**Resumen S.O**

**Primer Parcial**

**Introducción a los S.O (módulo 1)  
  
Definición de Sistema Operativo**

***Nota del Autor:*** *en NSO módulo 1 especifican claramente que no hay una definición exacta de sistema operativo, listando varias definiciones. Acá se muestran aquellas que transmiten la idea principal.*

El *software* para computadoras puede clasificarse en general, en 2 clases: los *programas de sistema*, que controlan la operación de la computadora en sí y los *programas de aplicación*, los cuales resuelven problemas de los usuarios. El *programa fundamental* de todos los programas de sistema, es el *Sistema Operativo*, que controla todos los recursos de la computadora y proporciona la base sobre la cual pueden escribirse los programas de aplicación.

Los Sistemas Operativos son básicamente administradores de recursos. El principal recurso que administran es el hardware del computador; además de los procesadores, los medios de almacenamiento, los dispositivos de entrada/salida, los dispositivos de comunicación, administran los procesos, el procesamiento y los datos.

Un ***Sistema Operativo* es un programa** que debe estar instalado en un computador y que actúa como intermediario entre el usuario y el hardware de un computador y su propósito es proporcionar un entorno en el cual el usuario pueda ejecutar sus programas. El objetivo principal de un Sistema Operativo es, entonces, lograr que el Sistema de computación se use de manera cómoda, y el objetivo secundario es que el hardware del computador se emplee de manera eficiente.

**Resumiendo, Sistema Operativo es:**

Un conjunto de programas que reside en la computadora y actúa como interfaz entre los

usuarios y los recursos, brindando los servicios necesarios para ejecutar programas.

Un conjunto de programas, rutinas y procedimientos manuales y automáticos que administran los recursos de un sistema de cómputos en operación para permitir a un grupo de usuarios, compartir eficientemente el mismo.

**Resumiendo más aún:**

Se podría decir que los Sistemas Operativos son un conjunto de programas que

crean la interfaz entre el hardware y el usuario, y que tiene cuatro funciones primordiales, que son:

* **Gestionar el hardware**: Se refiere al hecho de administrar de una forma más eficiente los recursos de la máquina.
* **Facilitar el trabajo al usuario**: Permite una comunicación con los dispositivos de la máquina.
* **Brindar un ambiente de ejecución seguro**. Protegiendo el uso correcto de los recursos.
* **Inicializar la máquina**.

**En cuanto a los Sistemas Operativos podemos afirmar que:**

* Depende del Hardware que controla debido a que complementa al juego de instrucciones del procesador.
* Puede ser ejecutado por el mismo procesador que ejecuta el programa del usuario.
* Cuando se está ejecutando el programa usuario, el S.O. está inactivo si es monoprocesador.
* Los detalles de interfaces internas varían de máquina en máquina, dado que está íntimamente ligado a los recursos del Hardware, que administra y controla.
* Las capacidades externas y la interfaz humana varían en función de las prestaciones que ofrece el S.O. Muchas veces sirven como argumentos de venta o atracción tecnológica sin ofrecer substanciales mejoras en los servicios que dice haber incorporado el fabricante o proveedor del Sistema Operativo.
* Los servicios que provee el S.O. a los programas usuarios cuando estos se ejecutan, son denominados llamadas al sistema (System Calls o SYSCALL).

**Definición de máquina extendida y Shell**

El S.O. debe presentar una interfaz que oculte toda la complejidad de la máquina real para transformarse en una "máquina" más fácil de usar (más *"amigable"* o sea *User Friendly*). A esto es lo que hacemos referencia con máquina extendida o interfaz Hombre – Máquina.Generalmente la visión de los usuarios de esta interfaz se conoce con el nombre de SHELL.

El Shell es la porción del S.O. que se encarga de hacer de interfaz Hombre - Máquina y de transmitir al S.O. los requerimientos del usuario al sistema. La interfaz humana es la que nos permite manipular el sistema y es la parte visible del S.O. y puede ser intercambiable.

**Las funciones principales de la máquina extendida son:**

* Separar la complejidad de la máquina desnuda al usuario. Una parte del S.O. se ocupa de eliminar la complejidad del Hardware y lo transforma en una máquina virtual más fácil de usar con una interfaz para el usuario más amigable y menos compleja.
* Actuar como interfaz de Entrada/Salida (E/S), y controlar el complejo manejo de los

dispositivos de E/S cuando el usuario requiere de un servicio.

* Facilitar la comunicación con el usuario (mediante el SHELL), ya sea a través del Intérprete de comandos, o del *Job Control Language* (J.C.L.), u órdenes verbales, o a través de Interfaz Gráfica compuesta por iconos.
* Aceptar entradas de nuevos Trabajos.
* Proteger el sistema, permitiendo solamente el acceso a usuarios correctamente identificados y autorizar en el uso del sistema, a través de distintos métodos de autentificación (usuario/clave, reconocimiento de huellas dactilares, etc.)

**Características de los S.O.**

En general, se puede decir que un Sistema Operativo tiene las siguientes características funcionales:

* **Conveniencia**: Un Sistema Operativo hace más conveniente el uso de una computadora.
* **Eficiencia**: Un Sistema Operativo permite que los recursos de la computadora se usen de la

manera más eficiente posible.

* **Habilidad para evolucionar**: Un Sistema Operativo deberá construirse de manera que permita el servicio del mismo.
* **Encargado de administrar el hardware**: El Sistema Operativo se encarga de manejar de una mejor manera los recursos de la computadora en cuanto a hardware se refiere, esto es, asignar a cada proceso una parte del procesador para poder compartir los recursos.
* **Relacionar dispositivos** (gestionar a través del kernel): El Sistema Operativo se debe encargar de

comunicar a los dispositivos periféricos, cuando el usuario así lo requiera.

* **Organizar datos**: para acceso rápido y seguro.
* **Manejar las comunicaciones en red**: El Sistema Operativo permite al usuario manejar con alta

facilidad todo lo referente a la instalación y uso de las redes de computadoras.

* **Procesamiento por bytes**: de flujo a través del bus de datos.
* **Facilitar las entradas y salidas**: Un Sistema Operativo debe hacerle fácil al usuario el acceso y manejo de los dispositivos de Entrada / Salida de la computadora.
* **Técnicas de recuperación de errores**: Debe proveer las rutinas necesarias para protegerse, y poder recuperarse de los errores (ya sean de hardware, o de software), con el menor perjuicio para los usuarios.
* **Evita que otros usuarios interfieran**: Debe evitar que los usuarios se bloqueen entre ellos, informándoles si esa aplicación está siendo ocupada por otro usuario.
* **Generación de estadísticas**: Para poder *cobrarle* a los diferentes usuarios del centro de costos la parte proporcional que les corresponde del gasto generado por el mantenimiento del mismo.
* **Permite que se pueda compartir**: tanto el hardware como los datos de los usuarios y entre los usuarios.

**Características comunes a todos los S.O**

En general los cuatro problemas comunes que deben tratar y resolver cualquier S.O. son:

* Gestión y reparto del conjunto de recursos.
* Designación de objetos y acceso a la información de esos objetos.
* Cooperación entre procesos paralelos también llamados concurrentes.
* Protección para preservar la integridad de los recursos y los usuarios.

**Modo dual de ejecución del procesador**

También conocido como juego de instrucciones diferenciadas del procesador.

El modo dual de ejecución del procesador es una medida de seguridad implementada en los sistemas operativos. Consiste en un modo privilegiado (modo kernel) que es el encargado de ejecutar instrucciones del sistema operativo, y un modo usuario que ejecuta instrucciones comunes, de programas del usuario.

De esta forma se logra proteger al S.O. de violaciones cometidas por los usuarios y también se

protege a los usuarios entre ellos mismos, simplemente clasificando como instrucciones privilegiadas aquellas que solo pueden ejecutarse en modo Kernel, como las instrucciones que pueden causar algún daño si son mal utilizadas por los programas de los usuarios (por ejemplo: un direccionamiento a áreas de memoria central no autorizadas).

El programa en modo usuario utiliza llamadas al sistema (system calls) cada vez que necesite realizar una instrucción cuyo modo de ejecución solo pueda ser hecho a través del kernel. Por ejemplo, si un programa necesita realizar una instrucción de E/S, lo hace a través de una syscall.

Las syscalls producen un cambio de contexto de ejecución. Se cambia la ejecución de modo usuario a modo kernel.

**Interrupciones**

Se llama interrupción a la detención del programa en ejecución debido a una condición externa al procesador, o sea, que el procesador es forzado a reconocer la ocurrencia de un evento en el sistema mediante una señal. En general, el procesador después de preservar los contenidos de todos los Registros y cierta información acerca del estado del proceso (salva en el stackde la memoria central como un VECTOR DE ESTADO o Bloque de Control del Proceso PCB), reemplaza el contenido del contador de programa (Program Counter PC), por una dirección predeterminada donde se encuentra la rutina de interrupción. A continuación, el procesador comienza a ejecutar la rutina de interrupción, que es un programa escrito para responder a la condición generada por el pedido de atención de esa interrupción.

Dependiendo del tipo de interrupciones, intervendrá el S.O. para dar un adecuado tratamiento al evento que generó el Pedido de Atención de la Interrupción.

Las interrupciones pueden clasificarse de la siguiente forma:

* **Interrupciones de hardware externo:** estas son asíncronas a la ejecución del procesador, es decir, se pueden producir en cualquier momento independientemente de lo que esté haciendo el CPU en ese momento. Las causas que las producen son externas al procesador y a menudo suelen estar ligadas con los distintos dispositivos de entrada o salida.
* **Interrupciones de hardware interno:** son las excepciones. Son aquellas que se producen de forma síncrona a la ejecución del procesador y por tanto podrían predecirse si se analiza con detenimiento la traza del programa que en ese momento estaba siendo ejecutado en la CPU. Normalmente son causadas al realizarse operaciones no permitidas tales como la división entre 0, el desbordamiento, el acceso a una posición de memoria no permitida, etc.
* **Interrupciones por software:** Las interrupciones por software son aquellas generadas por un programa en ejecución. Para generarlas, existen distintas instrucciones en el código máquina que permiten al programador producir una interrupción. En general, las son las llamadas a sistema (syscalls).

**Arquitecturas del kernel de un S.O**

**Estructura Monolítica:** el S.O entero se ejecuta dentro del kernel space. Todas las rutinas para las diversas tareas del S.O (file system, planificador de procesos, manejo de memoria) se ejecutan en el kernel space. El problema es que, si una de estas rutinas está mal programada, todo el kernel puede fallar, generando así una falla general en S.O.

Los primeros sistemas operativos utilizaban esta estructura.

**Estructura Microkernel:** solo los componentes indispensables se ejecutan en modo kernel. Lo demás es ejecutado en el user space. Esta arquitectura es más lenta, ya que los servicios ejecutados en user space deben comunicarse con el kernel a través de un proceso interno en vez de hacerlo directamente. La ventaja es que, si una rutina está mal programada, solo se afecta a sí misma, y no daña al resto del kernel.

**Estructura híbrida:** es una mezcla de ambas estructuras donde solo algunas rutinas esenciales son ejecutadas en modo kernel, mientras que otras son ejecutadas en modo usuario a medida que se las necesiten.

Cosas que no resumí pero que recomiendo leer:

* Tipos de sistemas operativos (Notas, Módulo 2, páginas 28 a 32). (leer por encima)

**De programas a procesos (módulo 2)**

**Definiciones generales**

* **Instrucción:** una instrucciónes 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.
* **Programa:** definimos como programaal 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.

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 **J**ob **C**ontrol **B**lock (**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.

* **Proceso:** un procesoes 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 (P**rocess **C**ontrol **B**lock **- 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** (**P**rocess **C**ontrol **B**lock), 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.
* **Hilos:** un **hilo o *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.

**El Kernel no es un 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.

**Descripción de los procesos**

Un programa es una entidad pasiva; 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.

**Bloque de control de procesos (PCB)**

El PCB es una estructura de datos donde se guarda información relevante al 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.

El Sistema Operativo mantiene para cada proceso un bloque de control 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 estructura:

* 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 más 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.

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.

**Estado de los procesos**

Inicialmente los programas y sus datos son colocados en la **Job Queue (Cola de Trabajos).**

Generalmente (los programas más datos más argumentos) residen en el almacenamiento secundario y esperan su ejecución.

A partir de que el S.O. los recibe va creando procesos y los ubica en una cola de Procesos creados con el estado de Nuevo en sus PCB. A partir de aquí, se planifica la ejecución sobre la base de los recursos disponibles (Memoria Central y Procesador) y la prioridad del proceso. Si se dispone de suficiente Memoria y de procesador, el S.O. los coloca en la Cola de Listos como proceso activo para su ejecución.

**Ready Queue (Cola de Procesos Listos o preparados):** procesos que residen en la Memoria

Central y que están listos esperando su ejecución. Es una lista enlazada con un encabezado de la 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.

***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).

***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).

Son de dos tipos:

* **Suspendido bloqueado (Suspended – Blocked):** Es el proceso que fue suspendido en espera de un evento, sin que haya desaparecido las causas de su bloqueo (e.g: aún espera por E/S).
* **Suspendido programado (Suspended – Ready)**: Es el proceso que ha sido suspendido, pero no tiene causa para estar bloqueado (e.g: ya que su petición de E/S ha sido contestada y puede volver a ejecución).

**Cambio de contexto o ejecución**

Un concepto muy importante que incluye el concepto de proceso, es el denominado ***cambio de***

***contexto***. Cuando 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 (context switch)*** 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, hay más información involucrada. Este cambio, que demora de unos pocos a varios microsegundos, dependiendo de la velocidad del procesador, es puro sobrecarga (**overhead**), 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).

**/!\** Siempre que hay un process switch, hay un context switch. Pero un context switch no necesariamente involucra un process switch. Un proceso puede generar una syscall para pedir E/S, produciendo así un cambio de contexto a modo kernel, pero no necesariamente ese cambio de contexto involucre que se haga un cambio de proceso en ejecución. **/!\**

**Estructuras de control de proceso**

* **Ubicación de los procesos:** un proceso constará al menos de la suficiente memoria para albergar sus programas y los datos del proceso mismo. Para que un S.O pueda administrar al proceso, necesita la información referente al mismo, que se encuentra en el PCB. No es necesario tener todo el PCB cargado en memoria central, parte del mismo puede permanecer en disco. Lo mismo sucede con el proceso en sí, no es necesario cargar todo a memoria principal, parte puede permanecer en disco (ver: parcial 2, administración de memoria con swapping).
* **El papel del PCB:** el PCB es una estructura de mayor importancia cuando queremos manejar procesos en un sistema multiprogramado. Si no tuviéramos dicha estructura, sería imposible poder manejar los diferentes procesos (ya que no podríamos saber su estado, su contexto de ejecución, etc).   
    
  Los PCBs pueden ser dañados debido a fallas. Pero esto se puede evitar con rutinas o funciones del S.O. que se encarguen de proteger los PCBs. Aunque esto genera overhead, lo que hace más lenta la ejecución en sí.

**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 (pero no confundir con paralelismo de procesos), 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 (en realidad, esta es la definición más usada, los procesos concurrentes son procesos que se ejecutan en un mismo periodo de tiempo).

***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.

**Hilos o Threads**

Un Thread (hilo), es una unidad elemental de uso del procesador. 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.

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** (**T**hread **ID**entifier) 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.

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 ejecutar 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 (solo si es KLT). 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.

Todos los hilos de un proceso comparten el estado y los recursos de ese proceso. Es decir, los hilos existen en el mismo espacio de memoria que el proceso que los creó, por lo que pueden acceder a datos del mismo. Si un hilo modifica un ítem de datos de su porción de memoria, otros hilos del mismo proceso ven el cambio si acceden al ítem.   
  
Los hilos pueden comunicarse entre sí sin involucrar al kernel, ya que comparten la misma memoria, archivos, y recursos del proceso, por lo que no es necesario utilizar algún medio de IPC.

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.

**Implementación de hilos**

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)

***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**, 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.

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.

***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.

***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.

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.

***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 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.

**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).

**Uso de 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.

* **Estructura Servidor/Jefe Trabajador:** un hilo tiene una tarea distinta de todos los demás: el hilo jefe genera o recopila tareas para realizar, las separa y se las entrega a los hilos trabajadores.

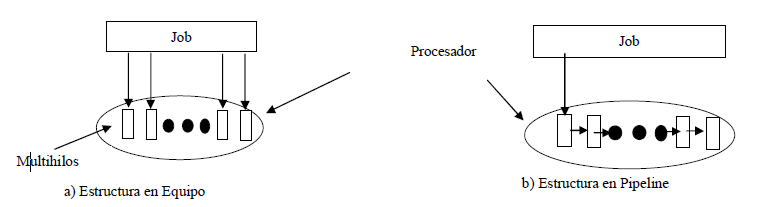
Este modelo es el más común para procesos que implementan servidores (es el modelo clásico del servidor Web Apache) y para aplicaciones gráficas (GUI), en que hay una porción del programa (el hilo jefe) esperando a que ocurran eventos externos. El jefe realiza poco trabajo, se limita a invocar a los trabajadores para que hagan el trabajo de verdad; como mucho, puede llevar la contabilidad de los trabajos realizados.

Típicamente, los hilos trabajadores realizan su operación, posiblemente notifican al jefe de su trabajo, y finalizan su ejecución.  
Hay 3 formas de encarar esta estructura: la primera se basa en crear un nuevo hilo por cada petición (paralelismo). La segunda se trata de un servidor de un solo hilo (sin paralelismo). La tercera es una máquina de estado finito que utiliza llamadas al sistema no bloqueantes (con paralelismo).

* **Estructura en equipo:** al iniciar la porción multihilos del proceso, se crean muchos hilos idénticos, que realizarán las mismas tareas sobre diferentes datos.

Su principal diferencia con el patrón jefe/trabajador consiste en que el trabajo a realizar por cada uno de los hilos se plantea desde principio, esto es, el paso de división de trabajo no es un hilo más, sino que prepara los datos para que éstos sean lanzados en paralelo. Estos datos no son resultado de eventos independientes (como en el caso anterior), sino partes de un solo cálculo. Como consecuencia, resulta natural que en este modelo los resultados generados por los diferentes hilos son agregados o totalizados al terminar su procesamiento. Los hilos no terminan, sino que son

sincronizados y luego continúan la ejecución lineal.

* **Estructura entubamiento (Pipeline):** si una tarea larga puede dividirse en pasos sobre bloques de la información total a procesar, cada hilo puede enfocarse a hacer sólo un paso y pasarle los datos a otro hilo conforme vaya terminando. Una de las principales ventajas de este modelo es que ayuda a mantener rutinas simples de comprender, y permite que el procesamiento de datos continúe, incluso si parte del programa está bloqueado esperando E/S.   
    
  Un punto importante a tener en cuenta en una línea de ensamblado es que, si bien los hilos trabajan de forma secuencial, pueden estar ejecutándose paralelamente sobre bloques consecutivos de información, eventos, etcétera.   
    
  Este patrón es claramente distinto de los dos anteriormente presentados; si bien en los anteriores los diferentes hilos (a excepción del hilo jefe) eran casi siempre idénticos (aplicando las mismas operaciones a distintos conjuntos de datos), en este caso son todos completamente distintos.  
    
  ****

**Aspectos importantes del diseño de paquetes de threads**

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.

**Implementación de un paquete de hilos  
  
Paquete de hilos en el espacio del usuario (ULT):** 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 (KLT):** 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.

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.

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.

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 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.

**El concepto de fibra**

Una fibra es una unidad de ejecución que debe ser agendada (Schedule, es decir, se le indica cuando debe ejecutarse) por la aplicación (programa del usuario). A diferencia de los hilos comunes, que están agendados por el sistema operativo, siendo éste el que controla cuando se ejecuta un thread o no.   
  
Si bien en un sistema que utilice ULT es el programa el que decide cuál hilo se ejecuta a continuación, el programa no tiene control sobre su propia ejecución, es decir, depende del scheduling realizado por el S.O para ejecutarse.

Esta es la principal diferencia de las fibras. Las fibras las maneja el programa del usuario, siendo éste el encargado de decirles cuando ejecutarse (a través de una syscall, e.g, SwitchToFiber) y cuando terminar su ejecución (también a través de una syscall). Esto nos asegura evitar las problemáticas del manejo de datos que provienen por la concurrencia de hilos, ya que la fibra no dejará de ejecutarse hasta que se le ordene explícitamente.

Como resultado de lo anterior, si la fibra entra en un loop infinito, ninguna otra fibra va a poder ejecutarse…

**Planificación (módulo 3)**

Debido a que los sistemas operativos evolucionaron a un modelo multiprogramado, fue necesario implementar algún método que permitiera organizar la forma en que se ejecutaría cada proceso.

La planificación puede definirse como un **conjunto de políticas y mecanismos incorporados al sistema operativo, a través de un módulo denominado planificador**, que debe decidir cuál de los procesos en condiciones de ser ejecutado conviene ser despachado primero y qué orden de ejecución debe seguirse.

**Niveles de planificación**

* **Planificación a extra-largo plazo:** la planificación a extra largo plazo es una planificación externa al sistema operativo, la cual se hace en una escala de tiempo humana. Es una planificación que se hace en concordancia con las políticas empresariales que definen cómo funcionará el centro de cómputos, que procesos o tareas son de mayor prioridad, y cómo se deberá atender a los usuarios. Son las reglas, procedimientos escritos, o políticas, que indican cómo debe usarse, cuál es la seguridad, quién tiene acceso, cuáles son las prioridades, etc.
* **Planificación a largo plazo:** se encuentra a cargo del Job Scheduler (o Long Term Scheduler). Decide cuál será el próximo trabajo que se ejecutará, para lo cual carga el programa y sus datos y crea los procesos (esto implica también crear un PCB para el proceso mismo). Si los recursos necesarios están disponibles, pone al proceso en la cola de listos (ready queue).   
    
  El objetivo del largo plazo es proporcionar una mezcla equilibrada de trabajos, tales como limitados por el procesador (CPU bound) y limitados por Entrada / Salida (I/O bound). Esto se conoce como ***Balance de carga del sistema***. En cierto modo actúa como un regulador de admisión de primer nivel para mantener la utilización de recursos al nivel deseado sin sobrecargar al sistema. Entonces debe seleccionar con cuidado que tipos de trabajos entran al sistema para mantener un adecuado balance de carga de procesos que hacen uso de Entrada/Salida intensivo o uso de procesador intensivo.  
    
  El Scheduler ejecuta con poca frecuencia; sólo cuando se necesita crear un proceso nuevo en el Sistema por la entrada de un trabajo, cuando termina un trabajo mediante un proceso de finalización, o ingresa un usuario al sistema, por lo que tiene prioridad máxima para ejecutar. Es el responsable de controlar el nivel de multiprogramación del sistema y generar el orden a seguir en la ejecución de los procesos.

El Shcheduler es el padre de todos los procesos.

* **Planificación a mediano plazo:** Middle Term Scheduler (o Planificador de Swapping) es el que decide sacar de memoria central y llevar al disco (swap-out) a aquellos procesos inactivos o a los activos cuyos estados sean bloqueado momentáneamente o temporalmente o los suspendidos y luego, cuando desaparezcan las causas de sus bloqueos, traerlos nuevamente a Memoria (swap-in) para continuar su ejecución.   
    
  Este tipo de planificación solo es usado en sistemas con mucha carga de procesos, ya que el procedimiento de swapping produce mucho overhead, haciendo bajar considerablemente el desempeño general.
* **Planificación a corto plazo:** el planificador de procesos (Short Term Scheduler) es el responsable de decidir quién, cuándo, cómo y por cuánto tiempo recibe procesador un proceso que está preparado en la Ready Queue para ejecutar, además en S.O. con esquemas expropiativos se ocupa de quitar el recurso procesador al proceso que está ejecutando. También verifica las interrupciones y las trata.  
    
  El planificador a corto plazo es invocado cada vez que un suceso (interno o externo) hace que se modifique el estado global del sistema. Por ejemplo:  
  + Tics de reloj (interrupciones basadas en el tiempo).
  + Interrupciones por comienzo y terminaciones de Entrada / Salida.
  + La mayoría de las llamadas al S.O.
  + El envío y recepción de señales.
  + La activación de programas interactivos.

**Proceso nulo o vacío**

El proceso nulo es un proceso que se ejecuta cuando el procesador no tiene nada para hacer. Es una forma de evitar que el procesador quede en un estado ocioso. El proceso nulo es de muy baja prioridad. Puede tener funciones como realizar estadísticas, limpieza del sistema, etc.

**Planificación de Trabajos**

Hola. Si vos como yo estás confundido sobre la definición de qué es un Job/Trabajo, tranquilo, no tengas miedo. No hay una definición clara, es un término ambiguo que tiene su historia en la época de las computadoras gigantes. La definición más usual de un Job es “un conjunto de procesos”. Pero acá lo podemos tomar como si fuera una tarea a cumplir, algo a ejecutar.

Si querés indagar más en esto, acá tenés un enlace: <https://stackoverflow.com/questions/3073948/job-task-and-process-whats-the-difference/31212568>

O también podés leer la definición que da el Notas en el módulo 1, acá abajo:

**TRABAJO (JOB)**: es un conjunto de programas (generalmente de aplicaciones) y datos de los cuales se quiere obtener un resultado mediante las ejecuciones en un computador. Un trabajo puede dividirse en ***pasos de trabajo (job steps*)**, que son los procedimientos o funciones incorporados por el programador en el programa. Cada uno de estos pasos es una unidad de trabajo que se hace en forma secuencial mediante procesos (por ejemplo, el ensamblado de un programa), una vez que el S.O. ha aceptado el Trabajo. Entonces podemos definir a un ***proceso (process)***, *como un conjunto de operaciones que se ejecutan en forma secuencial en el tiempo.*

La planificación de trabajos está a cargo del Job Scheduler. La tarea del Job Scheduler es decidir qué procesos crear y mandar a la cola de listos a medida que estos van llegando. A diferencia del planificador de corto plazo, este planificador no asigna ningún proceso a la CPU. Simplemente dice quién va primero en la cola de listos luego de ser creado.

Existen algunos algoritmos de planificación de trabajos, entre ellos:

* **FCFS (First Come First Serve):** el primero que llega es el primero en ser atendido. Se utiliza una cola FIFO para su implementación. Es el más sencillo de los algoritmos. Pero suele presentar problemas como tiempos medios de espera elevados, bajo nivel de utilización del procesador, y pobre tiempo de respuesta en procesos cortos en esquemas con mucha carga.
* **SFJ (Shortest Job First):** a través del uso de estadísticas y mediciones, se calcula el tiempo de ejecución de un proceso. A medida que los procesos llegan, se verifica cuál de estos tienen menos tiempo de ejecución, siendo estos los elegidos para ser creados primero y luego pasarlos a la cola de listos. Puede producir inanición, debido a que un proceso muy largo será siempre dejado para el final en el caso de que lleguen procesos más cortos.
* **Planificación por prioridad:** los trabajos tienen asignados una prioridad. Los trabajos de prioridad más alta son los primeros en ser asignados a la cola de listos. Si dos trabajos tienen la misma prioridad, se utiliza FCFS. También puede producir inanición en el caso de un proceso de muy baja prioridad.

**Planificación de procesos**

Acá es donde se decide qué proceso va a estar ejecutándose actualmente en la CPU. Es una planificación a corto plazo porque se hacen decisiones muy rápidamente en un período de tiempo corto.

Generalmente los algoritmos de planificación de corto plazo se basan en la siguiente propiedad de los procesos:

Durante su ejecución, un proceso alterna entre períodos de uso de CPU (***CPU bursts****)* o ráfagas de CPU y períodos de uso de E/S (***I/O bursts***) Los CPU burst pueden estar limitados por tiempo o por cantidad de instrucciones esto se conoce como ***CPU bound***.

* La ejecución siempre comienza y termina por un CPU burst.
* En sistemas con procesos E/S bound (limitado por E/S), la duración de los CPU bursts suele ser muy corta.
* En sistemas con procesos CPU bound, la duración de los I/O bursts es acotada de acuerdo al quantum de tiempo otorgado al proceso.

**Traffic controller**

Se ocupa del manejo de la cola de procesos listos. Es quien arma la cola de procesos listos o preparados para ejecutar para lo cual usa un algoritmo de planificación.

**Dispatcher**

El dispatcher es un módulo del sistema operativo involucrado en la planificación de procesos. Es el módulo que le da el control del CPU al proceso seleccionado por el planificador a corto plazo. El dispatcher es ejecutado en modo kernel como resultado de una interrupción o una syscall.

Las funciones del dispatcher son:

* Hacer los context switches, en los cuales el disptacher guarda el estado del proceso que estaba en ejecución, y carga el estado anterior o inicial del nuevo proceso a ejecutar.
* Cambiar a modo usuario.
* Saltar a la ubicación correcta dentro del programa para que continúe con su ejecución de acuerdo a su estado.

**Non preemtive vs preemtive**Multitarea cooperativa = non-preemtive 🡪 el proceso en ejecución decide cuándo soltar el procesador y decidir que se ejecute otra cosa. El S.O nunca fuerza un context switch.  
  
Multitarea aprobativa = preemtive. 🡪 los procesos solo son ejecutados por una cantidad de tiempo, el S.O fuerza context switch.

**Algoritmos apropiativos (non-preemptive)**

Este tipo de algoritmos tienen la particularidad de que, al ser elegido para ejecutarse un proceso, este no abandona el procesador hasta que termine su ejecución. Se APROPIAN del procesador y no lo sueltan.

* **FCFS:** su funcionamiento es igual al FCFS visto en largo plazo.
* **SPF (Shortest process first):** asocia a cada proceso la longitud de su siguiente ráfaga de procesador (CPU burst). Cuando el procesador queda libre, se asigna al proceso que tenga la siguiente ráfaga de CPU más pequeña. Si dos procesos tienen la misma ráfaga de CPU, se utiliza el FCFS. Obviamente que se requiere obtener la información de la longitud de la ráfaga para realizar la planificación y este valor es relativamente difícil determinar o, en el peor de los casos, no hay manera de saber la longitud de la siguiente ráfaga de CPU. Un enfoque que resuelve este problema consiste en tratar de aproximar la planificación del SPF, utilizando predicciones sobre las duraciones de las siguientes ráfagas.  
    
  SPF también puede ser preemptive. Es decir, un proceso puede ser interrumpido en su CPU burst (imposible de hacerlo en el esquema non-preemptive). En ese caso, se elige el proceso al que le queda menos tiempo para finalizar su CPU burst (shortest-remaining time first).
* **Planificación por prioridades:** Se asocia un número a cada proceso, ese número indica la prioridad con que será seleccionado para su ejecución. El significado de los números es arbitrario. Algunos S.O. consideran a mayor número mayor prioridad, en cambio otros a menor número, mayor prioridad. Un proceso puede entrar con una prioridad fija y luego ser modificada, ya sea por el propio usuario o por el S.O.

**Algoritmos no apropiativos (preemptive)**

Estos algoritmos liberan al procesador del proceso que lo está usando para seleccionar uno se la cola de listos como reemplazo. Dado que el proceso por sí mismo no deja el uso del procesador, se debe incorporar un mecanismo para forzar un context switch. Generalmente se recurre a una interrupción, por ejemplo, de reloj que requiere una atención del Kernel y este aprovecha esta circunstancia para cambiar el PCB del proceso en uso del procesador por otro PCB de la cola de Listos.

* **Round robin:** a cada proceso se le es asignado un tiempo de ejecución en el CPU llamado Quantum. Un quantum es una franja de tiempo pequeña que indica cuánto tiempo de uso de CPU puede tener cada proceso.   
    
  Para implementar la planificación RR, la cola se mantiene como una cola de procesos FIFO. Los procesos nuevos se añaden al final de la cola. El planificador de la CPU selecciona el primer proceso de la cola, fija un temporizador para interrumpir, tras cumplirse un quantum de tiempo, se produce la interrupción por reloj que despacha el proceso a la cola de listos y selecciona otro proceso.  
    
  No siempre ocurre una interrupción para ser removido un proceso de la CPU. Hay casos en que el proceso puede tener una ráfaga de CPU (**CPU burst**) inferior a un quantum de tiempo debido a que se completó o requiere de una operación de Entrada/Salida, en este caso, el proceso mismo libera la CPU voluntariamente, emitiendo un pedido de E/S o una llamada al sistema de terminando. Entonces se pasa al proceso siguiente en la cola.   
    
  En muchos casos, la ráfaga de CPU del proceso en ejecución es mayor que el quantum de tiempo, el temporizador sobrepasará su límite y ocasionará una interrupción al sistema operativo. Los registros del proceso interrumpido se guardan en su bloque de control de proceso, y el proceso se pone al final de la cola. Entonces el planificador selecciona el proceso siguiente de la cola y le asigna el

quantum de tiempo siguiente.

En el algoritmo de planificación Round Robin puro no se asigna la CPU a ningún proceso durante más de un quantum de tiempo consecutivo si hay otros procesos en la cola de Listos. Si la cola está vacía seguirá ejecutando el proceso hasta que haya un nuevo proceso en la Ready Queue.  
  
Un proceso puede salir del estado de ejecución por tres motivos:  
a) que termine su ejecución,  
b) se proceda al llamado a una entrada – salida y el proceso se quede bloqueado y

c) que se supere el quantum de ejecución del proceso.  
  
Una de las desventajas de este algoritmo es el overhead que es causado debido al cambio constante de procesos.

* **Menor tiempo restante (SRT):** es la versión expropiativa del SPF. En este caso el planificador siempre selecciona primero para ejecutar al proceso con menor tiempo restante o remanente, o sea, selecciona al proceso al que le queda menos tiempo esperado de ejecución en el procesador, Un proceso puede ser expulsado cuando otro proceso está listo.   
    
  Cuando un nuevo proceso que ya ejecutó al menos una vez, se agrega a la cola de listos, puede tener un tiempo restante de ejecución menor que el que se encuentra ejecutando actualmente, por lo que pasa a ejecución.   
    
  Al igual que con SPF se corre el riesgo de inanición de los procesos largos lo cual haría que el sistema no sea eficiente. Una ventaja que presenta es la de poseer el mejor rendimiento, ya que los procesos más cortos toman inmediatamente el control de la CPU.   
    
  Cada vez que un proceso es agregado a la cola por el planificador de medio o de largo plazo, y tenga un tiempo restante de ejecución menor del que se encuentra ejecutando, es inmediatamente puesto en ejecución por el dispatcher y retirado el que se encontraba en ese estado.
* **Planificación con múltiples colas:** Establece un conjunto de colas de planificación y sitúa los procesos en las colas, teniendo en cuenta, entre otros criterios, el historial de ejecución. Favorece a los procesos más nuevos y cortos que a los más viejos y largos. Un problema que presenta el sencillo esquema anterior es que el tiempo de retorno de los procesos mayores puede alargarse de forma alarmante. Puede ocurrir inanición si llegan regularmente nuevos trabajos al sistema.

Este tipo de planificación permite a un proceso pasar de una cola a otra. La idea es separar los procesos con diferentes características en cuanto a sus ráfagas de CPU. Si un proceso gasta demasiado tiempo en CPU, se le pasará a una cola con menor prioridad. Este esquema deja los procesos limitados por E/S y los procesos interactivos en las colas de más alta prioridad. Así mismo, si un proceso espera demasiado tiempo en una cola de baja prioridad pudiese pasarse a una de mayor prioridad, evitando así la inanición.  
  
Dentro de cada cola se utiliza el algoritmo de planificación FCFS, en el caso de la última se utiliza el algoritmo Round Robin.

Cosas que no resumí pero que recomiendo leer:

* Planificación de múltiples procesadores (Notas, Módulo 3, páginas 176 a 178).
* Planificación de hilos (Notas, Módulo 3, páginas 181 a 182).

**Sincronización de Procesos (módulo 4)**

En general, los procesos de un sistema operativo comparten un espacio en común de memoria principal (RAM) o un recurso. El acceso a estos recursos compartidos o espacios de memoria puede generar problemas si no hace con cuidado. Para resolver los problemas de competencia entre procesos se utilizan dos mecanismos: la sincronización y la comunicación.

**Sincronización entre procesos:** Ordenamiento de las operaciones en el tiempo debido a las

condiciones de carrera (acceder a los diversos recursos asincrónicamente, e.g: dos procesos que intentan acceder a una impresora al mismo tiempo).

**Comunicación entre procesos:** Intercambio de datos. La comunicación permite que los

procesos cooperen entre sí en la ejecución de un objetivo global, mientras que la sincronización permite que un proceso continúe su ejecución después de la ocurrencia de un determinado evento.

**Condición de carrera**

Las condiciones de carrera ocurren cuando dos o más procesos intentan acceder a un recurso al mismo tiempo. Esto puede generar problemas cuando dichos procesos intentan acceder al recurso en un orden no esperado por el programador. Si esto pasase, la salida del programa se vería afecta y los resultados serían impredecibles.

Las condiciones de carrera existen porque, en su estado puro, los procesos no tienen ninguna noción de lo que están haciendo otros procesos. Si un proceso necesita un recurso, irá a tomarlo lo más rápido posible. Y una vez que lo tenga, hará lo que deba hacer con ese recurso. Pero si no era el turno de dicho proceso de utilizar tal recurso, la salida del programa ejecutado será impredecible, o errónea.

**Concurrencia entre procesos**

En un sistema multiprogramado con un único procesador, los procesos se intercalan en el tiempo para dar la apariencia de ejecución simultánea. Aunque no se consigue un proceso paralelo real y se produce una cierta sobrecarga en los intercambios de procesos, esta ejecución produce beneficios importantes en eficiencia y estructuración de programas. En un sistema con varios procesadores, no sólo es posible intercalar los procesos, sino también superponerlos.

En ambos casos, los problemas de concurrencia parten del hecho de que la velocidad relativa de ejecución de los procesos no puede predecirse, ya que depende de la actividad de otros procesos, del tratamiento de las interrupciones y de las políticas de planificación.

Así surgen las siguientes dificultades:

1. Compartir entre procesos los recursos globales está llena de riesgos.
2. Para el S.O. resulta difícil asignar los recursos de forma óptima.
3. Resulta difícil localizar un error de programación porque los resultados no son normalmente reproducibles.

**/!\ Concurrencia vs Paralelismo /!\**

Esta es una típica pregunta de parcial. Concurrencia y paralelismo son dos conceptos que se confunden a menudo.

Concurrencia significa que hay muchos procesos que se inician, se ejecutan, y se completan en tiempos que se superponen, sin ningún orden en específico. El paralelismo se da cuando muchos procesos se ejecutan literalmente al mismo tiempo, por ejemplo, en un procesador multi-núcleo.

Para que exista paralelismo, los procesos deben estar ejecutándose en el mismo tiempo, el mismo instante (justamente, EN PARALELO).

Para que exista concurrencia, los procesos deben ejecutarse en el mismo período o franja de tiempo. No están los dos siendo ejecutados al mismo instante, solo uno se ejecuta a la vez. Pero ambos comparten una franja de tiempo donde son ejecutados.

**Problemas concurrentes**

**Programas secuenciales:** un programa secuencial especifica una secuencia de instrucciones que se ejecutan sobre un procesador que definimos como proceso o tarea.

**Programas concurrentes:** un programa concurrente especifica dos o más procesos secuenciales que pueden ejecutarse concurrentemente como tareas paralelas.

Un proceso secuencial se caracteriza por no ser dependiente de la velocidad de ejecución y de

producir el mismo resultado para un mismo conjunto de datos de entrada, mientras que en un proceso concurrente (o lógicamente paralelo) las actividades están superpuestas en el tiempo (una operación puede ser comenzada en función de la ocurrencia de algún evento, antes de que termine la operación que se estaba ejecutando).

La programación concurrente requiere de mecanismos de sincronización y comunicación entre los procesos.

**Mutua exclusión:** el objetivo de la exclusión mutua es que dos procesos no accedan al mismo recurso o sección crítica al mismo tiempo. Al ser los procesos concurrentes, debemos asegurarnos de que haya cierto control al momento en el que estos utilizan recursos compartidos. Por ejemplo, si tenemos procesos que editan la misma estructura de datos al mismo tiempo, sin ningún tipo de sincronización, el resultado de salida del programa puede verse afectado.   
  
La mutua exclusión entra en juego cuando un proceso o tarea comparte recursos con otro. Es decir, cuando los procesos no son independientes.

**Condiciones de concurrencia (Bernstein)**

Se utiliza la siguiente notación:

**Conjunto de Lectura R (Si) = (a1, a2, …, an):** El conjunto de lectura de la sentencia Si es aquel formado por todas las variables que son referenciadaspor la sentencia Si durante su ejecución sin sufrir cambios.

**Conjunto de escritura W (Si) = (b1, b2,... , bn):** El conjunto de escritura de la sentencia Si es aquel formado por todas las variables cuyos valores son modificadosdurante la ejecución de Si.

Dos sentencias cualesquiera Si y Sj pueden ejecutarse concurrentemente produciendo el

mismo resultado que si se ejecutaran secuencialmente sí y sólo sí se cumplen las siguientes condiciones:

1. **R (Si)** ∩ **W (Sj) = (** ø **)**
2. **W(Si)** ∩ **R (Sj) = (** ø **)**
3. **W(Si)** ∩ **W (Sj) = (** ø **)**

Es más fácil entender las condiciones de concurrencia si lo visualizamos como variables dentro de un programa. Lo que dicen estas tres condiciones es lo siguiente: un proceso no tiene que estar leyendo unas variables en el mismo momento que otro proceso las está escribiendo, y tampoco los dos procesos tienen que estar escribiendo en las mismas variables al mismo tiempo. La idea de las condiciones de concurrencia es evitar que los datos sean modificados erróneamente, produciendo resultados impredecibles.

**Procesos independientes e interactuantes**

Un proceso es **independiente** si no puede afectar o ser afectado por otros procesos corriendo en el sistema:

1. Su estado no es compartido con ningún otro proceso.
2. La ejecución es determinística: el resultado de la ejecución depende sólo de las entradas.
3. Su ejecución puede detenerse y reasumirse sin por eso causar efectos laterales al resto del sistema.

Un proceso es **interactuante** si puede afectar o ser afectado por otros procesos:

1. Su estado es compartido con otros procesos.
2. El resultado de su ejecución no puede ser predicho ya que depende de la ejecución de otros procesos.

**Conflictos relacionados a la concurrencia**

Dentro de la ejecución de un programa se presentan otros conflictos con el uso de recursos que el S.O. deberá resolver, a saber:

* **Inanición, o Postergación o Aplazamiento Indefinido (Starvation)**: Consiste en el hecho de que uno o varios procesos nunca reciban el suficiente tiempo de ejecución para terminar su tarea. Por ejemplo, que un proceso ocupe un recurso y lo marque como 'ocupado' y que termine sin marcarlo como 'desocupado'. Si algún otro proceso pide ese recurso, lo verá 'ocupado' y esperará indefinidamente a que se 'desocupe'.
* **Condición de Espera Circular**: Esto ocurre cuando dos o más procesos forman una cadena de espera que los involucra a todos. Por ejemplo, suponer que el proceso A tiene asignado el recurso 'cinta' y el proceso B tiene asignado el recurso 'disco'. En ese momento al proceso A se le ocurre pedir el recurso 'disco' y al proceso B el recurso 'cinta'. Ahí se forma una espera circular entre esos dos procesos que se puede evitar quitándole a la fuerza un recurso a cualquiera de los dos procesos.
* **Condición de No expropiación:** Esta condición no resulta precisamente de la concurrencia, pero juega un papel importante en este ambiente. Esta condición especifica que, si un proceso tiene asignado un recurso, dicho recurso no puede arrebatársele por ningún motivo, y estará disponible hasta que el proceso lo 'libere' por su voluntad.
* **Condición de Espera Ocupada**: Esta condición consiste en que un proceso pide un recurso que ya está asignado a otro proceso y la condición de no expropiación se debe cumplir. Entonces el proceso estará gastando el resto de su porción de tiempo (time slice) verificando si el recurso fue liberado. Es decir, desperdicia su tiempo ejecución en esperar. La solución más común a este problema consiste en que el sistema operativo se dé cuenta de esta situación y mande a una cola de espera al proceso, otorgándole inmediatamente el turno de ejecución a otro proceso.
* **Condición de Mutua Exclusión**: Cuando un proceso usa un recurso del sistema realiza una serie de operaciones sobre el recurso y después lo deja de usar. A la sección de código que usa ese recurso se le llama **'región crítica'**. La condición de mutua exclusión establece que solamente se permite a un proceso estar dentro de la misma región crítica. Esto es, que en cualquier momento solamente un proceso puede usar un recurso a la vez. Para lograr la mutua exclusión se ideo también el concepto de 'región crítica'. Para lograr la mutua exclusión generalmente se usan algunas técnicas para entrar a la región crítica como ser: semáforos, monitores, el algoritmo de Dekker y Peterson, los 'candados'.
* **Condición de Ocupar y Esperar (hold and wait):** Consiste en que un proceso pide un recurso y se le asigna. Antes de soltarlo, pide otro recurso que otro proceso ya tiene asignado.

**Beneficios de la concurrencia**

- Trata de evitar los tiempos muertos del procesador

- Comparte y optimiza el uso de recursos

- Permite la modularidad en las diferentes etapas del proceso

- Acelera los cálculos

- Da mayor comodidad

**Desventajas de la concurrencia**

- Inanición e interrupción de procesos

- Ocurrencia de interbloqueos (deadlocks)

- Que dos o más procesos compitan por el mismo recurso (No Apropiativo) complica su tratamiento.

**Región crítica**

Región (o sección) críticade un proceso es la fase o etapa en la vida de ese proceso concurrente en la cual accede a un recurso crítico (o compartido), que no debe ser accedido por más de un proceso,para modificarlo o alterarlo. Es un trozo de código en el que se utiliza un recurso compartido y que se ejecuta de forma exclusiva.

**Propiedades para usar la Región Crítica**

El problema de la región crítica consiste en sincronizar los procesos de forma tal que se cumpla el siguiente ***protocolo de sincronización***.

1. **Mutua exclusión:** sólo un proceso a la vez puede estar ejecutando en su región crítica (lo accede y la usa).
2. **Progreso:** un proceso fuera de su Región Critica no debe impedir la entrada de otro proceso a la misma. Sólo los procesos que quieren entrar a la Región Critica deben participar en la decisión. (La solución de estricta alternancia viola este requisito).
3. **Espera limitada:** un proceso debe poder entrar a la Región Crítica después de un número limitado de intentos.
4. **Abandono** (Tiempo limitado): debe dejar a la Región Critica en un tiempo finito. (Es “casi” una consecuencia de la 3).
5. **Penalidad:** un proceso no puede consumir tiempo de ejecución mientras espera por un recurso.
6. **Privilegio:** no debe haber ningún proceso privilegiado que monopolice la Región Crítica.

**Semáforos**

Un semáforo es una herramienta genérica de sincronización de procesos, o sea, permite el

ordenamiento de las operaciones que realizan los procesos en el tiempo. Es una especie de bandera (señal o ***flag***) que indica la posibilidad de acceder o no a un recurso.

* Es una función o un arreglo dentro del Kernel.
* Un semáforo es una *variable protegida* cuyo valor puede ser accedido y alterado tan sólo por las dos **primitivas independientes y atómicas\*** y una operación de inicialización del semáforo.
* Un semáforo S es una variable entera ***e(s)*** llamada ***valor del semáforo*** con las siguientes propiedades:

**1.** La variable puede tomar cualquier valor entero (positivo, negativo o nulo) en **e(s)** (valor del semáforo).

**2.** Se creamediante un system call o una declaración, donde se especifica el valor inicial del semáforo que, por definición, debe ser un entero no negativo.

**3.** Sólo es accesible mediante dos operadores primitivos atómicos **P(s)** (Proberen = comprobar en holandés, decrementa el semáforo) y **V(s)** (Verhogen = soltar, incrementa el semáforo) respectivamente.

*\*Que sea atómico implica la indivisibilidad e irreductibilidad del proceso, ya que éste debe realizarse en su totalidad o en caso de ser interrumpido poder deshacer sus acciones de modo que fuese como si no se hubiese realizado acción alguna.*

**Monitor**

Los monitores son un mecanismo de sincronización implementada por una primitiva de alto nivel (Mediante compilador) dentro de la cual se dispone de un conjunto de procedimientos, variables y estructuras de datos agrupados en un módulo.

Los recursos globales compartidos pueden ser declarados como pertenecientes a un monitor y

ningún proceso tendrá permiso para accederlo. En cambio, sí pueden accederlo a través de un (y solo un) procedimiento público proporcionado por el monitor y suministrar datos producidos como argumento.

El monitor manipula los datos, o sea los encapsula para que los procesos del usuario no pueda accederlos en forma directa.

Un ejemplo de un monitor, es la palabra clave synchronized en el lenguaje Java, que automáticamente hace que un método solo pueda ser utilizado por un proceso o hilo a la vez.

**Interprocess Comunication (IPC)**

La comunicación entre procesos es una de las funciones básicas de un sistema operativo. La IPC provee un mecanismo que permite a los procesos comunicarse y sincronizarse entre sí. El propósito es permitir que dos procesos se sincronicen o se envíen datos mediante dicho mecanismo.

Hay dos maneras de realizar comunicaciones entre dos procesos:

**1. Comunicación a través de un área común de memoria:** requiere contar con mecanismos de sincronización (semáforos, monitores, etc.) para garantizar la consistencia de los datos almacenados.

**2. Comunicación mediante el intercambio de mensajes (paso de mensajes):** aquí las variables utilizadas son locales, no hay necesidad de usar semáforos u otras alternativas.

Definimos como **mensaje** a una porción discreta de datos (generalmente compuesto por un conjunto de bits).

La comunicación se hace siguiendo un conjunto de reglas, normalmente conocidos como protocolos (sistema de reglas que permiten que dos o más entidades de un sistema de comunicación se comuniquen entre ellas para transmitir información).

Se dice que es una comunicación ya que un proceso al recibir un mensaje, obtiene los datos enviados por otro proceso. También se produce sincronización porque un mensaje solo puede ser recibido después de ser enviado. Generalmente después de ser recibido el mensaje éste destruye. Si se desea conservarlo se deben tomar los recados adecuados.

Generalmente el paso de mensajes se realiza a través de las primitivas **Send()** y **Receive()**.

Las mismas pueden tener diferentes variaciones que afecten al comportamiento que tiene el proceso luego de enviar un mensaje o mientras está en la espera de uno. Por ejemplo, una primitiva Send() puede ser bloqueante, lo que significa que el proceso quedará bloqueado hasta que el receptor reciba correctamente el mensaje. Pero a su vez, existen primitivas send no bloqueantes, que permiten que el mensaje sea enviado y el proceso emisor continúe con su hilo de ejecución.

**Direccionamiento**

Es necesario indicar en la primitiva **Send()** a qué proceso vamos a enviar el mensaje.

**Direccionamiento directo:** es aquel tipo de envío de mensajes en el cual el emisor envía directamente el mensaje al proceso receptor sin pasar por ningún otro intermediario antes. En algunos sistemas, es posible que el proceso receptor sepa de antemano quién le enviará un mensaje, en otros, es imposible saberlo. La desventaja de este tipo de sistema es que, si el proceso receptor no está preparado para recibir un mensaje en un momento dado, el mensaje se dará como perdido y no será atendido.

**Direccionamiento indirecto:** en este direccionamiento, los mensajes son enviados a un mailbox (buzón), el cual no es más que una estructura de datos formada por colas en donde los mensajes son guardados temporalmente. A diferencia del direccionamiento directo, aquí se cuenta con la ventaja de no perder aquellos mensajes que no pudieron ser recibidos y atendidos al instante, ya que los mismos permanecerán almacenados y podrán ser retirados uno a uno. Otra de las ventajas es que este sistema puede permitir una comunicación al estilo de uno a muchos, donde un proceso envía un mensaje y muchos pueden recibirlo al mismo tiempo.

**Tipos de sincronizaciones mediante mensajes**

* **Comunicación Sincrónica:** también conocida como **rendezvous** (cita en francés). Los procesos se reúnen para una comunicación sincrónica. El proceso emisor es bloqueado hasta que el proceso receptor esté listo para recibir un mensaje. Si el proceso receptor ejecuta el Receive() y no hay ningún mensaje, también queda bloqueado hasta que el mensaje le llegue. En resumen, ambos procesos bloquean su ejecución e implementan primitivas Send() y Receive() bloqueantes.

**Rendezvous extendido:** luego de enviar un mensaje, el proceso emisor queda bloqueado hasta que el proceso receptor le envíe un mensaje de respuesta (una confirmación, un OK, un ACK). Generalmente se utiliza cuando es necesario confirmar la ejecución de cierta acción luego de haber enviado un mensaje.

**Rendezvous asímetrico:** es similar al extendido, solo que ambos procesos quedan bloqueados hasta que se termine de ejecutar un bloque de código. El receptor no envía un mensaje de confirmación al emisor.

* **Comunicación Asincrónica:** ninguno de los procesos se bloquea ni al enviar ni al recibir mensajes. Ambos siguen ejecutándose. El receptor seguirá ejecutándose incluso aunque no haya recibido un mensaje. Depende de la implementación ver si los mensajes serán atendidos o no.
* **Comunicación semi-sincrónica:** se utiliza un Send() no bloqueante y un Receive() bloqueante.
* **Otros modos:** existen otros mecanismos, como la ejecución de procedimientos remotos. La llamada a procedimiento remoto (*Remote Procedure Call*, RPC) es un programa que utiliza una computadora para ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambas. De esta manera el programador no tenía que estar pendiente de las comunicaciones, estando estas encapsuladas dentro de las RPC.   
    
  Las RPC son muy utilizadas dentro de la comunicación cliente-servidor. Siendo el cliente el que inicia el proceso solicitando al servidor que ejecute cierto procedimiento o función y enviando este de vuelta el resultado de dicha operación al cliente.

**Tabla de comparación entre paso de mensajes y memoria compartida**

|  |  |
| --- | --- |
| **Paso de mensajes** | **Memoria compartida** |
| -Medio de comunicación entre procesos que ofrece el S.O.  -Se debe abrir una conexión y conocer el nombre del comunicador (proceso en la misma CPU o no)  -En red cada computador tiene el **nombre del anfitrión** y cada proceso tiene **nombre de proceso.**  -La fuente de la comunicación conocida como **cliente** y el demonio (daemons) que recibe **servidor**, intercambian mensajes mediante llamadas al sistema.  -Para intercambiar pequeñas cantidades de datos.  -Evita conflictos.  -Más fácil de usar para comunicación entre  computadoras. | -Utiliza llamadas para **correspondencia de memoria** con el propósito de obtener acceso a los registros de memoria que pertenecen a otros procesos.  -S.O. evita que un proceso tenga acceso a la memoria de otro.  Para compartir la memoria se requiere que dos o más procesos estén de acuerdo en eliminar esta restricción y así intercambiar  información leyendo y escribiendo datos en estas áreas compartidas.  -Forma de datos y ubicación están determinadas por esos procesos y no por el S.O., los cuales también se encargan que  no se escriba en posiciones iguales de memoria a la vez.  -Máxima velocidad y conveniencia en la comunicación (puede efectuarse a la velocidad de la memoria).  -Problemas en el área de protección y sincronización. |

**Addendum: Algunos medios de comunicación brevemente explicados**

**Pipes**

Una tubería (pipe) consiste en una cadena de procesos conectados de forma tal que la salida de cada elemento de la cadena es la entrada del próximo. Permiten la comunicación y sincronización entre procesos. Es común el uso de búfer de datos entre elementos consecutivos. La comunicación por medio de tuberías se basa en la interacción productor/consumidor, los procesos productores (aquellos que envían datos) se comunican con los procesos consumidores (que reciben datos) siguiendo un orden FIFO. Una vez que el proceso consumidor recibe un dato, éste se elimina de la tubería.

**Señales**

Una señal (signal) es una forma limitada de comunicación entre procesos empleada en Unix y otros sistemas operativos compatibles con POSIX. En esencia es una notificación asíncrona enviada a un proceso para informarle de un evento. Cuando se le manda una señal a un proceso, el sistema operativo modifica su ejecución normal. Si se había establecido anteriormente un procedimiento (handler) para tratar esa señal se ejecuta éste, si no se estableció nada previamente se ejecuta la acción por defecto para esa señal.

Al ser una notificación asíncrona, las señales son vulnerables a las condiciones de carrera. Una señal puede llegar al proceso mientras este todavía se encuentra ejecutando el procedimiento (handler) de la señal anterior.

**Sockets**

La palabra Socket designa un concepto abstracto por el cual dos procesos (posiblemente situados en computadoras distintas) pueden intercambiar cualquier flujo de datos, generalmente de manera fiable y ordenada.

Para que dos procesos puedan comunicarse entre sí es necesario que se cumplan ciertos requisitos:

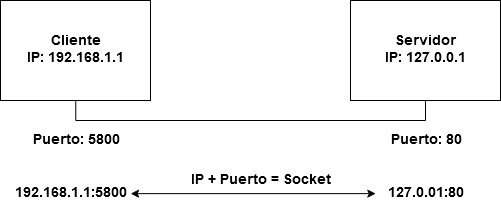
* Que un proceso sea capaz de localizar al otro.
* Que ambos procesos sean capaces de intercambiarse datos relevantes a su finalidad.

Para ello son necesarios los dos recursos que originan el concepto de *socket*:

* Un par de direcciones del protocolo de red (dirección IP, si se utiliza el protocolo TCP/IP), que identifican la computadora de origen y la remota.
* Un par de números de puerto, que identifican a un programa dentro de cada computadora.

Los sockets permiten implementar una arquitectura cliente-servidor. La comunicación debe ser iniciada por uno de los programas que se denomina programa "cliente". El segundo programa espera a que otro inicie la comunicación, por este motivo se denomina programa "servidor".

Un socket es un proceso o hilo existente en la máquina cliente y en la máquina servidora, que sirve en última instancia para que el programa servidor y el cliente lean y escriban la información. Esta información será la transmitida por las diferentes capas de red.



**Deadlock**

El deadlock (bloqueo mutuo) es el bloqueo permanente de un conjunto de procesos o hilos de ejecución en un sistema concurrente que compiten por recursos del sistema o bien se comunican entre ellos. A diferencia de otros problemas de concurrencia de procesos, no existe una solución general para los interbloqueos.

No hay que confundir deadlock con la inanición de un proceso. La inanición se produce cuando un proceso de baja prioridad no consigue los recursos que necesita, ya que le son asignados a otros procesos más importantes, quedando dicho proceso en espera hasta poder conseguir el recurso necesario para seguir ejecutándose. **Pero los demás procesos siguen ejecutándose.**

El deadlock es un estado general donde hay un **ciclo de procesos** que están a la espera de un recurso al cual no pueden acceder porque está asignado a otro proceso, generándose así un bloqueo de todos los procesos involucrado. Ninguno avanza en su ejecución. El deadlock es un caso extremo de la inanición.

**Condiciones (necesarias) de Coffman**

Son un conjunto de condiciones necesarias para que el bloqueo mutuo pueda existir.

Estas condiciones deben cumplirse simultáneamente y no son totalmente independientes entre ellas.

* **Mutua exclusión:** los recursos no deben ser compartibles, es decir que sólo un proceso a la vez puede usar el recurso.
* **Retener y esperar** (Hold & Wait)**:** significa que el proceso retiene los recursos que ya tiene asignados mientras espera por nuevos a adquirir del conjunto de recursos del sistema.
* **No expropiación** (No Preemption)**:** el proceso está reteniendo los recursos concedido y solo puede liberarlos y devolverlos al sistema como resultado de una acción voluntaria de ese proceso. El S.O. no puede obligarle a devolverlos esto se conoce como no expropiación.
* **Espera circular:** debe existir un conjunto de procesos {P0, P1, ..., Pn} tal que, P0 está esperando un recurso que posee P1, P1 está esperando por un recurso que posee P2, ..., y Pn está esperando por un recurso que posee P0, o sea, los procesos están esperando mutuamente a que el otro libere el recurso requerido formando una cadena circular entre dos o más procesos, en la que cada uno de ellos está esperando un recurso que tiene el próximo miembro de la cadena.

En rigor, espera circular implica retención y espera, pero es útil separar las condiciones, pues basta con que una no se dé para que no haya deadlock.

Si bien la mayoría de los autores toman a estas cuatro condiciones como necesarias y suficientes para que ocurra una situación de deadlock, en cambio W. Stallings destaca que las primeras tres condiciones mencionadas, son necesarias pero no suficientes, y la cuarta condición ocurre como consecuencia de las tres primeras. Entonces la diferencia que marca Stallings radica en que la cuarta condición es una consecuencia de las tres primeras. Esto es, si se cumplen las tres primeras condiciones, una secuencia de eventos llevará a que se incurra en una espera circular.

**Estrategias para tratar bloqueos mutuos**

1. **Ignorarlo**

Ignorar que puedan ocurrir deadlocks en el sistema es la solución menos costosa y más tentadora de todas. Sin embargo, solo hay que implementar esta opción cuando estemos seguro de que el riesgo de que ocurran bloqueos mutuos sea muy bajo. Podemos ignorar la aparición de deadlocks, pero en el caso de que aparezcan, el sistema tendrá que ser reiniciado, y eso puede llegar a implicar un gran costo.

Esta solución no puede ser aplicada en para sistemas como aeropuertos, centrales nucleares, o base de datos. Si en alguno de esos sistemas ocurriese un bloqueo mutuo, las consecuencias de tener que reiniciar todo el sistema serían bastante graves y costosas. La idea es aplicar esta solución en sistemas relativamente poco importantes, donde el costo de reiniciar el sistema sea bajo.

1. **Prevenirlo**

Existen diferentes formas de prevenir la aparición de deadlocks.

La primera de ellas es mediante el uso de un grafo de recursos, el cual lleve cuenta de los diferentes recursos que los procesos solicitan o liberan. Si se forma un ciclo en el grafo, se matan a todos los procesos del ciclo, o bien se van matando de a uno hasta que no haya más ciclos. El problema que esto acarrea es el siguiente: ¿qué procesos matar? ¿qué víctima elegir primero? ¿Y si el proceso que matamos está en medio de algo importante?

Otra de las soluciones es evitar las condiciones de Coffman.

**Mutua exclusión:** utilizar la virtualización de recursos para que varios procesos puedan acceder al mismo recurso sin tener que esperar a que otro proceso lo libere. El ejemplo clásico es el uso de una cola de impresión para la impresora, en la cual los procesos pondrían sus pedidos y seguirían su ejecución. En cuanto la impresora esté lista, imprimirá el pedido. Esta solución lo provee el SPOOLER simulando la impresora en el disco.

**Control y espera (hold and wait):** se busca que, cuando un proceso tenga un recurso asignado, no pueda solicitar otro.   
  
Se puede lograr de dos formas. La primera es dándole todos los recursos que necesite el proceso durante toda su vida antes de que empiece su ejecución. A esto se lo conoce como asignación estática. Sin embargo, esta solución es muy ineficiente y costosa, ya que si no están presentes todos los recursos que requiera el proceso, el mismo nunca iniciará. O también, puede que los recursos queden sin ser utilizados por mucho tiempo, tiempo en el cual otros procesos podrían haberlos utilizado.

La segunda forma es obligando al proceso a liberar los recursos que actualmente posee antes de solicitar un nuevo recurso. Solo puede solicitar un recurso cuando no tenga ninguno asignado. El problema que tiene esta solución es que la mayoría de los procesos necesitan dos recursos al mismo tiempo para su ejecución. ¿Qué pasaría si queremos copiar un archivo desde un disco duro a un USB? Primero pediríamos el disco, después lo liberaríamos para pedir al USB… y nos quedamos sin copiar el archivo porque no tenemos acceso al disco.

**No expropiación:** para evadir esta condición, se podría implementar la expropiación de recursos. Es decir, el sistema operativo podría quitarle recursos a un proceso en ejecución a medida que otro proceso los solicite. La desventaja es que puede llevar a inanición.

**Espera circular:** se puede evitar haciendo que los procesos soliciten recursos de forma lineal, siguiendo un orden. La idea sería que, si un proceso solicita el recurso número 10, no va a poder solicitar los recursos del 1 al 9, solo aquellos más grandes que el 10. El problema de esta solución es el hecho de tener que ir ordenando los recursos, cuando puede haber una cantidad grande de los mismos.

1. **Detectar y recuperar**

Consiste en abortar un proceso cuando se detecta o se presupone que puede ocurrir un deadlock. La ventaja de esta implementación es que ni el acceso a recursos, ni el accionar de los procesos, es limitado como en los otros tratamientos.

Sin embargo, se debe implementar un algoritmo que esté al tanto de los procesos y que analice si se va a producir un bloqueo mutuo.

El problema está en cuándo ejecutar este algoritmo. Si los ejecutamos cada vez que se solicita un recurso, podríamos detectar inmediatamente la situación de deadlock y así tratar de ver qué proceso es el que determinó tal situación. Pero hacer esto es muy costoso, es puro overhead. Y si limitamos la frecuencia de ejecución del algoritmo (para no producir tanto overhead), puede que perdamos de vista cuál es el proceso que produjo el deadlock…

Hay varias formas de recuperar una situación de deadlock. La primera es abortando todos los procesos involucrados. El problema se resuelve, pero el costo puede ser grande, y podríamos terminar matando procesos que estaban a punto de finalizar su tarea.

También se podría implementar un sistema de checkpoints, y en caso de haber un deadlock, volver el proceso a un punto anterior. Pero esto no nos asegura que el deadlock no vuelva a ocurrir.

Por último, podemos elegir uno de los procesos involucrados en el deadlock y matarlo. El problema reside en elegir bien a dicho proceso víctima. Se pueden tomar ciertos criterios como elegir al que menor cantidad de tiempo haya tenido en el CPU, menor cantidad de recursos asignados, menor prioridad…

Para todas estas cosas, es necesario que el sistema operativo monitoree de alguna forma los recursos, usando un grafo de recursos.

1. **Evitarlo dinámicamente**

Antes de explicar esta estrategia es necesario saber:  
  
El **Estado del Sistema**: es la asignación actual de recursos a procesos, definido por el número de recursos disponibles, número de recursos asignados y el máximo pedido de cada proceso. Este estado es registrado mediante una función de contabilidad en el S.O.

Un **estado es seguro** es cuando existe un orden tal en el que los procesos pueden llevarse a cabo por completo, sin resultar en Deadlock. Se dice que un estado es seguro si existe una secuencia de otros estados que lleva a que todos los procesos obtengan sus recursos y terminen su ejecución en un cierto tiempo. Un **estado inseguro** es aquél en que su ejecución conduce a un abrazo mortal.

Para evitar una situación de Abrazo Mortal se necesita contar con mucha información por adelantado que debe ser provista por el S.O.

Esta estrategia, se basa en asegurar que los procesos y los recursos permanezcan en un estado seguro: cuando un proceso solicita un recurso, se asume que le fue otorgado, entonces se chequea el sistema y se determina en que estado se encuentra el mismo.

De ser un estado seguro, el recurso es efectivamente entregado al proceso, ya que de lo contrario el proceso queda bloqueado hasta que sea seguro entregárselo.

Para ello se cuenta con dos protocolos:

1. No comenzar un proceso si las demandas pueden incurrir en Deadlock

2. No asignarle a un proceso en ejecución otro recurso si eso lo puede conducir a un Deadlock.

Las ventajas de esta estrategia con respecto a la de DETECCIÓN son las de no necesitar la aplicación del rollback ni la sustracción de un recurso a un proceso y la de menor existencia de restricciones.

Las desventajas son que no siempre un proceso conoce de antemano los recursos que va a necesitar durante su ejecución y debe existir un número fijo de recursos a asignar y un número fijo de procesos.

Para el desarrollo de esta estrategia varios autores definen matrices para guardar la información. Por ejemplo, Crocus presenta tres matrices. La primera describe el estado inicial del sistema, proporcionando la cantidad total de recursos que existen en cada clase, este vector permanece constante. La segunda matriz define los recursos asignados a los procesos y en la última los recursos solicitados por los procesos. Así cada vez que se solicite un recurso se modificaran los datos de las matrices y por ecuaciones se determinará si el estado del sistema es seguro o no.