Sistemas Operativos

Procesos concurrentes

Exclusión mutua y sincronización

Dr. Wenceslao Palma M.
<wenceslao.palma@ucv.cl>

Introducción

La concurrencia es importante en el diseño de sistemas operativos.

Se manifiesta en aspectos tales como la comunicación entre procesos, acceso a los recursos, sincronización de ejecución.

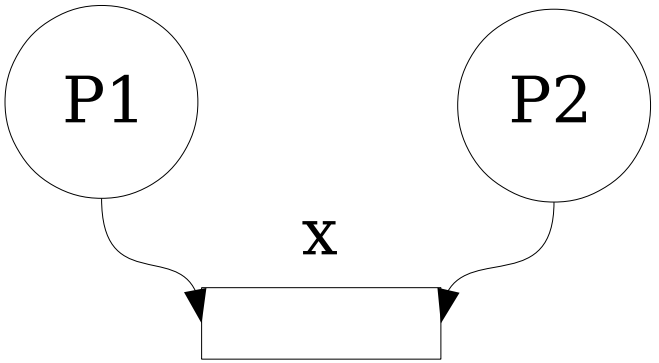
En el capítulo anterior vimos que la intercalación de procesos (ejecución concurrente) proporciona beneficios importantes. Ahora el problema esta cuando estos procesos acceden a un mismo recurso.

En términos generales, es importante controlar la concurrencia cuando :
dos ó más procesos concurrentes acceden a un mismo recurso y al menos uno de ellos realiza un acceso de escritura.

Consideremos el caso de 2 procesos accediendo a un segmento de memoria compartida.

```
x=3;
sleep();
x=x+1;
printf(“%d”,x);
```

el resultado
esperado es 4




```
x=x+1;
sleep();
printf(“%d”,x);
```

el resultado
esperado es 4

Si se origina la siguiente ejecución:

```
P1 x=3;
P1 sleep();
P2 x=x+1;
P2 sleep();
P1 x=x+1;
P1 printf(“%d”, x);        // la salida es 5!!
```

IMPORTANTE: es necesario proteger el acceso a un recurso compartido. La forma de acceso (r/w) es importante ya que puede afectar la semántica de los procesos.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHARED_SIZE 1
main(){
        int c;
        int shmid;
        key_t key;
        char *shmat();
        int *shm, *s;

        key = 123;
        if ((shmid=shmget(key,SHARED_SIZE, IPC_CREAT|
0666))        <0){
                perror("shmget");
                exit(1);
                }

        if ((shm=shmat(shmid,NULL,0))==(int *) -1){
                perror("shmat");
                exit(1);
        }

        s=shm;
        *s=3;
        printf("P1:[%d]\n",*s);
        sleep(2);
        *s=*s+1;
        printf("P1:[%d]\n",*s);
}
```

código fuente P1

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHARED_SIZE 1
main(){
        int c;
        int shmid;
        key_t key;
        char *shmat();
        int *shm, *s;

        key = 123;
        if ((shmid=shmget(key,SHARED_SIZE,0666))<0){
                perror("shmget");
                exit(1);
        }

        if ((shm=shmat(shmid,NULL,0))==(int *) -1){
                perror("shmat");
                exit(1);
        }


        s=shm;
        *s=*s+1;
        printf("P2:[%d]\n",*s);
        sleep(2);
        printf("P2:[%d]\n",*s);
}
```

código fuente P2

Interacción entre procesos

El modo en el cual interactúan los procesos se puede clasificar en función del nivel de conocimiento que cada uno tiene de la existencia de los demás.

Grado de Conocimiento	***Relación***	***Influencia de un proceso en los otros***	***Posibles problemas de control***
no hay conocimiento	competencia	resultados de un proceso son independientes de las acciones de los otros. los tiempos de los procesos pueden verse afectados.	exclusión mutua deadlock inanición
conocimiento indirecto (variables compartidas)	cooperación por compartimiento	resultados de uno pueden depender de la información obtenida de otro. tiempos de los procesos pueden verse afectados	exclusion mutua deadlock inanición coherencia de los datos
conocimiento directo(primitivas de comunicación	cooperación por comunicación	resultados de uno pueden depender de la información obtenida de otro. tiempos de los procesos pueden verse afectados	deadlock inanición

Competencia por los recursos

Dos o más procesos necesitan acceder a un recurso durante su ejecución. Ningún proceso conoce la existencia del otro y no se ven afectados por su ejecución.

Cada proceso debe dejar el recurso tal cual como esté el estado del recurso utilizado.

No hay intercambio de información entre los procesos, pero la ejecución de un proceso puede influir en el comportamiento de los procesos que compiten (espera e inanición).

Cuando los procesos compiten, se deben solucionar 3 problemas: exclusión mutua, deadlock e inanición.

Exclusión mutua: los procesos que acceden al recurso deben estar compuestos de una sección crítica la cual garantice que el proceso que ingrese en ella sea el único que puede acceder al recurso. Hacer que se cumpla la exclusión mutua puede provocar deadlock e inanición.

Deadlock: P1 y P2 necesitan acceder a R1 y R2. Ambos procesos necesitan acceder tanto a R1 como a R2 para cumplir con su trabajo. Además, si obtienen el acceso a un recurso no lo liberarán hasta que obtengan el acceso al siguiente recurso. En este caso P1-->P2 y P2-->P1.

Inanición: un proceso puede estar indefinidamente esperando por un recurso, lo cual no es deseable ya que no se garantiza el término de los procesos.

```
void P(int i){
        while (true){
                entrada_crítica(i);
                // accediendo al recurso.....
                salida_crítica(i);
                .......
        }
}

void main(){
        parbegin(P(R1),P(R2), P(R1), ....., P(Rn));
}
```

Cooperación por compartimiento

Los procesos interactúan sin tener un conocimiento explícito de la existencia del otro.

La interacción puede ser a través de variables compartidas, archivos, etc.

Si bien, no hay un conocimiento explícito de los otros procesos, se sabe que pueden acceder a los mismos datos.

Los datos se almacenan en recursos por lo tanto es necesario resolver la exclusión mutua y los problemas de deadlock e inanición.

Aquí interesa si el acceso es de lectura o escritura. Lo que agrega la noción de consistencia de los datos.

Luego, también es importante que los procesos esten “equipados” con una sección crítica.

Cooperación por comunicación

Cuando se coopera por comunicación, los procesos participan en una labor común que los une.

En este caso la comunicación es una manera de sincronizar o coordinar sus distintas actividades.

Las primitivas para el envío y recepción de los mensajes pueden venir como parte del lenguaje o por el kernel del sistema operativo. Por ejemplo: sendto() y recvfrom().

```
ssize_t  sendto(int  s,  const  void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);

 ssize_t  recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
```

En este caso no es necesario que los procesos tengan una seccion critica, sin embargo deadlock e inanición pueden estar presentes.

Requisitos para la exclusión mutua

Todo servicio que de soporte a la exclusion mutua debe cumplir los siguientes requisitos:

Solo un proceso debe tener permiso, en un momento dado, para ingresar en la seccion crítica

Un proceso que se interrumpe en una seccion no crítica debe hacerlo sin interferir con los otros procesos.

No puede existir deadlock o inanición.

Cuando ningún proceso se encuentre en la seccion crítica, cualquier proceso que solicite ingresar en ella debe hacerlo sin retraso alguno.

No se debe hacer suposiciones sobre la velocidad relativa de los procesos o el número de procesadores.

Un proceso permanece en la sección crítica por un tiempo finito

Soluciones por software

Una solución por software supone que los accesos a una misma posición de memoria se realizan en serie.

Algoritmo de Dekker

resuelve el problema de la exclusión mutua.

el programa es complejo, difícil de seguir y su corrección es difícil de mostrar.

Sin embargo Dijkstra desarrolló una serie de errores habituales que se producen en la construcción de programas concurrentes. Esto sirve para introducir el algoritmo de Dekker.

Primer Intento

Proceso 0

```
.....
while (turno!=0)
      // hacer nada
// sección crítica
turno = 1;
.......
```

Proceso 1

```
.....
while (turno!=1)
      // hacer nada
// sección crítica
turno = 0;
.......
```

turno es una posición de memoria compartida para ambos procesos.

Cuando el valor de **turno** es igual al número del proceso que la examina, este puede ingresar en su sección crítica.

Cuando un proceso no puede acceder a su sección crítica debe esperar y continuamente testear el valor de **turno**. La espera se denomina **espera activa** (busy waiting), es decir, consume su tiempo de procesador esperando por su oportunidad.

Si bien este intento garantiza la exclusión mutua, existen dos problemas:

- La alternancia en el uso de la sección crítica es estricta. Luego, el ritmo de ejecución viene dado por el proceso más “lento”.

- Si un proceso falla, el otro se bloquea en forma permanente.

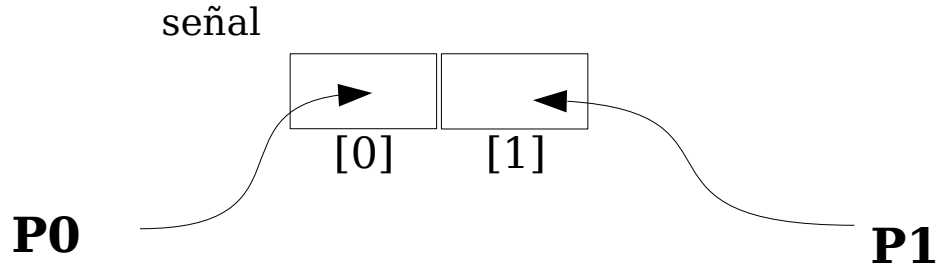
Cuando la decisión de quién toma el control es asumida por los procesos estaremos en presencia de **corrutinas**. Técnica que no resulta apropiada para dar soporte a ejecuciones concurrentes.

Segundo Intento

```
Proceso 0
......
while (señal[1])
      // hacer nada
señal[0]=cierto;
// sección crítica
señal[0]=falso;
.......
```

```
Proceso 1
......
while (señal[0])
      // hacer nada
señal[1]=cierto;
// sección crítica
señal[1]=falso;
.......
```

En este caso se incorpora más información para tratar de resolver el problema del bloqueo permanente cuando uno de ellos falla.



Cada proceso puede examinar la señal del otro pero no modificarla.

Si un proceso falla; antes de poner en cierto su señal o bien luego de ponerla en falso, el otro proceso no se bloquea en forma permanente.

Pero si un proceso falla luego de poner su señal en cierto o bien estando en su sección crítica, el otro proceso se bloquea en forma permanente.

Por otro lado, consideremos la siguiente secuencia:

P0 ejecuta while y encuentra que señal[1] es falso
P1 ejecuta while y encuentra que señal[0] es falso
P0 setea señal[0] a cierto e ingresa en su sección crítica
P1 setea señal[1] a cierto e ingresa en su sección crítica

Ambos procesos están en sus secciones críticas !!!

Tercer intento

Proceso 0

```
.....
señal[0]=cierto;
while (señal[1])
      // hacer nada
// sección crítica
señal[0]=falso;
.......
```

Proceso 1

```
.....
señal[1]=cierto;
while (señal[0])
      // hacer nada
// sección crítica
señal[1]=falso;
.......
```

En el caso anterior el problema está en que un proceso puede cambiar su estado después que el otro lo comprobó pero antes de entrar en su sección crítica.

En este caso se ubica señal[{0,1}]=cierto antes de while.

Pero, si un proceso falla dentro de su sección crítica el otro proceso se bloquea.

Esta solución garantiza exclusión mutua pero es propensa al deadlock (Cada uno piensa que el otro ha ingresado en su sección crítica).

Cuarto intento

Proceso 0

```
.....
señal[0]=cierto;
while (señal[1])
{
      señal[0]=falso;
      // esperar
      señal[0]=cierto;
}
// sección crítica
señal[0]=falso;
.......
```

Proceso 1

```
.....
señal[1]=cierto;
while (señal[0])
{
      señal[1]=falso;
      // esperar
      señal[1]=cierto;
}
// sección crítica
señal[1]=falso;
.......
```

Para evitar el deadlock los procesos activan su señal para indicar que desean acceder a su sección crítica, pero aademás estan listos para desactivarla y ceder la oportunidad al otro proceso.

La exclusión mutua esta garantizada pero aparece el livelock.

Consideremos la siguiente secuencia:

P0 setea señal[0] a cierto
P1 setea señal[1] a cierto
P0 comprueba señal[1]
P1 comprueba señal[0]
P0 setea señal[0] a falso
P1 setea señal[1] a falso
P0 setea señal[0] a cierto
P1 setea señal[1] a cierto

la cual se puede prolongar indefinidamente y ningún proceso ingresa en su sección crítica.

Una solución correcta: El Algoritmo de Dekker

```
void P0(){
      while (cierto){
            señal[0] = cierto;
            while (señal[1])
                  if (turno==1){
                        señal[0]=falso;
                        while (turno==1)
                              // hacer nada
                        señal[0] = cierto;
                  }
            // sección crítica
            turno = 1;
            señal[0] = falso;
            // resto del proceso
      }
}
```

```
void P1(){
      while (cierto){
            señal[1] = cierto;
            while (señal[0])
                  if (turno==0){
                        señal[1]=falso;
                        while (turno==0)
                              // hacer nada
                        señal[1] = cierto;
                  }
            // sección crítica
            turno = 0;
            señal[1] = falso;
            // resto del proceso
      }
}
```

```
void main(){
      señal[0] = falso;
      señal[1] = falso;
      turno = 1;
      parbegin(P0,P1);
}
```

```
boolean señal[2];
int turno;
```

El estado de ambos procesos se observa mediante el array señal.

Dado que el array y la “cortesía mutua” nos es suficiente, se utiliza la variable **turno** para indicar qué proceso tiene prioridad para exigir la entrada a la sección crítica.

Cuando P0 quiere ingresar a su sección crítica setea su señal a cierto y comprueba el estado de P1.

Si P1 no se encuentra en su sección crítica, P0 ingresa inmediatamente. En otro caso P0 consulta la variable turno. Si turno es 0, insiste y comprueba periódicamente el estado de P1.

En algún momento P1, cambia su estado (abandona su sección crítica) y P0 podrá ingresar.

Luego que P0 ejecuta se sección crítica, cambia su estado y cambia el valor de la variable turno para que P1 pueda insistir.

El algoritmo de Peterson

Desarrolló una solución más simple y elegante.

Al igual que en el caso anterior el array señal indica el estado de cada proceso respecto de la exclusión mutua y la variable turno resuelve los conflictos de simultaneidad.

```
void P0(){
      while (cierto){
            señal[0] = cierto;
            turno = 1;
            while (señal[1] && turno==1)
                  // hacer nada
            // sección crítica
            señal[0] = falso;
            // resto del proceso
      }
}
```

```
void P1(){
      while (cierto){
            señal[1] = cierto;
            turno = 0;
            while (señal[0] && turno==0)
                  // hacer nada
            // sección crítica
            señal[1] = falso;
            // resto del proceso
      }
}
```

```
void main(){
      señal[0] = falso;
      señal[1] = falso;
      parbegin(P0,P1);
}
```

```
boolean señal[2];
int turno;
```

Semáforos

Mecanismo utilizado en Sos y lenguajes de programación para dar soporte a la concurrencia.

Idea: Dos o más procesos pueden cooperar por medio de simples señales. Un proceso se puede deterner a la espera de la recepción de una señal específica. Cualquier requisito complicado de coordinación puede ser satisfecho por el uso de señales en forma adecuada.

Para la señalización se usan variables especiales llamadas **semáforos**

Para transmitir una señal por el semáforo s, los procesos ejecutan la primitiva signal(s).

Para recibir una señal del semáforo s, los procesos ejecutan la primitiva wait(s). Si la señal correspondiente aún no es transmitida el proceso se suspende.

Sobre los semáforos se definen 3 operaciones:

1.- Un semáforo puede inicializarse con un valor no negativo.

2.- La operación wait disminuye el valor del semáforo. Si el valor se hace negativo, el proceso que ejecuta wait se bloquea.

3.- La operación signal incrementa el valor del semáforo. Si el valor es negativo, se desbloquea un proceso bloqueado por la operación wait.

Aparte de estas operaciones, no hay forma de examinar o manipular los semáforos.

Las primitivas wait y signal son atómicas.

```
void wait(semaforo s){
        s.contador--;
        if (s.contador < 0){
                poner este proceso en s.cola:
                bloquear este proceso;
        }
}
```

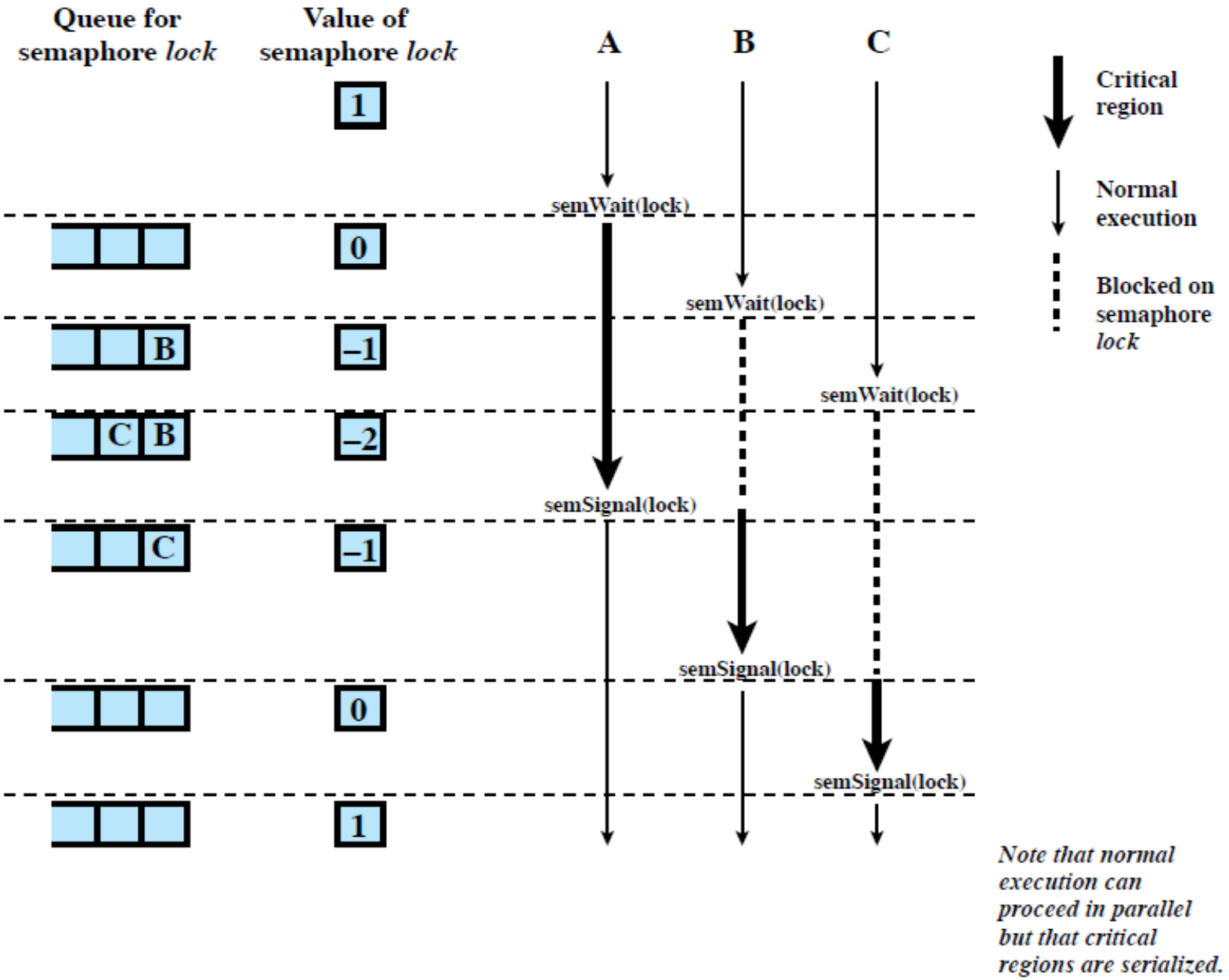
```
void signal(semaforo s){
        s.contador++;
        if (s.contador <= 0){
                quitar un proceso P de  s.cola:
                poner el proceso P en la cola de listos;
        }
}
```

```
struct semaforo{
        int contador;
        tipoCola cola;
}
```

Exclusión mutua por medio de semáforos

```
const int n = // número de procesos
semaforo s = 1;
void P(int i){
	while (cierto){
		wait(s);
		// sección crítica
		signal(s);
		// resto del proceso
	}
}
void main(){
	parbegin(P(1), P(2),....., P(n));
}
```

Procesos accediendo a datos compartidos protegidos por un semáforo.



Semáforos Binarios

Un semáforo binario solo puede tomar los valores 0 y 1.

Son más sencillos de implementar y tienen la misma potencia de expresión que los semáforos generales.

```
void waitB(semaforo_binario  s){
        if (s.valor == 1)
                s.valor=0;
        else{
                poner este proceso en s.cola;
                bloquear este proceso;
        }
}
```

```
void signalB(semaforo_binario  s){
        if (s.cola.esvacia())
                s.valor = 1;
        else{
                quitar un proceso P de  s.cola:
                poner el proceso P en la cola de listos;
        }
}
```

```
        struct semaforo_binario{
                enum (cero,uno) valor;
                tipoCola cola;
        }
```

En general:

Los semáforos generales y binarios utilizan una cola para almacenar los procesos en espera.

Generalmente la política para retirar los procesos es FIFO, debido a su equidad.

Cuando un semáforo incluye esta política se denomina **semáforo robusto**.

El problema del productor/consumidor

Uno o más productores generan cierto tipo de recursos (por simplicidad caracteres) y los almacenan en un buffer.

Un único consumidor saca, uno a uno, elementos del buffer.

Sólo un proceso (productor o consumidor) puede acceder al buffer en un determinado instante.

Consideremos un buffer limitado y las sgtes funciones para el productor y el consumidor:

productor

```
while (cierto){
        // producir v
        b[ent] = v;
        ent++;
}
```

consumidor

```
while (cierto){
        while(ent<=sal)
                // hacer nada
        v = b[sal];
        sal++;
        // consumir v
}
```

Una “solución” mediante semáforos binarios

Se utilizarán dos semáforos **s** y **retraso**:

s: para cumplir con la exclusión mutua.

retraso: para obligar al consumidor a esperar si el buffer está vacío.

```
void productor(){
        while (cierto){
                producir();
                waitB(s);
                añadir();
                n++;
                if (n==1)
                        signalB(retraso);
                signalB(s);
        }
}
```

```
void consumidor(){
        waitB(retraso);
        while (cierto){
                waitB(s);
                sacar();
                n--;
                signalB(s);
                consumir();
                if (n==0)
                        waitB(retraso);
        }
}
```

```
int n;
semaforo_binario s=1;
semaforo_binario retraso=0;
void main(){
        n=0;
        parbegin(productor, consumidor);
}
```

El productor inserta elementos al buffer en cualquier momento realizando waitB(s) y signalB(s) para garantizar exclusión mutua en el acceso al buffer.

Estando en la sección crítica el consumidor incrementa el valor de n. Si luego de esto n ==1, quiere decir que el buffer estaba vacio y es necesario comunicar al consumidor que ahora el buffer tiene un elemento (signalB(retraso))

Sin embargo, la solución propuesta tiene un problema: el consumidor puede consumir un elemento que no existe.

Lo anterior se produce cuando n indica que no hay elementos pero el semáforo no es consistente con ello.

	Producer	Consumer	s	n	Delay
1			1	0	0
2	waitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (signalB(delay))		0	1	1
5	signalB(s)		1	1	1
6		waitB(delay)	1	1	0
7		waitB(s)	0	1	0
8		n--	0	0	0
9		SignalB(s)	1	0	0
10	waitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (signalB(delay))		0	1	1
13	signalB(s)		1	1	1
14		**if** (n==0) (waitB(delay))	1	1	1
15		waitB(s)	0	1	1
16		n--	0	0	1
17		signalB(s)	1	0	1
18		**if** (n==0) (waitB(delay))	1	0	0
19		waitB(s)	0	0	0
20		n--	0	–1	0
21		signalB(s)	1	–1	0

Solución con semáforos binarios

```
void productor(){
        while (cierto){
                producir();
                waitB(s);
                añadir();
                n++;
                if (n==1)
                        signalB(retraso);
                signalB(s);
        }
}
```

```
void consumidor(){
    int m;
        waitB(retraso);
        while (cierto){
                waitB(s);
                sacar();
                n--;
                m = n;
                signalB(s);
                consumir();
                if (m==0)
                        waitB(retraso);
        }
}
```

```
int n;
semaforo_binario s=1;
semaforo_binario retraso=0;
void main(){
        n=0;
        parbegin(productor, consumidor);
}
```

Solución con semáforos generales

Es bastante más elegante considerando a n como un semáforo.

Lo anterior tiene sentido ya que ahora el semáforo n representa la cantidad de elementos del buffer.

```
void productor(){
        while (cierto){
                producir();
                wait(s);
                añadir();
                signal(s);
                signal(n);
        }
}
```

```
void consumidor(){
        while (cierto){
                wait(n);
                wait(s);
                sacar();
                signal(s);
                consumir();
        }
}
```

```
semaforo  s=1;
semaforo  n=0;
void main(){
        parbegin(productor, consumidor);
}
```

Qué sucede si se invierten wait(n) y wait(s) en el consumidor?

Monitores

Se ha mostrado la efectividad de los semáforos en cuanto exclusión mutua y coordinación. Sin embargo, la construcción de un programa utilizando semáforos no es sencilla.

Resulta complicado advertir el efecto de las operaciones wait y signal. Ademas, estas operaciones se deben emplear a lo largo de todo el programa.

Los monitores son estructuras a nivel de lenguaje de programación, son funcionalmente equivalentes a los semáforos y son mas fáciles de controlar.

El concepto de monitor fue planteado por HOAR en el artículo "Monitors: An Operating System Structuring Concept", Communication of The ACM, Octubre 1974.

El lenguaje Java implementa este concepto mediante synchronized, wait y notify (http://java.sun.com/docs/books/tutorial/essential/threads/)

Monitores con señales

Como modulo de software un monitor consta de uno o mas procedimientos, una secuencia de inicio y datos locales.

Las caracteristicas de un monitor son las siguientes:

Las variables locales son accesibles unicamente para los procedimientos del monitor.

Un proceso entra en el monitor invocando a uno de sus procedimientos.

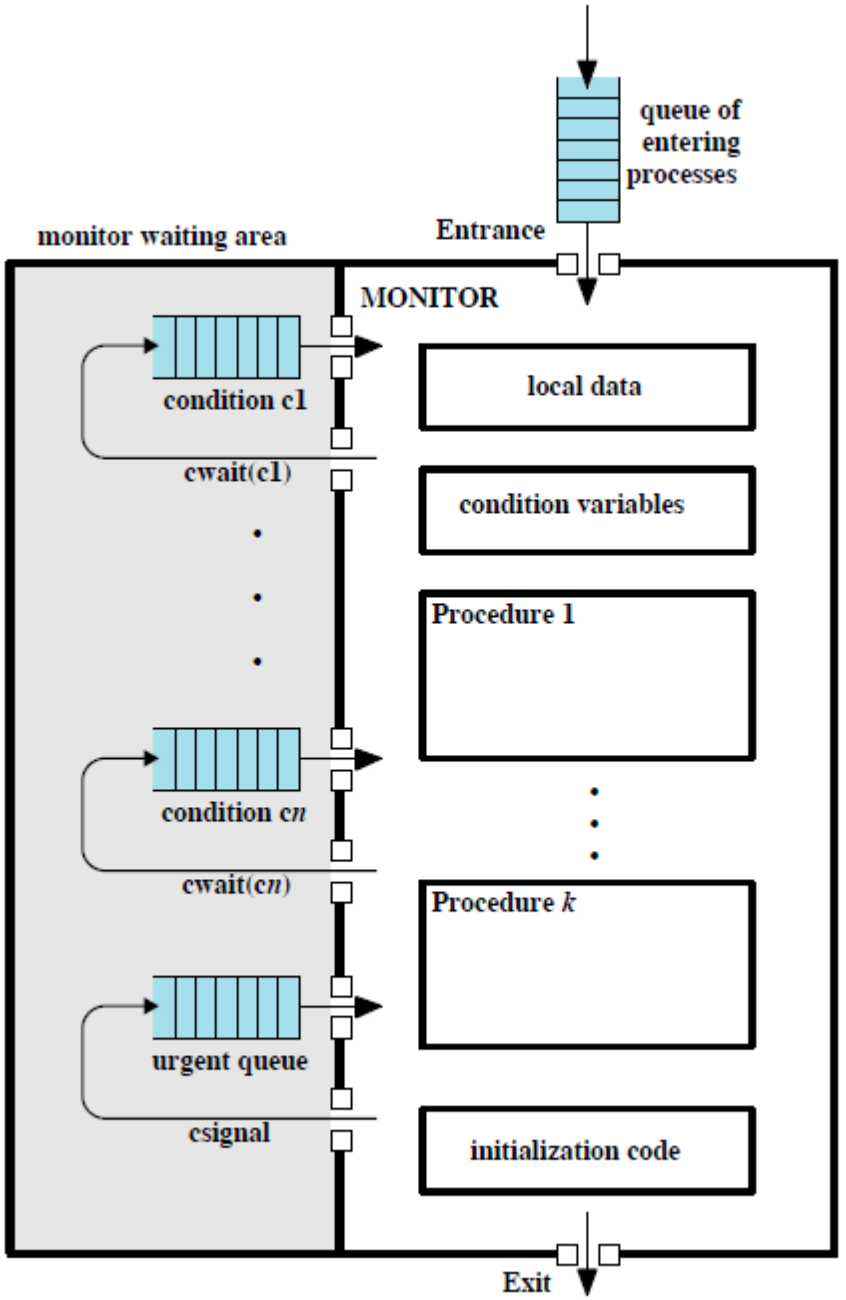
Solo un proceso puede estar ejecutando en el monitor en un instante dado, cualquier otro proceso que haya invocado al monitor se suspende a la espera que el monitor quede disponible.

De este modo una variable o estructura de datos compartida puede protegerse dentro de un monitor.

Un monitor proporciona sincronizacion por medio de variables de condicion.Hay dos funciones para operar con estas variables:

cwait(c): suspende la ejecucion del proceso que llama bajo la condicon c. El monitor esta disponible para ser usado por otros procesos.

csignal(c): reanuda la ejecucion de algun proceso suspendido por un cwait bajo la misma condicion. Si hay varios procesos, escoje uno de ellos, de lo contrario hace nada.


queue of
entering
processes
Entrance
monitor waiting area
MONITOR
condition c1
cwait(c1)
local data
condition variables
Procedure 1
condition cn
cwait(cn)
Procedure k
urgent queue
csignal
initialization code
Exit

Solución al problema del productor/consumidor con monitores y buffer acotado.

```
void productor(){
      char x;
      while (cierto){
            producir(x);
            añadir(x);
      }
}
```

```
void consumidor(){
      char x;
      while (cierto){
            sacar(x);
            consumir(x);
      }
}
```

```
void añadir(char x){
      if (contador==N)
            cwait(no_lleno);
      buffer[sigent]=x;
      sigent=(sigent+1)%N;
      contador++;
      csignal(no_vacio);
}
```

```
void sacar(char x){
      if (contador==0)
            cwait(no_vacio);
      x=buffer[sigsal];
      sigsal=(sigsal+1)%N;
      contador--;
      csignal(no_lleno);
}
```

```
monitor buffer_acotado;
char buffer[N];
int sigent=0, sigsal=0;
int contador=0;
condition no_vacio, no_lleno;
void main(){
            parbegin(productor,consumidor);
}
```

Paso de mensajes

Tiene la ventaja adicional que se puede implementar en sistemas distribuidos.

La funcionalidad del paso de mensajes se manifiesta mediante dos primitivas:

```
send(destino, msg);
receive(origen, msg);
```

Sincronizacion

Un receptor no puede recibir un mensaje hasta que sea enviado por otro proceso. Es necesario especificar que le sucede al proceso despues de ejecutar send o receive.

Cuando se ejecuta un send, hay dos posibilidades: el emisor se bloquea hasta que reancibe un mensaje o no se bloquea. Del mismo modo, cuando un proceso ejecuta receive, hay dos posibilidades:

1.- si previamente se envio un mensaje, este es recibido y continua la ejecucion.

2.- si no hay mensaje esperando, el proceso se bloquea hasta que llega un mensaje o continua se ejecucion abandonando el intento de recepcion.

De este modo tanto el emisor como el receptor pueden ser bloqueantes o no bloqueantes.

Las combinaciones mas habituales son: send bloqueante-receive bloqueante (rendesvouz),send no bloqueante-receive bloqueante y send no bloqueante-receive no bloqueante.

Direccionamiento

Es necesario disponer de una forma de especificar que proceso recibira el mensaje. Del mismo modo un receptor puede necesitar saber quien es el emisor del mensaje

Existen el direccionamiento directo y el direccionamiento indirecto.

Formato del mensaje:

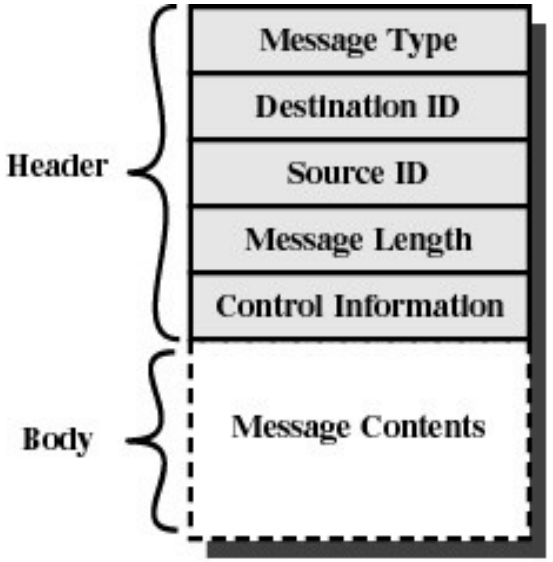


Figure 5.25 General Message Format

Disciplina de Cola

se utiliza para el despacho y recepcion de los mensajes.

de acuerdo a los requerimientos es posible incorporar prioridades o bien la posibilidad de examinar arbitrariamente la cola.

Exclusion mutua

```
const int n;
void P(int i){
	mensaje msg;
	while (cierto){
		receive(exmut,msg);
		// seccion critica
		send(exmut,msg);
		// resto
	}
}

void main(){
		crear_buzon(exmut);
		send(exmut,null);
		parbegin(P(1),.....P(n));
}
```

Se considera send no bloqueante y receive bloqueante.

El buzon funciona como un token que se pasa de un proceso a otro.

El problema del productor/consumidor usando mensajes

```
void productor(){
	mensaje msgp;
	while (cierto){
		receive(puede_producir, msgp);
		msgp = producir();
		send(puede_consumir,msgp)
	}
}
```

```
void consumidor(){
	mensaje msgc;
	while (cierto){
		receive(puede_consumir, msgc);
		consumir(msgc);
		send(puede_producir, null)
	}
}
```

```
void main(){
	crear_buzon(puede_producir);
	crear_buzon(puede_consumir);
	for(i=1;i<=capacidad;i++)
		send(puede_producir,null);
	parbegin(productor, consumidor);

}
```

El problema de la barbería

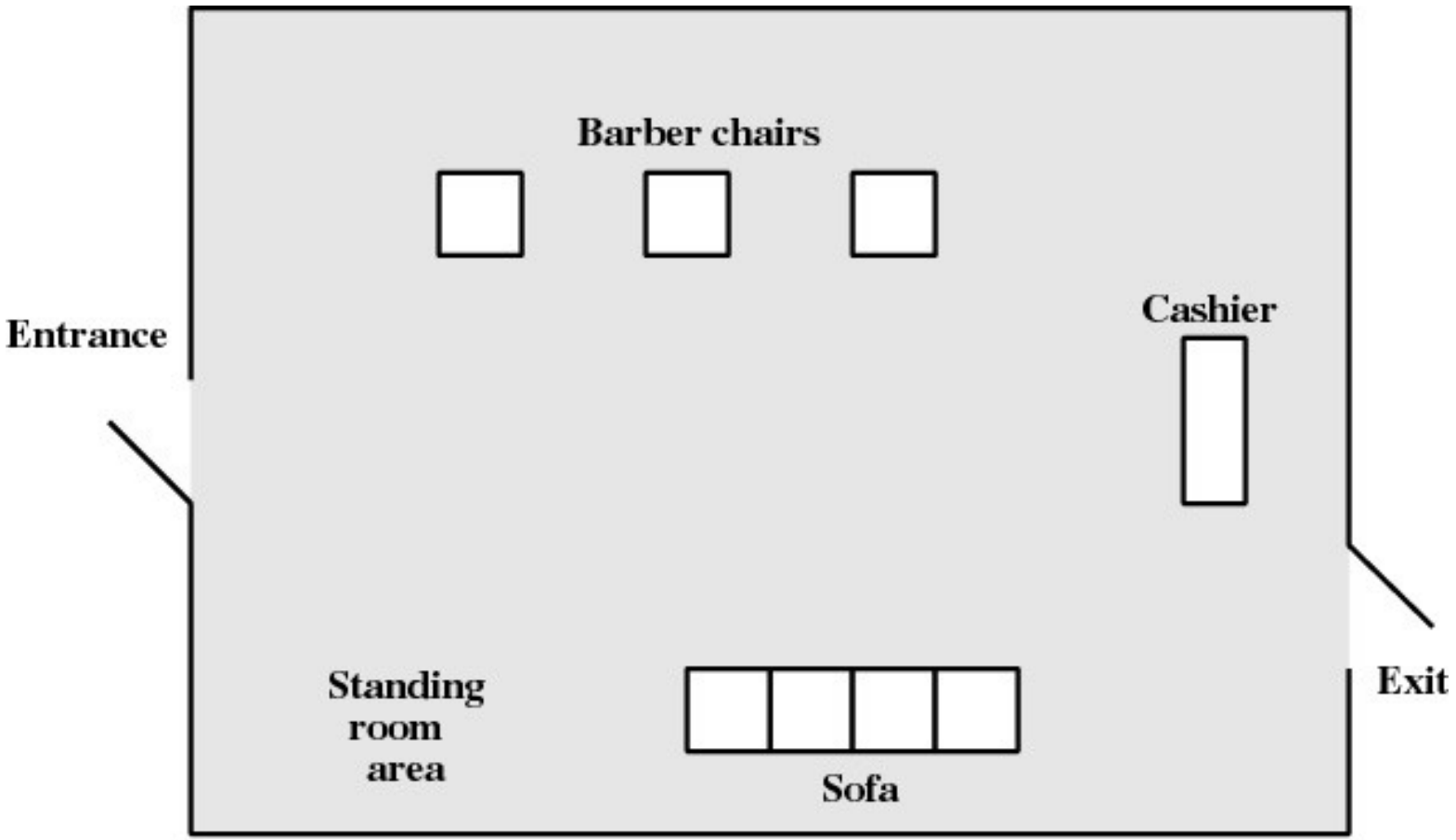


Figure 5.18 The Barbershop

Además:

• por cuestiones de seguridad se limita a 20 el número de clientes en la barbería.

• un cliente no entra en la tienda si su capacidad está completa.

• una vez dentro, el cliente toma asiento en el sofá o permanece de pie en la sala de espera.

• cuando un barbero está libre, se atiende al cliente que ha estado más tiempo en el sofá.

• para un cliente que está de pie, tomará asiento por orden de llegada si existe un lugar en el sofá.

•cuando un barbero finaliza un servicio, cualquiera puede aceptar el pago. Sin embargo, sólo un cliente paga a la vez ya que existe sólo una caja.

• los barberos dividen su tiempo:
  - cortando el pelo.
  - aceptando pagos.
  - durmiendo en su silla a la espera de un cliente.

Solución debe considerar:

- capacidad de la tienda y el sofá.
- capacidad de atención de clientes.
- asegurar que el cliente está en la silla del barbero.
- mantener al cliente sentado mientras no se termine el servicio.
- limitar un cliente por silla.
- pagar y esperar recibo.
- coordinar funciones de barbero y cajero.

Semáforo	***wait***	***signal***
max_capacidad	Cliente espera hasta que haya lugar en la barbería	Cliente que sale avisa a cliente que desea entrar.
sofa	Cliente espera por un lugar en el sofá	Cliente que deja el sofá avisa a otro que espera por un lugar.
silla_barbero	Cliente espera por una silla libre	Barbero avisa que su silla se encuentra disponible.
cliente_listo	Barbero espera que el cliente esté en la silla	Cliente avisa al barbero que acaba de sentarse en la silla.
terminado	Cliente espera que el servicio termine.	Barbero avisa cuando termina el servicio.
dejar_silla_b	Barbero espera que el cliente abandone la silla.	Cliente avisa al barbero que deja la silla.
pago	Cajero espera el pago de un cliente.	Cliente avisa al cajero que ya pagó.
recibo	Cliente espera el recibo de su pago.	Cajero avisa al cliente que su pago ha sido aceptado.
coord	Espera que un barbero esté libre para cortar el pelo o para cobrar.	Un barbero se encuentra disponible.

```
semaforo max_capacidad = 20;
semaforo sofa = 4;
semaforo silla_barbero = 3;
semaforo coord = 3;
semaforo cliente_listo=0,terminado=0,dejar_silla__b=0, pago=0,recibo=0;
```

```
void cliente(){
wait(max_capacidad);
entrar_tienda();
wait(sofa);
sentarse_sofa();
wait(silla_barbero);
levantarse_sofa();
signal(sofa);
sentarse_silla_barbero();
signal(cliente_listo);
wait(terminado);
levantarse_silla_barbero();
signal(dejar_silla_b);
pagar();
signal(pago);
wait(recibo);
salir(tienda);
signal(max_capacidad);
}
```

```
void barbero(){
   while (cierto){
      wait(cliente_listo);
      wait(coord);
      cortar_pelo();
      signal(terminado);
      wait(dejar_silla_b);
      signal(silla_barbero);
   }
}
```

```
void cajero(){
   while (cierto){
      wait(pago);
      wait(coord);
      aceptar_pago();
      signal(coord);
      signal(recibo);
   }
}
```

```
void main(){
   parbegin(cliente*50,barbero,barbero,barbero,cajero);
}
```

El problema de los lectores/escritores

• Existe un área de datos compartida entre una serie de procesos.

• El área de datos puede ser un archivo, un bloque de memoria o un conjunto de registros del procesador.

• Existen 2 tipos de procesos:
  - Lector: sólo lee los datos.
  - Escritor: sólo escribe datos.

• Se deben satisfacer las siguientes condiciones:
  - Cualquier número de lectores puede leer desde el área de datos.
  - Sólo un escritor puede escribir en el área de datos en cada instante.
  - Si un escritor está accediendo al área de datos, ningún lector puede leerlo.

Prioridad a los lectores: una vez que un lector ha comenzado a acceder a los datos, es posible que los lectores mantengan el control mientras exista al menos uno leyendo.

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true)
    {
     semWait (x);
     readcount++;
     if (readcount == 1)
          semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount--;
     if (readcount == 0)
          semSignal (wsem);
     semSignal (x);
    }
 }
void writer()
{
    while (true)
    {
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
    }
```

Figure 5.22 A Solution to the Readers/Writers Problem Using Semaphores: Readers Have Priority

Prioridad a los escritores: en el caso anterior los escritores están sujetos a inanición. Ahora no se permitirá acceder al área de datos

```
/*program readersandwriters*/
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true)
    {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1)
                         semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0)
               semSignal (wsem);
     semSignal (x);
    }
}
void writer ()
{
    while (true)
    {
     semWait (y);
          writecount++;
          if (writecount == 1)
               semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0)
               semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Figure 5. 23 A Solution to the Readers/Writers Problem Using Semaphores: Writers Have Priority