Resumen Sistemas Operativos – Primer Teórico

Threads

Procesamiento secuencial, concurrente y paralelo

Procesamiento concurrente: implica correr tareas simultáneamente, pero no necesariamente en paralelo. Puede ser implementado con context switch o paralelismo.

Procesamiento paralelo: mantiene tareas independientes que se asignan a diferentes CPU simultáneamente. Requiere arquitecturas multi-core. Este implica concurrencia.

Procesamiento secuencial: implica correr las tareas una tras otra.

Tareas simultaneas

Estas en el SO son los procesos o threads.

- Los procesos son más pesados.
    - La multiprogramación se logra con múltiples procesos.
    - Aíslan stacks y no comparten nada, se copia el PCB.
    - Los cambios de contextos son más costosos y la finalización es más lenta.
    - Los mensajes deben serializarse. Restricción volúmenes de datos a compartir.
- Los threads se caracterizan por ser más livianos.
    - La multiprogramación se logra por un proceso que creara threads.
    - Comparten memoria, por lo tanto, más rápido, pero menos seguros (no están aislados.
    - Cambios de contexto y finalización más rápida.

Las soluciones concurrentes son más complejas que las secuenciales. Ejecutar más tareas simultaneas que núcleos resultara en la saturación del CPU que se traduce en tiempos de respuesta altos.

Los context switch tiene un overhead que podría ralentizar la solución secuencial. La misma solución en paralelo debería arrojar mejores resultados.

Tipos de tareas

Es importante distinguir dos tipos de tareas:

- Ligadas a la CPU: estas tareas consumen muchos ciclos de CPU por lo que sacarán mejor provecho si se las paraleliza.
- Ligadas a la IO: tareas que esperan por operaciones de IO, reducen la latencia. El context switch alcanza.

GIL

Java utiliza todos los threads en paralelo: cada thread corre en cada CPU. Por otro lado Ruby y Python no, sólo usan un procesador, no aprovechando el paralelismo y generando overhead. El problema de estos es por el Global Interpreter Lock

GIL es un mecanismo utilizado por algunos intérpretes de lenguajes para sincronizar la ejecución de threads de forma tal que solo un thread nativo (por proceso) pueda ejecutar una operación critica a la vez usando mutex. Si se ejecutan dos instancias del intérprete, serán dos procesos y cada uno tendrá su propio GIL.

Este existe para para simplificar la concurrencia y la integración de librerías C.

Existen versiones de ambos lenguajes basadas en la JVM o Dotnet, pero a veces no están tan actualizadas como las versiones de referencia que están desarrolladas en C.

Hilos y Sistemas Operativos

Los lenguajes de programación brindan herramientas que nos permiten separar las diferentes tareas de los programas en unidades de ejecución diferentes.

Antiguos SO

La unidad básica de utilización de CPU eran los Procesos:

- Programa en Ejecución
- Unidad de asignación de los recursos
- Conceptos relacionados con proceso:
    - Espacio de direcciones
    - Punteros a los recursos asignados (stacks, archivos, etc.)
    - Estructuras asociadas: PCB, tablas
- En los primeros sistemas operativos, cada proceso tenía un espacio de direcciones y un solo hilo de control.
    - Un único flujo secuencial de ejecución.
    - Se ejecuta una instrucción y cuando finaliza se ejecuta la siguiente
    - Para ejecutar otro proceso, se debe llevar adelante un cambio de contexto

Actuales SO

Hay situaciones en las que es conveniente tener varios hilos de control en un mismo espacio de direcciones. Ejecutarlos en cuasi-paralelo como si fueran procesos (casi) separados (hay áreas comunes). Estos hilos comparten el espacio de direcciones. Se gana tiempo de CPU (no se pide permiso a SO).

Unidad básica de utilización de CPU: Hilos

Unidad básica de asignación de RECURSOS: Proceso

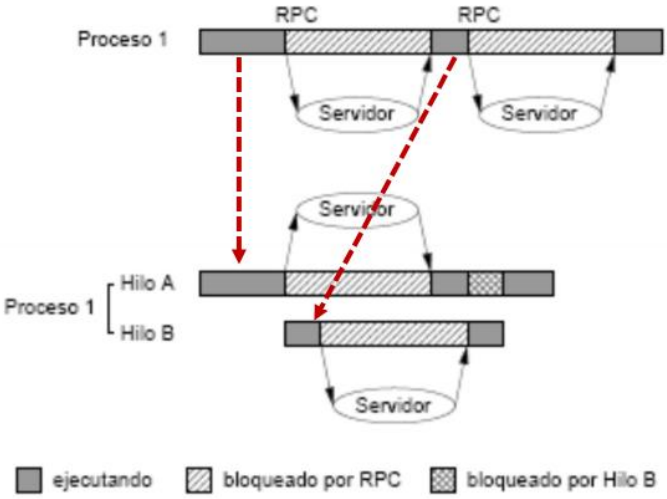
- Proceso:
    - Espacio de direcciones.
    - Unidad de propiedad de recursos.
    - Conjunto de threads (eventualmente uno).
- Hilos:
    - Unidad de trabajo (Hilo de ejecución).
    - Contexto del procesador.
    - Stacks de Usuario y Kernel.
    - Variables Propias.
    - Acceso a la memoria y recursos del PROCESO.

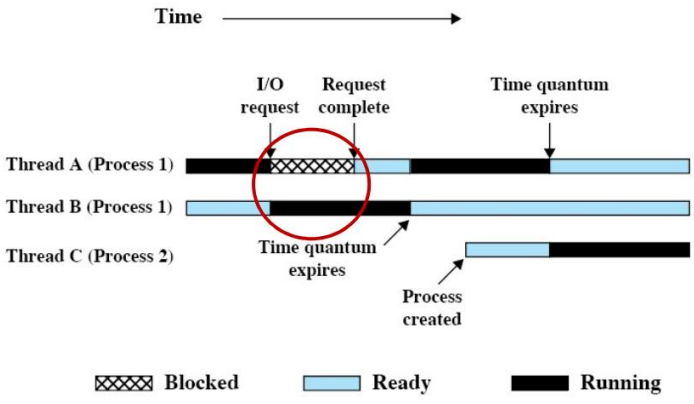
¿Por qué dividir una aplicación en Threads?

- Respuestas percibidas por los usuarios, paralelismo/ejecución en background.
- Aprovechar las ventajas de múltiples procesadores.
- Hay que tener en cuenta características complejas como sincronización y escalabilidad. Si se tiene como cantidad de threads por procesos la cantidad de CPU, hay excesivos cambios de contexto de hilos del mismo proceso.

Ventajas uso de hilos:

- Sincronización de procesos
- Mejor tiempo de respuesta
- Compartir los recursos
- Economía





Multithreading Simultaneo – Hyper Threading.

- Propietario de las CPU Intel
- Permite que el software programado para ejecutar múltiples hilos (multi-threaded) procese los hilos en paralelo dentro de un único procesador.

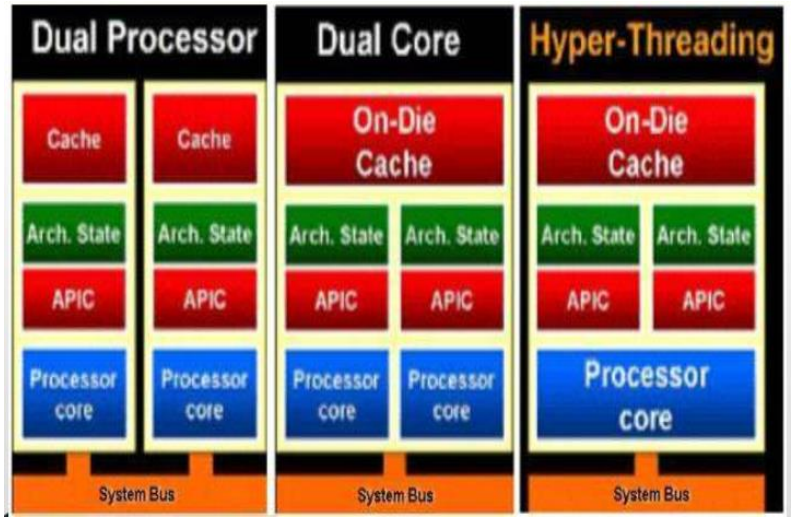
- La tecnología simula dos procesadores lógicos dentro de un único procesador físico
    - Duplica solo algunas “secciones” de un procesador
    - Registros de Control (MMU, Interrupciones, Estado, etc)
    - Registros de Propósito General (AX, BX, PC, Stack, etc.)
- Básicamente se tiene un solo core más grande. Se tienen varios registros. El cambio de contexto es más liviano.

Dual-Core

- Una CPU con dos cores por procesador físico.
- Un circuito integrado tiene 2 procesadores completos.
- Los 2 procesadores combinan cache y controlador.
- Bueno para implementar modelo productor consumidor ya que no implica acceso al bus.

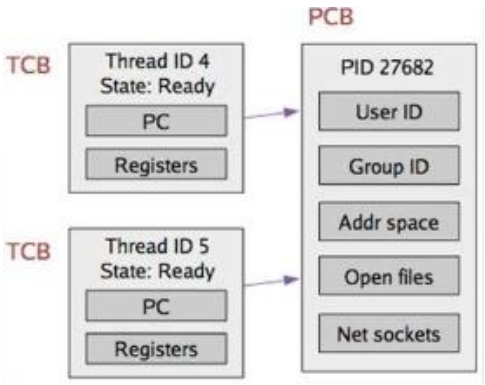
Dual-Processor

- Tiene 2 procesadores físicos en el mismo chasis.
- Pueden estar en la misma motherboard o no.
- Cache y controlador independientes.
- Si se tiene modelo productor consumidor es más costoso de implementar

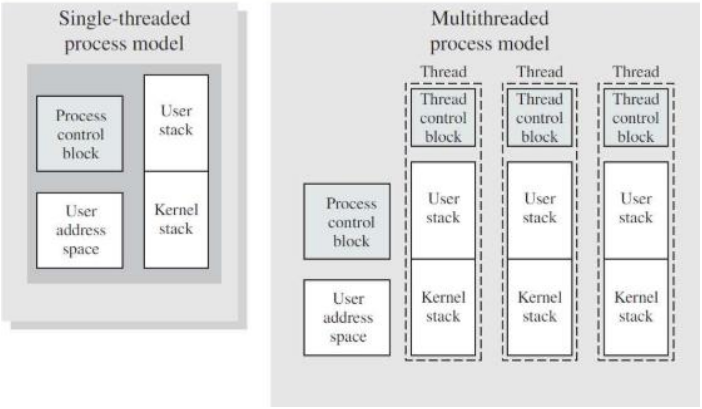


Estructura de un Hilo

- Un estado de ejecución
- Un contexto de procesador
- Stacks (modo usuario, modo kernel)
- Acceso a memoria y recursos del proceso
    - Archivos abiertos
    - Señales
    - Código
- TCB – Thread Control Block.



Hay una PCB por proceso, esta es manejada por el SO en el Kernel Space. Hay una TCB por hilo y la maneja por la librería del hilo en User Space.



Ventajas uso de hilos

- Context Switch:
    - El cambio de contexto solo se realiza a nivel de registros y no espacio de direcciones. Lo lleva a cabo el proceso sin necesidad de intervención del SO. En los procesos el SO interviene para salvar el ambiente del proceso saliente y recuperar el ambiente del nuevo.
    - Si hay 1 sola CPU:
        - Puede dar lugar a tiempos improductivos pero si hay varios hilos (2 o +) en 1 solo proceso, no lo hay.
- Creación:

    - La creación de hilos es mucho menos cargada y se encarga el proceso, no el SO. En los procesos implica que el SO cree de un nuevo espacio de direcciones, PCB, PC, etc.
- Destrucción:
    - La destrucción se realiza dentro del proceso sin intervención del SO. En los procesos el SO debe salvar el ambiente del proceso saliente y eliminar su PCB
- Planificación:
    - La planificación es responsabilidad del desarrollador y es menos costosa, aunque puede traer problemas. En los procesos el SO el cambio implica cambios de contexto continuos.
- Protección:
    - La protección debe darse desde el lado del desarrollo. Todos los hilos comparten el mismo espacio de direcciones. Un hilo podría bloquear la ejecución de otros (sencillo comunicar hilos pero más inseguro). En los procesos el SO tiene distintos mecanismos de seguridad que implican técnicas avanzadas.

Estados de un Thread

Estados mínimos: Ejecución, Listo y Bloqueado

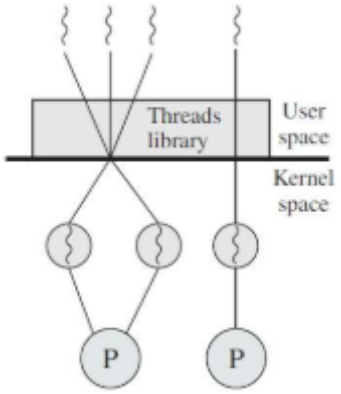
La planificación se hace sobre los threads y los eventos sobre procesos afectaran todos sus Threads (si duermo un proceso duermo todos sus threads).

Tipos de Threads

- User Level Thread

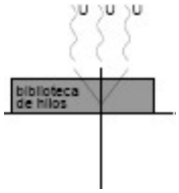
    - Se ejecutan a nivel de usuario por la aplicación quien se encarga de la gestión (a través de una biblioteca de hilos que debe permitir crea, destruir , planificar, etc.).
        - El kernel no se entera de la existencia de Threads.
            - Ej: Java VM, POSIX Threads, Solaris Threads.
    - Nos da ventajas como:
        - Intercambio entre hilos ya que comparten el espacio de direcciones, hay una planificación independiente, se podrían reemplazar llamadas al sistema bloqueantes por otras que no bloquean, hay mayor portabilidad en plataformas, no requiere cambios modo para su existencia y no es necesario que el SO soporte hilos.
    - Nos da desventajas como:
        - No se puede ejecutar hilos del mismo proceso en distintos procesadores, si un hilo produce page fault, todo el proceso se bloquea, un hilo podría monopolizar el uso de la CPU y el bloqueo del proceso durante una System Call bloqueante.
- Kernel Level Thread
    - La gestión completa se realiza en modo Kernel
    - Tiene ventajas como:
        - Se puede multiplexar hilos del mismo proceso en diferentes procesadores e independencia de bloqueos entre threads de un mismo proceso.
    - Da desventajas como:
        - Cambios de modo de ejecución para la gestión.
            - Planificación, creación, destrucción, etc.
- Combinaciones (SO actuales)
    - En este tipo de sistemas la creación de hilos se realiza a nivel de usuario y los mismos son mapeados a una cantidad igual o menos de KLT.
    - La sincronización de hilos en este modelo permite que un hilo se bloquee y otros hilos del mismo proceso sigan ejecutándose.
    - Permite que hilos de usuario mapeados a distintos KLT puedan ejecutarse en distintos procesadores.
    - Aprovecha las ventajas de ambos tipos.

- Básicamente hay hilos a nivel de usuario y dejo al SO verlos.

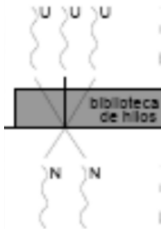


La relación entre ULT y KLT puede ser:

- Mapeados Uno a Uno: cada ULT mapea con un KLT. Si se bloquea un ULT, otro hilo del mismo proceso puede seguir ejecutándose. La concurrencia y7o paralelismo es máximo. Introduce un costo alto.



- Mapeado Muchos a Uno: muchos ULT mapean a un único KLT. Usados en sistemas que no soportan KLT. Si se bloquea un ULT; se bloquea el proceso.



- Mapeado Muchos a Muchos: muchos ULT mapean a muchos KLT. Este modelo multiplexa los ULT en KLT, logrando un balanceo razonable. No tiene el costo del modelo 1:1 y minimiza los problemas del bloqueo del modelo M:1

Threads: Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

Multiprocesadores

- Lograr mayor velocidad es físicamente complejo

    - Ninguna señal eléctrica se puede propagar más rápido que la velocidad de la luz
    - Existen problemas de disipación de calor y consumo eléctrico.
- Para solucionar esto se utiliza el computo en paralelo y/o distribuido.
    - Asignar varios problemas a varias CPU no es eficiente.
    - Debería existir un coordinador encargado de repartirlos (el SO básicamente).
    - Como hay varias CPU la complejidad aumenta en la distribución de tareas, el pasaje de mensajes y el acceso a memoria.

Multiprocesadores con memoria compartida

- Las CPU se comunican por medio de la memoria compartida.
    - Acceden a la misma por un único BUS físico.
    - Hay un único espacio lógico de direcciones para todos.
    - Todas las CPU tienen la misma velocidad de acceso.

Multiprocesadores con memoria independiente / pasaje de mensajes

- Cada memoria es local para una sola CPU y puede ser utilizada solo por esa CPU.
- Las CPU se conectan a una interconexión.
- Si una CPU quiere acceder a otra memoria solicita por pásame de mensaje a otra CPU los datos.
- Son fuertemente acoplados y son más sencillas de fabricar pero difíciles de programar.
- Hoy en día se utilizan en los servidores actuales.

Sistemas distribuidos

- Conecta sistemas de cómputo completos (cada nodo es una computadora completa) a través de una red.
- Son débilmente acoplados.
- Puede haber heterogeneidad de sistemas y hardware

    - Aparece una capa de homogenización (middleware que define estándar).
- Un nodo puede actuar como coordinador.

SO en multiprocesadores con Memoria Compartida

- Los programas se pueden ejecutar en cualquier CPU.
- Cada procesador ve un espacio normal de direcciones virtuales.
- Cuidado con la sincronización.
- Los SO realizan tareas como manejar syscalls, gestionar la memoria y la entrada/salida, además de funciones específicas para sistemas multiprocesadores, como sincronización o planificación de CPU.
    - Atacar las problemáticas que pueden ocurrir
- Cada procesador puede direccionar toda la memoria.

La arquitectura puede ser UMA o NUMA

Hardware – UMA (Uniform Memory Access)

- Hay un solo bus para comunicarse con la memoria.
- Antes de acceder al mismo se debe comprobar que no se encuentre ocupado.
    - Igualmente puede haber colisión.
- Si aumento cantidad de CPU, el acceso a memoria se vuelve ineficiente.
    - Las placas madre no son más grandes por problemas de censado de medio y de colisiones (por el tema de lo que tarda en viajar la señal).
- La limitación es el ancho de banda del bus, hay mucho tiempo de CPU ocioso

UMA (CPU con cache)

- Alternativa: agregar una cache a cada CPU para reducir el acceso al BUS compartido.
- Si el bloque se almacena en la cache en modo RO (read only), puede estar en varias cache.
- Si el bloque se almacena en modo RW, solo puede estar en una cache.
- Hay mecanismos de protección para evitar datos “sucios”

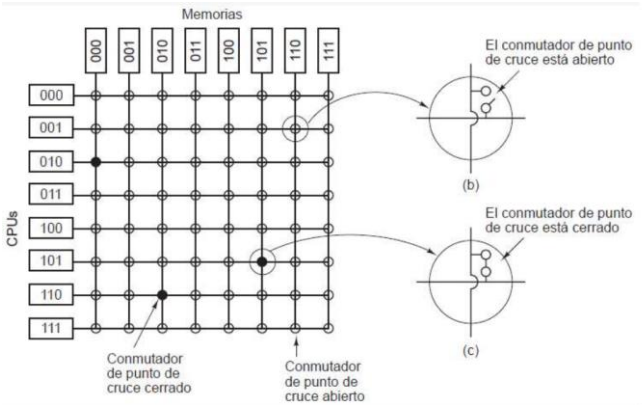
- Copias “limpias” vs. Copias “sucias”. Protocolo de coherencia de caché (se encarga el HW)
  - CPU intenta escribir una palabra que está en una o más caches
    - Se manda un mensaje al bus para informar.
      - Si hay CPU con copia limpia (la misma), se descarta para que solo quede la modificada.
      - Si hay CPU con copia sucia, se escribe a memoria e informa a la CPU para mantener coherencia.
  - Genera exceso en el uso de bus, vuelvo al mismo problema que tenía antes. No es conveniente tener tanta cache.

UMA (CPU con cache y memoria local)

- Alternativa: asignar a cada CPU una memoria privada.
- La memoria compartida solo se utiliza para escribir variables compartidas.
- Reduce el uso del bus, pero requiere de una asistencia activa por parte del compilador.
- Para datos privados debo seguir manejando coherencia.

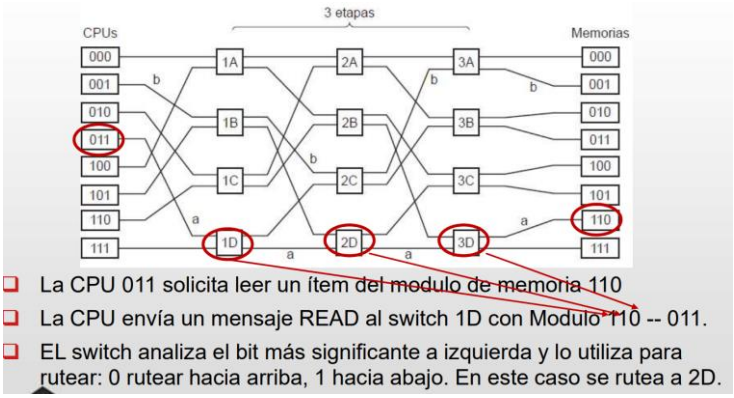
UMA con interruptores de barras cruzadas

- Un único bus limita a 16 o 32 CPU aproximadamente.
- Mejoro performance con interruptores de barras cruzadas: n CPU y n bancos de memoria, permite hasta n conexiones simultaneas.
- Abro y cierro switches asociados a CPU. Muchos switches.
- Implementado por hardware.



UMA con redes de conmutación multietapa

- Conmutación de n entradas x n salidas.
- Es una especie de grafo dirigido de caminos que permiten comunicar cada CPU con un módulo de memoria por un camino independiente.
- Implementado por hardware.

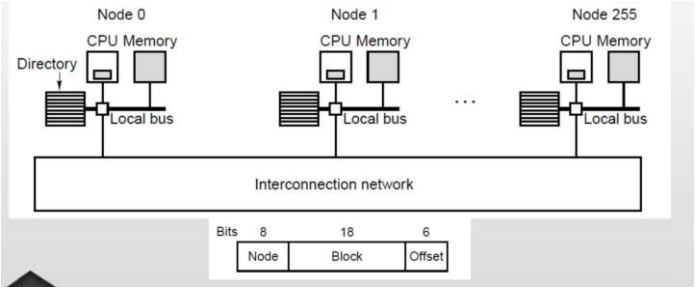


- Mensajes entre CPU y memoria están compuestos por 4 partes:
  - Modulo: módulo de memoria que se quiere acceder
  - Dirección: dirección de memoria dentro del módulo
  - CódigoOp: operación a realizar (READ o WRITE)
  - Valor (opcional): valor a escribir
- Solo requiere 12 switches.
- Mayor nivel de bloqueos en los circuitos.

Hardware – NUMA (Non-uniform Memory Access)

- Los procesadores UMA y sus técnicas son poco escalabres y costosas.
- Permite escalar en número de CPU.
- Se posee un único espacio de memoria visible por todas las CPU (como UMA).
- Cada unidad tiene cierto bloque de memoria local. Si una unidad quiere acceder a la memoria remota (de otro bloque), tarda más (porque se requiere acceso a un bus compartido).
- Rendimiento menor a UMA pero menos costoso. Mas escalable.
- Si dos procesos se comunicación mucho los pongo en la misma CPU.
- Mecanismos de cache para acelerar los tiempos de acceso a la memoria.
- Si no hay cache NC-NUMA si hay CC-NUMA:
  - En CC es importante mantener coherencia en las caches.

- Se usa el método multiprocesador basado en directorios:
    - BD que indica donde está cada línea y su estado (limpia o sucia (modificada)).
    - La BD debe almacenarse en hardware de propósito específico por cuestiones de performance.



Chips Multinúcleo

Con más transistores se puede:

- Agregar más memoria cache: la tasa de aciertos no se incrementa demasiado
- Agregar más velocidad de clock a una CPU: sigue existiendo un único hilo de ejecución.
- Agregar más CPU al mismo chip (núcleos): podrían compartir cache y memoria principal y se logra paralelismo.
- Importante diseñar un software (SO) para aprovechar el hardware.

Tipos de SO Multiprocesador

Varias metodologías para la administración en esquema de multiprocesadores

Cada CPU con su SO (poco utilizado)

- Se divide la memoria para cada CPU con su copia privada (las CPUs operan independientes).
- Se comparte el código de SO.
- Cada CPU tiene con su propio conjunto de procesos y por lo tanto su propia cola de listos (planificación independiente).
    - Cada CPU atrapa y maneja las syscalls de sus procesos
- Desbalance en la carga de trabajo.

- No se pueden compartir páginas.
    - Se debe usar pasaje de mensajes.
    - Hay memoria desperdiciada.
- Cada CPU tiene su propia copia de la cache llevando a posibles inconsistencias.

Maestro – Esclavo

- Única copia del SO y de su información.
- Todas las syscalls va a una CPU. Está puede ejecutar procesos si “le sobra tiempo”.
    - La CPU “Maestro” empieza a ser un cuello de botella.
- Asignar páginas entre los procesos de forma dinámica

SMP – Multiprocesadores Simétricos

- Única copia del SO en memoria y cualquier CPU puede ejecutarla.
- La syscall es atendida por la CPU que la invocó.
    - El proceso tiene partes del SO. Hay paginas propias del SO accesibles en modo kernel. Hay una interrupción, se pasa a modo kernel y se ejecuta el conjunto de páginas en modo kernel.
- Equilibro entre procesos y memoria (un único conjunto de tablas del SO)
- Problemas: se debe garantizar el acceso exclusivo a las estructuras propias del SO.
    - Locks:
        - Considerando a todo el SO como una gran sección critica.
            - Modelo poco utilizado, debido a la mala performance.
        - Por estructura(s):
            - Varias secciones criticas independientes cada una protegida por su propio mutex.
            - Mejor rendimiento pero es difícil determinar la sección critica.
            - Estructuras pueden pertenecer a más de una sección crítica
                - Puede generar deadlock.
            - Esquema que más se utiliza.

- Exclusión mutua con TSL (probar y establecer bloqueo):
    - Se lee la palabra de memoria, se almacena en un registro y se escribe un 1 en la memoria para hacer el lock.
    - Cuando termina libera escribiendo 0.
    - En uniprocesadores esta implementación es correcta, en multiprocesadores no.
        - Alternativas:
            - Cada CPU tiene su propia variable de lockeo en cache.
            - La CPU que no puede obtener el bloqueo se agrega a una lista y espera en su propio lock.
            - Agregar “delays” entre cada intento de TSL.

Planificación de multiprocesadores

Necesario determinar que se va a planificar:

- Que hilo (o proceso) se seleccionara
- En que CPU se ejecutara.
- Si se tienen ULT, el planificador los planifica a nivel de proceso.
- Si se tienen KLT, es posible tomar decisiones sobre su planificación (se va a analizar este).

Planificación de hilos independientes

- En ambientes de tiempo compartido.
- Usuarios ejecutan tareas no relacionadas entre sí.
- Una única cola de listos para todos los hilos o varias, una para cada prioridad.
- Ventajas:
    - Si los hilos son independientes entre si es una buena planificación.
    - Simple de implementar

    - Eficiente
    - Balanceo de carga, los trabajos son repartidos entre CPU.
- Desventaja:
    - Acceso a estructura única puede suponer cuello de botella.
- Espera activa:
    - Un hilo tiene un bloqueo de espera activa y finaliza su Quantum antes de liberar el mutex. Las otras CPU que esperan que se libere el bloqueo están ociosas mientras esperan.
    - Solución: usar un flag en cada proceso que indica que alguno de sus hilos tiene una espera activa:
        - No expulsar procesos que tengan el flag activado.
        - Cuando se libera el lock, se limpia el flag.
- Aprovechamiento de la Cache
    - Si un hilo se ejecuta en la misma CPU que antes hay más posibilidades de que los datos aun sigan en la cache. Se utiliza el algoritmo de Planificación de 2 niveles:
        - Se crea un hilo → se asigna a una CPU.
        - Cada CPU tiene propia colección de hilos y los planifica por separado.
        - Si queda una CPU ociosa, se reparten los hilos.

Planificación de hilos que trabajan en conjunto

- Se pueden planificar en conjunto (en varias CPUs).
- Mejora en trabajos en paralelo
- El grupo se planifica si hay CPUs libres para cada hilo, si no, el grupo espera.
- Cada CPU ejecuta solo 1 hilo bajando la productividad
- Si cada CPU ejecuta más de1 hilo podría ocurrir que los hilos no se ejecuten sincrónicamente ya que se planifican independientemente.
- Planificación por pandillas (una solución al problema anterior):
    - Los hilos relacionados se toman como una “Pandilla”.
    - Todos los miembros de una pandilla se ejecutan simultáneamente en distintas CPUs multiprogramadas.

- Todos los miembros de la pandilla inician y terminan sus intervalos en conjunto.
- La idea es ejecutar todos los hilos de un proceso juntos para realizar el pasaje de mensajes de forma más rápida y eficiente.

Multicomputadoras

- Conocidas como cluster de computadoras.
- Fuerte acoplamiento.
- No se comparte memoria, cada CPU tiene la suya.
- Son muchas PCs conectadas para distribuir el computo.
- Suelen carecer de Placa de Video, Sonido y en algunos casos disco.
- Tienen mucha CPU, memoria y placas de interconexión redundantes.
- Es necesario diseñar correctamente la red, debe ser rápida.
    - En la actualidad se suelen usar diseño estrella, anillo, Malla (o Mesh) Full Mesh (todos contra todos).

Transmisión de Datos

Dos esquemas:

1. Conmutación de paquetes y retransmisión:
    - Los bits se almacenan en un buffer hasta que se conforme un paquete y luego se envía.
    - El paquete es conmutado entre switches.
    - Mecanismos de retransmisión para solucionar pérdidas.
    - Se agrega latencia a medida que se atraviesan dispositivos de red.
- Placas de Red: proveen mecanismo de conexión hacia la red utilizada. Tienen buffers donde almacenan antes de copiar paquete a la RAM.
2. Conmutación de Circuitos:
    - El primer switch establece una ruta desde el origen hasta el destino.
    - Luego los bits se inyectan sin necesidad de almacenarlos en buffers.
    - Más rápida pero requiere de configuración adicional.
    - No hay control de flujo porque se asume que los bits no se pierden.

Problema en Multicomputadoras

- Copiado de paquetes en exceso entre la memoria principal y la placa de red puede generar una degradación del rendimiento si la placa de red está asignada al espacio del kernel (muchas syscalls).
    - Solución: asignar la interfaz de red al espacio de usuario, permitiendo que los procesos de usuario copien los datos directamente a la interfaz sin la intervención del kernel
        - Surge la cuestión de cómo gestionar la competencia en el acceso a la interfaz (el kernel también debería competir de igual a igual). Actualmente se utilizan técnicas DMA para esto.

Software de Comunicación a nivel de usuario

Los procesos en distintas CPUs en una multicomputadora se envían mensajes entre sí para comunicarse.

- Pasaje de mensajes:
    - Send y Receive:
        - SO provee interfaces a los procesos de usuario para realizar la comunicación.
            - Envío con bloqueo (síncronas): CPU inactiva durante transmisión
            - Envío sin bloqueo con copia (asincrónicas): desperdicio tiempo de la CPU por la copia adicional.
            - Envío sin bloqueo con interrupción: difícil de programar.
- RPC (Remote Procesure Call):
    - Proceso en una máquina invoca al procedimientos que se ejecuta en otra CPU (remoto) y se bloquea hasta que llegue la respuesta.
    - La comunicación es transparente al programador.

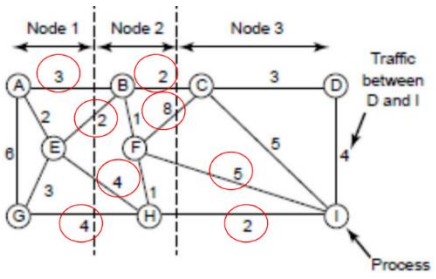
Planificación

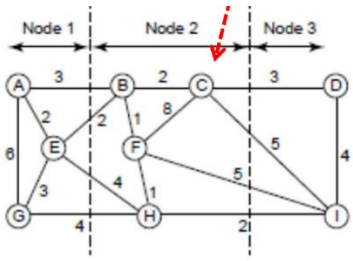
- Cada nodo tiene su propio conjunto de procesos.
- Un nodo no toma procesos de otro (seria costoso).

- Debe haber un balance de la carga.
- Se puede aplicar el concepto de planificación por pandilla.

Balanceo de la carga

- Se basa en lo que se “sabe” de cada proceso.
    - Tiempo de CPU, memoria requerida, comunicación con otros procesos.
- Objetivos:
    - Minimizar ciclos de CPU desperdiciados por falta de trabajo.
    - Minimizar el uso red.
    - Equidad para los procesos.
- Sistema representado como grafo.
    - Cada nodo es un proceso
    - Comunicación entre nodos representada a través de una arista.
        - Partir el grafo en tantos subgrafos como nodos se tengan considerando:
            - Requerimientos totales de CPU, de memoria.
            - Minimizar la cantidad de aristas entre nodos de distintos subgrafos (tráfico de red).
            - Buscar clusters con acoplamiento fuerte.





Se reduce tráfico de red (ej1: 30 , ej2: 28).

- Algoritmos distribuidos
    - Proceso se ejecuta en el nodo que lo creo al menos que el mismo este sobrecargado
        - Nodo sale a buscar uno no sobrecargado para “pasarle” el proceso.
            - Sobrecarga del enlace si todos los nodos se encuentran sobrecargados.
                - Mucho pasaje de mensajes.

- Además un nodo con poca carga puede que lo informe.

Sistemas distribuidos

- Cada nodo tiene su propia memoria privada.
- Menor acoplamiento
  - Pueden estar esparcidos por todo el mundo
- Cada nodo es una PC completa, incluyendo dispositivos
- Cada nodo puede ejecutar un SO y Hardware diferente, incluyendo su propio sistemas de archivos
- Middleware capa de software por encima del SO que garantiza uniformidad proveyendo una interfaz común a todos los procesos.

Resumen

Elemento	Multiprocesador	Multicomputadora	Sistema distribuido
Configuración de nodo	CPU	CPU, RAM, interfaz de red	Computadora completa
Periféricos de nodo	Todos compartidos	Compartidos, excepto tal vez el disco	Conjunto completo por nodo
Ubicación	Mismo bastidor	Mismo cuarto	Posiblemente a nivel mundial
Comunicación entre nodos	RAM compartida	Interconexión dedicada	Red tradicional
Sistemas operativos	Uno, compartido	Varios, igual	Posiblemente todos distintos
Sistemas de archivos	Uno, compartido	Uno, compartido	Cada nodo tiene el suyo
Administración	Una organización	Una organización	Muchas organizaciones

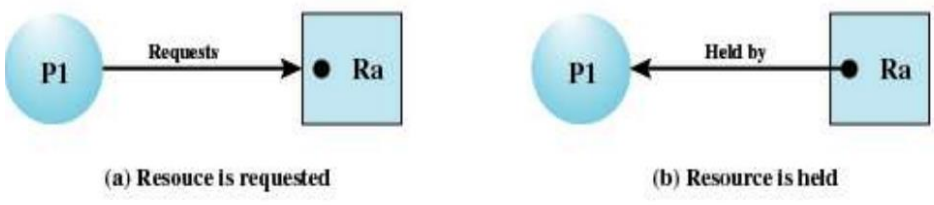
Deadlocks

Cada proceso está esperando por un recurso que está siendo usado por otro proceso del mismo conjunto.

Recursos

- Tipos:
  - Físicos: CPU, memoria, dispositivos.

    - Lógicos: archivos, registros, semáforos, etc.
    - Apropiativos: se le pueden quitar al proceso sin efectos dañinos (ej: memoria, CPU).
    - No apropiativos: si se le saca al proceso, éste falla (interrumpir una escritura a CD, impresora).
- Cada recurso puede tener varias instancias. Si son idénticas, se puede asignar cualquier instancia del recurso. El conjunto de instancias se lo conoce como "clase de recursos".
- Secuencia de empleo de recurso:
    1) Solicitud: Si no puede ser concedida inmediatamente, el proceso deberá esperar a que el recurso sea liberado
    2) Uso: El proceso puede operar sobre el recurso
    3) Liberación: Se libera el recurso para que pueda ser utilizado por otro proceso
- Si el recurso que se quiere utilizar está ocupado, se sigue un "Ciclo corto de solicitud": 1. Solicitud fallida, 2. Espera inactiva, 3. Nuevo intento de solicitud.
- Para representar la asignación de recursos se usa un Grafo de Asignación de recursos.
    - Permite visualizar el estado de los recursos del sistema y procesos en un momento determinado.
    - Proceso o recurso es representado por nodo.
    - Recurso con varias instancias tiene más de un punto dentro de nodo.
    - Arista representa una relación entre un Proceso y un Recurso
        - Dependiendo de la dirección indican distintos estados



(a) Resouce is requested (b) Resource is held

- Si el grafo no contiene ciclos no hay deadlock.
- Si el grafo contiene un ciclo:
    - Si hay una instancia por tipo de recurso hay deadlock.
    - Si hay varias instancias por tipo de recurso hay posibilidad de deadlock.
- Si se muere un proceso libera todos los recursos.

Condiciones para que se cumpla un deadlock

Se deben cumplir todas

1. Exclusión mutua: en un instante de tiempo dado, solo un proceso puede utilizar una instancia de un recurso
2. Retención y espera: los procesos mantiene los recursos asignados mientras esperan por los nuevos recursos.
3. No apropiación: a un proceso no se le puede sacar los recursos que ya tiene.
4. Espera circular: proceso forma parte de una lista circular en la que cada proceso de la lista está esperando por al menos un recurso asignado a otro proceso de la lista.

Manejo Deadlocks

Varios enfoques:

- Usar un protocolo que asegure que NUNCA se entrará en estado de deadlock.
    a. Prevenir: Que no se cumple alguna de las 4 condiciones. Restricciones en la forma en que los procesos REQUIEREN los recursos.
    b. Evitar: Tomar decisiones de asignación en base al estado del sistema. Asignar cuidadosamente los recursos, manteniendo información actualizada sobre requerimiento y uso de recursos.
- Importante saber diferencia entre prevenir y evitar.
- Permitir el estado de deadlock y luego recuperar.
- Ignorar el problema y esperar que nunca ocurra un deadlock.

Prevenir

1. Condición de exclusión mutua
    - Asigno abstracción de recurso, simulo que el recurso es compartible.
        - Se pueden implementar listas y que el recurso sea manejado por un proceso global (spooler). Igualmente el spooler podría llenarse y bloquear a los procesos.
        - Hay ciertos recursos que no se pueden compartir (BD)

2. Condición de retención y espera
    - Si un proceso requiere un recurso que no está disponible que libere los otros (no siempre se puede y además puede impedir que el proceso avance).
    - Alternativas:
        - Un proceso debe requerir y reservar todos los recursos a usar antes de comenzar la ejecución
        - El proceso puede requerir recursos sólo cuando no tiene ninguno
    - Ambas alternativas son ineficientes: baja utilización de recursos y posible inanición.
3. Condición de no apropiación
    - No siempre se puede expropiar
        - Soluciones:
            - Proceso requiere un recurso no disponible, los recursos de ese proceso son apropiados y se lo bloquea. Se desbloquea cuando el recurso pedido y los que tenía estén disponibles.
            - Proceso requiere un recurso no disponible, sistema chequea si el recurso lo tiene otro proceso que esté bloqueado esperando otro recurso. Si es así, se apropia el recurso del bloqueado y se asigna al solicitante. Si no es así, el solicitante es bloqueado hasta que el recurso esté disponible.
            - Virtualizar recursos:
                - El proceso no accede directamente al recurso, sino que accede a un demonio que lo administra
                - A medida que los procesos requieren el recurso, interactúan con el demonio quien almacena los trabajos en una cola (spooler).
                - Se logra que el recurso físico pueda ser “utilizado” por varios procesos al mismo tiempo (una forma ficticia de compartir el recurso, ya que en realidad solo 1 proceso a la vez lo está utilizando).
                - Esta es la menos traumática.

4. Condición de Espera circular
    - Se define ordenamiento de los recursos.
        - La función F asigna un numero único a cada recurso.
            - F:(R→N) N conjunto de los naturales.
    - Asigno recursos de acuerdo a la función en orden creciente.
    - Un proceso, que ya tiene R_i puede requerir R_j si y solo si $F(R_j) > F(R_i)$.
    - No es eficiente.

Evitar

- Baja la performance del sistema.
- Puede producir baja utilización de recursos.
- El SO cuenta con información sobre el uso de los recursos:
    - Cómo y en qué momento son requeridos, demanda máxima, etc.
- Se toman decisiones dinámicas sobre la asignación de recursos para prevenir Deadlocks; si se detecta riesgo de Deadlock, se niega la asignación.
- Se necesita conocer la secuencia de solicitud, uso y liberación de cada recurso requerido por el proceso.
- El algoritmo de evitación de interbloqueo examina dinámicamente el estado de asignación de recursos para evaluar las posibles consecuencias ante cada solicitud.
- El estado de asignación de recursos se define por el número de recursos disponibles y asignados, y las demandas máximas de los procesos, quienes deben declarar el número máximo de recursos de cada tipo que puedan necesitar.

Estado sano o seguro

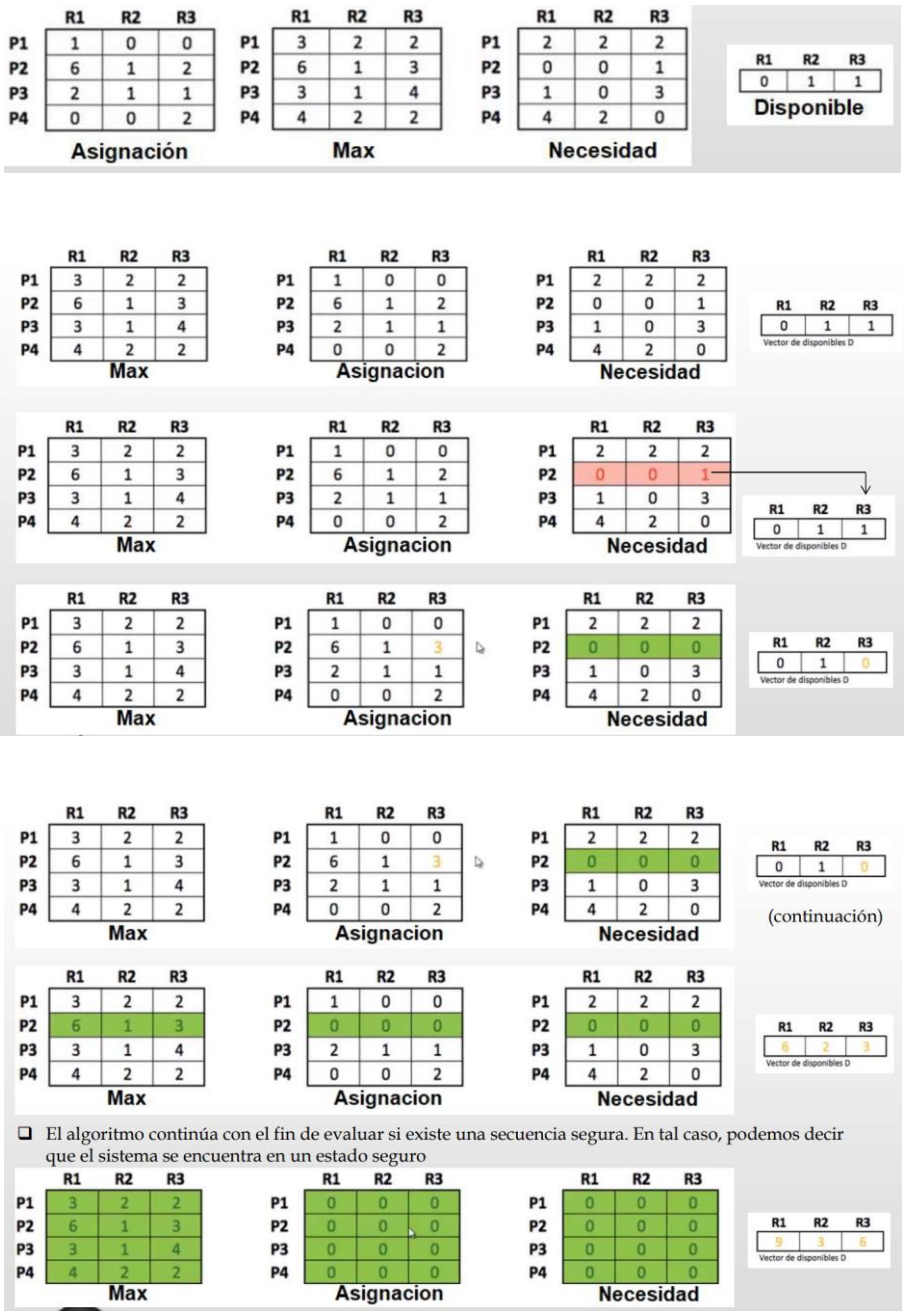
- Estado seguro: el SO puede asignar recursos a cada proceso de un conjunto de alguna manera, evitando el deadlock.
- Cuando un proceso solicita un recurso disponible, el sistema debe decidir si la asignación inmediata deja el sistema en un estado seguro.
    - Para que lo deje seguro debe haber una “cadena segura” de procesos que puedan ejecutarse con todos los recursos disponibles sin que haya deadlock.
    - Si no existe cadena segura se está en estado inseguro.

Importante:

- Un estado seguro garantiza que NO hay deadlock.
- Si hay deadlock, estoy en estado inseguro.
- En estado inseguro hay posibilidad que haya deadlock. No todo estado inseguro es deadlock. Si estoy en estado inseguro y a la larga no se hace nada hay deadlock.
- EVITAR/AVOIDANCE: trata de nunca entrar a estado inseguro. Garantiza lo seguro entonces no hay deadlock.

Algoritmos para evitar el deadlock

- Si se tiene una sola instancia por recurso:
    - Algoritmo determina el estado seguro de un sistema utilizando el grafo de asignación de recursos. Una secuencia segura es evitar los bucles.
- Si se tienen varias instancias por recurso:
    - Algoritmo del banquero, es teórico.
        - Busca encontrar una secuencia segura de asignación. Para esto los procesos necesitan declarar el número máximo de instancias que necesitan de los recursos y el SO decide en que momento asignarlos para garantizar estado seguro.
        - Se necesitan las siguientes estructuras:
            - n: cantidad de procesos
            - m: cantidad de tipos de recursos
            - disponible: vector con la cantidad de recursos disponibles para cada tipo.
            - asignación: matriz de n x m que indica cuantos recursos tiene asignados cada proceso.
            - max: matriz de n x m que indica la cantidad máxima de recursos que va a necesitar un proceso.
            - need: matriz de n x m que indica la cantidad de recursos que le faltan a un proceso para completar su ejecución.



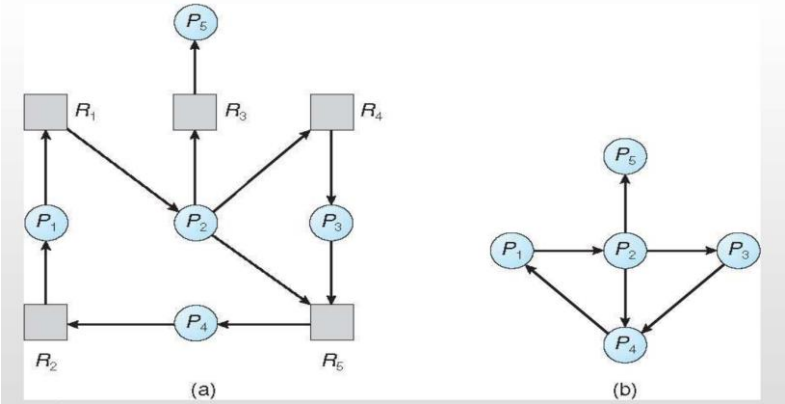
❑ El algoritmo continúa con el fin de evaluar si existe una secuencia segura. En tal caso, podemos decir que el sistema se encuentra en un estado seguro

Detección

Técnicas que permiten determinar si hay deadlock y si lo hubo poder recuperarse del mismo. Hay:

1) Algoritmo que examine si ocurrió un deadlock
   - Con recursos con una sola instancia:
     - Análisis del grafo de asignación.

- Como el que se vio mas arriba pero en este caso no hay nodos para los recursos. Nodos = Procesos.
- Hay una arista entre los nodos indicando que un proceso espera a que el otro (otro nodo) libere un recurso.
- El algoritmo busca básicamente si hay algún ciclo.
    - Si hay un ciclo hay bloqueo.



- Con recursos de varias instancias:
    - Algoritmo del Banquero
        - Como el que se vio más arriba.

- La frecuencia de ejecución de los algoritmos depende del sistema. Si es muy común que ocurra deadlock, lo corro a intervalos más regulares.
- Comprobar el estado cada vez que se solicita un recurso y estos no pueden ser asignados consume mucha CPU. Si se invoca arbitrariamente puede haber muchos ciclos en el grafo y no se puede saber qué proceso lo causo. Hay que encontrar un término medio.
- Se pueden activar ocasionalmente a intervalos regulares o cuando se detecte una inactividad prolongada de procesos o un menor uso significativo de la CPU.

2) Algoritmo para recuperación del deadlock
    - Hay varias maneras para eliminar el deadlock:
        - Caso 1: se puede resolver manualmente. (le aviso al SO)
        - Caso 2: el sistema automáticamente rompe el deadlock.
    - En cualquier caso para romperlo se puede:
        - Violar la exclusión mutua y asignar el recurso a varios procesos (no siempre se puede).
        - Cancelar los procesos suficientes para romperla espera circular.

- Expropiar recursos de los procesos que se presume están en estado de deadlock.

Eliminar procesos

- Dos métodos:
    - Matar todos los procesos en estado de deadlock: simple, pero a un muy alto costo. Si todos tienen un rollback a estado seguro conviene esta opción.
    - Matar de a un proceso por vez hasta eliminar el ciclo: se tiene overhead ya que por cada proceso que se va eliminando se debe re-ejecutar el Algoritmo de Detección para verificar si el deadlock desapareció o no.
- No es fácil eliminar un proceso.
    - Selección de la víctima: se debe seleccionar aquel proceso cuya terminación represente el costo mínimo para el sistema.
        - Criterios:
            - Prioridad más baja.
            - Menor cantidad de tiempo de CPU hasta el momento.
            - Mayor tiempo restante estimado para terminar.
            - Menor cantidad de recursos asignados hasta ahora.
        - Es ideal elegir un proceso que se pueda volver a ejecutar sin problemas.
- Se puede elegir un proceso que no esté directamente en el ciclo. Menor costo, pero puede resultar ineficiente.

Expropiación de recursos

Quitar los recursos de los procesos que los tienen y asignarlos a otros que los solicitan hasta conseguir romper el interbloqueo. Hay que considerar tres aspectos:

- Apropiación/Selección de Víctima - minimizar el costo, baja prioridad, recién lo empezó a usar, etc.
- Rollback - regresar al proceso que le sacamos los recursos a una instancia segura (check point).

- Inanición – si siempre escoge al mismo como víctima. Se evita esto asegurando que se elija un numero finito de veces. Incluir el número de retrocesos como factor de costo.

Los SO actuales se tienen que ir adaptando al entorno, por lo que usualmente combinan estas técnicas.