesador. A este nivel de granularidad, un proceso tiene un espacio de direcciones de memoria, una pila, sus registros y su registro puntero al programa 'program counter (PC)'.

Kernel que no es proceso. Este es el acercamiento más tradicional y simple. El sistema operativo tiene su propia región de memoria y su propia pila de sistema. El concepto de proceso solo se aplica a los programas de usuario, el código del sistema operativo es una entidad aparte que opera en modo privilegiado. Aunque hay Sistemas Operativos basados en procesos. El sistema operativo es una colección de procesos que corren en modo kernel. Nuevamente hay una pequeña porción de código (el de cambio de procesos) que corre fuera de cualquier proceso. Este tipo de sistema operativo impone un diseño modular y favorece el multiproceso.

Entonces, los programas se dividen en procesos en el momento de su ejecución. Un proceso es una porción de un programa cargado en Memoria Central al cual se le asocia su contexto de ejecución (**run time environment**) mediante una estructura de datos llamada vector de estado o process control block (PCB), de la cual se hablará en mayor detalle más adelante.

Un **Hilo o Hebra (thread)** también llamado proceso liviano, es un trozo o sección de un proceso que tiene sus propios registros, pila y 'program counter' y puede compartir la memoria con todos aquellos threads que forman parte del mismo proceso.

2.2. Introducción a los Procesos.

Todas las computadoras modernas ejecutan varias cosas al mismo tiempo. A la vez que ejecuta un programa del usuario, puede leer de un disco e imprimir en una terminal o impresora. En un sistema de multiprogramación, el Procesador también alterna de programa en programa, ejecutando cada uno de ellos por decenas o cientos de milisegundos. Aunque, en sentido estricto, el procesador (si es monoprocesador) ejecuta en cierto instante un solo programa, durante un segundo puede trabajar con varios de ellos, lo que da una apariencia de paralelismo. A veces, las personas hablan de **pseudoparalelismo** para indicar este rápido intercambio de los programas en el procesador, para distinguirlo del paralelismo real del hardware, donde se hacen cálculos en el procesador a la vez que operan uno o más dispositivos de entrada/salida. Es difícil mantener un registro de las distintas actividades paralelas. Por lo tanto, los diseñadores del Sistema Operativo han desarrollado con el tiempo un modelo que facilita el uso del paralelismo.

2.3. Definición, Concepto y Descripción de Procesos

El término "PROCESO", fue utilizado por primera vez por los diseñadores del sistema Multics en los años 60's. Desde entonces, el término proceso, utilizado a veces como sinónimo de tarea, ha tenido muchas definiciones. A continuación se presentan algunas:

- Un programa en ejecución más su contexto
- Una actividad asíncrona de ejecución
- Lo que se manifiesta por la existencia de un "bloque de control del proceso" en el Sistema Operativo
- La entidad a la que se asignan los procesadores.
- La unidad "despachable" de ejecución.
- Etc.

Como vemos no hay una definición universalmente aceptada, pero el concepto de "Programa en ejecución" parece ser el que se utiliza con más frecuencia. Un programa es una entidad inanimada; sólo cuando se ejecuta sobre un procesador, el S.O. le convierte en la entidad "activa" que se denomina proceso.

Un proceso pasa por una serie de estados discretos. Se dice que un proceso está ejecutando (**estado de ejecución**), si tiene asignado el procesador. Cuando un proceso está listo (estado listo) indica

que puede utilizar un procesador en caso de haber uno disponible. Un proceso está bloqueado (**estado bloqueado**) si está esperando que suceda algún evento (por ejemplo la lectura o escritura de datos sobre un disco) antes de poder seguir la ejecución.

Cuando un proceso se detiene en forma temporal, este debe volverse a inicializar en el mismo estado en que se encontraba al detenerse. Esto quiere decir que toda la información relativa al proceso debe almacenarse en forma explícita en alguna parte durante la suspensión. El lugar es el Stack de ejecución en la memoria central.

En muchos Sistemas Operativos, toda la información relativa en un proceso, distinta del contenido de su propio espacio de dirección se almacena en una tabla del Sistema Operativo llamada **tabla de procesos o PCB**, la cual consta de un arreglo (o lista enlazada) de estructuras, una por cada proceso existente en ese momento.

Así un proceso (**suspendido**) consta de un espacio de dirección, llamado imagen central y los datos de su tabla de control del proceso, que entre otras cosas contiene sus registros de CPU.

Para poder lograr la ilusión de varios programas ejecutando al mismo tiempo, el S.O. debe compartir un solo procesador entre todos los programas que se quieren ejecutar. Así nace el concepto de proceso.

Definimos como Proceso (tradicional) a la secuencia de acciones llevadas a cabo a través de instrucciones para proveer una funcionalidad del sistema o del programa usuario. Entonces Proceso es la porción del programa en ejecución más una estructura de datos llamado vector de estado o Process Control Block (PCB) que posee los valores actuales del espacio de nombres del programa y otros datos usados por el S.O., o sea, Proceso es una porción de Programa en ejecución más su contexto.

proceso = porción programa en ejecución + PCB

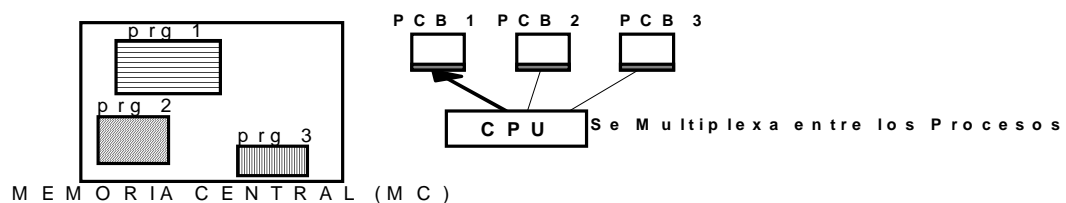


Fig. 2.04 Representación de la ejecución de los procesos

El PCB es utilizado para poder conmutar al procesador entre los distintos programas en ejecución como se muestra en la Fig. 2.04.

Proceso es entonces una CPU virtual, *idéntica* a la CPU real, que ejecuta un programa.

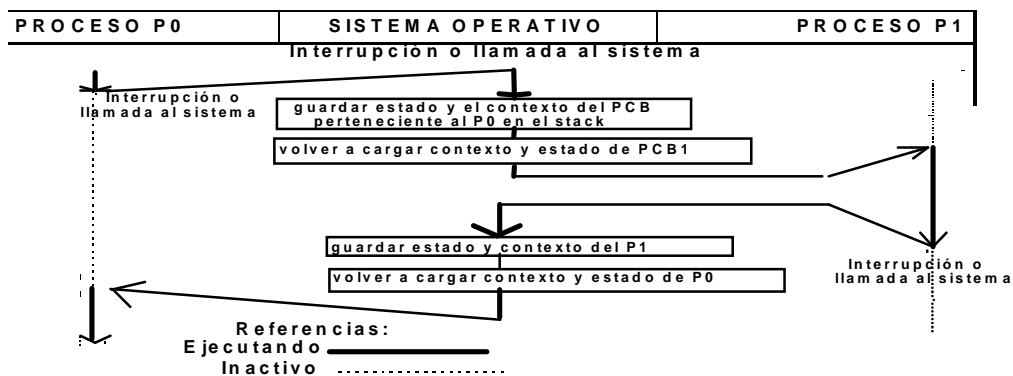


Fig. 2.05 La CPU puede cambiar de un proceso a otro

El PCB es utilizado para poder conmutar el procesador entre los distintos programas en ejecución. El PCB está almacenado en el área de Memoria (correspondiente al área del Stack). Cuando se ejecutan varios procesos (ejemplo P0 y P1 en la Fig. 2.05) sobre un procesador (Multiprogramación) la secuencia en el uso del procesador es la indicada en esa figura.

Entonces podemos definir:

- **Tarea (Task):** son distintas partes de un proceso que se ejecutan simultáneamente.
- **Instrucción:** unidad de ejecución que dura un tiempo finito, es determinística.
- **Procesador:** entidad cableada o no, capaz de ejecutar una instrucción.

- **Contexto:** conjunto de entidades que definen los estados del proceso. Proceso es entonces un procesador virtual, idéntico al procesador real, que ejecuta un programa.

2.4. Características fundamentales de los procesos.

El S.O. crea procesadores virtuales. Cada uno de ellos es una copia idéntica del procesador físico y ejecuta un programa sobre cada uno.

- Un Proceso es una secuencia dinámica de instrucciones mientras que un programa es una secuencia estática de instrucciones.
- Un proceso tiene asociado un programa que ejecuta.

Propiedades del espacio de nombres de un proceso:

- **Rango:** máximo número de objetos que puede especificar un proceso.
- **Resolución:** Tamaño mínimo de un objeto del espacio de nombres.

El Kernel y los procesos: El núcleo (Kernel) de un Sistema Operativo es un conjunto de rutinas cuya misión es la de gestionar los recursos de sistema como ser: el procesador, la memoria, la entrada/salida y también el resto de procesos disponibles. Toda esta gestión la realiza para atender al funcionamiento y pedidos que realizan los trabajos que se ejecutan en el sistema.

2.4.1. El Bloque de Control del Proceso (PCB, Vector de Estado o Descriptor del Proceso)

El PCB o Vector de Estado contiene:

1. el Estado inicial del Programa y
2. el conjunto de valores correspondientes a cada uno de los objetos a los que hace referencia el Programa.

Implementación de procesos. El Sistema Operativo mantiene para cada proceso un bloque de control (process control block - PCB), donde se guarda la información necesaria para reanudarlo (cuando es suspendido o desalojado del uso del procesador) además de otros datos. La información contenida en el PCB varía de S.O. en S.O. En general podemos mencionar la siguiente:

- Identificación del Proceso (única en el sistema)
- Identificadores de varios parientes del proceso (identificador del dueño, Padre, hijos, etc)
- Estado (ejecutando, listo, bloqueado, suspendido, nuevo, etc.)
- Program Counter
- Registros del procesador
- Información para planificación (p.ej., prioridad)
- Información para administración de memoria (p.ej., registros base y límite o la tabla de páginas)
- Información de I/O: dispositivos y recursos asignados al proceso, archivos abiertos en uso, etc.
- Estadísticas y otros datos contables: tiempo real y tiempo de CPU usado, etc.
- Privilegios.
- Otros objetos vinculados al proceso.

El sistema mantiene una cola con los procesos que están en estado de LISTO o Preparado. Todos los procesos bloqueados se ponen en otra cola, pero suele ser más eficiente manejar colas distintas según cuál sea la condición por la cual están bloqueados. Así, se tiene una cola de procesos para cada dispositivo de entrada/salida.

Los objetivos del bloque de control de procesos (PCB) son los siguientes:

- Localizar la información sobre el proceso por parte del Sistema Operativo.
- Mantener registrados los datos del proceso en caso de tener que suspender temporalmente su ejecución o reanudarla

El Bloque de Control de Proceso contiene el contexto de un proceso y todos los datos necesarios para hacer posible la ejecución del mismo y satisfacer sus necesidades. Cada PCB actual tiene un puntero al PCB anterior y uno al posterior mas una identificación del proceso, la palabra de control, los registros, si se está trabajando en un sistema de administración de memoria paginada un puntero a su tabla de páginas, dispositivos y archivos que esté usando, tiempos que hacen a la vida del proceso, el estado. Un esquema de un Bloque de Control de Procesos puede verse en la Tabla 2.1.

Esta información se encuentra en memoria central en el área del Stack y se accede a ella en los momentos en que se hace necesaria su actualización o consulta.

Si bien es cierto que es más fácil pensar al PCB como una matriz, este tipo de implementación es muy rígida y rápidamente podría desembocar en que el espacio reservado para PCB's se agote. Una solución mejor y más dinámica conviene implementar la Tabla PCB como un encadenamiento de PCB's mediante una estructura tipo cola. En forma más detallada cada uno de sus campos contiene:

- Identificación de Proceso (**PID**): identificación única para cada proceso que lo hace inconfundible con otro.
- Puntero al proceso anterior: dirección del PCB anterior (anterior en tiempo pues fue creado antes). El primer PCB tendrá una identificación que lo señale como tal y deberá ser conocida su ubicación por el Planificador de Procesos.
- Puntero al proceso posterior: dirección del PCB posterior (posterior en tiempo, pues fue creado después). El último PCB tendrá un proceso nulo o **nil** (no se descartan encadenamientos circulares, pero por ahora solo se presenta como lineales).
- Palabra de control: espacio reservado o puntero en donde se guarda el Program Counter (PC) y otros registros de uso general del procesador, cuando el proceso no se encuentra en ejecución.
- PTP (Page Table Pointer): puntero al lugar en donde se encuentra la Tabla de Páginas correspondiente a este proceso. En el supuesto de tratarse de otro tipo de administración de memoria en esta ubicación se encontraría la información necesaria para conocer en qué lugar de memoria está ubicado el proceso.
- Punteros a Tablas de Dispositivos: punteros a todos las Tablas de dispositivos o Drivers a los que tiene acceso el proceso al momento de ejecutarse. Esta información puede ser estática, entonces es necesario que el proceso declare antes de comenzar su ejecución todos los dispositivos que va a usar, o completamente dinámica si existe la capacidad de obtener y liberar dispositivos a medida que se ejecuta el proceso.

PID (Identificador del Proceso)
STATUS (Estado: New , Ready, Running, Wait, Halt, etc.)
Pointer (Puntero) (al próximo PCB)
Area de Pointers (Punteros a otros procesos parientes) (Padre, Hijos, Dueño, grupo, etc.)
CPU DATA: (Se deben salvar ante una Interrupción o un System Call) REGISTROS: Program Counter, Acumulador, Flags, RI, data pointer, segment pointer, index pointer etc. etc.
Memory Management: Límites, RB, RL, Tablas de Páginas o Segmentos, etc (Información para la administración de la Memoria Central)
FILE Management: Descriptores, Directorios, Path, Parámetros de llamada, protección, etc. (Inf. para la administración de los Archivos)
I/O Management: Status, Path, etc. (Inf. sobre los dispositivos y su uso)
ACCOUNT Información estadística y contable sobre los recursos para la administración.
Privilegios: Modos de ejecución, prioridades, protecciones, etc
(Área de Pointers a Threads)

Tabla 2.1 Estructura de datos de un proceso (PCB)

- Punteros a Bloque de Control de Archivos (File Control Block Pointer): puntero al Bloque de Control para cada archivo abierto del proceso.
- Tiempos: Tiempo de procesador utilizado hasta el momento. Tiempo máximo de procesador permitido a este proceso. Tiempo que le resta de procesador a este proceso. Otros tiempos.
- Estado del Proceso: Nuevo, Ejecutando, Listo, Bloqueado, Wait (En espera). Ocioso, etc.
- Puntero al PCB del proceso anterior en el mismo estado: dirección del PCB correspondiente al proceso anterior en ese mismo estado.
- Puntero al PCB del proceso posterior en el mismo estado: ídem anterior pero al proceso posterior.

- Información para el algoritmo de adjudicación del procesador: aquí se tendrá la información necesaria de acuerdo al algoritmo en uso.
- Puntero al PCB del Proceso Padre: dirección del PCB del proceso que generó el actual proceso.
- Puntero a los PCB de padre e Hijos: puntero a la lista que contiene las direcciones de los PCB hijos (generados por) de este proceso. Si no tiene contendrá nil.
- Punteros a procesos parientes: dueño, grupo, etc.
- Punteros a los Threads que dependen de ese proceso, si el procesamiento soporta multithreading.
- Accounting: información que servirá para contabilizar los gastos que produce este proceso (números contables, cantidad de procesos de E/S, tiempos, etc.)

También existe un **Bloque de Control de Sistema (System Control Block - SCB)** con objetivos similares al anterior y entre los que se encuentra el enlazado de los bloques de control de procesos existentes en el sistema. El cambio de contexto se producirá en caso de la ejecución de una instrucción privilegiada, una llamada al Sistema Operativo o una interrupción, es decir, siempre que se requiera la atención de algún servicio del Sistema Operativo.

2.4.2. La representación de los Procesos

Máquina Virtual: En la ejecución de cada proceso queda cableado en forma virtual todos los recursos necesarios para su ejecución. El S.O. controla la correspondencia dinámica de los Recursos de la Máquina Virtual y los físicos del sistema.

Como se observa en la figura siguiente el Sistema Operativo, a través del Kernel, crea esa máquina virtual que permite la ejecución de los procesos sobre un procesador (Hardware cableado para cada instrucción del programa en ejecución) haciendo una abstracción de la complejidad del procesamiento.

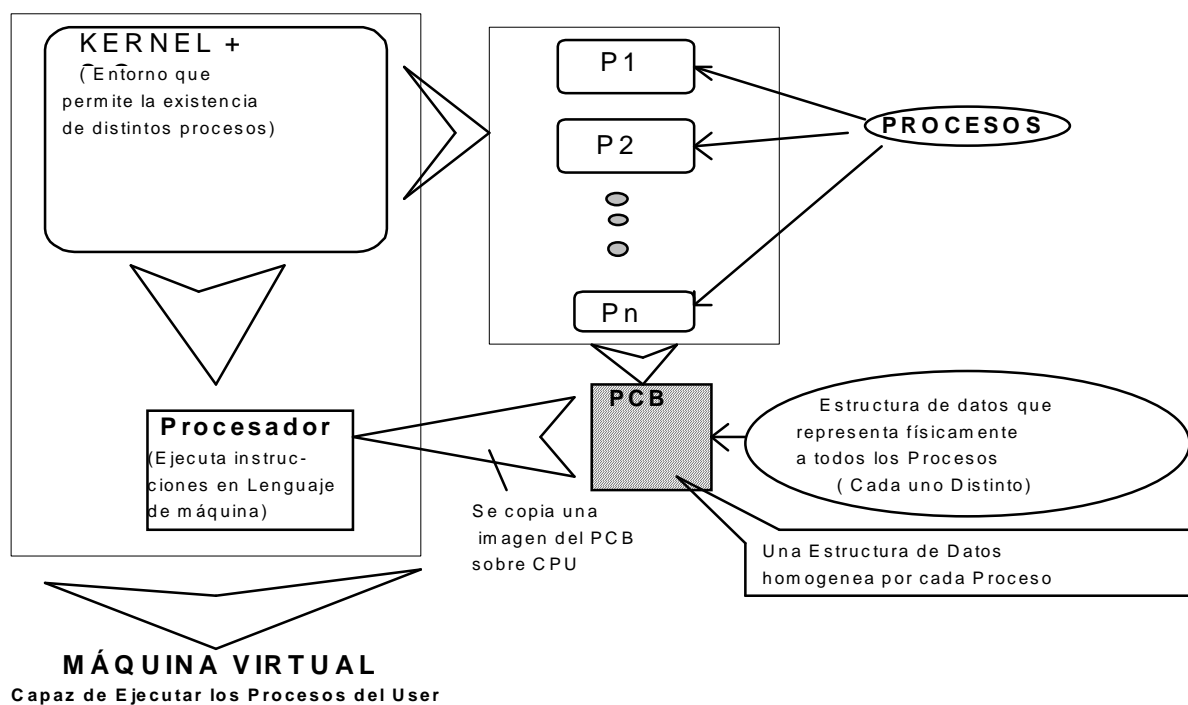


Fig. 2.06 a Representación de los procesos sobre el procesador

Dado que la cantidad de objetos que puede usar un proceso para su ejecución son variables (espacio de nombres del proceso) y la cantidad de objetos del procesador son fijos (espacio de nombres del procesador) es necesario homogenizar el espacio de nombres que se carga sobre el procesador (imagen del proceso). Si el espacio de nombres del proceso es más grande se requiere hacer referencia mediante un registro interno del procesador como puntero índice a la dirección del Stack donde están los objetos del PCB aún no cargados sobre el procesador. Sin embargo, si el espacio de nombres del proceso es más pequeño o igual al del procesador no es necesario este puntero.

cola de procesos listos y apuntadores al primer y último PCB de la lista. Cada PCB tiene un puntero al siguiente proceso de la cola READY.

A partir de aquí, los bloques de control de los procesos se almacenan en colas, cada una de las cuales representa un estado particular de los procesos, además de otras informaciones.

Los estados de los procesos son internos del Sistema Operativo y transparentes al usuario y se pueden dividir en dos tipos: activos e inactivos:

1- **Estados activos:** Son aquellos que compiten con el procesador o están en condiciones de hacerlo (instante en que se le ha asignado el recurso hasta que lo devuelve). Se dividen en:

- **Listo o Preparado (Ready):** Aquellos procesos que están dispuestos para ser ejecutados, pero no están en ejecución por alguna causa (Interrupción, haber entrado en cola estando otro proceso en ejecución, etc.). Disponen de todos los recursos para su ejecución y aguardan su turno en una cola de listos (Ready Queue).
- **Ejecución (Running):** Estado en el que se encuentra un proceso cuando tiene el control del procesador. En un sistema monoprocesador este estado sólo lo puede tener un proceso. En el estado de Ejecución, el proceso está en uso (control) del procesador. En un sistema con multiprocesadores, puede haber tantos procesos como procesadores.
- **Bloqueado (Blocked):** Son los procesos que no pueden ejecutarse de momento por necesitar algún recurso no disponible o esperan completar una operación de entrada/salida. Son los que aguardan momentáneamente por un recurso detrás de una cola de espera (Waiting Queue or Blocking Queue).

2- **Estados inactivos:** Son aquellos que no pueden competir por el procesador, pero que pueden volver a hacerlo por medio de ciertas operaciones. En estos estados se mantiene el bloque de control de proceso en colas (fuera de la memoria central (generalmente en el área de swap del disco) hasta que vuelva a ser activado. Se trata de procesos que aún no han terminado su trabajo y que han sido interrumpidos y que pueden volver a activarse desde el punto en que se quedaron sin que tengan que volver a ejecutarse desde el principio.

Son procesos que esperan sus recursos por largo tiempo y su estado es **Suspendido** o Bloqueado a la espera de la ocurrencia de un evento, generalmente por un dispositivo de entrada/salida (E/S). Algunos S.O. la llaman **hibernación** dado que su espera es por un gran tiempo.

Son de dos tipos:

- **Suspendido bloqueado (Suspend – Blocked):** Es el proceso que fue suspendido en espera de un evento, sin que haya desaparecido las causas de su bloqueo.
- **Suspendido programado (Suspended – Ready):** Es el proceso que ha sido suspendido, pero no tiene causa para estar bloqueado.

2.4.4. Ciclo de vida de un proceso

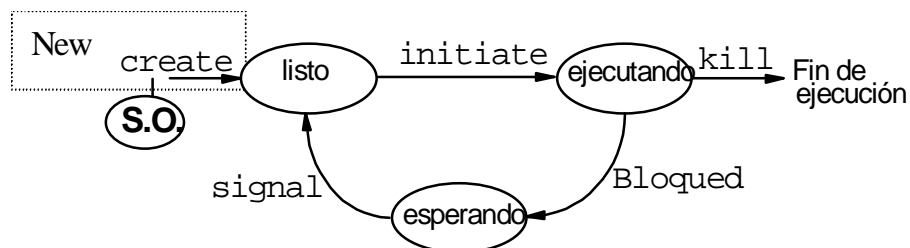


Fig. 2.07 Ciclo de vida de los procesos

Un Diagrama de estados de un proceso es el que se indica en la figura 2.07. Un Proceso nace mediante un System Call emitido por otro proceso o por el S.O.. A partir de esta acción pasa por la siguiente secuencia de estados

- Nuevo o New (create): El proceso está siendo creado.
- Listo (ready): El Proceso espera que se le asigne el procesador.
- Ejecutando (running): Se están ejecutando las Instrucciones del proceso.
- Esperando (waiting): El Proceso está esperando que suceda algún evento.
- Terminado (completed): El Proceso ha finalizado.

a) Creación de procesos:

La vida de un proceso esta limitada por su creación y su terminación. Construir la estructura de datos que se utilizan para administrar el proceso y asignar el espacios de direcciones que van a utilizar. Estas acciones constituyen la creación de un nuevo proceso.

Los sucesos que llevan a la creación de un proceso son:

1. En un entorno de trabajo por lotes, un proceso se crea como respuesta a la remisión de un trabajo.
 2. En un entorno interactivo, se crea un proceso cuando un nuevo usuario intenta conectarse.
- En estos dos casos, el S.O. es el responsable de la creación del nuevo proceso.
3. Si es el S.O. quien crea un proceso es parte de una ejecución.

Todos los procesos son creados por el S.O. de una forma transparente para el usuario o el programa de aplicaciones. Sin embargo, puede ser factible que un proceso origine la creación de otros procesos.

Cuando un proceso es creado por el S.O. tras la solicitud explícita de otro proceso, la acción se conoce como generación de procesos.

Cuando un proceso genera a otro, el proceso generador se conoce como proceso padre y el proceso generado es el proceso hijo. Normalmente estos procesos necesitan comunicarse y cooperar.

Crear un Proceso Significa:

1. Darle un nombre (referencias unívocas - no ambiguas-) generalmente queda representado en un **PID** (Process IDentifier) y en un espacio de direccionamiento en memoria.
2. En el PCB inicial se deben especificar los objetos y datos que va a usar el Proceso (Se asignan todos los recursos que va a utilizar durante su ejecución).
3. Se hace mediante una llamada al sistema.

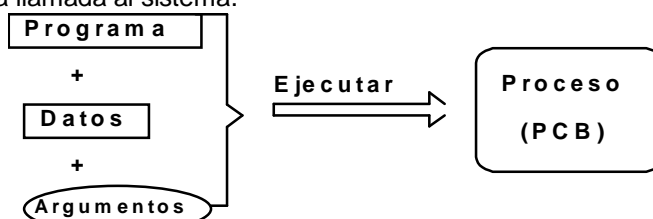


Fig. 2.08 La creación en el ciclo de vida de un proceso

Existen dos tipos de creación:

1. **Jerárquica:** Cada proceso que se crea es hijo del proceso creador y hereda el entorno de ejecución de su padre. Un proceso durante su ejecución puede crear varios procesos hijos a través de llamadas al sistema para la creación de procesos. El Proceso creador es el proceso padre y el proceso nuevo es el proceso hijo, el cual podrá crear otros procesos creando un árbol (tree) de procesos. Los subprocesos pueden obtener recursos del S.O. o restringirse a recursos del proceso padre, el cual divide recursos entre sus hijos o los comparte entre varios hijos. Al restringir un proceso hijo a un subconjunto de recursos del padre se evita que este sature al sistema creando demasiados procesos hijos. Al crearse un proceso éste obtiene recursos físicos y lógicos, más los datos iniciales, del proceso padre. Hay dos alternativas:
 - El Padre continúa ejecutando concurrentemente (paralelo) con sus hijos; o
 - El padre espera a que todos sus hijos hayan terminado y luego sigue él.
2. **No Jerárquica:** cada proceso creado por otro proceso se ejecuta independientemente de su creador en un entorno diferente. (no es muy común).

Motivaciones para crear un proceso: Hay cuatro motivaciones o razones para que un proceso sea creado:

- a) Llega un trabajo nuevo al sistema: generalmente en forma de batch entonces el S.O. debe recibirlo y comenzarlo a ejecutar creando una secuencia de procesos nuevos.
- b) Llegada de un usuario al sistema: Entonces el S.O. ejecuta un proceso llamado login.
- c) Un servicio al programa en ejecución: Creado por el S.O. por ejemplo realizar una lectura en disco (en ese caso el proceso que solicitó el servicio es bloqueado).
- d) Por un proceso existente: por razones de modularidad o paralelismo.

Una vez que el S.O. decide por alguna razón crear un nuevo proceso sigue los siguientes pasos:

- 1- asignar un único identificador al nuevo proceso.

- 2- Asignar espacio para el proceso
- 3- Inicializar el bloque de control de proceso
- 4- Establecer los enlaces apropiados con otras estructuras de datos.
- 5- Ampliar o crear otras estructuras de dato en el caso que fueran necesarias.

b) Descendencia de un proceso

Supongamos dos procesos (p y q) en que se dan las siguientes relaciones:

- q creado por $p \Rightarrow q \in D(p)$.
- $q \in D(p) \Rightarrow D(q) \in D(p)$.

Se dice que “q” fue creado por “p” pertenece a la descendencia de p y no existe otra forma de que un proceso pertenezca a D(p). Entonces “p” es el padre y “q” el hijo.

c) Muerte de un proceso:

Un proceso muere cuando el proceso termina de ejecutar (se completa) o por una falla o error. Puede ser por propia iniciativa o por iniciativa del Sistema Operativo u de otro proceso.

En cualquier sistema, debe haber alguna forma de que un proceso pueda indicar que ha terminado. Un trabajo por lotes debe incluir una instrucción de detención de fallos o una llamada explícita a un servicio del SO. En una aplicación interactiva, es la acción del usuario la que indica cuando termina el proceso. En una computadora personal o una estación de trabajo, el usuario puede abandonar una aplicación. Todas estas acciones provocan al final un pedido de un servicio al S.O. para terminar con el proceso.

Además de una serie de errores y condiciones de fallo pueden llevarnos a la terminación de un proceso. En algunos S.O. un proceso puede ser eliminado por el proceso que lo creó o al terminar el proceso padre.

Un Proceso puede terminar de ejecutar abruptamente en forma total o temporal por múltiples razones. Lo más deseable sería que el proceso termine en forma satisfactoria. Las causales posibles de abandono o muerte de un proceso son:

- FIN NORMAL (Proceso completado)
- ERROR DE PROTECCIÓN (Fin anormal) generalmente se produce cuando el proceso intenta acceder a un recurso no permitido, ejemplo leer o escribir un archivo.
- POR INTERVENCIÓN DE UN OPERADOR O POR EL S.O.: ejemplo cuando se “cuelga” un proceso.
- VIOLACIONES de ACCESO A ÁREAS DE MEMORIA: se dispara un mecanismo de seguridad no permitiendo que el proceso acceda a áreas no permitidas. Generalmente se emite en error de protección.
- FALLA DE E/S: se genera al producirse un error en las operaciones de entrada o salida. Por ejemplo se pretende leer un archivo inexistente.
- SE PRODUCE UNA EXCEPCIÓN (TRAP) EN LA EJECUCIÓN: Las excepciones pueden ser por una instrucción inválida o que no exista en el juego de instrucciones del procesador, o se pretenda ejecutar una instrucción privilegiada (del S.O.) en modo usuario, o la dirección es inválida porque no existe en memoria.
- ERROR DE DATOS: tipo de datos incorrectos o no inicializado.
- POR REQUERIMIENTO DE UN PROCESO PARIENTE: dado que los procesos parientes en el orden jerárquico ascendente tienen autoridad para finalizarlo. Ejemplo un padre le envía una señal para matarlo (en UNÍX: kill()).
- MUERTE O FINALIZACIÓN DE UN PROCESO PARIENTE: Toda la descendencia muere.
- ERROR ARITMÉTICO: generado por el procesador al realizar una división por cero o manipular un número mayor que el hardware pueda manejar.
- NO DISPONIBILIDAD DE MEMORIA: cuando el proceso requiere más memoria que el sistema dispone y no se trabaja con Memoria Virtual.
- LIMITE DE TIEMPO EXCEDIDO: generalmente se le asigna un tiempo prudencial para que se ejecute y el S.O. controla ese lapso. Si se excede puede producir una interrupción por time out y generalmente se aborta la ejecución. Esto vale también para la ejecución de procesador o E/S
- NECESIDAD DE RECURSOS (E/S, etc., por lo tanto pasa a Bloqueado)
- DESALOJO EN EL USO DE PROCESADOR (Por algún proceso de mayor prioridad o porque se le terminó el tiempo asignado) (Pasa a Listos)

Casi todas las causales de finalización se refieren al fin Total del proceso, en tanto solamente las dos últimas indican un fin Temporal del mismo. Desalojo significa que, por alguno de los algoritmos de

administración de procesador, se considera que el tiempo de uso del procesador por parte de ese proceso ha sido demasiado alto por lo que se le interrumpe su ejecución devolviéndole a la cola de listos.

Por otro lado un padre (pariente ascendente) puede causar la terminación de sus hijos por diferentes causas:

- el hijo se excedió en el uso de algún recurso asignado;
- ya no requiere más de la tarea del hijo.

Para esto el padre necesita conocer la identidad de sus hijos (cuenta con un mecanismo para inspeccionar los estados de sus hijos mediante la consulta de los PCB).

Cuando un proceso muere ocurren los siguientes pasos:

- Desaparece el PCB
- Recursos comunes son liberados
- Recursos locales son destruidos

Cuando un proceso termina (muere) también deben terminar sus hijos (normal o anormalmente). Esto se conoce como **terminación en cascada**.

Con la muerte de un proceso, el S.O. debe proveer un servicio de acuerdo al tipo de finalización.

2.4.5. Transiciones de Estado

Todo proceso a lo largo de su existencia puede cambiar de estado varias veces. Depende del modelo de estados implementado en el S.O.. Cada uno de estos cambios se denomina **transición** de estado. Estas transiciones pueden ser las siguientes:

- **Comienzo de la ejecución.** Todo proceso comienza al ser dada la orden de ejecución del programa mediante una llamada al Sistema Operativo quien crea el proceso y luego lo insertará en la cola de listos o lo enviará al disco. El encolamiento dependerá de la política de gestión de dicha cola.
- **Paso a estado listo o preparado:** Este paso puede ser producido por alguna de las siguientes causas. a) Orden de ejecución de un programa, con la cual el proceso pasa a la cola de listos. b) Si un proceso está en estado bloqueado por una operación de entrada/salida y esta finaliza, pasará de la cola de bloqueados a la de preparados o listos. c) Si un proceso está en ejecución y aparece una interrupción que fuerza al Sistema Operativo a ejecutar otro proceso, el primero pasará al estado de ejecución y el segundo su PCB a la cola de listos.
- **Paso de estado de ejecución.** Cuando el procesador se encuentra inactivo y en la cola de preparados exista algún proceso en espera de ser ejecutado, se pondrá en ejecución el primero de ellos.
- **Paso a estado bloqueado.** Un proceso que se encuentre en ejecución y que solicite una operación a un dispositivo externo, teniendo que esperar a que dicha operación finalice, será pasado de estado de ejecución a estado bloqueado insertándose su PCB en la cola correspondientes de bloqueado. A partir de este momento el procesador pone en ejecución el siguiente proceso, que será el primero de la cola de listos.
- **Activación.** Un proceso suspendido previamente sin estar bloqueado pasará al estado listo al ser activado nuevamente.
- **Paso a estado suspendido bloqueado.** Si un proceso está bloqueado y el Sistema Operativo recibe la orden de suspenderlo, su PCB entrará en la cola de procesos suspendidos bloqueados.
- **Paso a estado suspendido listo.** Este paso se puede producir bajo tres circunstancias:
 1. Suspensión de un proceso listo pasando éste de la cola de procesos listos a la de suspendidos listos.
 2. Suspensión de un proceso en ejecución, con lo cual el proceso pasa a la cola de suspendidos listos.
 3. Desbloqueo de un proceso suspendido bloqueado por desaparecer la causa que impedía el ser activado de nuevo.

a) Modelo de procesos de 2 estados

La responsabilidad principal del S.O. es llevar el control de los procesos; esto incluye determinar el patrón de intervalo de tiempo para la ejecución y asignación de recursos o procesos.

Lo primero a determinar para diseñar un programa para controlar procesos es describir el comportamiento que se desea que el proceso exhiba.

En un momento cualquiera, un proceso está siendo ejecutado o no. De esta manera, un proceso puede estar en 1 de estos 2 estados:

- Ejecutando (Running)
- No Ejecutando (Not running).

Cuando el S.O. crea un proceso, entra al sistema en el estado “not running”. El proceso existe (es conocido por el SO), y está esperando la oportunidad para ejecutarse.

Cada proceso debe ser representado de manera tal que el S.O. pueda seguirle el rastro. Para esto debe haber algún tipo de información con respecto a cada proceso (estado actual y ubicación en memoria, etc.). Los procesos que están corriendo son guardados en una cola, esperando su turno de ejecución. Este es el comportamiento del despachador en términos del diagrama de colas:

- Un proceso interrumpido es “metido” a la cola de procesos en espera (wait queue – not running).
- Cuando un proceso es finalizado o abortado, éste es descartado (Exit sale del sistema).
- Cualquiera sea el caso, el despachador elegirá un proceso de la cola para ejecutar.

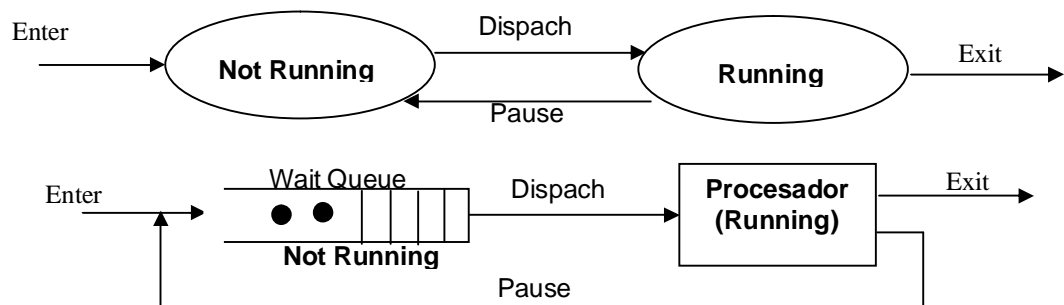


Fig. 2.09. Diagrama de 2 estados y su cola asociada

En este modelo las transiciones son muy simples:

- **Not Running ® Running:** El S.O. hace éste cambio de estado cuando es necesario elegir un nuevo proceso a ejecutar al existir procesos en la cola y el procesador esta desocupado.
- **Running ® Not Running:** De vez en cuando, un proceso será interrumpido por alguna razón. El S.O. elegirá un nuevo proceso para ejecutar o atenderá el evento. El proceso interrumpido pasará del estado “running” al “not running” y el evento (proceso) a ejecutar, realizará la transición inversa.

b) Modelo de 3 estados:

Para poder manejar convenientemente una administración de procesador es necesario contar con un cierto juego de datos. Ese juego de datos será una tabla (PCB) en la cual se reflejará en qué estado se encuentra el proceso, por ejemplo, si está ejecutando o no. Los procesos, básicamente, se van a encontrar en tres estados:

- Listo para la ejecución,
- Ejecutando, o
- Bloqueados por alguna razón.

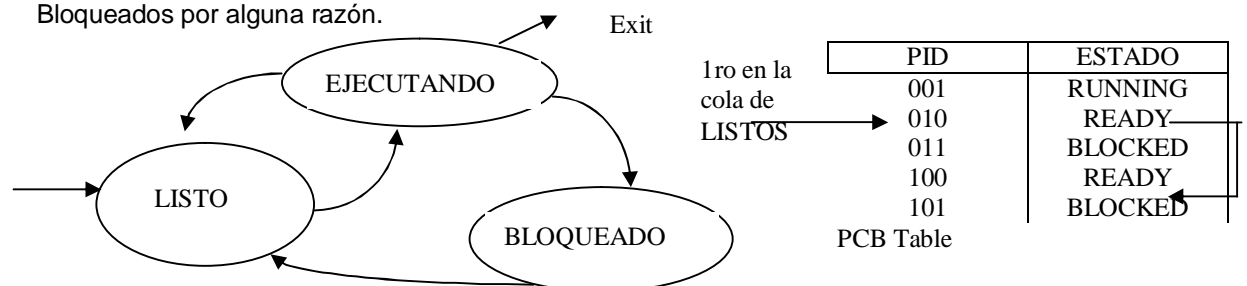


Fig. 2.10 Modelo de transiciones de 3 estados y su tabla asociado

Sobre la base de estos estados se construye lo que se denomina **Diagrama de Transición de Estados**. Estar en la cola de Listos significa que el único recurso que a ese proceso le está haciendo falta es el recurso procesador. O sea, una vez seleccionado de esta cola pasa al estado de Ejecución. Se tiene una transición al estado de Bloqueados cada vez que el proceso pida algún recurso de E/S. Una vez que ese

requerimiento ha sido satisfecho, el proceso pasará al estado de Listo esperando nuevamente el recurso procesador.

Para manejar esa cola de listos se requiere de una tabla, y esa tabla debe tener una identificación de los procesos (Fig. 2.10). Esta tabla contiene los Bloques de Control de Procesos (PCBT).

Como los listos pueden ser muchos, hará falta un puntero al primero de esa cola de listos, y posiblemente un puntero enganche entre los siguientes en el mismo estado.

c) Modelo de Cinco Estados

Como se ha explicado antes, el procesador opera en forma ejecutando – no ejecutando con los procesos. Sin embargo, esta forma de trabajo resulta ineficiente debido a que algunos procesos en el estado “not running” están listos para ser ejecutados, mientras los otros están bloqueados esperando por E/S. Si se trabajara de esta manera (con una sola cola), el despachador tendría que buscar en la cola el proceso no bloqueado que haya estado en ella más tiempo.

Para evitar éste trabajo, se ha dividido el estado “not running” en 2 estados (“ready” y “blocked”) y se han agregado 2 estados más, resultando un modelo de 5 estados:

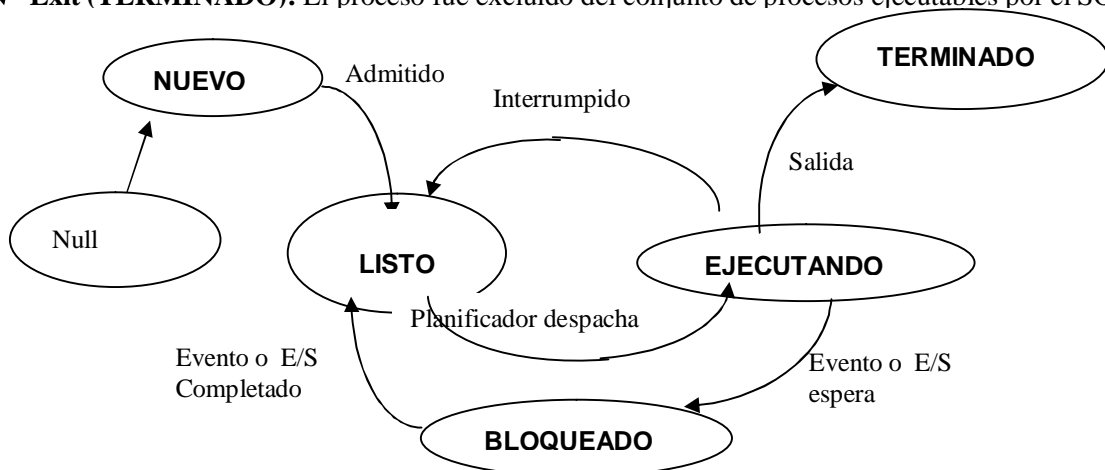
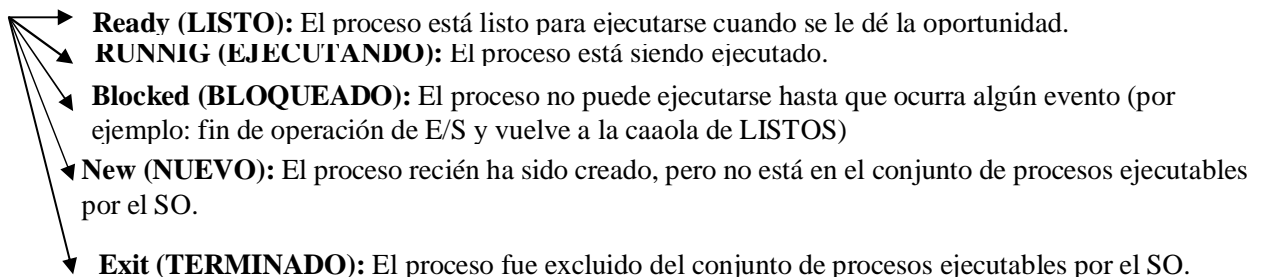


Fig. 2.11 Modelo de 5 estados

Los eventos que llevan a un proceso a cambiar de un estado a otro son:

1. **Null ® new:** El nuevo proceso es creado para ejecutar un programa.
2. **New ® ready:** El S.O. realiza esta transición cuando está preparado para comenzar a ejecutarse.
3. **Ready ® running:** El S.O. hace éste cambio de estado cuando es tiempo de elegir un nuevo proceso a ejecutar.
4. **Running ® exit:** Este cambio de estado se produce cuando el proceso actualmente en ejecución es finalizado o abortado.
5. **Running ® ready:** Esta transición puede ocurrir cuando se ha alcanzado el límite máximo de tiempo de ejecución ininterrumpida (técnica encontrada comunmente en los S.O. con multiprogramación) o cuando, al trabajar con distintos niveles de prioridad, un proceso es reemplazado por otro de mayor prioridad.
6. **Running ® blocked:** Esto ocurre cuando un proceso solicita algo por lo que deba esperar. Este pedido es realizado, por lo general, en la forma de un **system call** (llamado de un programa en

ejecución a un procedimiento que es parte del código del SO. Por ejemplo LEER un Archivo).

7. **Blocked ® ready:** Este cambio tiene lugar al ocurrir el evento por el que estaba esperando un proceso.
8. **Ready ® exit:** Esto ocurre cuando un proceso hijo es finalizado; ya sea por pedido del proceso padre o porque el padre en sí fue finalizado.
9. **Blocked ® exit:** Idem ready®exit.

d) Modelo de Seis y Siete Estados

A los cinco estados anteriores se le agrega el de Suspendido. Si bien el modelo de 5 estados resulta bastante eficiente, hay razones para agregar más estados al modelo. Para entender esto, se considerará un S.O. que no use memoria virtual (MV).

Cada proceso, para ser ejecutado, debe estar en memoria central (MC). Esto implica que todas las colas, con sus procesos, deben residir en MC. El problema es que, debido a la lentitud de la E/S en comparación con el procesador, se puede dar frecuentemente el caso de que todos los procesos estén esperando por E/S; estando el procesador inactivo la mayor parte del tiempo. La primera solución que viene a la mente es expandir la MC; pero esto no resuelve nada debido a 2 razones:

- El costo asociado a esa expansión tanto para su administración como de protección.
- La reciente tendencia de los programas a usar la mayor cantidad de memoria, debido fundamentalmente al bajo costo de la memoria por lo que no se optimiza la programación.

Por lo tanto, en caso de expandirla, se acabaría con procesos más grandes en lugar de más procesos. De todas formas se plantea el modelo de 6 estados puede ser con o sin swapping en que el estado suspendido no permanece o permanece en MC.

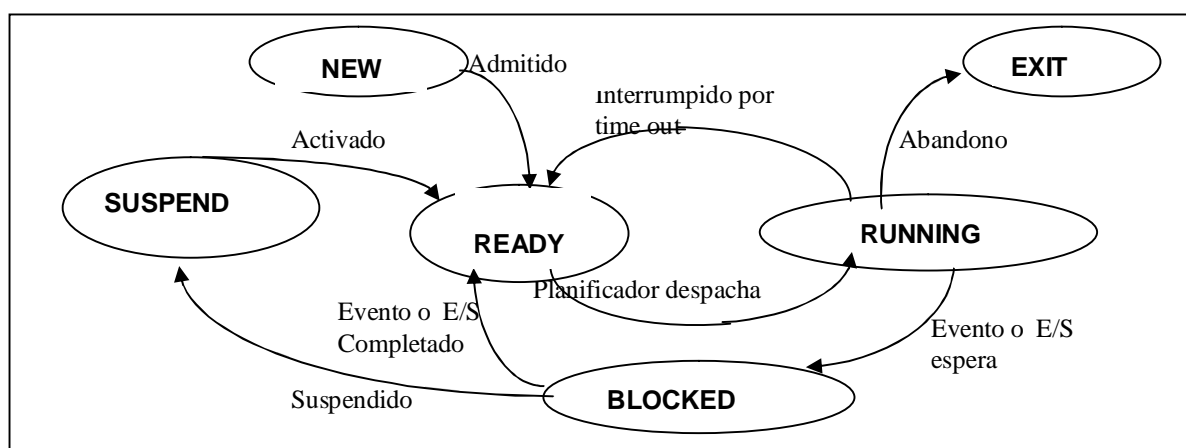


Fig. 2.12 Modelo de 6 estados.

Otra solución es el **swapping**¹ (mover un proceso de memoria a disco – swap out - y viceversa traerlo desde el disco a memoria – swap in -). Cuando todos los procesos están bloqueados, el S.O. suspende algunos procesos (pasándolo al estado “suspend”) y son transferidos al disco. Con ése espacio liberado de MC, se puede traer un procesos del disco que estaba suspendido y puede ser ejecutado o agregar un nuevo proceso al sistema.. La tendencia del S.O. será crear un nuevo proceso, ya que no tiene mucho sentido traer un proceso que no estará listo para ejecutarse (a pesar que esto acarrea un aumento en la cantidad de procesos).

El concepto de swapping trae nuevos conceptos de los estados existentes:

- **Listo (Ready):** El proceso está en MC y listo para su ejecución.
- **Bloqueado (Blocked):** El proceso está en MC, pero está esperando un evento generalmente de E/S.
- **Suspendido - Bloqueado (Blocked - suspend):** El proceso está en memoria secundaria y esperando un evento.
- **Bloqueado – Listo (Ready - blocked):** El proceso está en memoria secundaria y disponible para ejecución cuando se lo traiga a MC.

¹ Swapping significa la acción del intercambio entre dos niveles de memoria.

En un sistema con Memoria Virtual, es posible ejecutar un proceso que esté sólo parcialmente en MC (si el proceso está en MV, se trae la porción requerida a MC). Sin embargo, la MV no reemplaza el swapping, ya que el sistema de MV puede colapsar si hay demasiados procesos activos.

- En los siguientes diagramas vemos los 2 modelos planteados y en cada uno de ellos las transiciones de estados posibles. No son todos explicitados ya que solo se incluyen los nuevos.

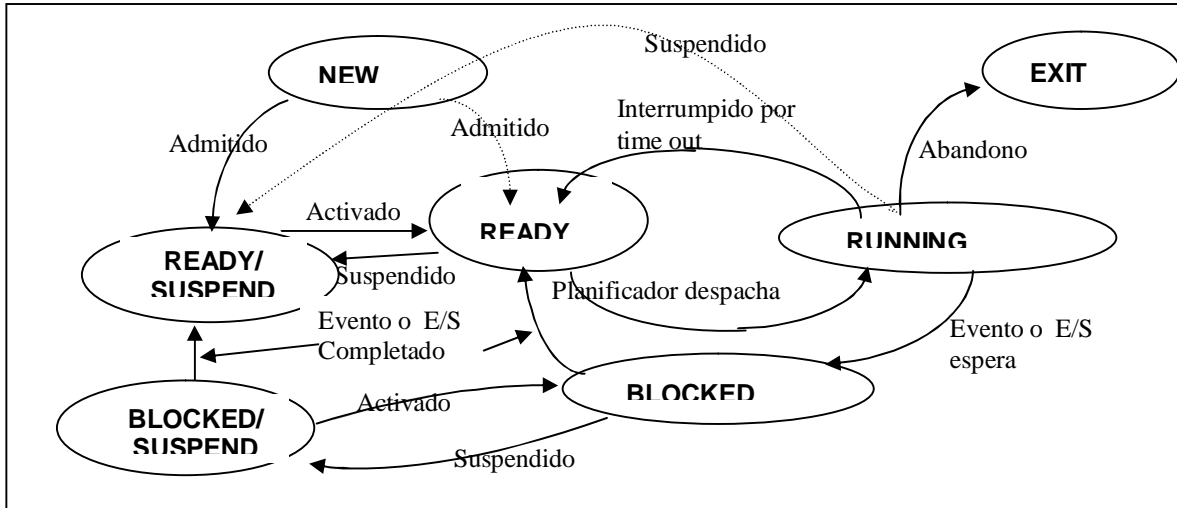


Fig. 2.13 Modelo de 7 estados.

- Blocked ® blocked - suspend**: Ocurre cuando un proceso bloqueado es llevado a disco para hacer espacio en memoria para otros procesos. También puede hacerse si hay procesos listos, pero el S.O. determina que el procesos en ejecución o un proceso listo requiere más MC para mantener el rendimiento.
- Blocked - suspend ® ready - suspend**: Esto pasa cuando ocurre el evento por el que estaba esperando el proceso. Esto requiere que la información del proceso suspendido esté accesible al S.O.
- Ready - suspend ® suspend**: Puede ocurrir si es la única manera de liberar un bloque grande de MC. También puede pasar si el S.O. prefiere suspender un proceso listo de menor prioridad a suspender uno bloqueado de mayor prioridad si éste considera que el "blocked" pasará a "ready" pronto.
- New ® ready - suspend y new ® ready**: Cuando un proceso es creado, éste puede ser agregado a la cola de listos o a la de "ready- suspend".
- Running ® ready - suspend**: Ocurre cuando a un proceso se le acaba el tiempo de ejecución ininterrumpida, o cuando un proceso de mayor prioridad que estaba en la cola "blocked- suspend" recién se ha desbloqueado.
- Varios ® exit**: Generalmente un proceso finaliza mientras está corriendo o si ocurrió algún error. También puede finalizar a pedido de su proceso padre o bien si éste ha sido finalizado.

e) Modelo de 9 estados.

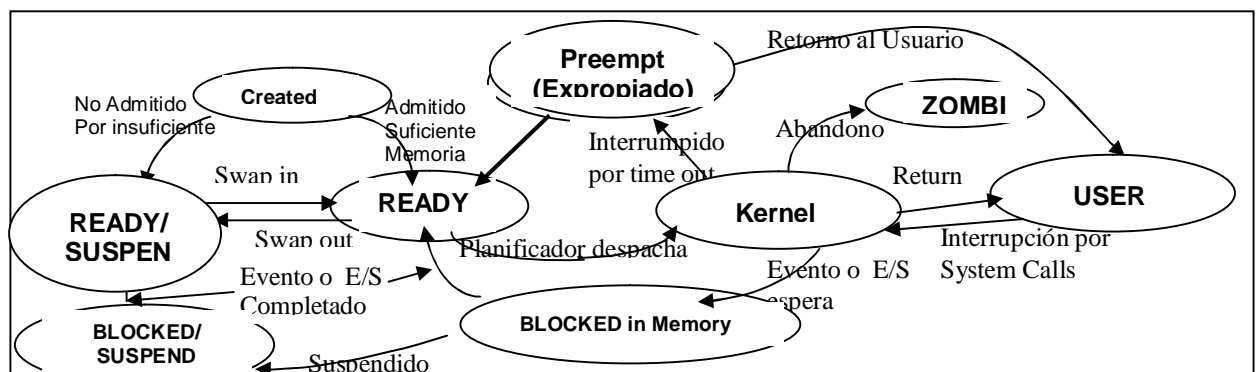


Fig. 2.14 Modelo de 9 estados de UNIX.

En particular UNÍX se tiene un modelo interesante de 9 estados posibles en que un proceso puede estar desde que se crea mediante la ejecución de una sentencia `fork()`.

f) Razones de un cambio de estado de Proceso

Las razones pueden ser múltiples. Se destacan particularmente 3, a saber: por interrupciones, por una excepción (trap) y por una llamada al sistema.

En una interrupción ordinaria, el control se transfiere primero a un gestor de interrupciones, quien lleva a cabo algunas tareas básicas y después salta a la rutina del S.O. que atenderá a la interrupción. Las interrupciones pueden ser de tres tipos: Por reloj cuando se expira un dado tiempo, por Entrada/Salida entonces el S.O. determina que acción a seguir y por falla de página en memoria cuando se utiliza memoria virtual. Las excepciones se deben a la ejecución de la instrucción corriente en que se da un tratamiento de error o una condición de excepción. Las llamadas al sistema se refieren a requerimientos explícitos de un servicio que necesita el proceso para continuar su ejecución, entonces llama a una función del S.O..

g) Cambios de contexto o de ejecución.

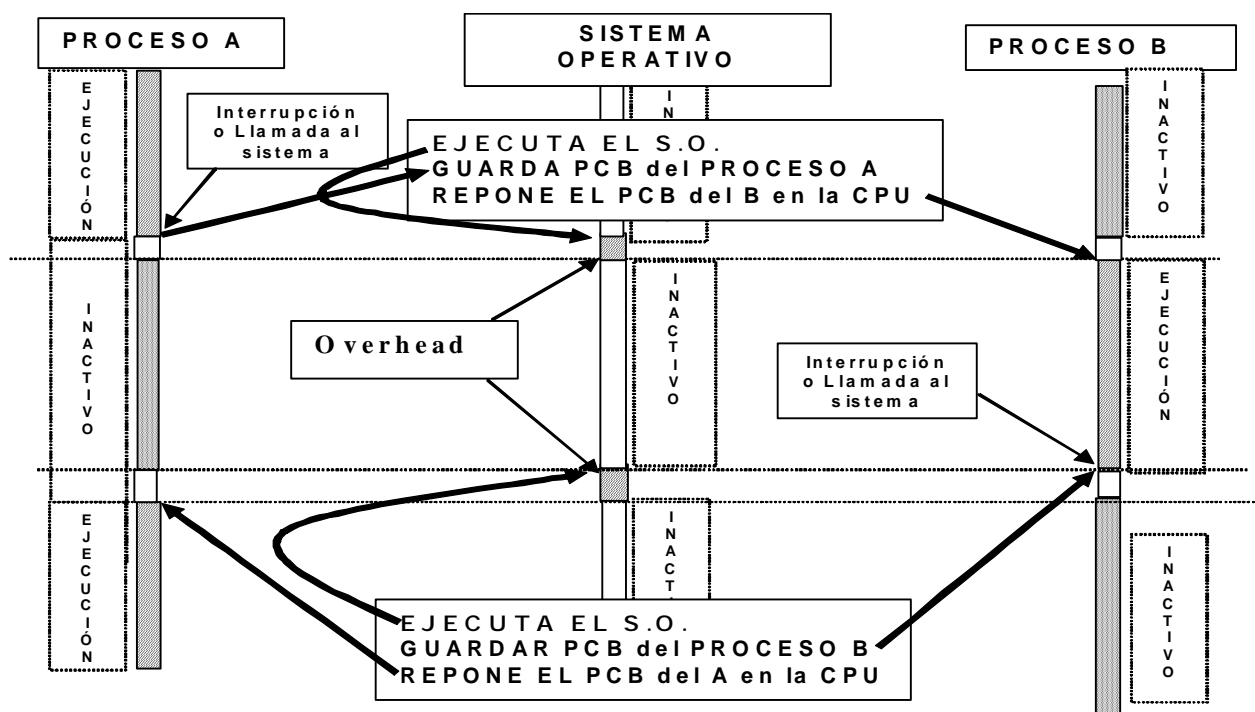


Fig. 2.15. Cambio de contexto en un Sistema Monoprocesador

Un concepto muy importante que incluye el concepto de proceso, es el denominado **cambio de contexto**. Este concepto está directamente relacionado con la idea de los sistemas de tiempo compartido (Time sharing System). En estos sistemas, existen muchos procesos ejecutándose concurrentemente, el sistema operativo se encarga de asignar a cada uno de los procesos un tiempo fijo de ejecución, cuando el proceso no termina de ejecutarse en ese tiempo el S.O. debe guardar la información que le corresponde a dicho proceso para poder recuperarla posteriormente cuando le asigne otra cantidad de tiempo de ejecución.

Se denomina **conmutación o cambio de contexto** al mecanismo mediante el cual el sistema almacena la información del proceso que se está ejecutando y recupera la información del proceso que ejecutará inmediatamente a continuación. Cuando el Sistema Operativo entrega a la CPU un nuevo proceso, debe guardar el estado del proceso que estaba ejecutando, y cargar el estado del nuevo.

El estado de un proceso comprende el Program Counter, y los registros de la CPU. Además, si se usan las técnicas de administración de memoria que veremos más adelante, hay más información involucrada. Este cambio, que demora de unos pocos a varios microsegundos, dependiendo de la velocidad del procesador, es puro sobrecarga (**overhead**²), puesto que entretanto la CPU no hace trabajo útil (ningún proceso avanza). Considerando que la CPU hace varios cambios de contexto en un segundo, su costo es relativamente alto.

Algunos procesadores tienen instrucciones especiales para guardar todos los registros de una vez. Otros tienen varios conjuntos de registros, de manera que un cambio de contexto se hace simplemente cambiando el puntero al conjunto actual de registros. El problema es que si hay más procesos que conjuntos de registros, igual hay que usar la memoria. Como sea, los cambios de contexto involucran un costo (overhead) importante, que hay que tener en cuenta. Para evitar el cambio de contexto y su costo en los S.O. modernos trabajan con Procesos Livianos o Hilos (Threads)

2.4.6. Las operaciones sobre un proceso

Operaciones sobre procesos. Los Sistemas Operativos actuales poseen una serie de funciones cuyo objetivo es el de la manipulación de los procesos. Las operaciones que se pueden hacer sobre un proceso son las siguientes:

- **Crear** el proceso (ya fue explicado)
- **Destruir o Eliminar** un proceso (se trata de la orden de eliminación o muerte del proceso con la cual el Sistema Operativo destruye su PCB).
- **Suspender** un proceso. Es un proceso de alta prioridad que paraliza un proceso que puede ser reanudado posteriormente. Suele utilizarse en ocasiones de mal funcionamiento o sobrecarga del sistema, por la falta de recursos.
- **Reanudar** un proceso. Trata de activar un proceso que ha sido previamente suspendido.
- **Cambiar la prioridad** de un proceso. Modifica su orden de ejecución.
- **Temporizar la ejecución** de un proceso. Hace que un determinado proceso se ejecute cada cierto tiempo (milisegundos, segundos, minutos,...) por etapas de una sola vez, pero transcurrido un periodo de tiempo fijo.
- **Despertar** un proceso. Es una forma de desbloquear un proceso que había sido bloqueado previamente por temporización o cualquier otra causa.
- **Prioridades.** Todo proceso por sus características e importancia lleva aparejadas unas determinadas necesidades de ejecución en cuanto a urgencia y asignación de recursos. Las prioridades según los Sistemas Operativos se pueden clasificar del siguiente modo:
 - Asignadas por el Sistema Operativo. Se trata de prioridades que son asignadas a un proceso en el momento de comenzar su ejecución y dependen fundamentalmente de los privilegios de su propietario y del modo de ejecución.
 - Asignadas por el propietario.
 - Estáticas.
 - Dinámicas.

2.5. El control de un Proceso

El S.O. es una entidad que administra el uso que hacen los procesos de los recursos del sistema.

Durante el curso de su ejecución cada proceso necesita tener acceso a ciertos recursos del sistema, entre los que se encuentran la memoria central, el procesador, los archivos y los dispositivos de E/S, entre otros.

Estructuras de control del SO:

Si el S.O. va administrar los procesos y los recursos, entonces tiene que disponer de información sobre el estado actual de cada proceso y de cada recurso. El método universal para obtener esta información sobre cada entidad es el uso de tablas. Estas tablas se encadenan mediante punteros.

² Overhead es el tiempo de ejecución del S.O. para realizar los servicios que solicitan los procesos. Es un tiempo de uso de CPU no productivo de ahí que se define como una sobrecarga en la ejecución.

- a) Las tablas de memoria se utilizan para controlar los espacios de direccionamiento de la memoria central y la virtual. Estas tablas de memoria deben incluir la siguiente información:
 - § La asignación de memoria central a los procesos.
 - § La asignación de memoria secundaria a los procesos.
 - § Cualquier atributo de protección de segmentos de memoria central o virtual.
 - § Cualquier información necesaria para gestionar la memoria virtual.
- b) Las Tablas de E/S son utilizadas por el S.O. para administrar los dispositivos y canales de E/S del sistema.
- c) Tabla de archivos, las cuales ofrecen información sobre la existencia de los archivos, su posición en la memoria secundaria, su estado actual y otros atributos.
- d) Tablas del sistema de procesos.

Todas estas tablas deben ser accesibles por medio del S.O. y por tanto, están sujetas a la gestión de memoria donde se localizan.

Cuando se inicializa el S.O., este debe tener acceso a algunos datos de configuración que definan el entorno básico y en algunos casos estos datos deben crearse fuera del S.O., con asistencia humana.

2.5.1. Estructuras de control de proceso:

a) Ubicación de los procesos.

Un proceso puede incluir un programa o un conjunto de programas a ser ejecutados. Un proceso constará al menos, de suficiente memoria para albergar los programas y los datos del proceso. Estos atributos se conocen como el bloque de control del proceso antes mencionado.

Para que el S.O. pueda administrar a un proceso, requiere que la información a usar por el S.O. debe mantenerse en memoria central. Cuando un proceso se descarga al disco, parte de su PCB permanece en memoria central. De esta forma, el S.O. puede seguir la pista de las partes del PCB de cada proceso que queda en memoria central.

El S.O. solo tiene que traer una parte de un proceso en particular. De ese modo que, en un momento dado, una parte del PCB del proceso puede estar en la memoria central y el resto en la memoria secundaria. Por lo tanto las tablas de procesos deben mostrar la ubicación de cada segmento o página del proceso tanto en memoria como en el disco.

Hay una tabla principal de procesos que administra el S.O. con una entrada para cada proceso.

b) Atributos del proceso:

En un sistema multiprogramado sofisticado se necesita una gran cantidad de información de cada proceso para su administración, se puede considerar que esa información reside en el bloque de control del proceso.

Se puede agrupar la información de los bloques de control del proceso en tres categorías generales siguientes

- § Identificación del proceso.
- § Información de estado del proceso
- § Información de control del proceso.

Con respecto a la identificación del proceso, el S.O. le asigna un identificador numérico único llamado PID. Si no hay indicadores numéricos, entonces debe haber una correspondencia que permita al S.O. ubicar las tablas apropiadas a partir del identificador de proceso. Es útil por ejemplo, cuando se comunican unos con otros, se utiliza el identificador de proceso para informar al S.O. el destino de cada comunicación en particular. También se puede tener un identificador de usuario que indica quien es el usuario responsable del trabajo.

La información del estado del procesador esta formada por el contenido de los registros del procesador. Cuando se interrumpe el proceso, toda la información de los registros debe salvarse de forma que pueda restaurarse sobre el procesador cuando el proceso reanude su ejecución.

Los registros del procesador visibles para el usuario son aquellas accesibles para los programas de usuario y que se usan para almacenar datos temporales. Se emplean varios registros de control y de estado para controlar la operación del procesador: la mayoría no son visibles para los usuarios.

Normalmente existe una PSW (Processor Status Word) que contiene información de estado de programa y los códigos de condición junto a otra información de estado. Los registros de pila (Stack Pointer) proporcionan los punteros a las pilas empleadas por el S.O. para controlar la ejecución de los programas y de las interrupciones.

La información de control de proceso, es una información adicional necesaria para que el S.O. controle y coordine los diferentes procesos activos. El bloque de control de proceso puede contener información de estructuración, incluyendo números que permiten enlazar los bloques de control de procesos, Mensajes y Banderas (Flags).

c) El papel del bloque de control:

El bloque de control de proceso, es la estructura de datos central y más importante de un S.O.. Cada PCB contiene toda la información de un proceso que necesita el S.O.. Los bloques son leídos o modificados por casi todos los módulos del S.O..

Se presentan algunos problemas como por ejemplo: un error en una rutina, como la de tratamiento de interrupciones, puede dañar los PCBs, lo que destruiría la capacidad del sistema para administrar y controlar los procesos. Otro tipo de problema puede ser un cambio de diseño en la estructura o en la semántica del bloque de control de procesos que podría afectar a varios módulos del S.O.. Estos problemas se pueden resolver exigiendo a todas las rutinas o funciones del S.O. que pasen a través de una rutina de manejo, cuya única tarea sería la de proteger los PCBs y que se constituirá en el único habilitado para leer y escribir en esos bloques. Obviamente esto genera una sobrecarga (Overhead) que hace más lenta la ejecución.

2.5.2. Control de procesos:

a) Modos de ejecución.

La mayoría de los procesadores dan soporte, por lo menos para dos modos de ejecución. Ciertas instrucciones pueden ejecutarse solo en modo privilegiado, otras lo hacen en User. Además se puede acceder a ciertas regiones de memoria solo en el modo privilegiado. El modo privilegiado, se conoce como modo Kernel, o Supervisor, o Master, o modo de sistema y se ejecutan solo instrucciones del S.O. o del usuario con ciertas protecciones, mientras en el modo usuario o común, en que se ejecutan los programas de usuario.

Para que el procesador sepa en que modo va a ejecutar normalmente existe un bit en la PSW que indica el modo de ejecución. Cuando se trata de un cambio de modo, se lleva a cabo ejecutando una instrucción privilegiada que hace el cambio de modo.

b) Creación de un proceso:

Decíamos que cuando el S.O. decide crear un proceso se pueden seguir los siguientes pasos:

1. Asignar un PID al nuevo proceso y crear una nueva entrada en la tabla de procesos
2. Asignar espacio en Memoria Central para el proceso y se debe asignar el espacio para el bloque de control del proceso.
3. Se debe inicializar el bloque de control del proceso.
4. Se deben establecer los enlaces apropiados con otras estructuras de datos.
5. Puede haber estructuras que cambiar o ampliar.

c) Cambio de procesos sobre el procesador:

Un cambio de proceso puede producirse en cualquier momento que el S.O. haya tomado control a partir del proceso que esta actualmente ejecutándose. Para el cambio, primero se va a tener en cuenta las interrupciones del sistema. Puede haber dos tipos de interrupciones, una es originada por un suceso independiente al proceso que se esta ejecutando, el otro tipo tiene que ver con un error o excepción generada dentro del proceso.

d) Cambio de contexto

Se produce un Context Switch solo en aquellos procesos pesados, o sea que no se basan en Threads. En el caso de los procesos pesados, si hay alguna interrupción pendiente el S.O. hace lo siguiente:

1. Salva el contexto del programa que actualmente se está ejecutando en el snack (memoria central)
2. Asigna al controlador de programa el valor de la dirección de comienzo del programa de tratamiento de interrupciones.

El contexto incluye cualquier información que pueda alterarse por la ejecución de la rutina de tratamiento de la interrupción y que pueda ser necesaria para reanudar el programa que fue interrumpido. Cuando se produce una interrupción, un trap o una llamada del kernel, el procesador se pasa al modo kernel y el control pasa al S.O.. Con tal fin, se salva el contexto del procesador y tiene lugar un cambio de contexto mediante una rutina del S.O.. Sin embargo, la ejecución continúa dentro del proceso de usuario en curso. De esta manera no se ha llevado a cabo un **cambio de proceso**, sino un cambio de contexto dentro del mismo.

e) Cambio de estado de los procesos:

Puede producirse un cambio de contexto sin cambiar el estado del proceso que esta actualmente en estado de ejecución. Los cambios involucrados en el cambio de procesos son los siguientes:

1. Salvar el contexto del procesador.
2. Actualiza el bloque de control del proceso que estaba en ejecución.
3. Mover el bloque de control del proceso a la cola apropiada en el Stack.
4. Seleccionar otro proceso para la ejecución
5. Actualizar el bloque de control del proceso seleccionado
6. Actualizar las estructuras de datos de gestión de memoria, o de E/S, o de otro proceso.
7. Restaurar el contexto del procesador a aquel que no existía en el momento en el que el proceso seleccionado dejó por última vez el estado de ejecución cargando los valores previos del contador de programa y de otros registros.
8. Reiniciar la ejecución.

2.5.3. Ejecución del SO

Cada función del S.O. también ejecuta como procesos.

a) Núcleo (Kernel) fuera del proceso de usuario:

Un enfoque bastante tradicional, es ejecutar el núcleo del S.O. fuera cualquier proceso.

Cuando el proceso en ejecución es interrumpido o hace una llamada del sistema, se salva el contexto de ejecución del proceso y se pasa el control al Kernel. El S.O. puede llevar a cabo cualquier función deseada y al concluir la función repone el contexto del proceso interrumpido para reanudar su ejecución o busca uno de mayor prioridad que el interrumpido.

b) Ejecución dentro de los procesos de usuario:

En este caso, se ejecuta todas las funciones del S.O. en un contexto de un proceso de usuario. El enfoque es que el S.O. es principalmente una colección de rutinas o funciones que el usuario llama para llevar a cabo diferentes tareas que son ejecutadas dentro del entorno del proceso usuario. El Stack propio del Kernel se utiliza para gestionar las llamadas y los retornos, mientras que el proceso esté en el "modo Kernel", el código de los datos y del S.O. está en el espacio de direcciones compartidas y son compartidas por todos los procesos de usuario.

Debido al concepto de modo de usuario y modo de Kernel, el usuario no puede ejecutar a las rutinas del S.O., mientras estas estén ejecutando en un servicio para el proceso de usuario.

c) S.O. basado en procesos:

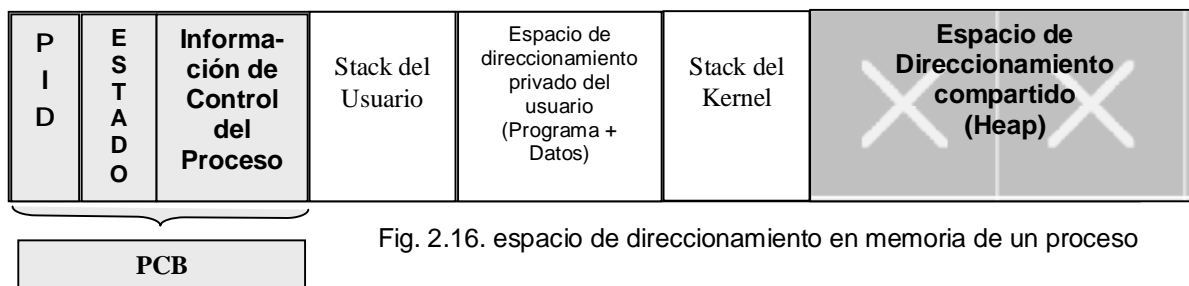


Fig. 2.16. espacio de direccionamiento en memoria de un proceso

Esta alternativa es interpretar al S.O. como una colección de procesos del sistema. Las funciones más importantes del núcleo se organizan en procesos separados. Puede haber una pequeña cantidad de código del Kernel que ejecutará en modo Kernel y no en modo user.

El espacio de direccionamiento en memoria será el de la figura 2.16

2.6. Tipos de Procesos

Procesos disjuntos

Los procesos de este tipo, también llamados independientes, son aquellos que solo tienen variables locales o comparten variables globales sin modificarlas, es decir la intersección de sus PCBs es vacía. La ejecución de uno de ellos no afecta al otro, por lo que pueden hacerlo en paralelo.

Procesos Concurrentes

También llamados procesos paralelos, se da cuando pueden usar simultáneamente un recurso. Si el recurso es modificado entonces es **crítico** y se usa mutua exclusión para sincronizar su uso.

También se denominan procesos concurrentes cuando sus ejecuciones se superponen en el tiempo.

Existen razones para la ejecución concurrente. Generalmente se debe a:

- Información compartida.
- Acelerar los cálculos.
- Modularidad
- Comodidad

Para que todo esto se cumpla sin problemas se requiere que exista la cooperación entre procesos, y se necesita un mecanismo para la sincronización y comunicación entre los procesos que se estudiará en el Módulo 4.

Procesos Interactuantes

Los procesos de este tipo son procesos concurrentes que comparten variables globales. Es decir dos procesos concurrentes están relacionados si la intersección de sus PCB no es vacía: uno de ellos puede hacer que un recurso quede accesible al otro, o privarlo de este recurso, es decir, uno de los procesos puede hacer cambiar el estado del otro.

Procesos reentrantes

Esta clasificación tiene lugar según el código del programa. Este tipo de procesos cuenta con código puro y no tienen asociados datos. Otro tipo de procesos que cumple con esta clasificación son los procesos reutilizables, que se describen en la próxima sección.

Procesos reutilizables

Este tipo de procesos, junto con los procesos reentrantes, dependen del código de programa que ejecuten, como se mencionó anteriormente. Los procesos reutilizables son los que pueden ser ejecutados con diferentes juegos de datos.

Otra forma de clasificar los procesos es por el tamaño de su PCB, en Procesos Pesados y Procesos Livianos o Threads.

2.7 Hilo o Hebra (Threads)

En la mayoría de los Sistemas Operativos tradicionales, cada proceso tiene un espacio de direcciones y un único hilo de control, sin embargo con frecuencia existen situaciones en donde se desea tener varios hilos de control que compartan un único espacio de direcciones, y se ejecutan de manera casi paralela, como si fuesen de hecho procesos independientes (excepto por el espacio de direcciones compartido).

Un thread es llamado también proceso liviano (**lightweight process**) debido a que mantiene la estructura de un proceso con su PCB, pero dispone de otra estructura más pequeña con un **TID** (Thread Identifier) además de un conjunto de información reducida del PCB, lo que hace que su ejecución sea más eficientemente. En sí un proceso es igual a uno o más tareas (**tasks**) cada una con su Hilo (**thread**) asociado.

Un Thread (hilo), es una unidad elemental de uso del procesador, y cada hilo posee un identificador, un Contador de Programa (PC - Program Counter), un juego de Registros del procesador (Register Set) y una Pila (Stack). En muchos sentidos los hilos son como pequeños miniprocesos.

Cada thread se ejecuta en forma estrictamente secuencial compartiendo el procesador de la misma forma que lo hacen los procesos, solo en un multiprocesador o en una arquitectura acorde se pueden realizar en paralelo. Los hilos pueden crear hilos hijos y se pueden bloquear en espera de llamadas al sistema, al igual que los procesos regulares. Mientras un hilo está bloqueado se puede ejecutar otro hilo del mismo proceso. Puesto que cada hilo tiene acceso a cada dirección virtual (comparten un mismo espacio de direccionamiento), un hilo puede leer, escribir o limpiar la pila de otro hilo.

No existe protección entre los hilos debido a que es imposible y no es necesario, ya que generalmente cooperan entre sí la mayoría de las veces. Aparte del espacio de direcciones, comparten el mismo conjunto de archivos abiertos, procesos hijos, relojes, señales, etc.

En muchos aspectos los procesos livianos son similares a los procesos pesados, dado que comparten el tiempo del procesador, y a lo sumo un thread está activo (ejecutando) a la vez, en un monoprocesador. Los otros pueden estar listos o bloqueados. Pero los procesos pesados son independientes, y el Sistema Operativo debe proteger a unos de otros, lo que genera una sobrecarga (overhead) en la ejecución. Los procesos livianos dentro de un mismo proceso pesado no son independientes, pues cualquiera puede acceder toda la memoria correspondiente al proceso pesado.

Entre sus características se destacan principalmente que los threads comparten procesador. Se ejecutan secuencialmente, pueden crear hijos y al ser parte del mismo proceso no son independientes, es decir que todos pueden acceder a una dirección de pila de otro thread.

Ventajas con respecto a los procesos:

Los beneficios de usar threads se observan en el desempeño del sistema dado que toma menos tiempo crearlos y eliminarlos (pues se crean dentro del contexto de un proceso). Por otro lado toma menos tiempo realizar el cambio sobre el procesador para procesar un nuevo thread. Comparten un mismo espacio de memoria y datos entre sí debido a que forman parte de un mismo proceso. No se efectúa un context switch completo sobre el procesador sino una pequeña parte del mismo perteneciente al Thread.

2.7.1. Implementación de hilos (Threads)

Los hilos pueden ser implementados en tres niveles por la forma en que son generados y tratados:

1. Nivel usuario (ULT – User Level Thread)
2. Nivel kernel (KLT – Kernel Level Thread)
3. Nivel de Proceso (PLT – Process Level Thread)
4. Mixtos (ULT-KLT)

a) Hilos a Nivel de Usuario (ULT):

Todo el trabajo del hilo es manejado por la aplicación, el kernel ni se entera de la existencia de los hilos. Cualquier aplicación puede ser programada para ser multithreaded mediante el uso de **threads library** (paquete de rutinas para ULT en el compilador). Las bibliotecas contienen código para crear y destruir hilos, pasar mensajes y datos entre hilos, ejecución planificada de hilos y para guardar y restablecer contextos de hilos. Entonces la generación de los ULT se hace en el momento de compilación y no se requiere la intervención del Kernel. Todo el manejo de los threads queda a cargo de la aplicación, el S.O. no está al tanto de la existencia de los thread. Para que la aplicación pueda manejar threads, se usa una biblioteca para las rutinas de este fin. Todo el trabajo de la biblioteca se hace en modo usuario sin que el del Kernel intervenga y el mismo sigue planificando procesos.

Por default, una aplicación comienza con un solo hilo y empieza corriendo en ese hilo. Esta aplicación y su hilo son asignados al espacio de direccionamiento de un solo proceso manejado por el kernel. En cualquier instante, mientras está ejecutando, puede hacer lo que se llama **spawn**³, que es crear un nuevo hilo para correr dentro del mismo proceso, esto se hace invocando la función **spawn()** de la biblioteca. Ésta, crea una estructura de datos para el nuevo hilo y luego a través de un algoritmo de planificación, le pasa el control a uno de los hilos que están en estado listo. Cuando el control pasa al planificador (la biblioteca), se guarda el contexto del hilo actual y cuando el control pasa del planificador al hilo, se restablece el contexto de ese hilo.

³ Spawn conceptualmente es reproducir como hijos o sea producir nuevos threads desde uno padre.

El kernel no se entera de lo anterior, continúa con la planificación del proceso como unidad y le asigna un solo estado de ejecución. (Listo, corriendo, etc.).

Ventajas:

1. El cambio de hilo sobre el procesador, no requiere el modo kernel, porque todas las estructuras de datos están dentro del espacio usuario. Es más, el proceso no cambia al modo kernel para manejar el hilo o sea que al no tener que cambiar de modo usuario a Kernel para hacer el manejo de los threads es una ventaja, por el poco overhead,
2. El algoritmo de planificación puede ser adaptado sin molestar la planificación del S.O. debido a que se puede especificar a la biblioteca que algoritmo de planificación de threads se va usar.
3. ULT puede correr en cualquier SO.
4. Es muy rápido en la ejecución

Desventajas:

1. En un S.O. típico, la mayoría de los system call son bloqueantes. Cuando un hilo ejecuta un system call no sólo se bloquea ese hilo, sino que también se bloquean todos los hilos del proceso., o sea que si se bloquea un thread, se bloquea todo el proceso.
2. En una estrategia pura de ULT, una aplicación multithreaded no puede tomar ventaja del multiprocesamiento. Un kernel asigna un proceso a sólo un procesador por vez y no se aprovechan los procesadores múltiples, no pueden correr dos threads de un mismo proceso en distintos procesadores simultáneamente

Solución para las desventajas:

Para los casos 1 y 2: escribiendo una aplicación como múltiples procesos en lugar de múltiples hilos. Pero así se pierde la principal ventaja de los hilos: cada cambio se convierte en cambio de procesos en lugar de cambio de hilos.

Para 1: usar una técnica llamada **jacketing**, cuyo objetivo es convertir un system call bloqueante en un system call no bloqueante.

b) Hilos a nivel de Kernel (KLT):

Todo el trabajo de manejo de hilos es hecho por el kernel. No hay código de manejo de hilo en el área de la aplicación. Cualquier aplicación puede ser programada para ser multithreaded. Todos los hilos dentro de una aplicación son soportados dentro de un solo proceso. El kernel mantiene la información de contexto para el proceso e individualmente para los hilos dentro del proceso. No tiene las desventajas que tenían los threads nivel usuario.

Ventajas:

1. Simultáneamente el kernel puede planificar múltiples hilos del mismo proceso en múltiples procesadores.
2. Si un hilo de un proceso se bloquea, el kernel puede planificar otro hilo del mismo proceso.
3. Las rutinas mismas del kernel pueden ser multithreaded.

Desventaja:

La transferencia de control de un hilo a otro dentro del mismo proceso le requiere al kernel un cambio de modo. Esto genera overhead extra.

c) Combinación de ULT- KLT

Los threads son creados en espacio de usuario. Los múltiples threads de una aplicación son mapeados en un número igual o inferior de threads nivel-Kernel. Si está bien diseñado, este enfoque logra combinar las ventajas de los dos anteriores.

Ejemplo: el S.O. SOLARIS de Sun.

Múltiples hilos dentro de una misma aplicación pueden correr en paralelo en múltiples procesadores y un system call bloqueante no necesariamente bloquea todo el proceso.

d) Hilos a nivel de Proceso.

Otra solución es crear los hilos por el proceso en el momento de su ejecución. Es la peor solución dado que es muy lento. Como ejemplo proponemos la siguiente tabla tomada de una publicación de la ACM⁴ en que las unidades de tiempo están en μs (Microsegundo = millonésimo de segundo). Como observación importante los Threads creados por las bibliotecas son unos 30 veces más rápidos que los soportados por el Kernel y unas 300 veces con respecto a los del proceso. Obviamente que la mayoría de las soluciones solo usan ULT y KLT (o la combinación de ambos) para su implementación.

Operación	ULT	KLT	PLT
Null Fork	34	948	11.300
Signal Wait	37	441	1.840

Tabla 2.2 Tiempo de latencia en operaciones de Threads.

e) Relación entre hilos y procesos.

Hilos:Procesos	Descripción	Ejemplo
1:1	Cada hilo es un único proceso con su propio espacio y recursos	La mayoría de las implementaciones UNIX
M:1	Un proceso define su espacio y recursos. Múltiples hilos pueden ser creados y ejecutados dentro de un proceso.	WinNT, Solaris, OS/2, MACH
1:M	Un hilo puede pasar del entorno un proceso a otro. Esto le permite al hilo ser fácilmente movido entre distintos sistemas.	Ra, Emerald
M : M	Combina atributos de M:1 y casos de 1:M	TRIX

Tabla 2.3 Relación entre hilos y procesos

Relación muchos a muchos (M:M).

En el S.O. TRIX, un dominio es una entidad estática, consistente en un espacio de direccionamiento y “puertos” a través de los cuales los mensajes pueden ser enviados y recibidos. Un hilo es sólo un camino de ejecución, con una pila, un estado de procesador y la información de planificación.

El uso de un solo hilo en múltiples dominios parece motivado por el deseo de proveer al programador de herramientas estructurales. Por ejemplo, consideremos un programa que usa un subprograma de Entrada/Salida (I/O). En un entorno multiprogramación, el programa principal podría generar un nuevo proceso para manejar la I/O y luego continuar la ejecución. Sin embargo, si el progreso del programa principal depende del resultado de la operación de I/O, entonces el programa principal tendrá que esperar que el programa de I/O termine. Hay varias formas de implementar esta aplicación:

1. El programa entero puede ser implementado como un solo proceso. Esta es la solución razonable y directa. Hay inconvenientes relacionados con el manejo de memoria. El proceso (todo) para ejecutarse eficientemente puede requerir un considerable espacio en Memoria Central, mientras que el subprograma de I/O puede requerir relativamente menor espacio para el buffer y para manejar la poca cantidad de código de programa. Debido a que el programa de I/O se ejecuta en el espacio del programa principal, entonces el proceso entero debe permanecer en Memoria Central durante la operación de I/O. Este manejo de memoria también existiría si el programa principal y el subprograma I/O fueran implementados como dos hilos en el mismo espacio.
2. El programa principal y el subprograma I/O pueden ser implementados como dos procesos separados.
3. Tratar al programa principal y al subprograma I/O como un solo hilo. Sin embargo, podría crearse un dominio para el programa principal y otro para el subprograma I/O. El S.O. puede manejar dos espacios independientemente y no se cae en el overhead. Aun más, el espacio usado por el subprograma I/O puede ser compartido con otros programas I/O.

Las experiencias en TRIX indican que la mejor es la 3.

⁴ Anderson T., Versad B., Lazowska E., and Levy H. “Scheduler Activations: Effective Kernel support for the User-Level Management of Parallelism” ACM Transactions on computer Systems, Febrero de 1992.
ACM son las siglas Association for Computer Machinery de Estados Unidos.

2.7.2. La creación de los Threads

Para la ejecución de un proceso se arma lo que se llama una máquina virtual que vincula al proceso con todos los recursos físicos que necesita. Este entorno es provisto por el kernel y el S.O. Cada proceso es dividido en un conjunto de instrucciones más pequeñas que ejecutan tareas llamadas **task** las cuales tienen asociadas un thread (hilo) de procesamiento. Los procesos, entre otros atributos, contienen variables que graban el número y tipo de operaciones realizadas de entrada/salida y otras variables que señalan el tipo y número de operaciones realizadas en memoria virtual de cada threads que le pertenece a un dado proceso.

KLT: Para esto el S.O. utiliza uno de los system calls: **Create_thread()** pasándole los parámetros del contexto en el que se va a ejecutar dicho thread y generando un TCB (Thread Control Block similar al PCB pero mucho mas pequeño) que está compuesto por ciertos atributos como: **TID** (Thread **ID**entifier, similar a los PID de los procesos (valor único de identificación), Thread context (Conjunto de registros y ciertos datos que definen la ejecución), Prioridad, Suspension count (veces que pasó por el estado suspendido), punteros a otros threads, Estado de salida (Exit status, que indica la razón por la cual se elimina al thread) entre otros. Esta forma de implementación mediante System calls debe ser soportada por el Kernel. Los Kernels de los S.O. Mach y OS/2, proporciona llamadas al sistema similares al que existe para procesos; consume más tiempo en el cambio de hilos, porque lo hace mediante interrupciones.

ULT: Hay una segunda forma de crearlo y es mediante bibliotecas especiales dentro del compilador. Esto es, por encima del Kernel a través de conjuntos de llamadas de bibliotecas a nivel del usuario (Ej. el S.O. Andrew); no involucra al Kernel; son rápidos para cambiar entre los hilos soportados por el Kernel; cualquier llamada al S.O. causa que el proceso entero tenga que esperar (causa: el Kernel planifica solo procesos y no conoce hilos).

Los cambios de hilos no necesitan llamar al S.O. y causar interrupciones al Kernel. Es independiente del S.O., en estos casos el bloquear y cambiar a otro hilo es una solución razonable al problema de como un sistema servidor (server) maneja muchos procesos eficientemente. Tiene como desventaja, un hilo único a nivel usuario ejecutando un System Call causará a la tarea entera esperar hasta que el System call sea devuelto.

2.7.3. La ejecución de los threads

A lo largo del tiempo fue avanzando la forma de ejecución. Algunos Sistemas Operativos proveían un único thread por proceso. Hoy día se utiliza lo que se llama **Multithreading** que es la habilidad que tienen los sistemas modernos para dividir un proceso en varios threads. En el caso de los sistemas multiprocesamiento (que cuentan con dos o más procesadores) los threads se ejecutan simultáneamente (mejorando por supuesto el desempeño del sistema)

Los threads son ejecutados secuencialmente, es decir cuando pasan al estado ejecutando (running) toman uso del procesador hasta que terminan (a esto se lo llama modo non preemptive o no expropiativo) y luego se ejecuta el thread siguiente hasta terminar con una tarea (task).

Son varios los servicios que se manejan para la ejecución de los threads además del antes mencionado **Create_thread()**, hay otros como **Open_thread()**, **Query_thread_information()**, **Get_context()**, **Set Context()**, **Suspend()**, **Resume()** que hacen al procesamiento y con ello pasar al thread por diferentes estados.

2.7.4. Estado de los threads

Para la ejecución de los procesos y por supuesto de los threads, se arman diferentes políticas de planificación que tienen en cuenta prioridades, tiempo de procesamiento, orden de llegada al sistema, entre otras.

Dentro de los diferentes estados, los threads se encuentran en la parte correspondiente al **short term scheduler** (lo estudiaremos en el módulo siguiente). Entonces los estados en los que puede estar un thread son: **listo (Spawn)**, **bloqueado (Block)**, **ejecutando (Running)** o **terminado (Finish)**.

- **Listo:** Al crearse un nuevo thread se agenda para una futura ejecución en una cola de este estado. Es el **dispatcher** (conjunto de servicios necesarios para la ejecución, que en realidad forman parte del kernel) es el encargado de asignar el orden de ejecución de los threads.
- **Bloqueado:** Cuando un thread necesita esperar que finalice un evento, el Kernel los va a encolar en esta cola grabando los registros asociados, program counter y punteros al stack. El procesador puede atender una rutina de interrupción o comenzar a ejecutar otro thread de la cola de listo.

- **Ejecutando:** Cuando el kernel selecciona un thread y realiza un “switch” para poder hacer que este se ejecute. La ejecución se realiza hasta que se bloquea o termina.
- **Terminado:** Cuando a thread completa su ejecución, se elimina su contexto y su stack asociado.

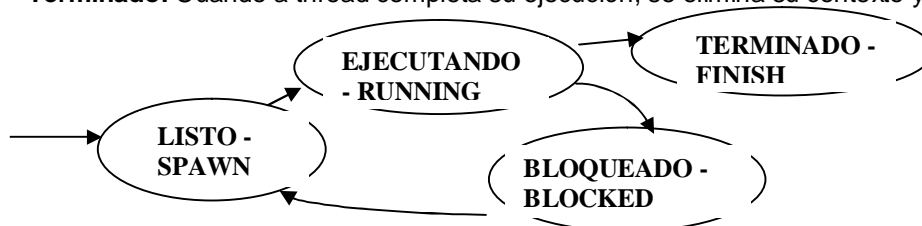


Fig. 2.17. Los estados posibles de un Hilo

2.7.5. Uso de los Hilos.

Los hilos se inventaron para permitir la combinación del paralelismo con la ejecución secuencial y el bloqueo de las llamadas al sistema. Existen 3 formas de organizar un proceso de muchos hilos en un Servidor (server).

- Estructura Servidor Trabajador
- Estructura En Equipo
- Estructura Entubamiento (Pipeline)

Estructura Servidor Trabajador. Existe un hilo en el servidor que lee las solicitudes de trabajo en un buzón del sistema, examina éstas y elige a un hilo trabajador inactivo y le envía la solicitud, la cual se realiza con frecuencia al colocarle un puntero al mensaje en una palabra especial asociada a cada hilo. El servidor despierta entonces al trabajador dormido (*un signal()* o *V()* al semáforo⁵ asociado).

El hilo verifica entonces si puede satisfacer la solicitud desde el bloque caché compartido, sino puede iniciar la operación correspondiente (por ejemplo podría lanzar una lectura al disco) se duerme nuevamente a la espera de la conclusión de ésta. Entonces, se llama al planificador para iniciar otro hilo, ya sea hilo servidor o trabajador. Otra posibilidad para esta estructura es que opere como un hilo único. Este esquema tendría el problema de que el hilo del servidor lanza un pedido y debe esperar hasta que éste se complete para lanzar el próximo (secuencial).

Una tercera posibilidad es ejecutar al servidor como una gran máquina de estado finito. Esta trata de no bloquear al único hilo mediante un registro del estado de la solicitud actual en una tabla, luego obtiene de ella el siguiente mensaje, que puede ser una solicitud de nuevo trabajo, o una respuesta de una operación anterior. En el caso del nuevo trabajo, éste comienza, en el otro caso se busca la información relevante en la tabla y se procesa la respuesta. Este modelo no puede ser usado en RPC⁶ puesto que las llamadas al sistema no son bloqueantes, es decir no se permite enviar un mensaje y bloquearse en espera de una respuesta.

En este esquema hay que guardar en forma explícita el estado del cómputo y restaurarlo en la tabla para cada mensaje enviado y recibido. Este método mejora el desempeño a través del paralelismo pero utiliza llamadas no bloqueantes y por lo tanto es difícil de programar.

Modelo	Características
Hilos	Paralelismo, llamadas al sistema bloqueantes
Servidor de un solo hilo	Sin paralelismo, llamadas al sistema bloqueantes.
Máquina de estado finito	Paralelismo, llamadas al sistema no bloqueantes

Tabla 2.4. Características principales en el uso de los hilos en la Estructura Servidor Trabajador

Estructura en Equipo. Todos los hilos son iguales y cada uno obtiene y procesa sus propias solicitudes. Cuando un hilo no puede manejar un trabajo por ser hilos especializados se puede utilizar una cola de trabajo. Esto implica que cada hilo verifique primero la cola de trabajo antes de mirar el buzón del sistema.

⁵ Semáforos se estudiará en el Módulo 4

⁶ RPC son las siglas de Remote Procedure Call o sea una llamada a un procedimiento remoto para que se ejecute en una máquina distante.

Estructura de Entubamiento (pipeline). El primer hilo genera ciertos datos y los transfiere al siguiente para su procesamiento. Los datos pasan de hilo en hilo y en cada etapa se lleva a cabo cierto procesamiento. Esta puede ser una buena opción para el modelo productor/consumidor, no así para los servidores de archivos.

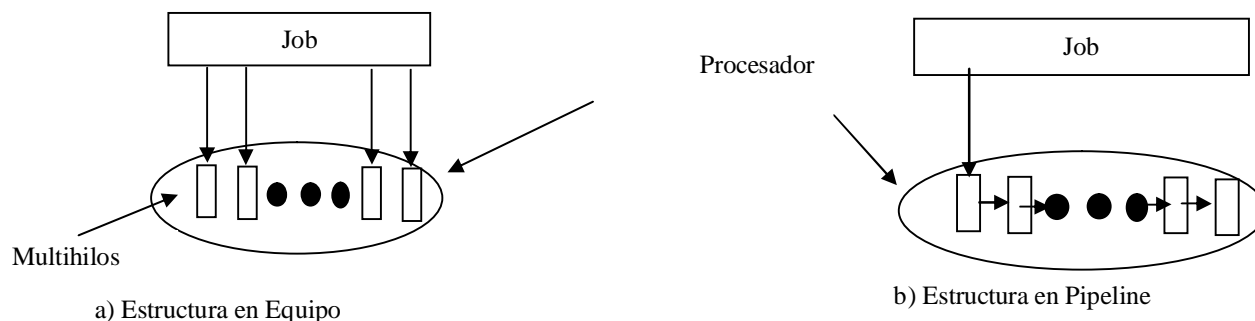


Fig. 2.18. modelos de implementación de Hilos

Otros usos de los hilos.

Por ejemplo, en un cliente los threads son usados para copiar un archivo a varios servidores, por ejemplo si un cliente quiere copiar un mismo archivo a varios servidores puede dedicar un hilo para que se comunique con cada uno de ellos. Otro uso de los hilos es el manejar las señales, como las interrupciones del teclado. Por ejemplo se dedica un hilo a esta tarea que permanece dormido y cuando se produce una interrupción el hilo se despierta y la procesa.

Existen aplicaciones que se prestan para ser programadas con el modelo de hilos, como es el caso de el problema de los productores-consumidores. Nótese que como comparten un buffer en común no funcionaría el hecho de tenerlos en procesos ajenos. Por último nótese la utilidad de los hilos en sistemas multiprocesadores en donde pueden realmente ejecutarse en forma paralela.

2.7.6. Aspectos del diseño de paquetes de Threads.

Un conjunto de primitivas disponibles relacionadas con los hilos para los usuarios se llama un paquete de hilos. Se pueden tener hilos estáticos o dinámicos. En un diseño estático se elige el número de hilos al escribir el programa, o durante su compilación. Cada uno de ellos tiene una pila fija (ULT). En el dinámico (KLT) se permite la creación y destrucción de los hilos en tiempo de ejecución. La llamada para la creación de un hilo establece el programa principal del hilo (como si fuera un puntero a un procedimiento o función) y un tamaño de pila, también otros parámetros, como ser prioridad de planificación. La llamada devuelve un identificador de hilo para ser usado en posteriores llamadas relacionadas con ese hilo. Entonces un proceso se inicia con un único hilo, pero puede crear el número necesario de ellos.

La terminación de los hilos puede ser de dos maneras, al terminar su trabajo o bien desde el exterior. En el caso de un servidor de archivos, los hilos se crean una vez iniciado el programa y nunca se eliminan. Ya que los hilos comparten un área de memoria común en el espacio de direccionamiento del proceso, el acceso a datos comunes se programa mediante regiones críticas que se implementan mediante el uso de semáforos, monitores u otras construcciones similares (que veremos más adelante como herramientas de sincronización de accesos a esas áreas compartidas).

Para un acceso único (de un solo thread a un recurso), se utiliza un semáforo de mutua exclusión llamado **mutex** sobre el cual se definen las operaciones de cierre (LOCK) y liberación (UNLOCK). Existe además una operación TRILOCK que para el cierre funciona igual que el Lock pero si el semáforo está cerrado en lugar de bloquear el hilo regresa un código de estado que indica falla. Se encuentra disponible también la variable de condición, similar en todo a la que se utiliza en monitores.

El código de un hilo consta, al igual que un proceso, de varios procedimientos o funciones, pudiendo tener variables locales, globales y variables del procedimiento o de la función. De éstas, las globales producen problemas, ya que el valor de una variable global puesto por un hilo que se duerme puede ser modificado por otro, creando así una incoherencia cuando se despierta. Las soluciones que se presentan son:

- a) **Prohibir las variables globales:** esto presenta conflictos con el software ya existente, como por ejemplo Unix.

- b) **Asignarle a cada hilo sus propias variables globales particulares.** Esto introduce un nuevo nivel de visibilidad, ya que las variables son visibles a todos los procedimientos de un hilo además de las variables visibles a un procedimiento específico y las visibles a todo el programa. Esta alternativa tiene el inconveniente de no ser posible de implementar en la mayoría de los lenguajes de programación. Una forma de hacerlo es asignar un bloque de memoria a las variables globales y transferirlas a cada procedimiento como un parámetro adicional.
- c) **Nuevos procedimientos en bibliotecas para crear, leer y escribir estas variables.** La creación de una variable global implicaría la asignación de un puntero en un espacio de almacenamiento dedicado a ese hilo de forma que solo él tiene acceso a la variable global definida.

Llamadas del sistema

Presentamos un resumen de las llamadas que pueden existir en los S.O. para el manejo de los hilos o threads:

- Llamadas de manejo de Threads: Create, Exit, Join, Detach
- Llamadas de Sincronización (Manejo de regiones críticas): Mutex_init, Mutex_destroy, Mutex_lock, Mutex_trylock, Mutex_unlock
- Llamadas de Condición Variables (usados para el bloqueo de recursos): Cond_init, Cond_destroy, Cond_wait, Cond_signal, Cond_broadcast
- Llamadas de Scheduling (administran las prioridades de los hilos): Setscheduler, Getscheduler, Setprio, Getprio
- Llamadas de Eliminación: Cancel, Set_cancel

2.7.7. Implementación de un paquete de Hilos.

Aquí trataremos la planificación de los hilos mediante distintos algoritmos, ya que el usuario puede especificar el algoritmo de planificación y establecer las prioridades en su caso. Existen dos formas de implementar un paquete de hilos:

- 1) Colocar todo el paquete de hilos en el espacio del usuario (el núcleo no conoce su existencia, caso ULT),
- 2) Colocar todo el paquete de hilos en el espacio del núcleo (el núcleo sabe de su existencia KLT)

Paquete de hilos en el espacio del usuario. La ventaja es que este modelo puede ser implantado en un Sistema Operativo que no tenga soporte para hilos. Los hilos se ejecutan arriba de un SISTEMA DE TIEMPO DE EJECUCIÓN (Run Time System) (Intérprete), que se encuentra en el espacio del usuario en contacto con el núcleo. Por ello, cuando un hilo ejecuta una llamada al sistema, se duerme, ejecuta una operación en un semáforo o mutex, o bien cualquier operación que pueda provocar su suspensión. Se llama a un procedimiento del sistema de tiempo de ejecución el cual verifica si debe suspender al hilo. En caso afirmativo almacena los registros del hilo (propios) en una tabla, busca un hilo no bloqueado para ejecutarlo, vuelve a cargar los registros guardados del nuevo hilo, y apenas intercambia el puntero a la pila y el puntero del programa, el hilo comienza a ejecutar. El intercambio de hilos es de una magnitud menor en tiempo que una interrupción siendo esta característica un fuerte argumento a favor de esta estructura. Este modelo tiene una gran escalabilidad y además permite a cada proceso tener su propio algoritmo de planificación. El sistema de tiempo de ejecución mantiene la ejecución de los hilos de su propio proceso hasta que el núcleo le retira el recurso CPU.

Paquete de hilos en el núcleo. Para cada proceso, el núcleo tiene una tabla con una entrada por cada hilo, con los registros, estado, prioridades, y demás información relativa al hilo (ídem a la información en el caso anterior), solo que ahora están en el espacio del núcleo y no dentro del sistema de tiempo de ejecución del espacio del usuario. Todas las llamadas que pueden bloquear a un hilo se implantan como llamadas al sistema, esto representa un mayor costo que el esquema anterior por el cambio de contexto. Al bloquearse un hilo, el núcleo opta entre ejecutar otro hilo listo, del mismo proceso, o un hilo de otro proceso.

Problemas de implementación de paquetes

- a) **Implementación de las llamadas al sistema con bloqueo.** Un hilo hace algo que provoque un bloqueo, entonces en el esquema de hilos a nivel del usuario no se puede permitir que el thread realice la llamada al sistema ya que con esto detendría todos los demás hilos y uno de los objetivos

es permitir que usen llamadas bloqueantes pero evitando que este bloqueo afecte a los otros hilos. En cambio en el esquema de hilos en el núcleo directamente se llama al núcleo el cual bloquea al hilo y luego llama a otro. Hay una forma de solucionar el problema en el primer esquema y es verificar de antemano que una llamada va a provocar un bloqueo, en caso positivo se ejecuta otro hilo. Esto lleva a escribir parte de las rutinas de la biblioteca de llamadas al sistema, si bien es ineficiente no existen muchas alternativas. Se denomina **jacket**. Algo similar ocurre con las fallas de página. Si un hilo produce una falla de página el núcleo que desconoce que dentro del proceso del usuario hay varios hilos bloquea todo el proceso.

- b) **Administración de los hilos (scheduling).** En el esquema de hilos dentro del proceso del usuario cuando un hilo comienza su ejecución ninguno de los demás hilos de ese proceso puede ejecutarse a menos que el primer hilo entregue voluntariamente el procesador. Dentro de un único proceso no existen interrupciones de reloj, por lo tanto no se puede planificar con quantum. En el esquema de hilos en el núcleo las interrupciones se presentan en forma periódica obligando a la ejecución del planificador.
- c) **Constantes llamadas al sistema.** En el esquema de hilos a nivel usuario se necesitaría la verificación constante de la seguridad de las llamadas al sistema, es decir es mucho más simple el bloqueo en los hilos a nivel núcleo que a nivel usuario puesto que en el núcleo solo bloquea al hilo y conmuta al próximo hilo mientras que a nivel usuario necesita antes de bloquearse llamar al sistema de tiempo de ejecución para conmutar y luego bloquearse.
- d) **Escalabilidad.** El esquema de hilos en el nivel usuario tiene mejor escalabilidad ya que en el esquema de hilos a nivel núcleo éstos requieren algún espacio para su tabla y su pila en el núcleo lo que se torna inconveniente si existen demasiados hilos.
- e) **Problema general de los hilos.** Muchos de los procedimientos de biblioteca no son reentrantes. El manejo de las variables globales propias es dificultoso. Hay procedimientos (como la asignación de memoria) que usan tablas importantes sin utilizar regiones críticas, pues en un ambiente con un único hilo eso no es necesario. Para poder solucionar estos problemas de manera adecuada habría que reescribir toda la biblioteca. Otra forma es proveer un **jacket** a cada hilo de manera que cierre un mutex global al iniciar un procedimiento. De esta forma la biblioteca se convierte en un enorme monitor. El uso de las señales (traps - interrupciones) también producen dificultades, por ejemplo dos hilos que deseen capturar la señal de una misma tecla pero con propósitos distintos. Ya es difícil manejar las señales en ambientes con un solo hilo y en ambientes multithreads las cosas no mejoran. Las señales son un concepto típico por proceso y no por hilo, en especial, si el núcleo no está consciente de la existencia de los hilos.

Como resumen sobre los procesos livianos (Threads) se puede decir que tienen las siguientes características:

- Un Thread (Hilo o Hebra) es básicamente, la unidad de utilización del procesador o de un módulo en las arquitecturas RISC (Reduced Instruction Set Computer).
- Tiene un pequeño estado no compartido.
- El ambiente en el cual un Thread se ejecuta es llamado Task.
- Un proceso tradicional (pesado) es igual a un Task con un Thread.
- Un Thread individual tiene su propio registro de estado y generalmente su propio Stack.
- Tiene las mismas características que los procesos pesados pero ejecuta más eficientemente.
- Los hilos pueden estar en uno de los siguientes estados: listos, bloqueados, ejecutando o terminados.
- Los hilos comparten procesador. Solo un hilo por vez puede estar ejecutándose.
- Un hilo dentro de un proceso se ejecuta secuencialmente.
- Pueden crear hilos hijos.
- Pueden bloquearse hasta que se complete un System Call. Si un hilo se bloquea otro puede correr pronto.
- A diferencia de los procesos pesados no son independientes, todos pueden acceder a cada dirección de una pila. Un hilo puede leer o escribir sobre cualquier otra pila de otro hilo. No se provee protección porque no es necesaria, ya que solo un usuario puede tener una tarea individual con múltiples hilos. Pueden asistirse. Así se permite cooperación de múltiples hilos que son parte del mismo Job (mayor resultado y performance).

- Una tarea no hace nada sin hilos y un hilo debe estar en una tarea.
- El compartir hace que el cambio de procesador entre grupos de hilos y la creación de hilos sea más fácil, comparado con el cambio de contexto de los procesos pesados.
- Un cambio de contexto en el hilo requiere un cambio en el juego de registros, pero el trabajo relacionado con manejo de memoria no es necesario hacerlo.

Resumen de Problemas con Threads:

- Un problema con el uso de procesos múltiples es que tenemos que “pagar” por los cambios de contexto, y éstos son caros.
- ¿Cómo mejoran los hilos los cambios de contexto? Los cambios de contexto entre hilos son más rápidos ya que hay que cambiar solamente conjuntos de registros.
- Un segundo problema es que puede el compartir los recursos como datos, archivos abiertos, I/O con el usuario, etc., puede ser difícil.
- Los hilos, o threads, son una solución. Un hilo es una cadena separada de ejecución dentro de un espacio existente de direcciones. Posee sus propio contador de programa, registros, y pila. Todos los otros recursos (heap, señales, etc.) son compartidos con otros hilos.
- Un proceso normal (un proceso pesado o una tarea) puede contener muchos hilos.
- Se pueden implementar bibliotecas de hilos a nivel de usuario (por ejemplo: no se necesita apoyo del Kernel). ¿Por qué mejora eso el tiempo de los cambios de contexto?
- No se necesita hacer una llamada al sistema, procesar una interrupción, etc.
- Empero, si no tenemos el apoyo de núcleo, tan solo un hilo puede bloquear todo el espacio de direcciones (por ejemplo: todos los hilos del proceso pesado) con una llamada al sistema. Por eso lo que proveen más y más sistemas es: apoyo de núcleo para hilos.
- ¿Cuál es un uso natural para los hilos? Servidores. Cada hilo en el servidor puede manejar una solicitud. El diseño es más sencillo y este esquema mejora la productividad, porque mientras que unos hilos esperan, otros pueden hacer trabajo útil.
- Especialmente con hilos se necesita controlar el acceso a los datos compartidos.

2.8. El Concepto de Fiber (Fibra)

Cuando se utilizan ULT (User Level Threads) para proveer más flexibilidad por parte de las aplicaciones, dichos threads pueden ser únicos o descomponerse en un conjunto de fibras que ejecutan un porción más pequeña.

Una fibra es una unidad de ejecución que debe ser agendada (**Schedule**) por la aplicación (programa del usuario). Las fibras corren en el contexto de los threads que las agendan. Cada thread puede agendar muchas fibras. En general las fibras no presentan ventajas sobre una aplicación multithreading bien diseñada. Sin embargo el uso de las fibras puede hacer más flexibles a las aplicaciones que fueron pensadas para agendar sus propios threads.

En sí es otra forma de organización al momento del procesamiento. Esta organización se realiza en modo usuario con las bibliotecas de threads provistas y programadas como parte de la aplicación a ejecutar.

Entonces, cada thread puede convertirse en una fibra o en un conjunto de fibras las cuales se ejecutan secuencialmente cuando se comienza a ejecutar el thread.

Se podría decir que un Proceso consta de dos partes:

- Una parte que maneja o controla los recursos que le fueron asignados al proceso, como la dirección de memoria en donde almacenar la imagen del proceso, o los canales de E/S, o dispositivos de E/S, o archivos.
- Otra parte que se encarga de la ejecución en sí (**Unit of Dispatching**), donde se guarda el estado del proceso, como ser Ejecutando, Ready u otro.

La idea es dividir estas partes y que funcionen, de alguna manera, por separado. Nace así el concepto de **Fiber** como una operación dentro de un Thread, que se asocia con la segunda parte mencionada anteriormente. Entonces se pueden tener múltiples Dispatching Units dentro de un Thread, cada una compartiendo los recursos del Thread. Si una Fibra es bloqueada, las demás pueden seguir ejecutándose sin problemas. Cuando el proceso muere, mueren todos sus Threads y Fibras que existieran dentro de él.

Las fibras presentan diferente información que los threads registrando solamente su stack, un conjunto de registros y datos provistos durante su creación.

Hay dos formas de ejecución de las fibras. Uno es **nonpreemptive** en la cual cada thread es ejecutado totalmente antes de abandonar el estado *running*, es decir que todas sus fibras creadas se ejecutan secuencialmente hasta terminar. La otra manera es la **preemptive** que hace que se pueda parar la ejecución de un thread sin haberse procesado todas las fibras en la que fue dividido.

Antes de comenzar con la ejecución de un thread se debe llamar a la función **Convert_Thread_To_Fiber()** la cual crea un área en la que se van a guardar todos los datos de la fibra y hace que el thread seleccionado pase a ser ahora la fibra a ejecutarse. A su vez cada fibra va creando nuevas fibras con la función **Create_Fiber()** ejecutándose con otra función llamada **Switch_To_Fiber()**. Luego se eliminan con **Delete_Fiber()**. De esta manera el ciclo continúa hasta finalizar la ejecución. Los nombres de las funciones varían de un S.O. a otro. Estos pertenecen al WinNT en sus distintas versiones.

2.9. Ejemplo de Sistemas Operativos basados en Threads.

2.9.1. Procesos en UNIX [Valley, 1991]

Procesos en UNIX [Valley, 1991]. Veamos algunas características particulares resumidas sobre Procesos en UNIX.

- Creación de un proceso: **int fork(void)**
 - a) Hace una copia completa del PCB del proceso actual creando así un nuevo proceso.
 - b) Después de **fork()** quedan 2 procesos (padre e hijo) corriendo el mismo programa.
 - c) Todo es idéntico (archivos abiertos, Stack, memoria, etc.), excepto:
 - i. El return value: 0 para el hijo, y > 0 para el padre (si ≤ 0 significa error).
 - ii. El **getpid()**.
 - iii. Valores de semáforos usados por el padre no son los mismos que los de los hijos.
 - iv. Bloqueos sobre archivos.
 - d) Uno de los dos procesos (normalmente el hijo) usa llamada al sistema **execve()** después del **fork ()** para reemplazar el espacio de memoria del proceso con el nuevo programa.
 Esta llamada carga en memoria un archivo binario (destruyendo el contenido en memoria del proceso que efectúa la llamada al sistema **execve()**) y comienza su ejecución.
- Inmediatamente después del **fork()** se debe verificar el valor de retorno para distinguir al proceso hijo del padre y separar sus ejecuciones.
- Número de procesos limitado. El Kernel tiene una Process Table de tamaño configurable, pero fijo.
- El proceso puede terminar usando la llamada al sistema **exit** (salida) y su proceso padre puede esperar con la llamada **wait** (espera).
- **wait** devuelve el identificador del proceso del hijo que ha terminado, para que el padre pueda determinar cuál de sus varios hijos posibles ha terminado.
- Es importante considerar que UNIX no divide recursos entre hijos. Planifica a cada hijo por separado y compete igualmente con otros procesos para poder obtener los recursos limitados (como memoria, acceso a disco, etc.).
- Por cada programa que se quiera ejecutar, el S.O. crea un proceso que lo ejecute.

Usuarios y grupos en UNIX.

Considera tres tipos de usuarios: El dueño (Owner), los compañeros de grupo (Group) y los otros.

- UNIX usa 7 identificadores por cada proceso.
 - PID (Process Id)** int 1 a 30000 Asignado con **fork ()**.
Query con **getpid ()**.
No se puede cambiar.
 - PPID(Parent Process Id)** Pid del padre (creador).
Asignado como return value de **fork ()**.
Query con **getppid ()**.
No se puede cambiar.
 - Padre e hijo pueden comunicarse vía signal.
 - RUID (Real User Id)** int 0 a 60000 Identifica al user. Se usa para seguridad
0 = super usr,
Asignado cuando se crea el usuario.

Query con **getuid ()**.

Alterado con **setuid ()**.

EUID (Effective User Id) Asignado cuando un programa ejecuta otro.

Si llamador (caller) tiene setuid Flag => EUID = RUID del caller, sino EUID = RUID.

Alterado con **setuid ()**.

Si super user=> Sets Two of ** 'Em.

Sino depende si el caller tiene o no setuid Flag.

RGID(Real Group Id) ídem RUID, pero con Group.

Query con **getgid ()**.

Alterado con **setgid ()**. Sólo s-usr (Super Usuario).

EGID (Effective Group Id) ídem EUID, pero con Group.

(Mismo algoritmo que Query con **getegid ()** UID).

Alterado con **setegid ()**.

PGID o PGRP (Process group id) Se usa para poder disponer de grupos de procesos => todas las señales se dirigen a un grupo de procesos.

El primer grupo lo crea getty cuando invoca a login.

El primer proceso que usa el PGID se transforma en el leader y todos los demás heredan el PGID.

Query con **getpgrp ()**.

Alterado con **setpgrp ()** y se transforma en process leader.

Procesos especiales en UNIX:

- **Proceso 0 Swapper.** La rutina de inicialización del kernel (I.P.L.) se ocupa de cargar el sistema y luego se transforma en proceso 0.
- **Proceso 1 I.P.L. fork ()** exec("/bin/init") => init.
Aclaración: finit ejecuta un script (/etc/inittab) que le dice todo lo que tiene que hacer **getty ()** por cada terminal entonces por cada **getty ()** hace login.
- **Proceso de inicialización** (Referencia Cap 1-3 Valley, 1991).
Bootstrap o I.P.L. La secuencia de inicialización es:
 - 1) La Máquina se enciende => Rutina en ROM busca un disco => lee rutina del primer bloque y la ejecuta.
 - 2) UNIX pregunta que kernel cargar y lo carga.
 - 3) La Rutina de inicialización del Kernel determina cuanta Memoria Central hay disponible. Reserva algo para el sistema y el resto para usuario.
 - 4) Llama a los ENTRY POINT de los I/O drivers para preparar los dispositivos para su funcionamiento normal.
 - 5) Luego el Kernel ejecuta => **fork ()** al proceso init.
 - 6) Salta a la rutina swapper y se transforma en el proceso 0.

Proceso init. Es un proceso de propósito general usado para ejecutar procesos necesarios para la inicialización del sistema. Es configurable ya que cada instalación necesita diferentes cosas. Las acciones se escriben en **/etc/inittab**. Es un archivo de texto donde están los comandos necesarios para configurar el sistema (similar al config.sys + autoexec.bat en DOS). Luego de configurar el sistema, init ejecuta **getty()** por cada terminal. Este proceso se ejecuta en modalidad **respawn**, de esta forma, cuando el usuario sale, se vuelve a ejecutar **getty()**

Proceso getty. Establece conexión física con cada terminal. Bit rate, stop, parity, data bits, handshaking protocol, character mapping, etc. Muestra mensaje "login:" y deja ingresar un nombre de usuario. Una vez ingresado el nombre, lo pasa como parámetro a login.

Proceso login. Busca el nombre en **/etc/passwd** donde se fija cual es el password cifrado, grupo, userid, groupid y home directory. Valida el password cifrado y ejecuta el **shell** asociado al usuario. El shell ejecuta el **script /etc/profile** y luego, el opcional profile del home directory.

Observación: El profile puede ser otro nombre según que shell sea.

Los únicos entes activos en UNIX son los procesos. Cada usuario puede tener varios procesos activos simultáneamente, así que en un sistema UNIX puede haber cientos o miles de procesos simultáneos. Incluso cuando no hay usuarios usando el sistema, hay procesos corriendo, llamados **demonios (daemons)**, que son creados cuando se inicializa el sistema.

Un ejemplo es el demonio **cron**, que despierta cada minuto para ver si hay algún trabajo que hacer. Gracias a él se pueden fijar actividades periódicas, como por ejemplo, hacer respaldos (backups) a las 4 AM. Otros demonios manejan el correo entrante, la cola de impresión, monitorean el uso de páginas de memoria, etc.

Los procesos en UNIX. El manejo de procesos en UNIX es por prioridad y **round robin**⁷. En algunas versiones se maneja también un ajuste dinámico de la prioridad de acuerdo al tiempo que los procesos han esperado y al tiempo que ya usó de CPU. El sistema provee facilidades para crear 'pipes' entre procesos, contabilizar el uso de CPU por proceso y una pila común para todos los procesos cuando necesitan ejecutarse en modo privilegiado (cuando hicieron una llamada al sistema). UNIX permite que un proceso haga una copia de sí mismo por medio de la llamada 'fork', lo cual es muy útil cuando se realizan trabajos paralelos o concurrentes; también se proveen facilidades para el envío de mensajes entre procesos. Recientemente Sun Microsystems, AT&T, IBM, Hewlett Packard y otros fabricantes de computadoras llegaron a un acuerdo para usar un paquete llamado ToolTalk para crear aplicaciones que usen un mismo método de intercambio de mensajes.

2.9.2. Procesos en LINUX

Observación: Ya mencionamos casos de Procesos en UNIX para que se observe la secuencia automática para inicializar el sistema y abrir una sesión al usuario. Lo propuesto se utiliza como ejemplo, dado que el S.O. UNIX es un sistema multiusuario ampliamente utilizado por ejemplo en comunicaciones o servidores de red. UNIX Tiene un S.O. clon que se llama **LINUX**.

Procesos en LINUX. Para que Linux pueda gestionar los procesos en el sistema, cada proceso se representa por una estructura de datos: `task_struct` (las tareas (task) y los procesos son términos intercambiables en Linux). El vector `task` es una lista de punteros a cada estructura `task_struct` en el sistema. Esto quiere decir que el máximo número de procesos en el sistema está limitado por el tamaño del vector `task`; por defecto tiene 512 entradas. A medida que se crean procesos, se crean nuevas estructuras `task_struct` a partir de la memoria del sistema y se añaden al vector `task`. Para encontrar fácilmente el proceso en ejecución, hay un puntero (`current`) que apunta a este proceso.

Linux soporta procesos de tiempo real así como procesos normales. Estos procesos tienen que reaccionar muy rápidamente a sucesos externos (de ahí el término "tiempo real") y reciben un trato diferente del planificador. La estructura `task_struct` es bastante grande y compleja, pero sus campos se pueden dividir en áreas funcionales:

State (Estado) A medida que un proceso se ejecuta, su estado cambia según las circunstancias. Los procesos en Linux tienen los siguientes estados:

- **Running:** (Ejecutándose) El proceso se está ejecutando (es el proceso en curso en el sistema) o está listo para ejecutarse (está esperando a ser asignado a una de las CPUs del sistema).
- **Waiting:** (Esperando) El proceso está esperando algún suceso o por algún recurso. Linux diferencia dos tipos de procesos; interrumpibles e interrumpibles. Los procesos en espera interrumpibles pueden ser interrumpidos por señales mientras que los interrumpibles dependen directamente de sucesos de hardware y no se pueden interrumpir en ningún caso.
- **Stopped:** (Detenido) EL proceso ha sido detenido, normalmente porque ha recibido una señal. Si se están reparando errores en un proceso, este puede estar detenido.
- **Zombie:** Es un proceso parado cuya estructura `task_struct` permanece aún en el vector `task`. Es, como su nombre indica, un proceso muerto.

2.9.3. Procesos en NT-Windows

MANEJO DE PROCESOS E HILOS EN WINDOWS NT. Algunos Sistemas Operativos como el D.O.S., presentaba en sus versiones viejas, el problema de permitir que únicamente se ejecutara un programa a la vez; era necesario esperar a que éste terminara de ejecutarse para proseguir con otro.

Con el desarrollo a pasos agigantados del hardware y nuevas arquitecturas de procesadores, se vio la posibilidad de aprovechar de una mejor manera los recursos de la máquina, eliminando en gran proporción, tiempos ociosos. Así se dio paso a la investigación, en busca de la solución de este problema. Muchas compañías comenzaron a estudiar por su cuenta la solución al problema logrando así productos que permitieran aprovechar y explotar los recursos de la máquina de una manera más eficiente. Un ejemplo de

⁷ round robin es una política de planificación de procesos que se verá en el módulo 3

estos es WINDOWS, creado por Microsoft, que ha evolucionado de ser una simple plataforma a convertirse en un Sistema Operativo.

La base fundamental de estos avances, radica en el hecho de poder ejecutar varios programas al mismo tiempo en una máquina. Esto se logra gracias a la introducción del concepto de "proceso", que puede confundirse con "programa". Un programa es un conjunto de instrucciones, mientras que un proceso es un conjunto variable de llamadas a un programa, junto con todos los recursos que requiere para su ejecución.

Todos los procesos deben ser administrados de alguna manera en estos Sistemas Operativos. Así es

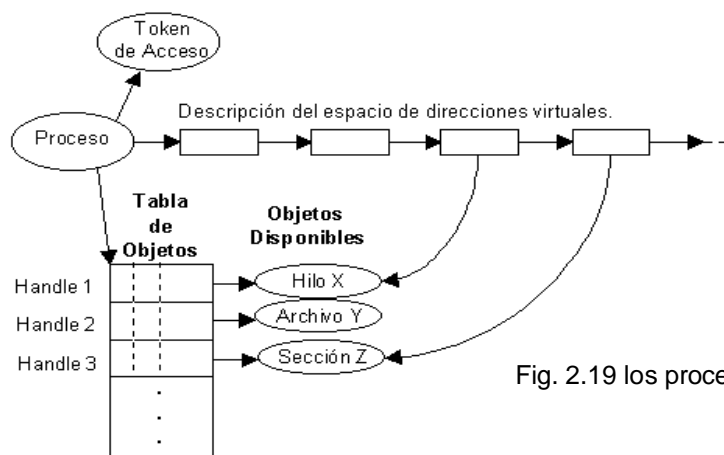


Fig. 2.19 los procesos en Windows NT

como en máquinas que sólo constan de un procesador, parece que varios procesos se estuvieran ejecutando a la vez, pero lo que en realidad está sucediendo es que cada proceso o tarea tiene asignada un tiempo específico en el procesador, logrando así una apariencia de ejecución simultánea. Dado que es más fácil solucionar los problemas dividiéndolos en partes, los Sistemas Operativos usan el mismo esquema de trabajo con los procesos. Cada proceso puede usar varios hilos para su ejecución, dependiendo de la dificultad del problema, donde cada hilo se encarga de una parte específica de dicho problema.

PROCESOS. En algunos Sistemas Operativos como OS2 se caracteriza el manejo de los procesos por la utilización de estructuras jerárquicas padre-hijo. Esto consiste en que un proceso a la hora de su ejecución crea un hilo que comanda la ejecución de la tarea.

Dependiendo de su complejidad, dicho hilo puede delegar tareas a otros hilos, aumentando así la facilidad en la ejecución; estos hilos a su vez pueden delegar más tareas y así sucesivamente. Cuando cada hilo hijo va terminando su ejecución rinde cuentas al hilo padre y estos a su vez a sus hilos padres, concluyendo así la ejecución total del proceso.

En WINDOWS NT, el manejo de los procesos es diferente; son manejados por el ADMINISTRADOR DE PROCESOS y tienen las siguientes características:

- Debido a que WINDOWS NT fue implementado en ambiente de objetos, sus procesos heredan todas las características de dicho ambiente y son manejados como tales.
- Un proceso de WINDOWS NT puede manejar varios hilos, pero es importante resaltar que el administrador de procesos no maneja relaciones padre/hijo entre los procesos que crea.

Un proceso en WINDOWS NT estaría compuesto por un programa, un conjunto de recursos necesarios para su ejecución (semáforos, puertos de comunicación, archivos, etc.), un espacio de memoria virtual reservado y privado y por lo menos un hilo de ejecución que será planificado por el kernel. El manejo de los espacios en memoria virtual, permite al usuario ver la ejecución de sus procesos y tareas en una forma organizada, a diferencia de lo que sucede en espacios de memoria física, en las cuales toda la información se encuentra entremezclada con otro y sólo se le encuentra el orden por un acceso de direcciones.

El espacio de direcciones en memoria virtual podría ser visto como una máscara sobre la memoria física. Cada espacio virtual es exclusivo para la ejecución de un proceso (que es lo que ve el usuario en una forma organizada) pero así mismo, cada dirección virtual hace referencia a una dirección en memoria física. Este manejo de memoria virtual tiene dos propósitos: darle un espacio de ejecución más grande a cada proceso y evitar que el usuario manipule directamente la memoria física previniendo así dañar al Sistema Operativo y errores o conflictos con otras aplicaciones.

WINDOWS NT maneja dos modos de ejecución para sus procesos: modo kernel y modo usuario. Todos los procesos por lo general se ejecutan en modo usuario, pero en el momento en que necesiten de algún servicio del Sistema Operativo, el procesador cambia su modo de ejecución a modo kernel, mientras se ejecuta el servicio, regresando de nuevo a modo usuario al terminar la operación. Complementando la idea, el modo kernel evalúa todas las peticiones que el hilo haga al sistema, desechando los que puedan provocar alguna falla al sistema. Como ya se había mencionado, un proceso tiene asociado un conjunto de recursos y un espacio de direcciones privado.

Tipo de objeto	Proceso
Atributos del cuerpo de objeto	ID de proceso Token de acceso Prioridad base Afinidad con el procesador por defecto Límites de cuota Tiempo de ejecución Contadores de E/S Contadores de operaciones de VM Puertos de excepción/depurado Estado de salida
Servicios	Crear proceso Abrir proceso Consultar información del proceso Configurar información del proceso Actualizar proceso Finalizar proceso Reservar/liberar memoria virtual Leer/escribir en la memoria virtual Proteger la memoria virtual Cerrar/abrir la memoria virtual Consultar la memoria virtual Activar la memoria virtual

Fig. 2.20. Atributos y servicios de un proceso en Windows NT

El conjunto de recursos se compone de un TOKEN DE ACCESO que contiene la identificación de cada proceso que permite el acceso a los diferentes recursos del sistema; TABLA DE OBJETO que contiene los handles que apuntan a direcciones de memoria virtual donde se encuentran hilos, archivos y demás recursos necesarios para la ejecución del proceso; CONJUNTO DE LIMITES DE CUOTAS DE RECURSOS que restringen la memoria que utiliza cada uno de los hilos para abrir handles a objeto.

Para concluir con el tema de los procesos es necesario hablar del objeto proceso.

El objeto proceso es creado y eliminado por el administrador de objetos, que también está encargado de asignar los atributos, servicios y parámetros de inicialización del proceso. En la figura 2.20. se muestran los atributos y servicios del objeto proceso.

HILOS. Como se comentó en párrafos anteriores resulta más sencillo atacar un problema si antes lo partimos en sub-problemas más pequeños. Haciendo la analogía, un hilo sería cada una de las subtarefas que debe ejecutar un proceso; un proceso termina su ejecución, cuando cada uno de sus hilos termina de ejecutarse.

Un hilo en WINDOWS NT tiene un área privada de almacenamiento llamada CONTEXTO DEL HILO, que se caracteriza por tener un único identificador (ID_cliente), un área de almacenamiento empleada por subsistemas y bibliotecas, una pila para ejecución en modo usuario, otra para ejecución en modo kernel y registros que indique el estado del procesador.

Es importante recordar que cada hilo se ejecuta en un espacio virtual separado para cada proceso. Cuando un proceso consta de varios hilos, todos ellos se ejecutan en el mismo espacio virtual, permitiendo así que los resultados de un hilo, sirvan como entradas de procesamiento de otros, permitiendo así la ejecución completa del proceso.

Para la ejecución de los hilos, el Sistema Operativo se puede ayudar de los siguientes conceptos:

MULTITAREA: Consiste en ejecutar varios programas aparentemente en forma simultánea. Esta impresión se logra asignando tiempos de procesador para la ejecución independiente de cada hilo. En WINDOWS NT la clave del cambio del hilo que está en el procesador ejecutándose por otro en espera de ejecución es la CONMUTACION DE CONTEXTO (controlada por el kernel) que se compone de los siguientes pasos:

§ Ejecución de un hilo hasta su terminación o vencimiento de Quantum.

§ Grabar el contexto del hilo que actualmente se ejecuta para que cuando regrese dicho hilo a estado de ejecución, sepa donde continuar.

§ Cargar el contexto de otro hilo.

Esta secuencia debe repetirse mientras haya hilos en espera de procesador. La ventaja que acarrea la multitarea es que el procesador siempre estará ocupado con la ejecución de un hilo, eliminando los tiempos ociosos. Esto lo consigue haciendo que procesos que están actualmente en ejecución y necesitan esperar por algún servicio del sistema son mandados a estado suspendido; mientras se espera por dicho servicio, el procesador comienza a ejecutar un nuevo hilo, retomando luego el hilo inicial en el contexto donde había quedado.

La multitarea que maneja WINDOWS NT es del tipo "pre-empted", que introduce el concepto de Quantum de tiempo o espacios de tiempo fijos que el procesador asigna para la ejecución de cada hilo. Cuando el procesador está ejecutando un hilo y se consume su Quantum efectúa la conmutación de contexto. Además del tiempo de ejecución asignado a cada hilo, al hilo se le asigna una prioridad, que va siendo reducida, a medida que el hilo se ejecuta; también hay que aclarar que el hilo a ejecutar será aquel que tenga la más alta prioridad.

En algunos casos varios hilos necesitan comunicarse entre ellos para lograr un objetivo común; dado que cada proceso tiene un espacio de memoria diferente para su ejecución, es necesario proporcionar un espacio de memoria compartido por dichos hilos. Este espacio de memoria compartido se denomina "pipe" y el tipo de ejecución de estos procesos se denomina "ejecución concurrente".

MULTIHILO: En algunas ocasiones el usuario queriendo acelerar la ejecución de un programa utiliza varios procesos cada uno de los cuales tendrá siempre un hilo para ejecutar, proporcionando así mayor velocidad de ejecución. Una mejor solución para este problema consiste en crear procesos multihilos. Un **proceso multihilo** consta de varios hilos donde cada uno de ellos cumple una labor específica. La ventaja está en que todos los hilos pertenecen al mismo proceso y por lo tanto tienen el mismo espacio de direcciones en común y los mismos recursos asignados, evitando así pérdida de tiempo mientras se carga el espacio de memoria de otros proceso. La forma de garantizar que todos los hilos se ejecutan es como ya se explicó anteriormente, asignando prioridades a cada hilo e ir reduciéndolos a medida que se ejecutan. Como siempre entra a ejecución el hilo de mayor prioridad, queda garantizado la ejecución de todos los hilos.

OBJETO HILO: Un proceso para ser ejecutado necesita un hilo. Una vez creado dicho hilo, pueden ser creados hilos adicionales. Como se puede apreciar en la figura 2.21 la prioridad base de un hilo entra a depender directamente de la prioridad base del proceso, ya sea sumándole o restándole 2 niveles. Cuando el hilo entra a ejecución, el ejecutor de WINDOWS NT podrá subir su prioridad sin restricción, pero no bajarla por debajo del nivel donde fue creado.

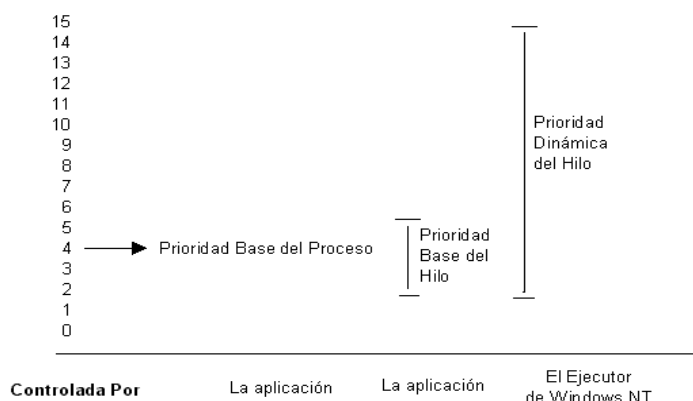


Fig.2.21 prioridades en Windows NT

SINCRONIZACION Y ALERTAS: En algunos casos cuando se ejecutan procesos concurrentes es necesario que los hilos de dichos procesos se comuniquen entre sí, necesitando así de mecanismos que indiquen a otros hilos que deben esperar mientras se obtienen resultados del que actualmente se encuentran en ejecución. Este mecanismo se conoce como "sincronización". Para que la ejecución de los hilos se ejecute de forma controlada, WINDOWS NT proporciona varios objetos de sincronización como los objetos proceso, objetos hilo, objetos semáforo y objetos timer entre otros. Los objetos de sincronización manejan 2 estados principales: el estado señalado que indica que un hilo no está en ejecución y estado no señalado que indica lo contrario. Otra operación que está muy relacionada con la sincronización son las alertas que permiten que un hilo notifique a otros que debe detener su ejecución, o que un hilo alerte a otro en qué momento debe ejecutarse. Todas estas operaciones las realiza WINDOWS NT por medio de los APC (Llamadas de Procedimiento Asíncrono)

Las llamadas a procedimientos locales (LPC) son usadas para pasar mensajes entre dos diferentes procesos corriendo dentro de un mismo sistema NT; estos sistemas fueron modelados utilizando como modelo las llamadas a procedimientos remotos (RPC); los RPC consisten en una manera estandarizada de pasar mensajes entre un cliente y un servidor a través de una red. Similarmente los LPC's pasan mensajes de un procedimiento cliente a un procedimiento servidor en un mismo sistema NT.

Cada proceso cliente en un sistema NT que tiene capacidad de comunicación por medio de LPC's debe tener por lo menos un objeto de tipo puerto asignado a él, este objeto tipo puerto es el equivalente a un puerto de TCP/IP en un sistema UNIX.

CONCLUSIONES

- § A diferencia del manejo de los hilos de los procesos en otros ambientes, WINDOWS NT no utiliza el concepto de jerarquías, queriendo decir con esto que nunca un hilo hijo tendrá que rendirle cuentas a un hilo padre sino que por el contrario todos los hilos se encontrarán a un mismo nivel con las mismas capacidades para ejecutarse (dependiendo de las prioridades)
- § La forma en la cual los procesos e hilos se manejan en WINDOWS NT es muy diferente al manejo jerárquico de la mayoría de los Sistemas Operativos actuales, dado que WINDOWS NT facilita la ejecución tratando a los componentes como objetos, evitando al mismo tiempo complejidad.
- § A la hora de programar el usuario tiene que tener muy claros los conceptos de procesamiento en multitarea y procesamiento multihilo así como también el tipo de operaciones y datos a ejecutar, dado que dependiendo de la situación aplicar una cosa u otra puede permitir mejorar el desempeño y aprovechamiento de la máquina.
- § Los hilos de un mismo proceso se pueden comunicar fácilmente usando espacios de memoria compartida, pero si no fuera por las llamadas de procedimiento local sería imposible la transferencia de información entre hilos de diferentes procesos.
- § Uno de los mayores avances en los Sistemas Operativos han sido el manejo de los procesos en hilos, esto permite mayor aprovechamiento del hardware y la creación de sistemas que trabajen en multitarea y multiprocesamiento lo cual amplía el campo de las posibilidades del manejo de aplicaciones más complejas y completas.

2.9.4. Manejo de procesos en VMS (Virtual Machine Systems)

Soporta muchos ambientes de usuario tales como: Tiempo crítico, desarrollo de programas interactivos, batch, ya sea de manera concurrente, independiente o combinado.

El planificador VAX/VMS realiza planificación de procesos normales y de tiempo real, basados en la prioridad de los procesos ejecutables en el Balance Set. Un proceso normal es referido a como un proceso de tiempo compartido o proceso background mientras que los procesos en tiempo real se refieren a los de tiempo crítico.

En VMS los procesos se manejan por prioridades y de manera expropiativa. Los procesos se clasifican de la prioridad 1 a la 31, siendo las primeras quince prioridades para procesos normales y trabajos en lote, y de la 16 a la 31 para procesos privilegiados y del sistema. Las prioridades no permanecen fijas todo el tiempo sino que se varían de acuerdo a algunos eventos del sistema. Las prioridades de los procesos normales pueden sufrir variaciones de hasta 6 puntos, por ejemplo, cuando un proceso está esperando un

dispositivo y éste fue liberado. Un proceso no suelta la unidad central de procesamiento hasta que exista un proceso con mayor prioridad.

El proceso residente de mayor prioridad a ser ejecutado siempre se selecciona para su ejecución. Los procesos en tiempo crítico son establecidos por el usuario y no pueden ser alterados por el sistema. La prioridad de los procesos normales puede ser alterada por el sistema para optimizar **overlap** (superposición) de computación y otras actividades I/O.

Un aspecto importante del planificador de procesos en VMS es la existencia de proceso 'monitor' o 'supervisor', el cual se ejecuta periódicamente para actualizar algunas variables de desempeño y para re-planificar los procesos en ejecución.

Existen versiones de VMS que corren en varios procesadores, y se ofrece bibliotecas para crear programas con múltiples 'threads'. En específico se proveen las interfaces 'cma', 'pthread' y 'pthread-exception-returning'. Todas estas bibliotecas se conocen como DECthreads e incluyen bibliotecas tales como semáforos y colas atómicas para la comunicación y sincronización entre threads. El uso de threads sirve para enviar porciones de un programa a ejecutar en diferentes procesadores aprovechando así el multiproceso.

Servicios del Sistema para el Control de Procesos

- **Crear un proceso:** El servicio de creado de sistema permite a un proceso crear otro. El proceso creado puede ser un subproceso o un proceso completamente independiente. (se necesitan privilegios para hacer esto).
- **Suspender un proceso:** Esto es que le permite a un proceso suspenderse a sí mismo o a otro (también necesita tener privilegios).
- **Reanudar un proceso:** Permite a un proceso reanudar a otro si es que este tiene privilegios para hacerlo.
- **Borrar un proceso:** Permite que se borre el proceso mismo o a otro si es que es un subproceso, o si no tiene que tener privilegios de borrado.
- **Dar Prioridad:** Permite que el proceso mismo se ponga prioridad o a otros, para el planificador.
- **Dar el modo de espera:** Permite que el proceso escoja de dos modos: el modo por default es cuando un proceso requiere un recurso y está ocupado y espera a que esté desocupado, y el otro modo es cuando está ocupado el recurso, el proceso no espera y notifica al usuario que el recurso no se encuentra disponible en ese momento en lugar de esperar.
- **Hibernar:** Es cuando un proceso se hace inactivo pero está presente en el sistema. Para que el proceso continúe necesita de un evento para despertar.
- **Wake:** Esto activa a los procesos que están hibernando.
- **Exit:** Es cuando se aborta un proceso.
- **Dar nombre al proceso:** Este puede dar un nombre al proceso mismo o cambiarlo (el PCB contiene el nombre).

2.9.5. Manejo de procesos en OS/2

OS/2 utiliza un esquema de planificación expropiativa, es decir, los procesos pueden ser suspendidos para darle su turno de ejecución a otro diferente. Los procesos pueden estar divididos en 'threads' que cuentan con sus propios registros, pila y contador de programa y todos los 'threads' de un mismo proceso comparten la memoria. Esto facilita la comunicación entre ellos y la sincronización. También es posible que un proceso genere un proceso hijo, en tal caso el hijo hereda todos los atributos del padre como son los descriptores de archivos abiertos, los valores en memoria, etc.; prácticamente igual que el Sistema Operativo UNIX.

Otra facilidad de OS/2 es la facilidad de crear 'conductos' (pipes) lo cual también es una función heredada de UNIX.

La planificación de procesos o 'threads' se hace por prioridad y dándoles una intervalo de ejecución a cada proceso o 'thread'. Se manejan tres niveles de prioridades: procesos preferentes, procesos preferentes interactivos y procesos normales. OS/2 eleva a la categoría de preferentes a aquellos procesos que hacen mucha E/S.

Otra facilidad notable de OS/2 es la carga dinámica de bibliotecas, que consiste en la generación de aplicaciones cuyas bibliotecas no forman parte del código compilado, sino que son cargadas cuando el programa es ejecutado. Esto sirve bastante sobre todo cuando las bibliotecas son de uso común. Como se ve, esta facilidad es parecida a las del Sistema Operativo UNIX SunOS.

2.9.6. SOLARIS: Ejecución de threads

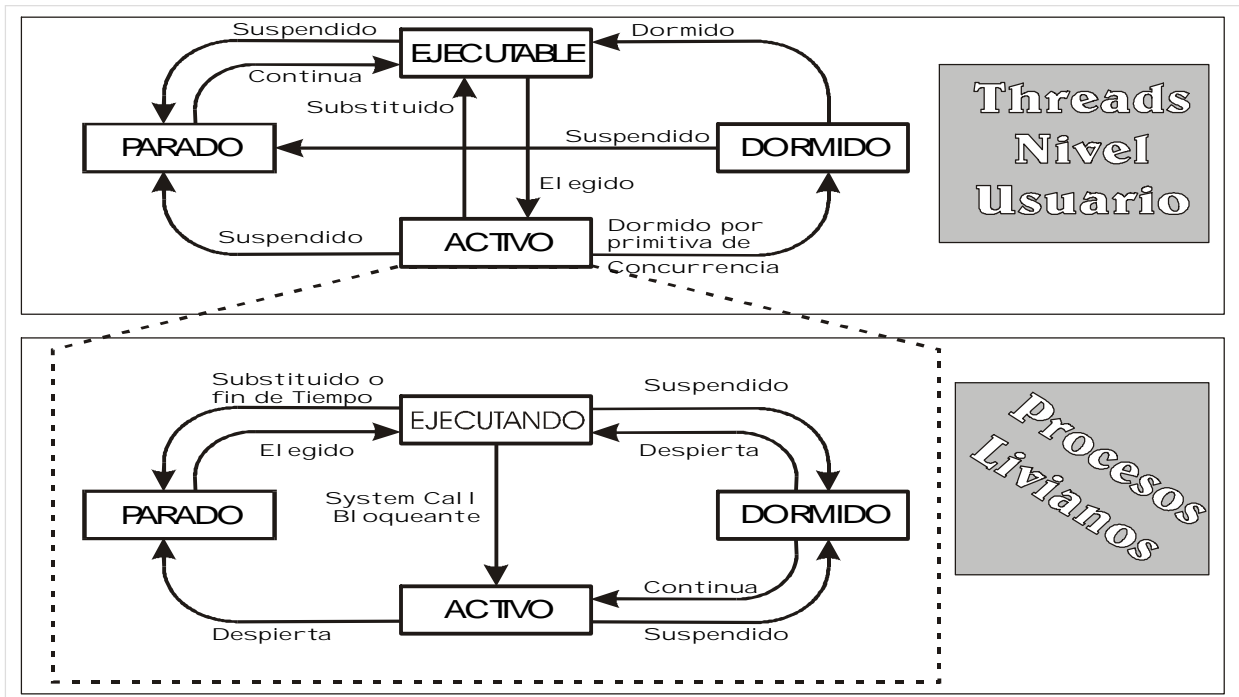


Figura 2.23 Diagrama de estados de ejecución en Solaris

Solaris combina hábilmente el enfoque de thread nivel usuario y thread nivel kernel.

Arquitectura multithread

- **Proceso:** concepto típico de proceso de cualquier sistema UNIX.
- **Thread nivel usuario:** implementados a través de una librería, en el espacio de direcciones del proceso.
- **Thread nivel kernel:** son las entidades fundamentales que pueden ser planificadas y elegidas para ejecución.
- **Procesos livianos:** se pueden ver como un mapeo entre thread nivel usuario y thread nivel kernel. C/u soporta uno o más thread nivel usuario para mapearlo en un solo thread nivel kernel. Son planificados por el kernel.

Con esto se logra implementar los dos enfoques sobre threads y sus combinaciones (fig. 4-14 / pag. 177). Esto permite elegir el grado de concurrencia de un programa.

Existen los thread nivel kernel sin procesos livianos, el kernel los usa mucho para sus funciones.

La estructura con la que Solaris administra c/proceso es como la de cualquier otro sistema UNIX, salvo que reemplaza el bloque de estado del procesador por una lista de estructuras con información de los procesos livianos

Interrupciones como Threads

Solaris maneja las interrupciones con un conjunto de thread nivel kernel. Así disminuye el overhead de la sincronización entre interrupciones.

Los manejadores de interrupción trabajan sobre información que es compartida por todo el kernel. Al convertirlos en threads, se pueden aprovechar las primitivas de mutua exclusión (tema del cap. 5) que se usan para sincronizar procesos y/o threads, para proteger la información mencionada y lograr la sincronización entre interrupciones.

Los threads de interrupción tienen una prioridad más alta que cualquier otro thread del kernel.

2.10. Multiprocesamiento⁸

Organización de Symmetric Multi Processing (SMP)

En SMP hay varios procesadores y una Memoria Central compartida, también c/dispositivo de E/S es compartido por los procesadores. La comunicación entre estos elementos suele hacerse mediante un bus. La memoria suele estar organizada de manera que el acceso simultáneo a distintos bloques sea posible. Un problema que siempre cuesta resolver es lograr la consistencia de la caché de c/procesador con la del resto y con Memoria Central.

2.10.1. Consideraciones de diseño

Procesos y/o threads simultáneos concurrentes

Las rutinas del kernel deben ser 'reentrantes' para permitir a distintos procesadores ejecutar el mismo código del kernel al mismo tiempo. Debe manejarse propiamente para evitar deadlocks y operaciones inválidas.

Planeamiento

Lo hace cualquier procesador. Si hay multithreading se pueden planificar distintos threads de un mismo proceso en distintos procesadores. Hay que evitar que dos procesadores planifiquen el mismo proceso al mismo tiempo.

Sincronización

Debe hacer valer la mutua exclusión y el ordenamiento de eventos.

Administración de memoria

Debe explotar el paralelismo del hardware disponible para alcanzar el mejor desempeño posible.

Tolerancia a fallos y confiabilidad

El planificador y otras porciones del sistema operativo deben reconocer la pérdida de un procesador y en concordancia con esto se deben reestructurar las tablas de administración del sistema.

2.11. Bibliografía recomendada para el módulo 2.

Operating Systems. (5th Edition), Internals and Design Principles, Stallings Willams, Prentice Hall, Englewood Cliff, NJ., 2004,

Operating Systems. (Fourth Edition), Internals and Design Principles, Stallings Willams, Prentice Hall, Englewood Cliff, NJ., 2001,

Applied Operating Systems Concepts (Fifth Edition), Silberschatz, A. and Galvin P. B., Addison Wesley, 1998,

Operating Systems Concepts First Edition, Silberschatz, A. and Galvin P. B., Gagne G. Wiley, 2003,

Operating Systems Concepts and Design (Second Edition), Milenkovic, Millan., Mc Graw Hill, 1992

Modern Operating Systems, Tanenbaum, Andrew S., Prentice - Hall International, 1992

Operating Systems, Design and Implementation (Second Edition), Tanenbaum, Andrew S., Prentice - Hall International, 1997

Fundamentals of Operating Systems., Lister, A.M. , Macmillan, London; 1979

⁸ Se ampliará el concepto en el Modulo 3.

Operating System; Lorin, H., Deitel, H.M.; Addison Wesley; 1981

The UNIX Operating System; Christian, K.; John Wiley; 1983

UNIX System Architecture; Andleigh, Prabhat K.; Prentice - Hall International; 1990

The UNIX Programming Environment; Kernighan, B.W. and Pike, R.; Prentice - Hall International; 1984

The UNIX System V Environment; Bourne, Stephen R.; Addison Wesley; 1987

Sistemas Operativos Conceptos y diseños.(Segunda Edición); Milenkovic Milan.; Mc Graw Hill; 1994..

Sistemas de Explotación de Computadores; CROCUS; Paraninfo; 1987 424 pág.

GLOSARIO DE TÉRMINOS EN IDIOMA INGLÉS

Mainframe	Host	Process	Communication Manager
Workstation	Personal Computer	Thread	I/O Manager
Batch	Time Sharing	Fiber	Memory Manager
Job	Feedback	User Level Thread	File Manager
Job Scheduler	Low Scheduler	Kernel Level Thread	Run Time Environment
Remote Job Entry	Shell	Process Level Thread	New
Job Control Language	Microkernel	Lightweight process	Ready
Loader	Kemel	Task	Suspend
Dispatcher	Switcher	Super User	Running
Traffic Controller	Overhead	User	Zombi
Prompt	I/O Completed	Preempt	Exit
Supervisor Call	Atomic Action	Stack	Lock
System Calls	Top Down	Programm Counter	Unlock
Program Calls	Botton Up	Multithreading	Process Control Block
System Program	Trap	Owner	System Control Block
Microkernel Architecture	Invalid Instruction	Spawn	Dispatching unit
Switches	Instruction set	Finish	Init
Exit to User	Run Time	Register set	Profile
I/O Interrupt	Program Counter	Token	Password
First Level Interrupt Handler	Program Check	File Control Block Pointer	Master - Slave
InterruptService Routine	Resource	Message	Context switch
Interrupt Acknowledge	Resource Allocation	Message passing	Direct Memory Access
Interrupt Request	Resource Manager	Status Information	Fork
Run Time Environment	Account	Status	End
Programming Language Support	Program Loading and Execution	Application Programs	Reposition
Set	Delete	End	Abort
Load	Execute	Terminate	Get
Attribute	Set	Wait for time	Wait for event
Signal	Allocate Memory	Free	Suspend task
Sleep task	Resume Task	Set priority	Reset Priority
Enable	Disable	Mailbox	Send
Receive	Accept Control	Semaphore	Open
Close	Read	Write	Assign
Halt	Loop	Device	Kill
Time	Date	Transfer	Ready Queue
Event Driven	Iddle	Bussy	Queue
Process Identifier	Server	Initiate	Waiting Queue
Query	Ready	Return Value	Job Queue
Deadlock	Waiting		Completed
	Tree		Suspend Queue

GLOSARIO DE TÉRMINOS EN CASTELLANO

Sistema	Multiusuario	Sistema Operativo
Control de Procesos	Monousuario	Estrategia
Bloque de Control del Proceso	Sistema de Gestión de Operaciones	Contabilidad
Modo Kernel	Entrada Remota de Trabajo	Vector de Estado
Kernel	Ciclo de vida de un Proceso	Programa
Programa Ejecutable	Recurso Computacional	Programa Fuente
Espacio de Nombres de un Proceso	Recurso común	Contexto de Ejecución
Poder de un Proceso	Recurso Compartido	Espacio de Nombres del Procesador
Descendencia de un Proceso	Proceso Pesado	Espacio de Nombres de un Programa
Variables Locales	Proceso Liviano	Espacio de nombres del proceso
Procesos interactuantes	Fibra	Espacio de Memoria
Procesos reentrantes	Programas del Sistema	Crear un Proceso
Ejecución dual de instrucciones	Programa Objeto	Terminación en cascada
Llamada al Sistema	Programa ejecutable	Muerte de un Proceso
Primitivas	Tabla de procesos	Contexto de ejecución
Procesos Disjuntos	Maquina Virtual	Tarea
Procesos reutilizables	Modo Usuario	Poder de un Proceso
Procesos Concurrentes	Interrupción	Transiciones de estado
Procesos Interactuantes	Rutina de atención de interrupción	Estado Activo
Procesos Reentrantes	Servicios del S.O.	Estado inactivo
Activado	Listo	Preparado
Admitido	Suspendido	Bloqueado
Variables Globales	Variables locales	Hilo o hebra
Multitarea	Multihilo	Estado de un proceso

ACRÓNIMOS USADOS EN ESTE MÓDULO

S.O. /SO	Sistema Operativo	GUI	Graphical User Interface
PC	Program Counter	M.C./MC	Memoria Central
P.C.	Personal Computer	JCB	Job Control Block
R.J.E.	Remote Job Entry	V.M. o VM	Virtual Machine
E/S	Entrada / Salida	MV	Máquina Virtual
I/O	Input / Output	USR	User
P()	P de un semáforo	PCB	Process Control Block
V()	V de un semáforo	PCBT	Process Control Block Table
ROM	Read Only Memory	ULT	User Level Thread
RAM	Random Access Memory	KLT	Kernel Level Thread
J.C.L.	Job Control Language	PLT	Process Level Thread
HW	Hardware	SCB	System Control Block
SW	Software	PID	Proces IDentifier
CPU	Central Processing Unit	TID	Thread IDentifier

AUTOEVALUACIÓN DEL MODULO 2:

Preguntas:

1. ¿Un proceso se puede definir como una porción de un programa cargado en memoria central?.
2. ¿Siempre es necesario que una parte de la imagen de un proceso resida en memoria central?.
3. ¿El Sistema Operativo mantiene tablas para administrar los procesos?.
4. ¿Cuáles son los distintos tipos de procesos?.
5. ¿Cuáles son los pasos para crear un proceso por parte del Sistema Operativo?
6. ¿Por qué se utilizan dos modos de ejecución?
7. ¿Cuáles son los estados de un hilo?
8. ¿Cuáles son las ventajas de la utilización de MicroKernel?.
9. ¿Qué es la imagen de un proceso?
10. ¿Qué es Jacketing?.
11. ¿Cuáles son los mecanismos para producir un cambio de proceso?.
12. ¿Qué es un PCB?

Multiple Choice:

1. El PCB: a) Contiene los recursos que va a usar el proceso. b) Permite conmutar el procesador entre varios procesos. c) En algunos casos está almacenado en el área del swap del disco. d) Todas las anteriores son correctas. e) Ninguna de las anteriores son correctas.	2.- El PCB es utilizado: a) Por el S.O. para saber el Estado del proceso. b) Para enlazar al proceso en la cola de listos. c) Para enlazar al proceso en la cola de Jobs d) Por el S.O. para tener una imagen del proceso actualizada en todo momento. e) Por los procesos para abrir y cerrar sus archivos. f) Todas las anteriores g) Ninguna de las anteriores
3. Un hilo a nivel de usuario: a) Es supervisado enteramente por el Kernel. b) Es generado automáticamente cuando una aplicación comienza. c) Una de las ventajas de este nivel es que su planificación puede ser adaptada de tal forma que sea independiente de la planificación del Sistema Operativo. d) Todas las anteriores son correctas. e) Ninguna de las anteriores son correctas.	4.- Durante el ciclo de vida un proceso esta habilitado para: a) Bloquearse a la espera de un evento b) Matar sus procesos hijos c) Compartir CPU con otros procesos d) Cambiar su prioridad e) Copiar su PCB para crear procesos hijos f) Ejecutar programas g) Todas las anteriores h) Ninguna de las anteriores
5.- En un modelo de: a) 6 estados, si un proceso está bloqueado permanece en memoria secundaria. b) 7 estados, un proceso nuevo siempre es admitido, o sea, pasa a la cola de procesos Listos. c) 9 estados, un proceso nuevo puede ir al disco por falta de memoria d) Todas las anteriores son correctas. e) Ninguna de las anteriores son correctas.	6.- Los threads ULT a) Pueden trabajar en varios procesadores en paralelo. b) No pueden compartir memoria. c) Se crean mediante un biblioteca de funciones. d) Todas las anteriores son correctas. e) Ninguna de las anteriores son correctas.
7.- La creación de un proceso: a) En forma Jerárquica, el proceso hijo hereda todo el entorno de ejecución del proceso padre. b) Siempre está en manos del Sistema Operativo. c) En forma Jerárquica, el proceso padre puede esperar a la finalización de sus hijos para poder continuar. d) Todas las anteriores son correctas. e) Ninguna de las anteriores son correctas.	8.- Los Threads KLT a) Son creados por el compilador. b) Ejecutan en varios Procesadores c) Tiene un Pequeño estado no compartido d) Tienen la misma Planificación que los Procesos. e) Todas las anteriores f) Ninguna de las anteriores
9.- Las fibras a) Presentan la misma información que los threads que las agendan b) Divide la parte de ejecución de un proceso conocida como Unit of Dispatching c) Corren en el contexto de los threads que las agendan d) Se utilizan en ambientes de KLT (Kernel Level Threads) e) Todas las anteriores son correctas. f) Ninguna de las anteriores son correctas.	10.-En un sistema donde la ejecución se basa en hilos, el hilo: a) El hilo es la unidad de propiedad de los recursos. b) Un hilo puede contener varios procesos. c) El proceso es la unidad de propiedad de los recursos. d) Comparte los recursos con su proceso e) Todas las anteriores son correctas. f) Ninguna de las anteriores son correctas.
11.- Un proceso pesado... a) Es un elemento creado por el SO para lograr la multiprogramación b) Sólo se utiliza para ejecutar programas del usuario c) Contiene una copia exacta del programa en ejecución	12.-Proceso Interactuante se caracteriza por... a) No compartir variables globales b) La intersección de su PCB con otros procesos es vacía c) Puede afectar y ser afectado por otros procesos d) No comparte su estado con otros procesos

<ul style="list-style-type: none"> d) Es la unidad "despachable" de ejecución. e) Todas las anteriores f) Ninguna de las anteriores. 	<ul style="list-style-type: none"> e) El resultado de su ejecución puede predecirse ya que depende de la ejecución de otros procesos f) Es concurrente: compite por recursos con otros procesos g) Todas las anteriores son correctas. h) Ninguna de las anteriores son correctas.
<p>13. Proceso Independiente se caracteriza por....</p> <ul style="list-style-type: none"> a) Sólo tener variables locales b) Compartir variables globales sin modificarlas c) La intersección de su PCB con el resto de los procesos es vacía d) No poder afectar ni ser afectado por otro proceso e) No compartir su estado con ningún otro proceso f) Su ejecución puede detenerse y reasumirse sin causar efectos laterales al resto del sistema g) Todas las anteriores son correctas. h) Ninguna de las anteriores son correctas. 	<p>14.- Cual de las siguientes afirmaciones pertenecen a un context...</p> <ul style="list-style-type: none"> a) Salvar el contexto del proceso. b) Actualizar el PCB del procesador c) Mover el PCB a la cola apropiada. d) Seleccionar otro proceso para ejecución. e) Actualizar las estructuras de datos de gestión de memoria. f) Todas las anteriores son correctas. g) Ninguna de las anteriores son correctas
<p>15.- Los estados que puede tomar un hilo son....</p> <ul style="list-style-type: none"> a) Creado b) Ejecución c) Listo d) Bloqueado e) Terminado f) Suspendido g) Desbloqueado h) Todas las anteriores son correctas. i) Ninguna de las anteriores son correctas. 	<p>16.- La parte de control de procesos se responsabiliza de....</p> <ul style="list-style-type: none"> a) La planificación de los procesos. b) Distribución de los procesos. c) De la sincronización entre los procesos. d) Comunicación entre los procesos. e) La gestión de memoria. f) Todas las anteriores son correctas. g) Ninguna de las anteriores son correctas
<p>17.- Las operaciones sobre procesos entre otras son...</p> <ul style="list-style-type: none"> a) Renombrar b) Identificar c) Reanudar d) Despertar e) Suspendir f) Todas las anteriores son correctas. g) Ninguna de las anteriores son correctas. 	<p>18.- Bloque de Control de Sistema tiene por objetivos....</p> <ul style="list-style-type: none"> a) Enlazar los bloques de control de procesos existentes en el sistema. b) Facilitar el cambio de contexto entre procesos. c) Facilitar el control de la ejecución de un proceso. d) Requerir la atención de algún servicio del Sistema Operativo. e) Todas las anteriores son correctas. f) Ninguna de las anteriores son correctas
<p>19.- Los beneficios de usar los threads son....</p> <ul style="list-style-type: none"> a) Lleva menos tiempo crear un thread que crear un proceso. b) Lleva menos tiempo hacer un switch entre dos threads dentro del mismo proceso. c) Es más eficiente usar una colección de threads que usar una colección de procesos separados. d) Mejoran la eficiencia en la comunicación entre programas que están ejecutando. e) Los hilos dentro del mismo proceso comparten memoria y archivos y pueden comunicarse el uno con el otro sin involucrar al kernel. f) Todas las anteriores son correctas. g) Ninguna de las anteriores son correctas. 	<p>20.- Multithreading (hilos multiples) es la habilidad de...</p> <ul style="list-style-type: none"> a) Un SO para soportar múltiples hilos de ejecución dentro de un proceso. b) Un proceso para soportar múltiples hilos de ejecución dentro de un S.O. c) Un Procesador para ejecutar múltiples hilos de un S.O.. d) Un compilador de crear múltiples hilos de ejecución dentro de un proceso. e) Todas las anteriores son correctas. f) Ninguna de las anteriores son correctas

Respuestas a las preguntas

1. **Falso.** Un proceso es una porción de programa en Memoria central más su PCB o su contexto de ejecución.
2. **Verdadero,** siempre es necesario que una parte de la imagen de un proceso resida en memoria central para su ejecución.
3. **Verdadero,** El Sistema Operativo mantiene tablas para administrar los procesos.
4. Los procesos pueden ser Disjuntos, Concurrentes, Reentrantos, Interactuantes o Reutilizables.
5. Para crear un proceso, el Sistema Operativo debe: Asignar un único identificador al nuevo proceso, asignar espacio para el proceso, inicializar el PCB, establecer los enlaces apropiados y crear o ampliar otras estructuras de datos.
6. Para proteger al Sistema Operativo y a las tablas importantes del mismo.
7. Los estados son: ejecución, listo y bloqueado.
8. Algunas de las ventajas de la implementación de MicroKernels son: Flexibilidad, Portabilidad, Confiabilidad, Soporte de sistemas distribuidos y Soporte para Sistemas Operativos orientados a objetos.
9. Es un conjunto de datos para CPU, que incluye el PCB, datos del usuario, programa de usuario (o sea el programa que se ejecutará) y el stack del sistema.
10. Tiene como objetivo convertir un System Call Bloqueante en un System Call No Bloqueante.
11. Para producir un cambio de proceso se puede utilizar interrupciones, traps o System Calls.
12. Es el bloque de control de procesos, la estructura que utilizan los sistemas operativos para almacenar información sobre el proceso.

Respuestas del múltiple choice.

- | | | | | |
|-----------|---------------|--------------|-------------|---------------|
| 1.- d. | 2.- a, b. | 3.- b, c. | 4.- a, c. | 5.- c. |
| 6.- a, c. | 7.- a, c. | 8.- b, c, d. | 9.- b, c. | 10.- c, d. |
| 11.- d | 12.- c, f. | 13.- g. | 14.- c,d,e. | 15.- b,c,d,e. |
| 16.- f. | 17.- c, d, e. | 18.- e. | 19.- f. | 20.- a. |