Secciones Críticas

y

Problemas Clásicos De Sicronización

El Problema de la Sección Crítica

- **Problema**: Coordinación de procesos que acceden a datos compartidos (memoria principal o a través de archivos).
- Las condiciones de competencia se resuelven mediante la definición de **secciones críticas** en las **partes de un proceso (código) que accedan a recursos compartidos**.
- **Definición.** Una sección crítica es una secuencia de actividades (o instrucciones) dentro de un proceso, durante el cual un área de memoria compartida debe ser accesada en forma mutuamente excluyente.
- La solución al problema de secciones críticas debe satisfacer los siguientes tres requerimientos:
  - **Exclusión Mutua.** dos procesos no pueden estar ejecutando simultáneamente sus **SC's**.
  - **Progreso.** sólo procesos en espera para ejecutar sus **SC's** pueden decidir quién es el siguiente en ejecutarla. En concreto, un proceso que no esté en su **SC** no debe bloquear otros procesos.
  - **Espera Limitada (sin hambruna).** Existe un número limitado de veces que los procesos ejecutan sus **SC** para permitir que otro proceso también pueda ejecutar su propia **SC**.

Ejemplo: Recorrido de un Arbol: versión recursiva

```
int count = 0;

void arbol(struct nodo *raiz) {
        count++;

        if ( raiz->left == NULL && raiz->right == NULL )
              return;
        else {
              if (raiz->left != NULL ) arbol(raiz->left);
              if (raiz->right != NULL ) arbol(raiz->right);
         }
   return;
 }
```

Versión Concurrente

```
#include <thread.h>
int count;

void  *arbol(struct nodo *raiz) {
   thread_t th0, th1;
   void *status;

   count++;  /*La Sección Crítica */
   if (raiz->left == NULL && raiz->right == NULL ) return;
   else {
      if (raiz->left != NULL) task1 = thr_create(NULL, 0, arbol,(struct nodo *)raiz->left,th0);
      if (raiz->right != NULL) task2 = thr_create(NULL, 0, arbol,(struct nodo *)raiz->right,th1);

     if (raiz->left != NULL) thr_join(th0,NULL,NULL);
     if (raiz->right != NULL) thr_join(th1,NULL,NULL);
     thr_exit(status);
   }
}
```

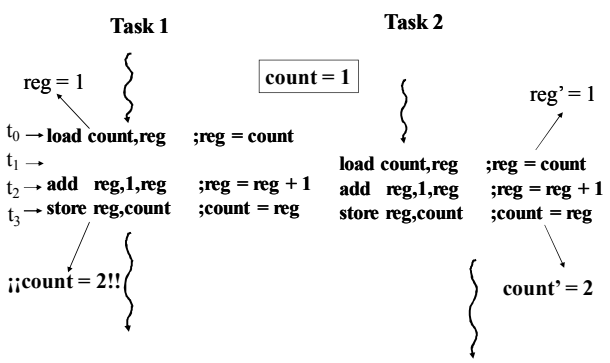
Detectando la Sección Crítica

```
int count = 0;
void *arbol(struct nodo *raiz) {
                count++;  /*S_C*/
                if ( ...) ...
                else ...
 }
```

La instrucción count++ en lenguaje de máquina:

```
load  count,reg     ;reg = count
add   reg,1,reg     ;reg = reg + 1
store reg,count     ;count = reg
```

Ejecución Concurrente.



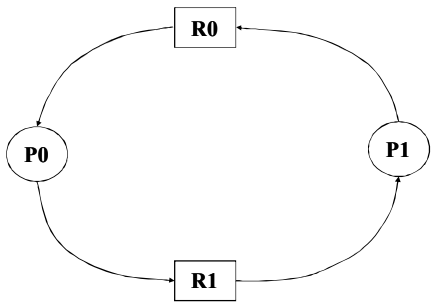
Solución con Mutex: Librería de Solaris

```
#include <thread.h>
int count;
mutex_t  mutex;

void  *arbol(struct *raiz) {
   thread_t task1, task2;
   void *status;
   mutex_lock(&mutex);
    count++;  /*La Sección Crítica */
   mutex_unlock(&mutex);
   if (raiz->left == NULL && raiz->right == NULL 1 )  return;
   else {
     if (raiz->left != NULL ) task1=thr_create(NULL, 0, arbol,(struct nodo *)raiz->left,th0);
     if (raiz->right != NULL ) task2=thr_create(NULL, 0, arbol,(struct nodo *)raiz->right,th1);

     if (raiz->left != NULL ) thr_join(th0,NULL,NULL);
     if (raiz->right != NULL ) thr_join(th1,NULL,NULL);

     thr_exit(status);
   }
}
```

Definiciones

- **Deadlock.** Situación en la cual cada uno de los procesos que se encuentra en un ciclo Proceso/Recurso, está esperando por recursos que son mantenidos por el siguiente proceso en ese ciclo, y que nunca los va a obtener.
- **Starvation (Hambruna).** Situación en la cual un proceso continuamente se le niega un recurso que éste necesita, incluso aunque ese recurso esté siendo otorgado a otros procesos.

Situación de Deadlock



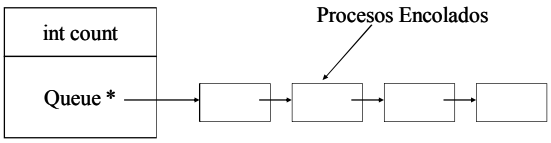
Primitivas para Exclusión Mutua

- Deshabilitar Interrupciones (solución hardware). Peligroso si lo hacen los programas de usuario.
- Variables switch o banderas (asumir lectura y escritura atómica). Problema: Espera Activa.
- Locks o spin-locks (cerrojos o candados). Solución Hardware. Espera Activa.
- Semáforos. Solución Software.
- Mensajes. Con o sin memoria compartida. Solución Software.
- Monitores de Hoare. Solución Software.
- Rendezvous de Ada. A través de mensajería. Solución Software.

Semáforos

- Un semáforo es un asignador de tickets.
- Las operaciones que acepta un semáforo son:
  - **Wait(sem):** pide un ticket al semáforo. Si el semáforo no tiene tickets disponibles, el proceso se bloquea hasta que otro proceso aporte tickets a ese mismo semáforo.
  - **Signal(sem):** aporta un ticket al semáforo. Si había algún proceso esperando un ticket, éste se desbloquea. Si había más procesos esperando tickets, se desbloquea el primero que llegó pidiendo tickets.
- Debe haber funciones que permitan crear y destruir un semáforo.
- Un mutex es un semáforo que proporciona solo un ticket.
- Un semáforo con 2 o mas tickets se llama semáforo contador.

Estructura de un Semáforo




```
typedef struct sem {
        int         count; /* Contador de tickets */
        Queue   *q;      /* La cola evita la Espera Activa */
} *Semaforo;
```

Semáforos en Unix

Llamadas al Sistema para Semáforos

- Para crear y operar sobre un sem´aforo en los sistemas Unix, se debe utilizar las siguientes funciones:
  - **semget**. Que permite obtener el identificador de un conjunto de sem´aforos. Su forma general es la siguiente:

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>

int semget (key_t key, int nsems, int semflg);
```

- La función devuelve el identificador del conjunto de semáforos asociado al valor del argumento key. El nuevo conjunto de nsems semáforos se crea si key tiene el valor IPC_PRIVATE.
- Si hubo éxito, el valor devuelto será el identificador del conjunto de semáforos (un entero positivo), de otro modo, se devuelve -1.

Llamadas al Sistema para Semáforos (2)

- La función semctl. Permite operaciones de control sobre semáforos. Su forma general es la siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

union semun {
    int     val; /* valor para SETVAL */
    struct semid_ds *buf; /* buffer para IPC_STAT, IPC_SET */
    unsigned short int *array; /* array para GETALL, SETALL */
    struct seminfo *__buf; /* buffer para IPC_INFO */
};

int semctl(int semid, int semnun, int cmd, union semun arg);
```

- La función realiza la operación de control especificada por cmd en el conjunto de semáforos (o en el semáforo semnum-avo del grupo) identificado por semid. El primer semáforo del conjunto está indicado por el valor 0 para semnum.
- El parámetro SETVAL pone el valor de semval a arg.val para el semnum-avo semáforo del conjunto.

Llamadas al Sistema para Semáforos (3)

- La operación semop. Realiza operaciones sobre semáforos. Su forma general es la siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop ( int semid, struct sembuf *sops, unsigned nsops )
```

- Esta función ejecuta operaciones en los miembros seleccionados del semáforo indicado por semid. Cada uno de los nsops elementos en el array apuntado por sops especifica una operación a ser realizada en un semáforo por struct sembuf incluyendo los siguientes miembros:
  - short sem_num; número de semáforo: 0 = el primero.
  - short sem_op; operación sobre el semáforo.
  - short sem_flg; flag (indicadores/parámetros) de la operación.
- Si semop es un entero positivo, la operación añade este valor a semval.

Ejemplo: Semáforos en Unix

Por ejemplo, la siguiente rutina permite inicializar un sem´aforo:

```
int inicia(int valor)
{
int semval, id;
unsigned short int init_values[] = {0, 0, 1}; /* Para inicializar tres
 semáforos */
val.array = init_values;
union semun {
    int val; /* Para inicializar un solo semáforo. Usar flag SETVAL */
    struct semid_ds *buf;
    unsigned short *array; /* Para inicializar dos o más semáforos. Usar
 flag SETALL *
 } arg;

if ( (id=semget(IPC_PRIVATE, N, (IPC_CREAT|0666))) == -1 )
     return (-1); /* el grupo estar´a compuesto por N sem´aforos */
arg.val = valor; /* valor es la cantidad de tickets*/
if ( semctl(id, 0, SETALL, arg) == -1 ) return (-2); /*error en
 inicializacion*/
    return(id); /* retorna el identificador del grupo de sem´aforos */
}
```

Operaciones sobre un Semáforo

- Por otro lado, para realizar una operación sobre un semáforo, debemos utilizar la siguiente rutina:

```
semaf_call(int semaforo, int op)
{
   struct sembuf sb;

   sb.sem_num = 0; /* ¿cuál semáforo estamos manejando {0, 1, 2, 3,...}? */
   sb.sem_op = op; /* Incrementamos o decrementamos la cantidad de tickets?
   */
   sb.sem_flg = 0; /* se puede usar SEM_UNDO para evitar problemas de
   hambruna */

    return ((semop(semaforo, &sb, 1))); /*devuelve -1 si hay error */
}
```

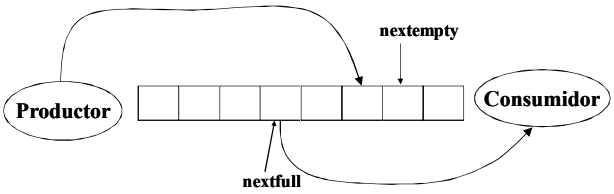
- Donde sem es el identificador del semáforo dentro del conjunto y op es una valor que permitirá incrementar o decrementar la cantidad de tickets del semáforo.

Productores y Consumidores

Descripción

- Cada cierto tiempo, el productor coloca un ítem en el buffer.
- El consumidor retira un ítem desde el buffer.
- Se requiere de una sincronización cuidadosa.
- El consumidor debe esperar si el buffer está vacío.
- El productor debe esperar si el buffer está lleno.
- También conocido como el problema del Buffer Limitado.
- Ejemplo en Unix: uso de pipes, cola de impresora.

El Problema del Buffer Limitado



count = 3

Solución usando semáforos: un Productor y un Consumidor

```
Semaforo empty;  /*con N tickets  */
Semaforo full;  /* con 0 tickets */
Item buffer[N];
int nextempty=0, nextfull=0;

void Productor() {
 for (;; )
  {
    Item x = Produce();

    Wait(empty);
    buffer[nextempty] = x;
    nextempty = (nextempty + 1) % N;
    Signal(full);
  }
}
```

```
void Consumidor() {
 for (;; )
  {
    Item x;

    Wait(full);
    x = buffer[nextfull];
    nextfull = (nextfull + 1) % N;
    Signal(empty);
    Consume(x);
  }
}
```

Solución usando semáforos: N Productores y M Consumidores

```
Semaforo empty;  /*con N tickets  */
Semaforo full;     /* con 0 tickets */
Semaforo mutx;   /* con 1 ticket */
Item buffer[N];
int nextempty=0, nextfull=0;

void Productor() {
 for (;; )
  {
    Item x = Produce();

    Wait(empty);
    Wait(mutex);
    buffer[nextempty] = x;
    nextempty = (nextempty + 1) % N;
    Signal(mutex);
    Signal(full);
  }
}
```

```
void Consumidor() {
 for (;; )
  {
    Item x;

    Wait(full);
    Wait(mutex);
    x = buffer[nextfull];
    nextfull = (nextfull + 1) % N;
    Signal(mutex);
    Signal(empty);
    Consume(x);
  }
}
```

El Problema de los Lectores y Escritores

Descripción

- En este problema varios procesos concurrentes comparten una misma estructura de datos y necesitan consultarla o actualizarla.
- Un Proceso Lector es aquel que está consultanto la estructura en algún momento.
- Un Proceso Escritor es aquel que está modificando la estructura de datos.
- Las características del problema son las siguientes:
  - Se permite a varios procesos lectores al mismo tiempo.
  - Sólo se permite un escritor en un instante dado. Mientras el escritor modifica la estructura no puede haber lectores u otros procesos escritores.
- Es clásico en los sistemas de Base de Datos.

Solución utilizando semáforos: prioridad para los lectores/hambruna para los escritores

```
Semaforo  escritura;   /* con 1 ticket */
Semaforo  mutex;   /* con 1 ticket *
int lectores = 0;


void Escritor() {
 for (;; )
  {
     Wait(escritura);
      escribir();
     Signal(escritura);
  }
}
```

```
void Lector() {
 for (;; )
  {
     Wait(mutex);
     lectores++;
     if (lectores==1) Wait (escritura);
                      //primero en llegar.
     Signal(mutex);
     leer();
     Wait(mutex);
     lectores--;
     if (lectores==0) Signal(escritura);
                   //último en salir.
     Signal(mutex);
  }
}
```

Solución del Problema de Productores y Consumidores con Semáforos de Unix.

```
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <errno.h>
#include <signal.h>
#define N            10
#define DOWN     -1
#define UP           1
#define MUTEX    0
#define FULL       1
#define EMPTY     2

int *next_empty = NULL;
int *next_full = NULL;
int *fin_buffer;
```

```
union semun{
     int                  val;
     struct semid_ds   *buf;
     unsigned short int *array;
 };

int memoria; // ID del segmento
     compartido.
int *inicio; //Dirección de memoria
               // Segmento Compartido.
int sema_set; //Conjunto de semáforos.
```

La función main() (1)

```
int main() {
     int pid, status;
     time_t times;

     signal(SIGINT, catch_int);
     srand((unsigned)time(&times));
      // Inicialización del semáforo binario
     sema_set = crea_semaforos();
      // Solicitamos memoria al sistema operativo
     if ((memoria=shmget(IPC_PRIVATE, 4096, 0660)) == -1) {
          perror ("Error en acceso a memoria compartida de
  sistema.");
          exit(-1);
      }
      // Obtenemos el puntero a dicha memoria
     inicio = (int *)shmat(memoria, NULL, 0);

     // Inicializamos la posición compartida
     fin_buffer = inicio + 10;
```

La función main() (2)

```
    //Creación de Procesos
    pid = fork();
    if ( pid == 0 ) productor(inicio, sema_set);
    pid = fork();
    if ( pid == 0 ) productor(inicio, sema_set);
    pid = fork();
    if ( pid == 0 ) consumidor(inicio, sema_set);
    pid = fork();
    if ( pid == 0 ) consumidor(inicio, sema_set);
    pid = fork();
    if ( pid == 0 ) consumidor(inicio, sema_set);

    wait(&status);

    shmdt((char *)inicio); /* Despega el segmento del proceso */
    shmctl(memoria, IPC_RMID, 0); /* destruye la memoria. */
    borra_s(sema_set); /* destruye el conjunto de semáforos semaforo */

    exit(0);
}
```

Creación de los Semáforos

```
int crea_semaforos()
 {
    int      idsem;
    union    semun argum;
    unsigned short int arg[]={1,0,N}; //mutex=0,full = 1,empty = 2

    if ( (idsem = semget(IPC_PRIVATE, 3, (IPC_CREAT|0666)) ) < 0 )
       return (-1);

    argum.array = arg;
    if ( semctl(idsem, 0, SETALL, argum) < 0 ) return (-2);

    return (idsem);
 }
```

Las Operaciones Wait() y Signal()

```
int semaf_call(int sem_set,int id_sem, int op){
    struct sembuf buff;

    buff.sem_num = id_sem;
    buff.sem_op  = op;
    buff.sem_flg = SEM_UNDO;

    return((semop(sem_set, &buff, 1)));
}

WaitSem(int set_sem, int id_sem) {
    semaf_call(set_sem,id_sem, DOWN);
 }

SignalSem(int set_sem, int id_sem) {
    semaf_call(set_sem,id_sem, UP);
 }
```

El Productor

```
productor(int *segment, int set_sem){
   if ( next_empty == NULL ) next_empty = segment;
   for ( ;; ) {
       WaitSem(set_sem, EMPTY);
       WaitSem(set_sem, MUTEX);

       // Producir un elemento
       *next_empty = Producir();

       // movemos el puntero a la siguiente entrada libre
       if ( next_empty == fin_buffer ) next_empty = segment;
       else next_empty++;

       SignalSem(set_sem, MUTEX);
       SignalSem(set_sem, FULL);
    }

   exit(0);
}
```

El Consumidor

```
consumidor(int *segment, int set_sem) {
   int dato;

   if ( next_full == NULL ) next_full = segment;
   for (;;) {
       WaitSem(set_sem, FULL);
       WaitSem(set_sem, MUTEX);

       dato = *next_full;
       Consumir(dato);

       // movemos el puntero a la siguiente entrada con
   elementos, de lo contrario al principio
       if ( next_full == fin_buffer ) next_full = segment;
       else next_full++;

       SignalSem(set_sem, MUTEX);
       SignalSem(set_sem, EMPTY);
    }
    exit(0);
}
```

¿Cómo termina el Programa?

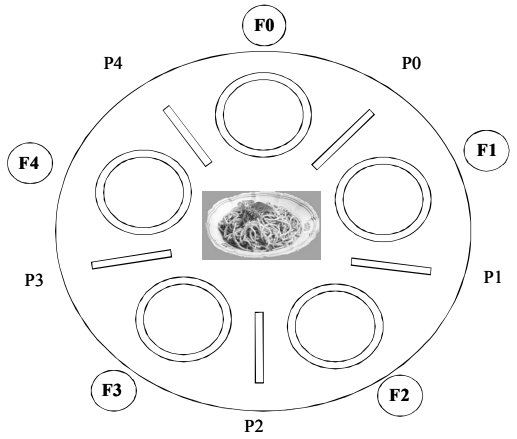
```
void catch_int(int sig_num)
 {
    signal(SIGINT, catch_int);
    printf("Adios Productores y Consumidores %d\n",getpid());
    fflush(stdout);

    // A este punto se llega después de que hijos y padre
    // hayan terminado.
    shmdt((char *)inicio);

    exit(0);
}
```

El Problema de los Filósofos Cenando

Descripción

- Planteo:
  - Cinco filósofos sentados en una mesa circular.
  - En la mesa hay cinco platos con tallarines y cinco tenedores (o palillos chinos).
  - Cada filósofo necesita dos tenedores o palillos para comer (el derecho y el izquierdo).
  - Vida del filósofo: Sucesión de ciclos alternativos de comer y pensar.
    - Si tiene hambre:
      - Toma los tenedores de derecha e izquierda (de uno en uno y si están libres).
      - Come.
      - Deja los tenedores (de uno en uno) sobre la mesa.
    - Cuando deja de comer, piensa.

El Problema de los Filósofos Cenando



Solución correcta utilizando Semáforos

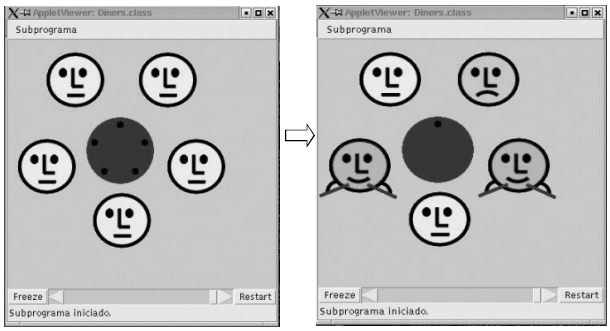
Primera Solución

```
Semaforo tenedor[N]; */ 1 ticket c/u */

void Filosofo(int i)
 {
    for(;;)
     {
        Wait(tenedor[min(i,(i+1)%N)];
        Wait(tenedor[max(i,(i+1)%N)];
          Comer();
        Signal(tenedor[min(i,(i+1)%N)];
        Signal(tenedor[max(i,(i+1)%N)];
          Pensar();
     }
 }
```

Segunda Solución

```
Semaforo tenedor[N];
Semaforo sala;  */ 4 tickets */

void Filosofo(int i)
 {
    for(;;)
     {
        Wait(sala);
        Wait(tenedor[i];
        Wait(tenedor[(i+1)%N];
          Comer();
        Signal(tenedor[i];
        Signal(tenedor[(i+1)%N];
        Signal(sala);
          Pensar();
     }
 }
```

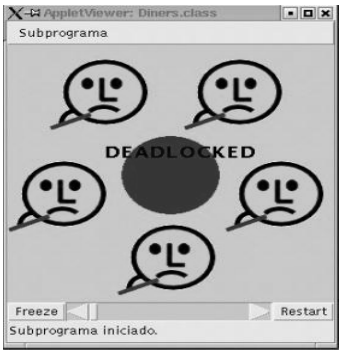
Estado normal de ejecución



Solución Errónea con Semáforos

```
Semaforo tenedor[N]; /* 1 ticket c/u */

void Filosofo(int i)
 {
    for(;;)
     {
        Wait(tenedor[i];
        Wait(tenedor[(i+1)%N];
          Comer();
        Signal(tenedor[i];
        Signal(tenedor[(i+1)%N];
          Pensar();
     }
 }
```

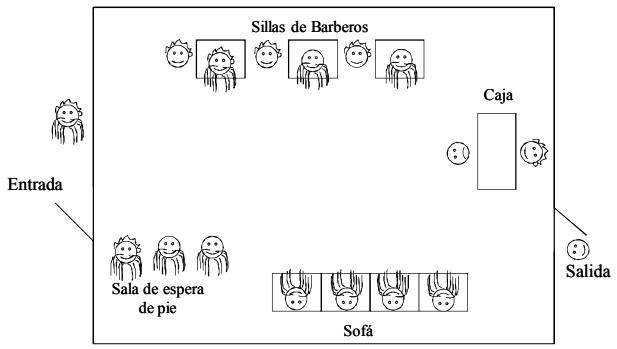
Situación de Deadlock



El Problema de la Barbería

Los Barberos Dormilones

La Barbería



Posible Solución: PROBLEMA DE LA BARBERÍA

- Representa una situación de coordinación de procesos cliente-servidor donde el servidor es idempotente (hasta que no termina de atender a un cliente no empieza con el siguiente). La barbería tiene tres sillas, tres barberos, una zona de espera, y un sofá con capacidad para cuatro personas.

Solución No Equitativa

```
Semaforo max_capacidad; /* 20 tickets */
Semaforo sofa; /* 4 tickets */
Semaforo cliente_listo, terminado, dejar_silla_b, pago, recibo;
/*Inicialmente 0 ticket c/u*/
Semaforo silla_barbero; /*Inicialmente 3 tickets */
Semaforo coord; /*Inicialmente 3 tickets */
```

Posible Solución: PROBLEMA DE LA BARBERÍA (2)

```
void barbero(void)
{
  while (1) {
    Wait (cliente_listo);
    Wait (coord);
      corta_pelo();
    Signal (coord);
    Signal (terminado);
    Wait (dejar_silla_b);
    Signal (silla_barbero);
  }
}
```

```
void cajero(void)
{
  while (1) {
    Wait (pago);
    Wait (coord);
      aceptar_pago();
    Signal (coord);
    Signal (recibo);
  }
}
```

Posible Solución: PROBLEMA DE LA BARBERÍA (3)

```
void cliente(void) {
    Wait (max_capacidad);
      entrar_barberia();
    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_barberia();
    Signal(max_capacidad);
}
```

Los Monitores

¿Qué es un Monitor?

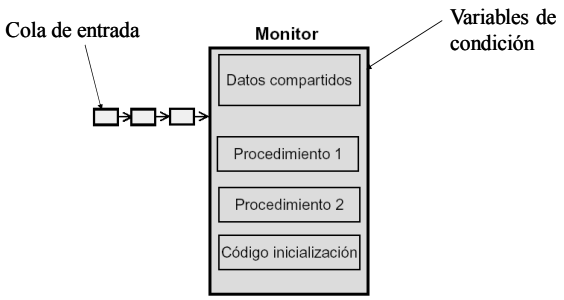
- Un Monitor es una versión concurrente de una estructura de datos.
- Paquete o módulo especial que contiene estructuras de datos, variables y procedimientos.
- Posee un estado interno, mas un conjunto de operaciones.
- Estas operaciones se invocan concurrntemente, pero el monitor las ejecuta secuencialmente.
- Fueron Inventados por Hoare.
- Java proporciona Monitores con una variable de condición.

¿Cómo se definen?: Sintaxis de Pascal Concurrente

- **Especificación de un monitor:**

```
type <nombre_monitor> = monitor
    var variables compartidas;

    procedure entry Proc1 (...)
    begin ... end;

    procedure entry Proc2 (...)
    begin ... end;

    begin
      código de inicialización;
    end;
```

Estructura de un Monitor



Ejemplo: Solución para Productores y Consumidores

```
type BUFFER = monitor
var pool: array[0..N-1] of Item;
in, out, count: integer;
noempty, nofull: condition;

Procedure entry Put(x: Item);
begin
   if (count = n) then nofull.wait;
   pool[in] := x;
   in := (in + 1) mod N;
   count := count + 1;
   noempty.signal;
end;
```

```
Procedure entry Get(var x: Item);
begin
   if (count = 0) then noepmty.wait;
   x := pool[in];
   out := (out + 1) mod N;
   count := count - 1;
   nofull.signal;
end;
```