Sistemas Operativos II

Resumen Primera Parte

Emiliano Salvatori

Agosto 2019

1. Clase nº 2

1.1. Concepto de Programa y sus Estados

Un programa es una entidad pasiva, como el ejemplo en C escrito en clase. Solamente que es compilado y ejecutado. Esto está estático en un disco, cuando se compila se convierte en una entidad activa, ya que se convierte en un proceso.

El concepto de proceso es mucho más que código compilado, tiene muchos más atributos que una simple ejecución de un programa.

Pregunta de examen: realizar un grafo de los estados y explicarlos y tener en cuenta en poner los motivos de las flechas.

1.1.1. Estados

Los tipos de Estados son los siguientes:

- **Nuevo**: El proceso está siendo creado. Es cuando apenas se lanza un programa. El equivalente sería a darle doble click a un lanzador en el Sistema Operativo o cuando se ejecuta por terminal: *./ejecutable*. El proceso está siendo creado.
- **Preparado**: El proceso está a la espera de que le asignen tiempo para ejecutarse en el procesador. Lo que hace el SO es poner el programa en espera para los programas que están listos para correr en el procesador; se pueden visualizar por ejemplo cuando se corre *htop* se listan estos estados de procesos. Que están esperando que el kernel le asigne tiempo para que se corra en el micro.
- **En ejecución**: Es cuando el SO le asigna tiempo dentro del procesador para poder ejecutar el código. El estado significa que se están ejecutando las instrucciones pertenecientes al proceso.
- **Espera**: Es un estado que se le asigna el SO cuando se produce por ejemplo un fallo de página. [1].

  O también se produce cuando realiza operaciones de E/S que tardan mucho tiempo en realizar la petición al hardware. Como esta petición tarda mucho tiempo, el SO pone al proceso en Espera y cuando le es devuelta la petición, lo vuelve a ejecutar. Según la filmina, el proceso está esperando a que se produzca un suceso.
- **Terminado**: Es el estado que el SO asigna a un proceso cuando terminó de ejecutar lo que debía de procesar.

Significado de Interrupción: El sistema operativo le otorga a cada proceso según un quantum un determinado tiempo para ejecutarse dentro del procesador para luego quitarlo si es que no termina. Cuando se le acaba el tiempo para correr, el proceso pasa nuevamente a *preparado*. Importante: Las interrupciones se hacen por tiempo.

Los estados en los que pueda estar un proceso depeden también de lo que esté ejecutando el programa. Por ejemplo: Si se pone un *for* que hace muchas cuentas matemáticas, sólo estará en dos estados: Preparado y Ejecución porque la ALU por sí sola puede realizar estas cuentas matemáticas, sin la necesidad de recurrir a ninguna otra petición externa. En cambio si en ese emphfor se pone un imprimir por pantalla estará más tiempo en el estado de Espera.

[1] Se denomina de esta forma a una excepción arrojada cuando un programa informático requiere una dirección que no se encuentra en la memoria principal actualmente. Recordar que los programas cargan en memoria de a pedazos, cuando no se encuentra ese pedazo en la siguiente instrucción a ejecutar, se produce el fallo de página

1.2. Bloque de control de procesos

Se denomina Bloque de Control de Procesos al conjunto de atributos que tienen los procesos y que NO tiene un programa compilado. Estas características no las poseen los programas estáticos.

Pregunta de examen: liste las caracteristicas del Bloque de Control de Procesos (PCB).

- **Estado**: sería en qué lugar del grafo me encuentro. Qué estado le otorga el SO al proceso.
- **PID** : process ID es como el DNI del proceso.
- **Contador de programa**: puntero a la siguiente instrucción a ejecutar. Esto es indispensable porque el SO cada vez que quita un proceso del procesador porque se le acabó el tiempo de ejecución, requiere guardar las instrucciones en las que se quedó ejecutando, tiene que saber en qué parte de la ejecución quedó pendiente.
- **Registros del CPU**: Al igual que el punto anterior, también al quitar del procesador un proceso es necesario que se guarden las variables que estaba utilizando. Cuando el proceso se guarda porque se le acabó el quantum, debe tomar el registro de todas las variables que tiene el programa y guardarlas y saber en dónde terminó para que cuando se le vuelva a dar tiempo en el procesador saber desde dónde seguir.
- **Información de la planificación de la CPU**: Se tiene información acerca de la prioridad que el SO le da al proceso.
- **Información de gestión de memoria**: son los valores referidos a memoria utilizadad por el proceso, como ser: valor de registros básicos, tablas de paginación o segmentos, etc.
- **Información contable**: cuántos milisegundos corrió el proceso sobre el procesador, por ejemplo. Lleva la cuenta de cuánto tiempo estuvo corriendo el proceso.
- **Información de entrada salida**: qué dispositivo se está uttilizando, por ejemplo si hay un proceso que utiliza la impresora, el SO sabe que no puede asignar a otro proceso mientras el otro lo está utilizando.

1.3. Concurrencia

Dos procesos son concurrentes entre si, sólo si es que son independientes. Es decir, son independientes entre si cuando no importa el orden en que se ejecute, el resultado es siempre el mismo. Por ejemplo: si hay un proceso que imprime "HOLA.ºtro proceso imprimie "PERROτ se corren al mismo tiempo, son independientes porque no importa el orden el resultado siempre es el mismo. El resultado de un proceso NO depende del resultado del otro.

Para que exista independencia en la ejecución de procesos, estos deben tener comunicación entre si. Por ejmplo cuando se visita la página de Youtube y se reproduce un video. Un proceso está encargado de ir llenando el buffer del video mientras que otro proceso está pendiente de si se aprieta play para poder comenzar a poner en secuencia las imágenes. El primero de ellos siempre va adelante y siempre haciendo peticiones de E/S a la memoria asignada como buffer, manejándose de forma independiente.

Para el usuario final, el resultado es el mismo, se ve el video. Pero los procesos que atienden estos servicios NO son lo mismo, son varios procesos intercomunicados entre si brindando un mismo servicio. El programa en este caso es un sistema concurrente. [2]

Las cuestiones a tener en cuenta a la hora de diseñar procesos concurrentes:

- Comunicación entre procesos.
- Compartición y competencia por los recursos.
- Sincronización de la ejecución de varios procesos.
- Asignación del tiempo del procesador a los procesos.

[2] Dos o más procesos decimos que son concurrentes, paralelos, o que se ejecutan concurrentemente, cuando son procesados almismo tiempo, es decir, que para ejecutar uno de ellos, no hace falta que se haya ejecutado otro.

1.4. Paralelismo

Capacidad de ejecutar procesos de manera paralela. Está asociado al hardware. La única manera de ejecutar dos procesos A y B a la vez, es que lo permita el hardware. Se tiene cuando hay mas de un cause por los que se puede ejecutar un proceso, y esto lo dispone el hardware. Antes tenían procesadores con un solo cause, se corría un solo proceso dentro de un solo procesador. En los 80 se inventaron varios causes y al día de hoy los causes se duplicaron.

Recordar que **Paralelismo NO implica concurrencia**. Si se tiene paralelismo (con hardware adecuado) se va a necesitar la capacidad de correr procesos concurrentes. **Pero para que haya Concurrencia es necesario que haya Paralelismo**

1.5. Gestión de procesos

Multiprogramación: tener un solo cause y dividir en tajadas de tiempo varios programas y tener la sensación de que sucede todo a la vez. Se denomina multiprogramación a una técnica por la que dos o más procesos pueden alojarse en la memoria principal y ser ejecutados concurrentemente por el procesador o CPU.

Hay que tener en cuenta que los procesos van intercambiandose entre si un tiempo el proceso P1, otro tiemp el P2 y puede haber un P3 que lo utilice otro tiempo. Lo que NO puede pasar es que se superpongan los Procesos.

Multiprocesamiento: es tener varios causes. Se tiene varios procesadores para varios procesos. Multiprocesamiento o multiproceso es el uso de dos o más procesadores (CPU) en una computadora para la ejecución de uno o varios procesos (programas corriendo). Si se tiene multiprocesador se puede tener por ejemplo a P1 y P2 corriendo en simultaneo (dependiendo de cuántos cause tenga el procesador).

Procesamiento distribuido: Consiste en la gestión de varios procesos, ejecutándose en sistemas de computadoras múltiples y distribuidos. Es la idea de cluster.

La computación distribuida o informática en malla (grid) es un modelo para resolver problemas de computación masiva utilizando un gran número de ordenadores organizados en clústeres incrustados en una infraestructura de telecomunicaciones distribuida.

1.6. Función Fork

La función fork(), se utiliza para crear un nuevo proceso duplicando el proceso de llamada actual, siendo el proceso recién creado conocido como **proceso hijo** y el proceso de llamada actual conocido como **proceso padre**. Entonces podemos decir que fork () se usa para crear un proceso secundario al proceso de llamada.

Con esta instrucción, el cause de ejecución no es de arriba hacia abajo como en un programa simple realizado en C, sino que se tiene un cause distinto, bifurcado. Se puede correr de forma concurrente. Se tene dos procesos simultaneamente por ejemplo:

Como se puede ver *fork()* lo que hace es crear un proceso nuevo, es decir un proceso hijo pero dentro de la ejecución del proceso padre. Lo que hace el SO cuando encuentra esta instrucción es duplicar el espacio de memoria del padre, compartiendo todas las variables. Copia toda la memoria estática al proceso hijo.

Se debe recordar que Minix tiene una tabla de procesos que sólo puede albergar 100 procesos (de 0 a 99). Si se llena esa tabla, el fork() da error, simbolizando su valor de retorno con un -1. Solo puede crear 99 procesos nada más. El fork() da -1 y sale con error y este error lo toma el SO en el cause del proceso que lo invocó.

Con el fork no se puede saber cuándo un proceso se ejecuta primero. Por lo que se puede utilizar la función *sleep()* en el del hijo. Para dormir al hijo y que ejecute el hijo y no el padre.

Pregunta de examen: ¿existe otro estado que no sea el de *terminado*? Este estado posible se denomina *Estado Zombie*, que es cuando el hijo no tiene nada que ejecutar y está esperando que el padre muera o al revés. En sistemas operativos Unix, un proceso zombi o "defunct"(difunto) es un proceso que ha completado su ejecución pero aún tiene una entrada en la tabla de procesos, permitiendo al proceso que lo ha creado (padre) leer el estado de su salida. Metafóricamente, el proceso hijo ha muerto pero su .ªlma.ªún no ha sido recogida.

Preguna de examen: Escriba un código en C donde un proceso A cree un proceso hijo B. Esto sería similar a poner el *ejemplo5.c* citado anteriormente.

Se debe tener en cuenta que el PID puede tener 3 valores:

- Un número mayor que cero.
- Un número igual a cero. Puede ser cero por ejemplo para el PID del hijo de un proceso padre
- Un número igual a -1.

1.7. Hilos y procesos multihilo

Proceso padre tiene un espacio de memoria asociado y luego tiene un código. En el espacio tiene la pila, los datos, el contador de programa. Y tiene un solo cause. Cuando se hace invoca a la función fork(), se crea un hijo idéntico, lo que cambia solo es el PID. Por lo que es costoso en tiempo y en recurso invocar un fork().

Un proceso de Unix es cualquier programa en ejecución y es totalmente independiente de otros procesos. El comando de Unix *ps* nos lista los procesos en ejecución en nuestra máquina. Un proceso tiene su propia zona de memoria y se ejecuta "simultáneamente.ª otros procesos

La idea de hilo es agarrar el procso y utilizar el mismo tamaño de memoria de un solo proceso pero hacer correr varias hebras. Esto tiene muchos más beneficios que desdoblar un proceso en dos, ya que los hilos comparten espacio de memoria. Los hilos ya de por sí comparten espacio de salida. Cada hebra corre por distintos causes, si una se bloquea el proceso sigue consumiendo tiempo en el procesador ejecutando otros hilos y no saca el proceso de su ejecución como SI sucedería con un proceso hijo y padre.

Dentro de un proceso puede haber varios hilos de ejecución (varios threads). Eso quiere decir que un proceso podría estar haciendo varias cosas "a la vez". Los hilos dentro de un proceso comparten todos la misma memoria. Eso quiere decir que si un hilo toca una variable, todos los demás hilos del mismo proceso verán el nuevo valor de la variable. Esto hace imprescindible el uso de semáforos o mutex (EXclusión MUTua, que en inglés es al revés, funciones *pthread_mutex*) para evitar que dos threads accedan a la vez a la misma estructura de datos. También hace que si un hilo "se equivocaτ corrompe una zona de memoria, todos los demás hilos del mismo proceso vean la memoria corrompida. Un fallo en un hilo puede hacer fallar a todos los demás hilos del mismo proceso.

Conclusión: Un proceso es, por tanto, más costoso de lanzar, ya que se necesita crear una copia de toda la memoria de nuestro programa. Los hilos son más ligeros.

Ventajas de la programación multihilo:

- **Capacidad de respuesta**: permite que un programa continúe ejecutándose incluso aunque parte de él esté bloqueado.
- **Compartición de recursos**: por omisión, las hebras comparten la memoria y los recursos del proceso al que pertenecen.
- **Economía**: asignar memoria y recursos para crear procesos es costosa, es más económico realizar cambios de contexto entre hebras.
- **Multiprocesador**: se pueden ejecutar en paralelo en diferentes procesadores (un proceso monohebra solo se puede ejecutar en un sólo microprocesador).

1.8. Instancias de Kernel

Modelo Muchos a Uno: Otra forma es que muchas hebras llaman a una sola instanacia de kernel. Es un costoso cuando hay muchas hebras. Es eficiente pero el proceso completo se bloquea si una hebra realiza una llamada bloqueante al sistema.

Modelo Uno a uno: Asigna cada hebra de usuario a una de Kernel. Todo el tiempo se hacen solicitudes al kernel (como por ejemplo imprimir en pantalla). Cuano se tienen hebras se pueden una instancia de kernel que le responda a cada hebra. Cada hebra hace una solicitud a una instanaica del kernel. Esto se denomina Uno a Uno: un hilo solicita una instancia de Kernel. Este modelo proporciona mayor concurrencia que el anterior, se permiten herbas bloqueantes mientras se ejecutan otras. El incovenviente de ello es que crear una hebra de usuario requiere crear su correspondiente hebra de kernel por lo que repercute en el rendiemiento

Modelo muchos a muchos: La solución que se da es un híbrido entre ambas: muchos hilos solicitan al kernel una instancia en particular y el kernel evalúa cuándo crear una instancia para cada hebra. Multiplexa muchas hebras de usuario sobre un número menor o igual de hebras de Kernel. Se pueden crear tantas hebras de usuario como sea necesario y las correspondientes hebras del Kernel pueden ejecutarse en paralelo en un multiprocesador. Cuando una hebra realiza una llamada bloqueante, el kernel planifica otra hebra.

1.9. Hilos en modo Usuario

Los hilos en modo usuario tienen determinadas ventajas:

- El núcleo no sabe que existen.
- Tabla de subprocesos probada para cambios de contexto.
- Cambio de contexto mucho mas rápido entre hilos (no se pasa al kernel).

- Cada proceso puede tener su algoritmo de planificación.

Inconvenientes:

- Llamadas bloqueantes al sistema.
- Fallos de página.
- Tienen que ceder la CPU entre ellos: conmutación en el mismo proceso.
- Precisamente queremos hilos en procesos con muchas E/S para obtener paralelismo, es decir que se están bloqueando muy frecuentemente.

1.10. Hilos en Modo Kernel

Ventajas:

- El núcleo mantiene la tabla de hilos, que es un subconjunto de la de procesos.
- Las llamadas bloqueantes no necesitan funciones especiales.
- Los fallos de página no suponen un problema.
- Al bloquearse un hilo, el núcleo puede conmutar a otro hilo de otro proceso.

Inconvenientes

- Las llamadas bloqueantes son llamadas al sistema.
- La creación y destrucción de procesos es mas costoso, por lo que se trata de reutilizar hilos.

1.11. Diferencia entre procesos e hilos

Creación:

- **Procesos**: son costosos para crear
- **Hilos**: son bastante ligeros

Recursos(memoria):

- **Procesos**: Independientes
- **Hilos**: Compartidos

Comunicación:

- **Procesos**: compleja
- **Hilos**: Sencilla

Cambio por Sistema operativo:

- **Procesos**: Muy lento
- **Hilos**: Rápido

Proramación:

- **Procesos**: Reducida
- **Hilos**: Alta

2. Clase nº 3

2.1. Recursos

En SO se puede determinar que los Recursos:

- **Apropiativos**: Un recurso apropiativo es uno que se puede quitar al proceso que lo posee sin efectos dañinos. La memoria es un ejemplo de un recurso apropiativo. Por ejemplo, un recurso de este tipo es el microprocesador, el cual se cede por determinado tiempo a cada recurso y luego se lo expropia.
- **No apropiativos**: Un recurso no apropiativo es uno que no se puede quitar a su propietario actual sin hacer que el cómputo falle. Si un proceso ha empezado a quemar un CD-ROM y tratamos de quitarle de manera repentina el grabador de CD y otorgarlo a otro proceso, se obtendrá un CD con basura.

Para las gráficas, los Recursos se dibujan como cajas y los procesos como círculos o burbujas. Hay recursos que tienen dos puntos que significa que el recurso tiene dos instancias, es un mismo recurso como subdividdo; cuando se toma el puntito en el diagrama, se debe especificar qué tipo de sub-recurso está tomando de ese recurso (de la caja general).

2.2. Deadlock/Interbloqueo/Abrazo Mortal

Para que exista un Deadlock deben darse 4 condiciones:

- **Exclusión Mutua**: debe haber recursos no compartidos de uso exclusivo, es decir, cada recurso se asinga a un solo proceso en un momento dado. Recursos no compartidos de uso esclusivo, es cuando ese recurso se lo hace un proceso, ese recurso queda bloqueado para ese proceso. Cada recurso se asigna en un momento dado a sólo un proceso, o está disponible.
- **Hold and Wait (contención y espera)**: se toma un recurso y se espera por otro. (Flecha que entra hacia el proceso, quiere decir que ese recurso esta reternido por el proceso, cuando la flecha está desde el proceso al recurso, el proceso está pidiendo ese recurso). Los procesos que actualmente contienen recursos que se les otorgaron antes pueden solicitar nuevos recursos.
- **Condicion no apropiativa**: significa que una vz que se le dio un recurso a un proceso, no se lo puede expropiar. El micro NO estaría en esta condición ya que se cede a un proceso y luego se lo saca todo el tiempo. Los recursos otorgados previamente no se pueden quitar a un proceso por la fuerza. Deben ser liberados de manera explícita por el proceso que los contiene.
- **Espera circular**: es una cadena circular donde cada proceso está esperando un recurso que está siendo tomado/utilizado por otro. Debe haber una cadena circular de dos o más procesos, cada uno de los cuales espera un recurso contenido por el siguiente miembro de la cadena.

2.2.1. Algoritmo de la avestruz

El algoritmo del avestruz es un concepto informático para denominar el procedimiento de algunos sistemas operativos. Esta teoría, acuñada por Andrew S. Tanenbaum, señala que dichos sistemas, en lugar de enfrentar el problema de los bloqueos mutuos (deadlock en inglés), asumen que estos nunca ocurrirán. Ingenieros vs Matemáticos: los últimos quieren darle solución y los ingenieros decían que no hacía falta porque la probablilidad de que suceda un interbloqueo era baja, por lo tanto no hay que hacer nada.

¿Cómo se da cuenta de un deadlock? Hay una solución que se puede realizar realizando Grafos [ver carpeta]. Se comienza siempre por algún proceso externo (en el caso es B) y se siguen las flechas.

2.2.2. Detección y Recuperación

Lo que se hace es tomar el primer elemento del arreglo y se compara contra todos los elementos restantes. Si no existe repetido entonces se pasa al siguiente elemento y así con todos. La relación es todos contra todos.

Hay momentos en donde se pueden generar más sub-listas porque pueden haber otros caminos (recursos y procesos posibles) lo que genera un grafos más grande, haciendo que el algoritmo se vuelva muy costoso.

La ventaja de éste algoritmo de Detección y Recuperación es que *es simple* de implementar. Pero la desventaja como se nombró anteriormente, es que *es costoso* en lo que respecta a la capacidad de cómputo.

$$Lista = [B]$$

$$Lista = [B, T]$$

$$Lista = [B, T, E]$$
$$Lista = [B, T, E, V]$$
$$Lista = [B, T, E, V, G, U]$$
$$Lista = [B, T, E, V, G, U, D]$$
$$Lista = [B, T, E, V, G, U, D, T]$$

Se puede ver que en la última iteración vuelve a a parecer el recurso T, por lo que existe una espera circular. Se puede observar que además este tipo de algoritmo debe ser recursivo por lo que se tiene una complejidad mucho mayor en consumo de recursos de memoria.

2.2.3. Algoritmo del Banquero

El algoritmo mantiene al sistema en un *estado seguro*. Un sistema se encuentra en un estado seguro si existe un orden en que pueden concederse las peticiones de recursos a todos los procesos, previniendo el interbloqueo. El algoritmo del banquero funciona encontrando estados de este tipo.

Los procesos piden recursos, y son complacidos siempre y cuando el sistema se mantenga en un estado seguro después de la concesión. De lo contrario, el proceso es suspendido hasta que otro proceso libere recursos suficientes.

Nota: Si hubiera un deadlock se harán varias iteraciones y la cantidad de disponibles no podrá satisfacer ls procesos. En este algoritmo no importa el orden de los procesos, se irán terminando sin orden, sino que depende de los recursos que vaya slicitando para temrinar. En los examenes si llega el caso de producirse un Deadlok, terminaría el ejercicio señalandose que se llegó a un deadlock.

2.2.4. Inanición

En informática, inanición (starvation en inglés) es un problema relacionado con los sistemas multitarea, donde a un proceso o un hilo de ejecución se le deniega siempre el acceso a un recurso compartido. Sin este recurso, la tarea a ejecutar no puede ser nunca finalizada.

La inanición NO es sinónimo de interbloqueo, aunque el interbloqueo produce la inanición de los procesos involucrados. La inanición puede (aunque no tiene porqué) acabar, mientras que un interbloqueo no puede finalizar sin una acción del exterior.

2.3. Comunicación entre procesos (IPC)

Hay que tener en cuenta que la función *fork()* tiene el siguiente funcionamiento: cuando se ejecuta un código cuyo flujo es de arriba hacia abajo y se encuentra con la función fork(), lo que sucede es que a partir de ese momento el programa se **duplica**, es decir que el SO coloca en el espacio de memoria dos imagenes totalmente iguales del mismo código. Es por ello que luego de la llamada a fork() es necesario escribir el código del proceso hijo y luego el código del proceso padre mediante el condicional sobre el número del PID. Cuando el programa copia A ejecuta el mismo código que su hijo B, encuentra el condicional y entra al código escrito para su PID. Cuando el otro proceso B con el mismo código que A encuentra el código con el IF haciendo mención a su PID, entra a ejecutar ese código.

También sucede que PID dentro de la memoria del proceso, será 0 y para el padre será mayor que 0.

```c
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>  //tiene la función de exit para salir con el error


//Int es para poder enviarle un entero al sistema operativo
int main(void){

    pid_t PID;  //es como un tipo int PID,
    int i = 20;

    /*
     * resultado de fork() en PID y si ese es igual a uno, si eso
     * significa que se produjo un error. Esto es posible porque en minix no
     * puede tener más de 100 procesos ejecutandose.
     */
    if( (PID = fork() ) == -1 ){

        printf("Error creando al hijo\n");
        exit(1);

```

```
22      }
23      if( PID == 0 ){ //proceso hijo
24
25          printf("Soy el hijo. El PID de mi padre: %i\n", getppid());
26          sleep(10);
27          printf("Soy el hijo muriendo\n");
28
29      }else{
30
31          printf("Soy el padre. PID del Hijo: %i. El PID de mi padre: %i\n", PID,
32                  getppid());
33          sleep(15);
34          printf("Soy el padre muriendo\n");
35
36      }
37
38      return 0;
39
40  }
```

Se utiliza la función *getppid()* (get process PID) que se encuentra en la librería *unistd.h* para que cada uno de los procesos devuelva el PID de su proceso padre. Tener en cuenta que si esta función es invocada desde el proceso padre (en nuestro caso el A), devolverá el PID del proceso que está corriendo la consola.

Es posible que dos procesos generados a partir de fork() se comuniquen entre si, y es utilizando tuberías o *pipelines*.

Las funciones *sleep()* se utilizan para poder detener la ejecución del padre durante los segundos que se le indiquen como parámetro y poder visualizar su aparición en la tabla de proceso mediante el comando *top* de la siguiente forma:

$$\$top\ -U\ usuario$$

Existen dos formas para comunicar procesos:

1. **Por Memoria compartida**: Se utiliza Un espacio de memoria común entre procesos donde un proceso escribe y el otro lee. La ventaja de ello es que al utilizar memoria es más rápida en lo que respecta a escribir y leer. Lo complejo de este método es que para que ambos procesos compartan una parte de memoria en común, se requiere que exista sincronía entre ellos, de forma contraria se podrían producir colapsos. En sistemas multiusuarios como los que existían antaño, es imposible implementarlo, como tampoco en sistemas distribuidos. Tampoco se puede implementar en algunos microprocesadores de IBM los cuales al tener la característica de tener varios procesadores juntos conectados en forma de anillo, y donde cada procesador tiene una memoria caché exclusiva, se hace imposible que esa memoria dedicada a un procesador la comparta con otro procesador.

2. **Por pasaje de mensaje**: La característica que tiene es que es menos rápida ya que no comparte un espacio de memoria como en el anterior, pero al no verse obligado a implementar un sistema de sincronización, es mucho más fácil de implementar. Ejemplo de estos pasajes son: sockets, pipes, mensajes, buzones. Se suele utilizar esto por ser más simple.

2.3.1. Características de las comunicaciones entre procesos

- **Comunicación Directa/indirecta (en el SO)**: Cuando es directa cada proceso debe nombrar explícitamente al otro proceso con el que se quiere comunicar.
- **Comunicación indirecta**: asociado con lo que tiene que ver con buzones, NO se utiliza el PID del proceso con el que se comunica
- **Sincronica**: suelen ser bloqueantes porque es como hablar con el teléfono, el proceso no puede seguir haciendo otra cosa que no sea comunicarse y reservarse para eso.
- **Asincronica**: es que puede enviar el mensaje y luego pasar a hacer otra cosa. Correo postal o mensaje de whatsapp.
- **Envío y recepción Bloqueante**:

  El emisor espera hasta que el mensaje ha sido entregado

  El receptor espera hasta que le llegue el primer mensaje

- **Envío y recepción No Bloqueante**:

  El emisor no resulta bloqueado en ningún caso

  El receptor tampoco

  La función recibir devuelve error si no hay mensajes disponibles

- **Simetrica**: los dos procesos tienen que tener los PID para comunicarse. Nombra a quien escribe y el otro nombra de quien recibe. Se basa en saber si cada uno tiene el PID del otro.
- **Asimetrica**: solo un proceso que tiene que tner el PID de uno para escribirle, el otro no lo tiene. Uno lo sabe y el otro no.

2.3.2. Buzones

Los Buzones son una herramienta que provee el So para poder comunicarse entre procesos. Existen tres formas de realizarlos:

- **Uno a uno**: un proceso manda a un buzón y el otro lee
- **Muchos a uno**: más de un proceso que escriba y que uno solo lea desde el buzón.
- **Uno a muchos**: comunicación de broadcasting, como la antena de la radio emitiendo para todos.

¿De qué depende qué buzon implementar? Depende de la velocidad de los procesos a la hora de emitir los mensajes y cuánto tarden en recepcionarlo y leerlos, por eso es que algunas veces es necesario poner muchos emisores porque el receptor es lento o varios receptores porque los procesos son lentos al leer los mensajes enviados.

2.3.3. Características del canal

Según su Capacidad:

- **infinita**:, ninguno tiene infinito, toda comunicación tiene un máximo
- **cero**: que está cortada la comunicación. Que no existe un canal.
- **Limitada**: todas las comunicacionee son limintadas.

Según el Tipos de mensaje:

- **Largo fijo**:
- **Largo variable**: TCP, el que se usa en internet.

2.3.4. Tuberias o Pipelines

Para que dos procesos se comuniquen entre sí es necesario que lo hagan mediante lo que se denomina *tuberías o pipelines*. 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.

El siguiente es un ejemplo de cómo proceder en lenguaje C para poder establecer una tubería:

```c
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>  //tiene la función de exit para salir con el error
#include<string.h>  //se utiliza strlen, devuelve el largo de la cadena

//Devuelve un entero  para poder enviarle un entero al sistema operativo
int main(void){

    pid_t PID;
    int fd[2];  //se crea un vector con dos posiciones. Por acá se envia msj

    int nbytes;
    char cadena[] = "Hola mundo\n";
    char buffer[80];
    pipe(fd);

    if( (PID = fork() ) == -1 ){        //en caso de error

        printf("Error creando al hijo\n");
```

```
20          exit(1);
21
22      }
23      if( PID == 0 ){ //proceso hijo
24
25          close(fd[0]);
26          printf("Soy el hijo. El hijo enviando\n");
27          write(fd[1], cadena, strlen(cadena)+1);
28
29      }else{          //Proceso padre
30
31          close(fd[1]);
32          nbytes = read(fd[0], buffer, sizeof(buffer));
33          printf("El padre recibiendo\n");
34          printf("%d caracteres\n", nbytes);  //Se imprime la cantidad de bytes
35          printf("Mensaje: %s\n", buffer);    //Se muestra lo que hay dentro del buffer
36
37      }
38
39      return 0;
40
41  }
```

Se deben aclarar los siguientes puntos:

- **string.h**: se incluye la librería para el tratamiento de cadenas.
- **stdlib.h**: se incluye la librería para la función *exit* en caso de que haya algún error creando el proceso hijo.
- **fd[2]**: es un vector de dos elementos que sirve como file descriptor para la comunicación de procesos. Sería la tubería respectivamente.
- **buffer[80]**: es la variable que alojará lo recibido por el proceso A. Dentro de *buffer* se guardará lo que el proceso padre recibe, que en este caso es una cadena que dice "hola mundo".
- **pipe()**: es la función encargada de tomar el vector de enteros denominado *fd* y convertirlos en file descriptos para que se comuniquen los procesos. Según la página de manual de *pipe()* dice lo siguiente: *The pipe() function shall create a pipe and place two file descriptors, one each into the arguments fildes[0] and fildes[1], that refer to the open fileescriptions for the read and write ends of the pipe. Data can be written to the file descriptor fildes[1] and read from the file descriptor fildes[0]. A read on the file descriptor fildes[0] shall access data written to the file descriptor fildes[1] on a first-in-first-out basis.*
- **close()**: Siempre se debe invocar esta función tanto sea en el padre como en el hijo antes de utilizar cada *fd[]*. Se debe cerrar el *fd* que no se utilizará y utilizar el otro para la comunicación.
- **write()**:

$$ssize\_twrite(intfildes, constvoid * buf, size\_tnbyte);$$

  Según el manual, la función *write()* permite escribir *nbytes* de tamaño desde el buffer apuntado como *buf* al archivo asociado con el file descriptor abierto *fields*. La función devuelve un tamaño de bytes escritos.
- **read()**:

$$ssize_tread(intfildes, void * buf, size_tnbyte);$$

  Según el manual: la función *write()* intentará leer *nbytes* desde el archivo apuntado por el file descriptor abierto *fildes*, al buffer apuntado como *buf*. La función devuelve un tamaño de bytes leídos.
- **sizeof()**: Permite saber el tamaño en bytes de su argumento, sin importar que sea un dato primitivo o un dato creado por el usuario, como ser un struct.

Hay otras formas de hacer una tubería mediante consolas:
En la primer consola colocar:

1. *mknod tubo p*
2. *ls -l*
3. *cat < tubo*

En la segunda consola colocar:

1. *cat > tubo*

3. Clase nº 3 - Sincronización

3.1. Procesos Cooperativos

Definición: Son aquellos que puede afectar o verse afectado por otros procesos que estén ejecutándose en el sistema. Estos pueden compartir espacio de direcciones y comparten datos a través de mensajes.

Las razones por las que se implementan estos procesos

- **Compartir información**: varios usuarios pueden estar interesados en compartir información.
- **Acelerar cálculos**: atomizar tareas y ejecución en paralelo.
- **Modularidad**: Construcción del sistema en forma modular
- **Convivencia**: múltiples tareas o procesos ejecutándose al mismo tiempo.

3.2. Memoria Compartida

Los procesos intercambian información leyendo y escribiendo en lo que se denomina *zonas compartidas*. Estas zonas son más rápidas para el transpaso de mensajes ya que como se dijo en la clase anterior, al utilizar memoria, es mucho más rápido el acceso a esta, que otros medios.

La memoria compartida es aquel tipo de memoria que puede ser accedida por múltiples programas, ya sea para comunicarse entre ellos o para evitar copias redundantes. La memoria compartida es un modo eficaz de pasar datos entre aplicaciones. Dependiendo del contexto, los programas pueden ejecutarse en un mismo procesador o en procesadores separados. La memoria usada entre dos hilos de ejecución dentro de un mismo programa se conoce también como memoria compartida

3.3. Productor/Consumidor

El problema Productor/Consumidor consiste en el acceso concurrente por parte de procesos productores y procesos consumidores sobre un recurso común que resulta ser un buffer de elementos. Los productores tratan de introducir elementos en el buffer de uno en uno, y los consumidores tratan de extraer elementos de uno en uno.

Para asegurar la consistencia de la información almacenada en el buffer, el acceso de los productores y consumidores debe hacerse en exclusión mutua. Adicionalmente, el buffer es de capacidad limitada, de modo que el acceso por parte de un productor para introducir un elemento en el buffer lleno debe provocar la detención del proceso productor. Lo mismo sucede para un consumidor que intente extraer un elemento del buffer vacío.

Cuando se utiliza memoria compartida los procesos que utilicen la memoria compartida o también denominado *Zona Crítica o Sección Crítica* deben estar **sincronizados** para que el consumidor no intente consumir cuando el productor no escribió aún.

Buffer: Buffer es un vector de memoria que se organiza de forma consecutiva. Es por ello que se plantea el problema de ¿Qué sucede si se sigue produciendo datos cuando termina el vector? Los programas deben tener en cuenta esto por lo que se deben programar de manera circular para que apunte a la primera posición una vez que el buffer se encuentra desbordaro. Hay que tener en cuenta que cuando los dos punteros tanto del productor como del consumidor, apuntan a la misma dirección de memoria dentro del buffer, significa que está vacio.

Todos los problemas de consumidor/producor tiene una variable entera que se denomina *counter* que es la encargada de llevar el conteo de los elementos encerrados dentro del buffer. Por lo tanto:

- El proceso Productor lo que hace es counter++, es decir, aumentar en una unidad la variable contador.
- El proceso Consumidor lo que hace es counter–, es decir, disminuir en uno la variable contador.

Hay que tener en cuenta que tanto la variable que se utiliza de contador, como el buffer se encuentra en la Sección critica y tanto el proceso Productor como el Consumidor utilizarán ambas cosas. Es por ello que se debe asegurar que sólo uno de ellos pueda entrar por vez a la Sección Crítica. Esto se denomina *exclusión mutua*.

Un ejemplo de cómo proceden los procesos que hacen de Productor y Consumidor:

- **Productor**: El productor se le da la posibilida para poder utilizar la *Sección Crítica* por lo que accede al buffer y produce un dato. Acto seguido le suma a la variable contador 1 unidad y el proceso finaliza dejando libre la Sección Critica.
- **Consumidor**: El proceso que auspicia de Consumidor, al tener libre el acceso a la memoria, incresa y consume un dato del buffer y le resta a la varible contador una unidad. El proceso al finalizar esto, deja libre la Sección Crítica.

El código en Assembler sería algo así:
counter++ reg1 = counter reg1 = reg1 + 1 counter = reg1

Secuencia: En la primera línea lo que sucede es que se adelanta en una posición el puntero del contador. Cuando la variable *reg1* carga la variable counter se guarda en el registro de la memoria, luego se le suma el literal y se vuelve a guardar en el registro de memoria, con el nuevo valor *counter + 1.*

Esto lo realiza el primer proceso denominado Productor (ya que produce un dato). Como son dos procesos los involucrados en la Sección Crítica, entonces puede suceder que el Schedule no le asigne más tiempo en el procesador al proceso del Productor, pasándolo a un estado *En Espera* en el medio de la creación del dato y la actualización de las variables *counter* y *reg1*, y acto seguido le asigne el microprocesador al proceso del Consumidor.

El ejemplo de la ejecución sería el siguiente: reg1 = counter == reg1 = 3 reg1 = reg1 + 1 == reg1 = 4 reg2 = counter reg2 = reg2 - 1 == reg2 = 3 counter = reg2 == reg2 = 2 counter = 4 counter = reg1 == counter = 4

La posible solución a esto es que cuando el proceso Productor se ejecute, se haga para sí de la memoria compartida, produzca el dato, adelantar el puntero a la siguietne espacio de memoria. Todo ello lo debe hacer SIN que el proceso de consumidor se meta antes de que termine.

En la filmina se trata de ejemplificar esto con la idea de una autopista que cuenta la cantidad de coches que pasan por un peaje. ¿Qué sucede si pasan dos coches a la misma vez? El contador entraría en problemas ya que no podría divisar esto como dos autos.

3.4. Posibles Soluciones

La solución filosófica al problema del Productor/Consumidor la esgrimió Decker, pero fue Peterson quien volcó al código e implementó las ideas del primero.

3.4.1. Solución nº 1

La aproximación que plante Deker se explica mediante un iglú que funcionaría como una variable, y dos esquimales, que vendrían a ser dos procesos.

Sólo un proceso entra por la puerta del iglu, dentro ve en una pizarra, escrito el nº 1, sale del iglú, toma la lanza (es el recurso compartido), y va a cazar, vuelve y escribe el nº 2. El proceso nº 2 entra al iglu, ve el número 2 escrito en la pizarra, toma la lanza y sale a cazar, cuando vuelve pone el 1 y el proceso se repite una y otra vez.

- **Ventajas**: Se asegura la exclusión mutua, no hay ninguna manera de que los dos estén dentro en un mismo momennto.
- **Desventajas**: si el proceso nº 1 muere cazando un oso por ejemplo, no vuelve nunca, por lo que el otro no entra a la sección crítica nunca más. Los procesos están acompasados, lo que significa que si existe un proceso que utiliza mucho la sección crítica, y el otro poco tiempo, el problema es que al estar acompasados, el proceso que marcará el paso es el que lo utiliza menos tiempo, por lo que volverá la performance lenta.

3.4.2. Solución nº 2

Se plantea el mismo problema que el nº 1, donde existe otro iglu (una segunda variable variable) por lo que serían dos iglu. Los dos procesos entrarían en distintos inglus. Las pizarras se encuentran por defecto en *falso* (libres) ¿Cómo hacen los esquimales para entrar en la Sección Crítica?

El procedimiento es el siguiente: El proceso nº 1 entra al iglu nº 2 y ve falso, vuelve al suyo y pone verdadero, por lo que da a entender que se encuentra en la Sección Crítica. El proceso nº 2 va al iglu nº 1 y ve *verdadero* entonces lo que hace es esperar. El proceso nº 1 cuando termina pone falso y se repite esto para ambos procesos cuando quieran acceder a la Sección Crítica.

La problemática que conlleva esta solución:

- Si dos esquimales se cruzan y ponen verdadero las dos pizarras. Se complejiza el problema debido a que en este planteo existen dos iglu, a diferencia del anterior que existía sólo uno, al haber uno se asegura la exlusión mutua.
- El problema del planteo nº 1 es el tiempo que lleva cada proceso dentro de la Exclusión Mutua.

3.4.3. Solución nº 3

La tercer solución que se plantea también existen dos iglues como en la solución anterior. El Proceso nº 1 entra a su iglu, pone *veradero*, y a partir de ahí se dirige al otro iglu y ve si es *falso*, lo que significaría que está en la Sección Crítica.

- **El problema**: surge cuando ambos procesos quieran poner *verdadero* en sus respectivos iglues y salen a poner en la pizarra del otro proceso lo mismo; si ambos realizan lo mismo, entonces se convierte en un problema ya que nuevamente nos encontramos en una Exlusión Mutua
- **Interbloqueo**: Si la situación anterior se repite de manera continua, se produce un **interbloqueo**, porque cada uno vuelve a poner en su pizarra falso, luego los dos ponen verdadero, y así continuamente; la solución nº 3 tiene el problema del interbloqueo.

3.4.4. Solución nº 4

Esta solución es algo parecido con lo que hace el protocolo *TCP/IP*. Se implementa utilizando las características de la solución nº 3 pero con tiempo, es decir que en caso de que un proceso vea dos verdadso en ambos iglues, lo que hace es esperar un tiempo aleatorio para poder volver a chequear las pizarra, por lo que cada proceso en esta situación, podrá en algún momento entrar, ya que los tiempos que debe esperar son *aleatorios*. El problema que tiene acá es si los tiempos aleatorios se repiten, por lo que se vuelve al problema nº 3, pero esto sucedería con una baja probabilidad.

3.4.5. Solución nº 5

Se implementa lo planteado por la solución nº 4 pero agregando un nuevo iglú que oficiará de árbitro, donde estaría escrito el nº del proceso que está dentro de la Sección Crítica.

Pero es muy complicado de implementarlo para varios procesos.

Pregunta de examen: ¿Qué pasa si un proceso que está en la sección crítica muere? Esto es la problemática que atraviesa a todos los intentos, si un proceso está ocupando la memoria criticia y muere, entonces se vuelve problemático, ya que se estaría utilizando un recurso que nunca pasará a desbloquearse.

3.5. Solución por Hardware

Como las posibles soluciones anteriores eran difíciles de implementarlas vía software, se implementó una solución via hardware. Se pensó que haya en assembler una instrucción que permita bloquear el acceso a la Sección Critica.

3.5.1. TSL

TSL es una instrucción atómica en Assembler que permite crear un registro candado. Esto permite implementar una solución de Software amparada en hardware

Como se dijo anteriormente,TSL (Test and Set Lock) es una operación que se utiliza frecuentemente cuando se tiene que lidiar con problemas de índole general en Exclusión Mutua. Es una instrucción indivisible

- "candado": es una variable compartida.
- El valor 0: permite paso (cualquiera lo puede poner a 1)

En informática, la instrucción test-and-set es una instrucción utilizada para escribir 1 (set) en una ubicación de memoria y devolver su antiguo valor como una operación **atómica** única (es decir, no interrumpible). Si varios procesos pueden acceder a la misma ubicación de memoria, y si un proceso está realizando una instrucción test-and-set, ningún otro proceso puede comenzar otra instrucción de la misma índole hasta que finalice el test-and-set del primer proceso.

Se declara una variable candado y se comparte entre todos los procesos que quieren entrar a la sección crítica. La variable Candado es una variable como la que uno declara en C.

Cuando el Candado se encuentra en estado *abierto* el valor por defecto es 0, por lo que significa que la sección crítica está abierta para que se utilice la memoria. Un proceso pide el acceso a la Sección Crítica por lo que se fuerza el 1 al registro Candado. Cuando otro proceso quiera acceder, se evalúa el registro Candado y al encontrarse en 1 (ocupado) se espera a que la sección sea desocupada.

Esta variable se comparte con el restro de los procesos que interactúan en la misma sección de memoria, no se sabe si hay un proceso en la SC (Sección Critica), se guarda en un registro y se fuerza candado a 1, todo esto se hace de manera ATOMICA; Esto NO es lo mismo que las soluciones por software.

Lo que permite esto es que haya si o si un solo proceso que pueda cambiar la variable Candado a la vez, y por ende que entre a la Sección Crítica, garantizando de esta manera la Exlusión Mutua. Cuando se va de la SC se pone el candado a Cero (se hace con "MOV candado, 0")

Cabe destacar que la palabra atómica es INDISPENSABLE. Todo funciona porque el intercambio de valor de candados se hace de manera indivisible, atómica, en un ciclo de reloj. No se pude dividir. Con esta instrucción se asegura que el intercambio se produzca de manera indivisible, no interrumpible.

3.5.2. XCHG

XCHG (exchange data) es una instrucción en Assembler que permite el intercambio del contenido de dos operandos. Lo que genera es el intercambio a nivel atómico del valor registro con el valor de la variable Candado.

- *reg = 1*: por lo tanto se intercambiará con el valor *Candado = 0*.
- *reg = 0*: por lo tanto se intercambiará con el valor *Candado = 1*.
- *reg = 1*: por lo tanto se intercambiará con el valor *Candado = 0*. (Esto es lo mismo que hace TCL cuando registro tiene el valor 1)

3.6. Semáforos (Mutex)

Los Semáforos son soluciones que se implementan vía software. Se proporcionan herramientas (liberías) que permiten generar semáforos (estas soluciones, por debajo ejecutan instrucciones de tipo TSL).

```c
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>

int main(void){

    pid_t PID;
    int i = 20;
    int s; //variable compartida que es utilizada como semáforo

    if( (PID = fork() ) == -1 ){

        printf("Error creando al hijo\n");
        exit(1);

    }
    if( PID == 0 ){ //proceso hijo

        wait(s); //se bloquea para que ningún proceso más pueda ingresar
            printf("Soy el hijo. El PID de mi padre: %i\n", getppid());
            sleep(10);
            printf("Soy el hijo muriendo\n");
        signal(s); //habilita vía semáforo el ingreso de otro proceso a la región crítica

    }else{

        printf("Soy el padre. PID del Hijo: %i. El PID de mi padre: %i\n", PID, getppid());
    }

    return 0;
}
```

Se debe recordar que la variable ”s” implementada en el código anterior es de tipo entera y es compartida por los procesos como variable Candado.

Función de la función *wait()*:

- **wait(s)**: si s es ¿0 entonces s = s - 1. Decrementa el valor de s.
- **wait(s)**: si s es ¡0 entonces espera a que la variable retorne a un valor mayor a 0.
- **signal(s)**: lo que genera es s = s + 1, incrementa el valor de S.

Modo de operar:

Un proceso si quiere entrar a la sección de memoria compartida, hace wait(s) sin saber si hay otro proceso en la Sección Critica. La implementación dentro de la función *wait(s)*, es lo mismo que se hace en *TSL*. Cuando está en s = 1, está en verde, lo que hace es hacerle un s–. Cuando se termina de hacer lo que se tiene que hacer, se hace un signal(s) que lo que haría sería un s++.

3.7. Problema de la cena de los filósofos

Consiste en 5 filósofos y en el medio de la mesa hay un plato. Cada filósofo tiene un tenedor. Cada filósofo es un proceso y el tenedor es un recurso. Lo que hace un filósofo es o comer o filosofar. Solo pueden comer con 2 cubiertos. El problema se plantea cuando dos filósofos contiguos quieren comer (ya que tiene los cubiertos, recursos ocupados).

Pregunta de examen: ¿Cómo podría darse un interbloqueo en la cena de los filósofos? Se debe dibujar el esquema de la mesa reonda y explicar cómo se toman los recursos . Lo que se hace es programar todos los filósofos iguales, que pidan el recurso de la derecha y esperen el de la izquierda. Si hacen todos esto al mismo tiempo, se produce un deadlock.

3.8. Problema de la barrera

Otro problema que se plantea con procesos concurrentes.

En un punto de sincronización donde los 3 procesos tienen que llegar para poder seguir ejecutando del otro lado de la barrera. La barrera es un punto de sincronización donde todos los procesos a distintos tiempos deben llegar para poder seguir ejecutando del otro lado de la barrera.

La barrera se pude ver como un array lleno de 0 que cuando llga cada uno de los procesos se guarda el 1.

3.9. MicroKernel

Se parte del siguiente análisis: *Cada 1000 líneas de código se encuentran 10 errores.* Los caules tienen que ver en su gran mayoría con la seguridad del sistema.

Siguiendo lo anterior, En un sistema operativo con 5 millones de lineas se tienen 50 mil errores. Por lo que hay muchos problemas sobre todo de seguridad que hay que resolver.

Como hay tantos errores entonces se pensó lo siguiente: depurar el kernel, por lo que se toman las rutinas más importantes y son colocadas como bases del kernel para que sea lo más simple posible.

3.9.1. Kernel Monolítico

En un Kernel cuya organización es monolítica; debajo de todo se encuentra el hardware, por encima de ello los Servicios de Scheduling (proceso que se encarga de elegir qué proceso y por cuánto tiempo le corresponde correr en el micro procesador). La misma es una rutina de software del sistema operativo.

Por sobre el anterior, está el Memory Manager: administrador de la memoria que le indica qué cantidad de memoria le correspnde cada programa.

Por sobre ellos se encuentran los Servicios de I/O Manager el cual administra las entradas y salidas del sistema; y el FS (que es el que se encarga de que todos los archivos parezcan como que están organizadas)

Sistem Servicess: Ataja las solicitudes de las aplicaciones al sistema operativo. Por sobre ello se encuentra el modo usuario de los procesos de usuario.

Modo Kernel: Lo que no se puede tocar, lo que está ya compilado.

Los sistemas de tipo GNU/Linux se encuentran estructurados como un Kernel Monolítico.

3.9.2. MicroKernel

Por debajo se encuentra el hardware. Sobre ello una rutina que se denomina "microkernel" y por sobre ella el *modo usuario*. En el *espacio de usuario* es donde se encuentran las rutinas que antes pertenecían al modo kernel. Es por ello que gran parte del SO puede ser manipulado por el usuario.

La idea es tener la rutina de MicroKernel lo más chica y simple posible para que no hayan bugs de seguridad en contraposición al otro tipo de kernel (monolítico).

Ventajas:

- Manejador de IO por ejemplo, si se quiere cambiar un driver, se puede desinstalar e instalar otro como si fuera una aplicación de usuario. Se tiende a la seguridad por ser el MircroKernel ser lo más simple posible. Con esto se gana en modularidad y en seguridad.
- Podemos cambiar un servicio del SO, cambiando el proceso que lo implementa. Podemos ejecutar programas realizados para otros distinto.
- Un posible error de un servicio del SO queda confinado en el espacio de direcciones del proceso que lo implementa. Es extensible y personalizable.

Desventajas:

- Por otro lado, sus principales dificultades son la complejidad en la sincronización de todos los módulos que componen el micronúcleo y su acceso a la memoria, la anulación de las ventajas de Zero Copy, la Integración con las aplicaciones. Además, los procesadores y arquitecturas modernas de hardware están optimizadas para sistemas de núcleo que pueden mapear toda la memoria.

- Esto mejora la tolerancia a fallos y eleva la portabilidad entre plataformas de hardware, según los defensores de esta tendencia. Sus detractores le achacan, fundamentalmente, mayor complejidad en el código, menor rendimiento, o limitaciones en diversas funciones.

Pregunta de examen: ¿Cual es la motiviación para implementar SO de Micro kernel? ¿cuáles son las ventajas de ello?

4. Clase nº 5 (Repaso antes del parcial)

Ejercitación en C

Se hace un repaso de lo que entra en el parcial en la parte **práctica**. Lo que entra es sólo lo que se refiere a Lenguaje C y la comunicación entre procesos.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main (void){

    //File Descriptor necesario para generar la conexión entre procesos
    int fd[2];
    //Variable donde se almacenaran la cantidad de caracteres leidos
    int nbytes;
    //Process ID para saber cuándo ejecuta el hijo y cuándo el padre
    pid_t pid;      //utilizado para poder atajar el proceso hijo
    //Cadena que se pasará al proceso que quiera leer
    char cadena[] = "hola mundo\n";
    //Buffer donde se almacenará la cadena leída
    char buffer[80];

    //Necesario para poder generar el pipe por donde se comunicarán los procesos
    pipe(fd);

    //Se evalúa si existe un error al generar el proceso hijo.
    if( ( pid = fork() ) == -1 ){

        printf("Error creando hijo\n");
        exit(1);

    }

    //Proceso hijo el cual va a escribir la cadena
    if( pid == 0 ){

        printf("Soy el hijo\n");
        close(fd[0]); //Se debe cerrar el fd por el que NO se transmitirá nada
        //strlen es una función que permite obtener el largo de la cadena que se le pasa.
        write(fd[1], cadena, strlen(cadena)+1);
        sleep(1);

    }else{

        //Proceso padre, el cual va a recibir la cadena que le pasa el hijo
        printf("Soy el padre\n");
        close(fd[1]); //Se debe cerrar el fd por el que NO se leerá nada
        //sizeof() permite saber el tamaño en bytes de lo que se le pasa, en este caso
            devolverá 80
        //read() devuelve la cantidad de bytes leídos, por lo que se almacena en nbytes
        nbytes = read(fd[0], buffer, sizeof(buffer));
        printf("La cadena recibida fue: %s", buffer);
        printf("Se recibieron: %d\n", nbytes);
    }

    return 0;
}
```

Explicación de la función fork():

En la ejecución del proceso, cuando el control llega hasta la función *fork()* comienzan a existir dos procesos donde uno se denomina *padre* y el otro *hijo*. El Sistema operativo tiene asignado un espacio de memoria único para el proceso main que se está ejecutando. En el proceso padre declara una variable *pid* la cual, cuando el

control se topa con la función fork() el espacio de memoria se copia tal cual para el padre como para el hijo. Ambos códigos son iguales, por lo que el padre y el hijo utilizan las mismas variables; la única forma de reconocer el código del padre y del hijo es mediante la variable *pid* el cual, el del hijo será 0, y el padre un número mayor que cero. Como el código es el mismo en memoria es exactamente el mismo, el hijo entra cuando se consulta por cero y el del padre cuando se sale por el *else*, es decir, cuando es un número distinto de cero.

Ejercicio con conversión de cadena a número

Se agregará una complicación al proceso que recibe la cadena (quien la lee), obligandolo a realizarle algún tratamiento. En el siguiente ejemplo, el proceso hijo le pasa una cadena de números al padre y el padre deberá de almacenarlos como números.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main (void){

    int fd[2];
    int nbytes;
    pid_t pid;      //utilizado para poder atajar el proceso hijo
    char cadena[] = "135.23"; //numeros almacenados como cadena
    float numero;
    char buffer[80];

    pipe(fd);       //funcion que crea la conexión entre procesos

    //Si hubo un error al crear el hijo
    if( ( pid = fork() ) == -1 ){

        printf("Error creando hijo\n");
        exit(1);

    }
    //El HIJO será el que escribe
    if( pid == 0 ){

        printf("Soy el hijo\n");
        close(fd[0]);
        //strlen() devuelve el largo de la cadena +1 para el \0
        write(fd[1], cadena, strlen(cadena)+1);
        sleep(1);

    }else{

        //Proceso PADRE quien es el que recibe la cadena
        close(fd[1]);
        nbytes = read(fd[0], buffer, sizeof(buffer));
        printf("La cadena recibida fue: %s\n", buffer);
        printf("Se recibieron: %d\n", nbytes);

        for( int i = 0; i < nbytes; i++ ){

            if ( i == 3 ){
                printf("El valor en %d es: %c\n", i, buffer[i]);
            }else{
                buffer[i] = buffer[i] - 48;
                printf("El valor en %d es es: %d\n", i, buffer[i]);
            }
        }

        printf("El valor de nbytes es: %d\n", nbytes);

        //El error esta en que no se puede visualizar con el
        //formato float porque hay un caracter de punto en la posición
        //3
        numero += buffer[0] * 100;
        numero += buffer[1] * 10;
        numero += buffer[2];
        numero += buffer[4] * 0,1;
        numero += buffer[5] * 0,01;

        printf("El numero es %f\n", buffer);

```

```
64      }
65
66      return 0;
67  }
```

Ejercicio con conversión entre mayúsculas y minúsculas

También se puede pedir en un ejercicio que se realice algún tratamiento con la cadena que se recibe, como por ejemplo invertir las minúsculas a mayúsculas y viceversa. En el caso siguiente se sabe que en el código ASCII la diferencia entre la letra a y la A se encuentra a 32 caracteres de distancia. Por lo que si se quiere pasar de mayúscula a minúscula se debe restarle 32. Como dato se tiene que A mayúscula es 65, que la a minúscula es 97 y que la Z mayúscula es 122. Ejemplo en código C:

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main (void){

    int fd[2];
    int nbytes;
    pid_t pid;
    //Cadena con mayúsculas y minúsculas
    char cadena[] = "HoLa mUnDo\n";
    char buffer[80];

    pipe(fd);       //Función necesaria para crear la conexión

    //Si hubo un error al crear el hijo
    if( ( pid = fork() ) == -1 ){

        printf("Error creando hijo\n");
        exit(1);

    }

    if( pid == 0 ){

        //El hijo será quien envie el mensaje
        printf("Soy el hijo\n");
        close(fd[0]);
        write(fd[1], cadena, strlen(cadena)+1);
        sleep(1);

    }else{

        //Será el proceso padre quien reciba el mensaje
        printf("Soy el padre\n");
        close(fd[1]);
        nbytes = read(fd[0], buffer, sizeof(buffer));
        printf("La cadena recibida fue: %s", buffer);
        printf("Se recibieron: %d\n", nbytes);

        int i = 0;

        //Pasamos a mayúscula
        while( cadena[i] != '\n' ){

            //Se pregunta si la letra se encuentra entre las mayúsculas.
            //Caso efectivo se le resta 32 para poder pasarlas a minúscula.
            if( (cadena[i] >= 97) && (cadena[i] <= 122) ){
                cadena[i] = cadena[i] - 32;
            }
            i++;

        }

        printf("La cadena convertida es: %s", cadena);

    }

    return 0;
}
```

4.1. Algoritmo del Banquero

Para comenzar con éste algoritmo, lo que se tiene que tener en cuenta es que se cuenta con 3 matrices:

- **Matriz de Asignación**: En esta matriz los recursos se representan con las letras en la parte superior y los procesos en la columna primera. Esta matriz informa los recursos asignados a determinados procesos en un momento dado.
- **Matriz de Requerimientos**: Es la matriz que nos informa qué recursos requieren determinados procesos para finalizar su ejecución.
- **Matriz de Disponibilidades**: Indica cuántos recursos disponibles hay para otorgar a los procesos en ejecución de la *matriz de asignación*.

Ejercicio

Se cuenta con la siguiente distribución:
Matriz de Asignación:

-	A	B	C
P0	2	0	1
P1	1	1	1
P2	0	0	1
P3	0	0	1
Total	3	1	4

Matriz de Requerimientos:

-	A	B	C
P0	0	0	1
P1	1	1	0
P2	1	0	0
P3	2	0	1

Matriz de Disponibles:

-	A	B	C
-	1	0	0
Total	1	0	0

Hay que tener en consideración que, para obtener determinadas matrices se puede hacer:

$$A = 3 \text{ (Suma de Asignación más Disponibles)}$$
$$B = 1 \text{ (Suma de Asignación más Disponibles)}$$
$$C = 4 \text{ (Suma de Asignación más Disponibles)}$$

El paso a seguir es analizar la matriz de Asignación contra la de Requerimientos para saber qué recurso otorgar para satisfacer a cada uno de los procesos solicitantes.

- **Proceso nº 0**: Se puede ver en la matriz de *Requerimientos* que el proceso solicita una instancia de C, que en la matriz de *Disponibles* no se encuentra apto para poder otorgar. Por lo tanto se pasa al siguiente proceso.
- **Proceso nº 1**: No se puede satisfacer la solicitud de la instancia de B porque en la matriz de *Disponibles* no hay ninguna instancia disponible. Se pasa al siguiente proceso.
- **Proceso nº 2**: Se puede satisfacer la solicitud del proceso nº 2 ya que requiere sólo una instancia de A que se encuentra disponible para ser otorgada según la matriz de *Disponibles*. Se procede con este proceso.
- **Proceso nº 3**: No se puede satisfacer la solicitud de la instancia de C, ya que no se encuentra disponible en la matriz de *Disponibles*.

Primera iteración

Matriz de Asignación (Iteración nº 1) :

-	A	B	C
P0	2	0	1
P1	1	1	1
P2	1	0	1
P3	0	0	1
Total	3	1	4

Matriz de Requerimientos (Iteración nº 1):

-	A	B	C
P0	0	0	1
P1	1	1	0
P2	0	0	0
P3	2	0	1

Matriz de Disponibles (Iteración nº 1):

-	A	B	C
-	0	0	0
Total	0	0	0

Al finalizar la primera iteración se obtiene que el Proceso nº 2 pudo finalizar su proceso, por lo que vuelven a estar las instancias disponibles para otros procesos. El proceso nº 2 desaparece de las matrices.

Matriz de Asignación (Iteración nº 1) :

-	A	B	C
P0	2	0	1
P1	1	1	1
P3	0	0	1
Total	3	1	4

Matriz de Requerimientos (Iteración nº 1):

-	A	B	C
P0	0	0	1
P1	1	1	0
P3	2	0	1

Matriz de Disponibles (Iteración nº 1):

-	A	B	C
-	1	0	1
Total	1	0	1

Segunda iteración

Se puede observar que el siguiente proceso que puede ser satifecho mirando tanto la matriz de *Requerimientos* como la de *Disponibles* es el **Proceso nº 0**. Por lo que se procede con ese proceso:

Matriz de Asignación (Iteración nº 2) :

-	A	B	C
P0	2	0	2
P1	1	1	1
P3	0	0	1
Total	3	1	4

Matriz de Requerimientos (Iteración nº 2):

-	A	B	C
P0	0	0	0
P1	1	1	0
P3	2	0	1

Matriz de Disponibles (Iteración nº 2):

-	A	B	C
-	1	0	0
Total	1	0	0

Luego de la segunda iteración las matrices quedan de la siguiente forma:
Matriz de Asignación (Iteración nº 2) :

-	A	B	C
P1	1	1	1
P3	0	0	1
Total	1	1	2

Matriz de Requerimientos (Iteración nº 2):

-	A	B	C
P1	1	1	0
P3	2	0	1

Matriz de Disponibles (Iteración nº 2):

-	A	B	C
-	3	0	2
Total	3	0	2

Tercer iteración

Se puede observar que el siguiente proceso que puede ser satifecho mirando tanto la matriz de *Requerimientos* como la de *Disponibles* es el **Proceso nº 3**. Por lo que se procede con ese proceso:
Matriz de Asignación (Iteración nº 3) :

-	A	B	C
P1	1	1	1
P3	2	0	2
Total	3	1	3

Matriz de Requerimientos (Iteración nº 3):

-	A	B	C
P1	1	1	0
P3	0	0	0

Matriz de Disponibles (Iteración nº 3):

-	A	B	C
-	1	0	1
Total	3	0	2

Luego de finalizado la ejecución del proceso nº 3, las matrices quedan de la siguiente manera:
Matriz de Asignación (Iteración nº 3) :

-	A	B	C
P1	1	1	1
Total	1	1	1

Matriz de Requerimientos (Iteración nº 3):

-	A	B	C
P1	1	1	0

Matriz de Disponibles (Iteración nº 3):

-	A	B	C
-	3	0	3
Total	3	0	3

Como se puede ver, en esta instancia del ejercicio, nos encontramos en un **Interbloqueo**, lo que significa que no se puede satisfacer ningún requisito solicitado por los procesos, dado que no se encuentra en la tabla de *Disponibles* para las solicitudes operadas. Este es el caso del **Proceso nº 1** que solicita una instancia más de B, pero no existe esa instancia para podersela otorgar. Finaliza el ejercicio indicando que se llegó a una instancia de **Interbloqueo**, por lo cual no puede seguir continuando.

4.2. Algoritmo de búsqueda para interbloqueos

Se declara una variable L como una lista que contiene tanto procesos como recursos en una computadora.

Dibujo en la carpeta
L = [B]
L = [B,T]
L = [B,T,E]
L = [B,T,E,V]
L = [B,T,E,V,G]
L = [B,T,E,V,G,T]

Este algoritmo trata de encontrar los posibles interbloqueos en la tabla de procesos operante en el Sistema Operativo. Lo que hace es mantener una lista de todos los procesos y solicitudes a instancias de recursos, y recorre esta lista una y otra vez para encontrar si hay recursos o procesos repetidos, en caso que se encuentre alguna, se reconoce como una *espera circular*. La comparación que realiza es del orden de todos contra todos.

El problema que produce este tipo de algoritmos es que:

- Es enorme la cantidad de procesamiento que consume para realizar cada comparación recursivamente.
- En un Sistema Operativo normal, se encuentran cientos de procesos con ciento de recursos otorgados y disponibles, por lo que la lista se vuelve inmensa, ya que el alogoritmo debe de tener en cuenta todos los posibles caminos por los que un recurso puede ser poseído por un proceso. Por cada bifurcación, pueden realizar varios caminos distintos por lo que hay que también tenerlos en cuenta para realizar las comparaciones.
- El algoritmo como está planteado, debe de ser recursivo, por lo que se vuelve más pesado aún, generando un uso enorme de la memoria.

También cabe resaltar que este tipo de algoritmos es imposible de implementar por el tiempo de cómputo que consume.