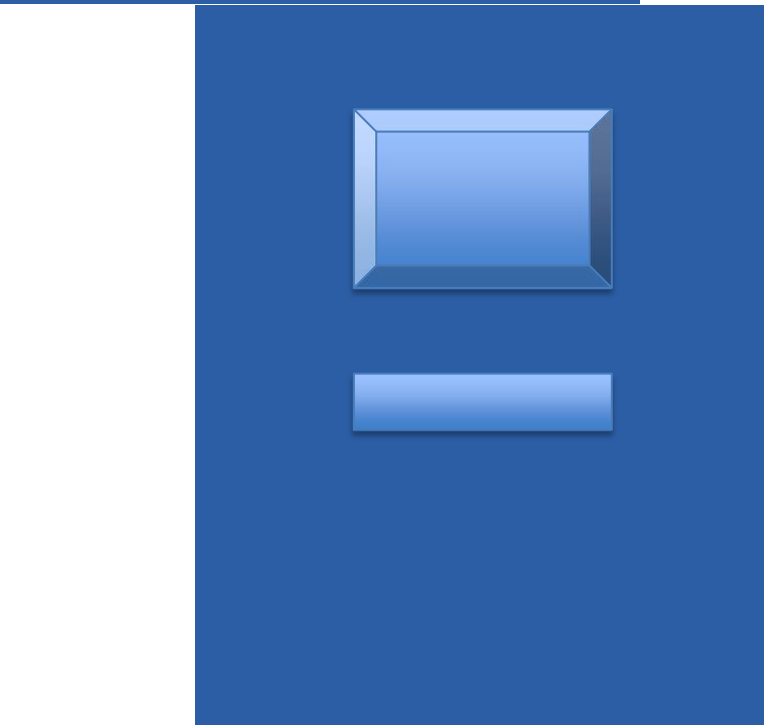
Sistemas Operativos II



NO ENTRA EN EL EXAMEN

Monitores

RPC

Paso de mensajes

TEMA 4.1
Sincronización y comunicación

4.1.1 PROBLEMAS DE SINCRONIZACIÓN Y COMUNICACIÓN ENTRE PROCESOS

Sincronización y Comunicación

Concurrencia: ejecución asíncrona de varios procesos con acceso a los mismos recursos.

La concurrencia de procesos puede representarse en **tres contextos** diferentes:

1. Varias aplicaciones ejecutándose a la vez sobre el mismo procesador.
2. Aplicaciones implementadas como conjuntos de procesos concurrentes.
3. Estructuración de los SSOO como conjunto de procesos concurrentes.

En un sistema donde está permitida la ejecución concurrente de procesos se debe garantizar la existencia de:

- Mecanismos de **acceso exclusivo a los recursos compartidos**.
- Mecanismos de **sincronización y comunicación que permitan establecer un orden de ejecución en los procesos**.

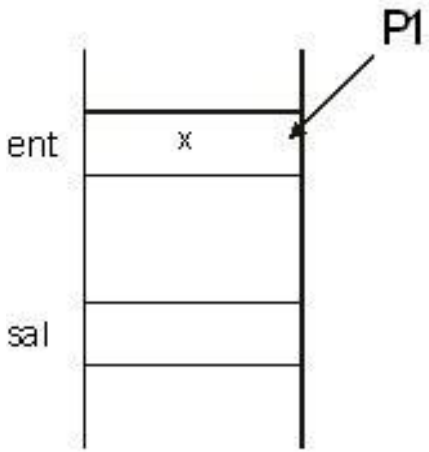
Sincronización y Comunicación

Ejemplo: E/S por teclado y pantalla.

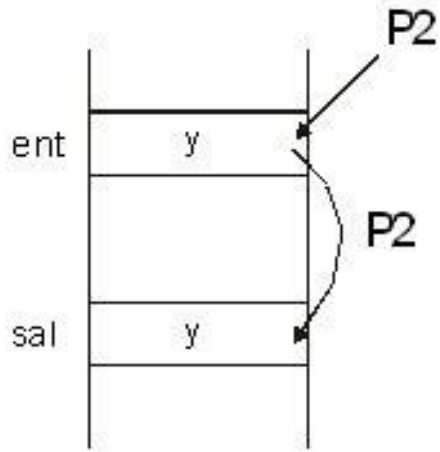
```
static char sal, ent;

void echo() {
   extern char sal,ent;
(1)
   getc(ent);
   sal := ent;
   puts(sal);
} // echo

void main () {
   ...
   echo();
   ...
} // main
```

P1 y P2 ejecutan a la vez este código

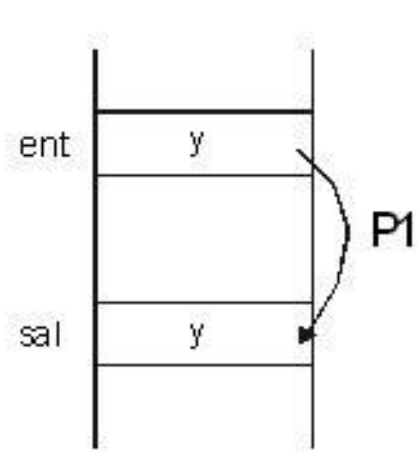
1. P1 ejecuta main, llega a la llamada a echo y ejecuta hasta (1) cargando *x* en ent.
2. P2 ejecuta todo el código hasta el final, cargando *y*, y mostrandola por pantalla.
3. P1 termina su ejecución, muestra *y*.



(1) P1 pone el valor x en la variable ent y se interrumpe.



(2) P2 se ejecuta por completo machacando el valor de ent y colocando el valor y en ella.



(3) P1 continúa la ejecución pero con un valor ERRONEO en la variable ent.

Sincronización y Comunicación

Ejemplo: (continúa)

Para solucionar la ejecución errónea se puede plantear el siguiente requisito:

- echo() es una función global pero **sólo lo puede utilizar un proceso a la vez**. La secuencia de ejecución sería la siguiente:
- P1 llama a echo() y es interrumpido justo después de llamar a la función de entrada, en (1).
- P2 llama a echo() pero no puede utilizarlo porque está bloqueado (está siendo usado por otro proceso), por lo que se detiene P2 hasta que pueda acceder a echo().
- P1 retoma la ejecución y completa echo().
- P2 puede ahora ejecutar echo() sin cometer ningún error.

Conclusión: **los recursos globales deben estar protegidos para evitar errores**.

En sistemas multiprocesador el problema es similar: varios procesos ejecutando a la vez pueden realizar accesos simultáneos a recursos.

Sincronización y Comunicación

Funciones de Gestión del SO relacionadas con Concurrencia:

- Seguir la pista de los procesos activos: control de **PCBs**.
- **Asignar y retirar recursos** a los procesos en función de:
  - Tiempo de procesador (planificación)
  - Memoria consumida (gestión de memoria virtual)
  - Utilización de archivos
  - Utilización de recursos de E/S
- **Protección de datos y recursos asignados** a cada proceso.
- Los **resultados** de un proceso deben **ser independientes de su velocidad de ejecución**. No se deben ver afectados por interrupciones en la propia ejecución de un proceso.

4.1.2 EXCLUSIÓN MUTUA. SOPORTES.

Exclusión Mutua

Requisitos para la exclusión mutua:

- Debe cumplirse la **exclusión mutua**, es decir, **debe permitirse a los procesos acaparar el uso de recursos críticos con exclusividad**.
- Dos procesos no pueden estar simultáneamente en sus secciones críticas.
- No se pueden hacer suposiciones acerca de la **velocidad de ejecución** de los procesos o **cantidad** de ellos en ejecución.
- Cuando ningún proceso está en la sección crítica, cualquier proceso que lo solicite debe poder entrar en ella.
- **Ningún** proceso **esperará indefinidamente** para entrar en sección crítica.
- Los procesos solo pueden estar en la sección crítica durante un **periodo finito de tiempo**.
- Ningún proceso que se ejecute fuera de su región crítica puede bloquear a otros procesos.

Estos requisitos pueden ser implementados bien por SW, bien por HW.

Soporte SW para Exclusión Mutua

Todas las soluciones SW suponen la existencia de unas primitivas HW para la exclusión mutua en el acceso a memoria.

Primer intento: *solución por turnos*.

- ***Protocolo del Iglú***: "vivienda unipersonal con una pizarra dentro, por cuya puerta solo cabe una persona".
- Los procesos del sistema se asemejan a los esquimales de este protocolo.
- **En la pizarra estará escrito el proceso cuyo turno está activo**.
- Si un proceso "**entra**" **y** "**ve**" **su turno → ejecuta su sección crítica y** después **cambia el turno**.
- Si un proceso "**entra**" **y NO está su turno → se sale y espera**, luego vuelve a entrar para comprobar si ya es su turno.

Soporte SW para Exclusión Mutua

Primer intento: *solución por turnos.*

La **implementación**, considerando a turno:0..1 variable compartida sería:

PROCESO 0	PROCESO 1

...
while (turno != 0) { /*nada*/ }
<sección crítica>
turno = 1;
...
	...
	while (turno != 1) { /*nada*/ }
	<sección crítica>
	turno = 0;
	...

Garantiza exclusión mutua.

Inconvenientes:

- Los procesos deben alternar estrictamente sus secciones críticas.
- Si uno de los procesos falla, el otro puede quedarse esperando indefinidamente para entrar en la sección crítica.
- Problema de la solución: ***sólo considera el estado de uno de los procesos*** a la hora de entrar en la sección crítica. Debería considerar el de los dos.

Soporte SW para Exclusión Mutua

Segundo intento: *variables cerrojo.*

- Cada esquimal tiene su propio iglú en el que puede leer y escribir, pero en el iglú del prójimo sólo puede leer.
- A la hora de entrar en la sección crítica, el proceso "mira" la pizarra del otro para ver si está a FALSE. Si es así,
  - Escribe en su pizarra TRUE para que el otro sepa que está en su sección crítica.
  - Ejecuta su sección crítica.
  - Escribe FALSE en su pizarra.

Implementación considerando int senial[2] las "pizarras" compartidas:

PROCESO 0	PROCESO 1

`...`
`while (senial[1]) { /*nada*/ }`
`senial[0] = 1;`
`<sección crítica>`
`senial[0] = 0;`
`...`
	`...`
	`while (senial[0]) { /*nada*/ }`
	`senial[1] = 1;`
	`<sección crítica>`
	`senial[1] = 0;`
	`...`

Soporte SW para Exclusión Mutua

Segundo intento: *variables cerrojo.*

Esta solución **garantiza**:

- Que cada proceso puede entrar en su sección crítica todas las veces que lo desee, independientemente de lo que el otro haga.
- Si uno de ambos procesos falla fuera de su sección crítica, el otro todavía puede seguir ejecutándose.

Inconvenientes:

No garantiza totalmente la exclusión mutua. Ejemplo:

- P0 ejecuta la sentencia while y encuentra senial[1] == 0.
- P1 ejecuta la sentencia while y encuentra senial[0] == 0.
- P0 pone senial[0] = 1 y entra en su sección crítica.
- P1 pone senial[1] = 1 y entra en su sección crítica.

¡ Los dos procesos están a la vez en sus secciones críticas !

La solución no es independiente de la velocidad relativa de ejecución de los procesos.

Soporte SW para Exclusión Mutua

Tercer intento:

El problema de la propuesta anterior radica en que ***los procesos pueden cambiar su estado después de que el otro proceso lo haya comprobado***. Solución:

PROCESO 0	PROCESO 1

...
senial[0] = 1;
while (senial[1]) { /*nada*/ }
<sección crítica>
senial[0] = 0;
...
	...
	senial[1] = 1;
	while (senial[0]) { /*nada*/ }
	<sección crítica>
	senial[1] = 0;
	...

Esta solución **garantiza**:

- **Exclusión mutua**.
- Cada proceso puede entrar en su sección crítica todas las veces que lo desee, independientemente de lo que el otro haga.
- Si uno de los procesos falla fuera de su sección crítica, el otro todavía puede seguir ejecutándose.

Soporte SW para Exclusión Mutua

Tercer intento:

Inconvenientes:

- Se puede dar ***interbloqueo***. Por ejemplo, en la siguiente situación:
- P0 pone senial[0] = 1
- P1 pone senial[1] = 1
- **Ninguno de los dos puede entrar en su sección crítica** y como consecuencia, ambos esperarán indefinidamente a que el otro cambie el valor de las variables cerrojo.

Los procesos cambian su estado antes de conocer el estado del otro.

Sin embargo, el **problema** de este intento es que ***cada proceso fija su estado sin tener en cuenta el estado del otro proceso***.

Solución: **"lecciones de educación"**

Soporte SW para Exclusión Mutua

Cuarto intento:

Los procesos, tras recibir "lecciones de educación" actúan así:

1. Activan su señal.
2. Se preparan para desactivar su señal y "ceder el paso" al otro proceso.

PROCESO 0	PROCESO 1

...
senial[0] = 1;
while (senial[1]) {
  senial[0] = 0;
  <esperar un cierto tiempo>
  senial[0] = 1;
}
<sección crítica>
senial[0] = 0;
...
	...
	senial[1] = 1;
	while (senial[0]) {
	  senial[1] = 0;
	  <esperar un cierto tiempo>
	  senial[1] = 1;
	}
	<sección crítica>
	senial[1] = 0;
	...

- Garantiza **exclusión mutua**.
- Gran **retraso en llegar a la solución**, sin llegar al interbloqueo.

Soporte SW para Exclusión Mutua

Cuarto intento:

Ejemplo de secuencia de ejecución:

1. P0: `senial[0] = 1`
2. P1: `senial[1] = 1`
3. P0 comprueba `senial[1]`, que es 1, y por tanto entra en el bucle.
4. P1 comprueba `senial[0]`, que es 1, y por tanto entra en el bucle.
5. P0: `senial[0] = 0` (primera instrucción del bucle de espera)
6. P1: `senial[1] = 0` (primera instrucción del bucle de espera)
7. P0: `senial[0] = 1` (última instrucción del bucle de espera)
8. P1: `senial[1] = 1` (última instrucción del bucle de espera)
9. Vuelta al paso 3

- **Retraso en llegar a la solución** hasta que alguno de los procesos realice el cambio a 0 de su señal antes de que el otro compruebe su valor.

Soporte SW para Exclusión Mutua

Algoritmo de Dekker:

- Diseñado por Dijkstra en 1965 para solucionar el problema de acceso en exclusión mutua planteado por el matemático holandés Dekker.
- Suma a la observación del **estado** de los procesos la inclusión de un **turno**.

PROCESO 0	PROCESO 1

...
senial[0] = 1;
while (senial[1])
  if (turno == 1) {
  senial[0] = 0;
  while (turno == 1) {}
  senial[0] = 1;
  }
<sección crítica>
turno = 1;
senial[0] = 0;
...
	...
	senial[1] = 1;
	while (senial[0])
	  if (turno == 0) {
	  senial[1] = 0;
	  while (turno == 0) {}
	  senial[1] = 1;
	  }
	<sección crítica>
	turno = 0;
	senial[1] = 0;
	...

Soporte SW para Exclusión Mutua

Algoritmo de Dekker:

Esta solución garantiza:

- **Exclusión mutua**.
- Cada proceso puede **entrar en su sección crítica todas las veces que lo desee**, mientras que el otro no muestre intención de entrar en su sección crítica (a través de **senial**).
- **Si uno** de los procesos **falla fuera de su sección crítica**, **el otro** todavía **puede seguir** ejecutándose porque el que falló puso su senial a 0 justo al salir de la sección crítica.
- **No existe interbloqueo** gracias a la variable compartida **turno**.

Inconvenientes:

- Demasiado **complejo** para demostrar su corrección en cualquier caso.
- **Difícilmente extensible a *n* procesos**.

Soporte SW para Exclusión Mutua

Algoritmo de Peterson:

PROCESO 0	PROCESO 1

...
senial[0] = 1;
turno = 1;
while (senial[1] && turno==1) {};
<sección crítica>
senial[0] = 0;
...
	...
	senial[1] = 1;
	turno = 0;
	while (senial[0] && turno==0) {};
	<sección crítica>
	senial[1] = 0;
	...

Estudio de casos: supongamos **P0 bloqueado en while**. P0 sólo puede entrar cuando una de las dos condiciones deje de cumplirse:

- P1 no está interesado en entrar en su SC: imposible porque senial[1] == 1.
- P1 está esperando a entrar en su SC: imposible porque si turno == 1, P1 puede entrar en su SC.
- P1 monopoliza SC: no puede ocurrir porque P1 pone turno = 0 antes de la SC.

El algoritmo de Peterson es fácilmente generalizable para *n* procesos

Soporte HW para Exclusión Mutua

Inhabilitar Interrupciones:

- El SO, a través de las interrupciones, toma el control y cambia el proceso en ejecución.
- Se puede resolver la exclusión mutua haciendo que **un proceso que vaya a ejecutar su sección crítica deshabilite las interrupciones** para que nada le detenga, restaurándolas una vez que terminó su sección crítica. Implementación:

```
repeat
   <inhabilitar interrupciones>
   <sección crítica>
   <habilitar interrupciones>
   ...
forever
```

- **Ventaja**: **garantiza exclusión mutua**.
- **Inconvenientes**: **pérdida de rendimiento** (disminuye el grado de multiprogramación), no soluciona el problema en sistemas multiprocesador.

Soporte HW para Exclusión Mutua

Instrucciones Máquina Especiales:

- Problemas de soluciones SW:
- Necesario utilizar dos instrucciones para *leer* y *escribir* .
- Necesario utilizar dos instrucciones para *leer* e *intercambiar* valores en las variables cerrojo.
- Todas las soluciones SW suponen la existencia de unas primitivas HW para la exclusión mutua en el acceso a memoria.

- **Solución**: proponer **instrucciones máquina atómicas** para estas operaciones.
- Estas instrucciones se ejecutan "en un ciclo de reloj".
- Implementadas en ensamblador como atómicas.

- **En este curso estudiaremos *Test and Set* e *Intercambiar***

Soporte HW para Exclusión Mutua

Test and Set:

- **Definición:**

```
int TestAndSet (int *i) {
  if (*i == 0) {
        *i = 1;
        return (1);
  } else {
        return (0);
    }
} // TestAndSet
```

Si el valor de la variable compartida **es 0, la pone a 1** para indicar que se entra en sección crítica **y devuelve 1** (TRUE).

Si el valor de la variable compartida es 1 significa que alguien está en sección crítica, así **que TS devuelve 0** (FALSE).

- **Solución al probl**

PROCESO 0

```
...
while (!TestAndSet(&cerrojo)) {};
<sección crítica>
cerrojo = 0;
...
```

Soporte HW para Exclusión Mutua

Intercambiar:

- **Definición:**

```
void Intercambiar (int *r, int *m) {
  int temp;
  temp = *m;
  *m = *r;
  *r = temp;
} // Intercambiar
```

Idealmente: intercambio de valores entre un registro (r) y una posición de memoria (m).

Implementación en "un ciclo".

- **Solución al problema de ex. mutua:**
- **Inicialmente cerrojo = 0.**
- **El único proceso que puede** ejecutar la sección crítica es aquel que encuentre cerrojo a 0.

PROCESO 0

```
...
clave = 1;
do {
   Intercambio (&clave, &cerrojo);
while (clave != 0);
<sección crítica>
Intercambio (&clave, &cerrojo);
...
```

Soporte HW para Exclusión Mutua

Ventajas de las soluciones HW:

- Se pueden aplicar a cualquier número de procesos en sistemas con memoria compartida, sean mono o multiprocesador.
- Simples y fáciles de verificar.
- Se pueden usar para definir varias secciones críticas.

Desventajas de las soluciones HW:

- Se emplea espera activa, consumiendo tiempo de procesador.
- Puede producirse **inanición**, porque la selección del proceso a entrar en ejecución es arbitraria.
- Puede producirse **interbloqueo**. Ejemplo: sistema basado en prioridades.
  - P0 entra en la sección crítica y pone cerrojo = 1
  - P1, con mayor prioridad que P0, le interrumpe e intenta entrar en la SC.
  - P1 se mantiene en espera activa debido a que P0 jamás podrá interrumpirle (tiene menos prioridad) y desbloquear la variable cerrojo.

4.1.3 SEMÁFOROS.

Semáforos

- **Herramienta SW que permite resolver el problema de la sincronización entre procesos evitando la espera activa.**
- Inventados por *Dijkstra*. Ideas desarrolladas:
- Dos o más procesos cooperan y se sincronizan por medio de señales.
- Posibilidad de forzar a un proceso a detenerse en una posición concreta hasta que reciba una determinada señal.
- **Un semáforo S contendrá una variable entera** a la que, una vez asignado un valor inicial, **sólo podrá accederse a través de operaciones atómicas** (primitivas) estándar: **wait** y **signal**.
- **wait(s):** permite detener un proceso hasta la llegada de una señal. **Decrementa el valor del semáforo**. Si s < 0 el proceso que invoca wait se bloquea.
- **signal(s):** envía una señal al semáforo. **Incrementa el valor del semáforo**. Si s <= 0 tras ese incremento, significa que habrá algún proceso bloqueado, de modo que se desbloqueará al primer proceso bloqueado.

Semáforos

Definición formal de las primitivas de los semáforos:

- **Implementación:**

```
struct semaforo {
   int contador;
   struct cola *buffer_proc;
}
```

- **Primitiva wait:**

```
void wait(semaf) {
        semaf.contador--;
        if (semaf.contador < 0) {
                semaf.cola.input(proc);
                // Coloca en la cola de procesos al que invocó wait.
                bloquear();
                // El proceso que invocó wait es bloqueado.
                  }
}
```

- **Inicialización:**

```
struct semaforo semaf;
semaf.contador = 1;
// Un wait antes de bloqueo
```

Un semáforo puede inicializarse a un valor no negativo

Semáforos

Definición formal de las primitivas de los semáforos (sigue):

Primitiva signal**:**

```
void signal(semaf) {
   semaf.contador++;
   if  (semaf.contador <=  0) {        // Algún proceso bloqueado
      proc = semaf.cola.primero();    // Se saca al primero
      desbloquear(proc);
   }
}
```

Más características:

- Los procesos en espera se gestionan mediante una FIFO: equitatividad.
- Los procesos no pueden permanecer indefinidamente en la cola del semáforo.
- **Semáforo binario** (ó **mutex**) es aquel cuyo contador solo toma los valores 0 y 1. Los semáforos binarios son equipotentes a los generales.
- Proporcionan una solución al problema de exclusión mutua para ***n*** procesos.

Semáforos y exclusion mutua

Solución para dos procesos

```
void main() {
   semaforo semaf;

   // Inicialización
   semaf.contador = 1;
   entero = 0;                    // Segmento de memoria compartida (variable)
   pid_t pid = fork();            // Creación de proceso hijo
   switch(pid) {
      case -1: printf("Error, no funcionó fork"); break;
      case 0: P(); break;                          // Código del hijo.
      default: P(); break;          // Código del padre.
   } // switch
} // main

void P() {
   while(1) {
      wait(semaf);
      entero++;                                    //// Zona de exclusión mutua ////
      printf("%i", entero);          //// Zona de exclusión mutua ////
      signal(semaf);
   } // while
} // P
```

Implementación de Semáforos

- Las primitivas wait y signal deben implementarse como operaciones atómicas. Posibilidades:
- Soluciones SW: Algoritmo de Dekker, Algoritmo de Peterson
- Soluciones HW: Inhabilitación de interrupciones (preferida), Instrucciones especiales: *Test and Set*, *Intercambiar*

Soporte en lenguajes de programación:

- C: librerías sem.h, ipc.h
- C++: varias implementaciones de clases para sincronizar hebras
- Dependientes del SO.
- Independientes del SO: ZThreads

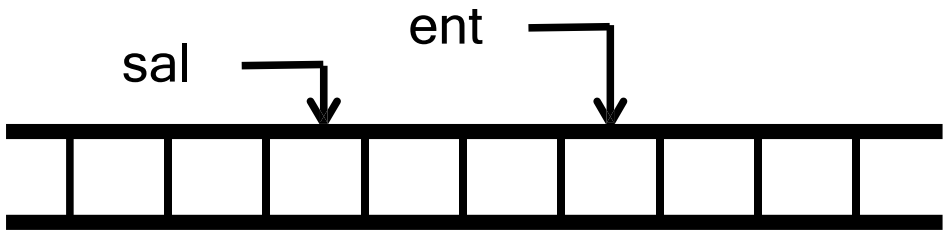
  http://zthread.sourceforge.net/html/classZThread_1_1Semaphore.html
- JAVA: clase Semaphore (Java 2, SE 5.0)

  http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Semaphore.html

Productor / Consumidor - Semáforos

Enunciado:

- Uno o más procesos productores generan datos de un cierto tipo y los sitúan en un buffer.
- Uno o varios consumidores sacan los elementos del buffer de uno en uno.
- Supóngase un buffer ilimitado:



Buffer Ilimitado

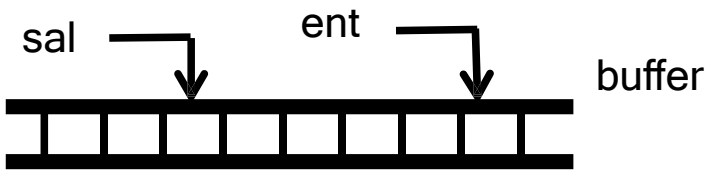
Situaciones "delicadas":

- Dos productores tratan de acceder al buffer a la vez.
- Dos consumidores tratan de acceder al buffer a la vez.
- Un consumidor accede a la vez que un productor...

Productor / Consumidor - Semáforos

Solución 0: sin sincronización

```
// Compartidas
int buffer[INF];
int ent = 0;
int sal = 0;
```



Espera Activa

```
// Productor
int v;
while (1) {
   v = Produce();
   buffer[ent] = v;
   ent = ent + 1;
}
```

```
// Consumidor
int w;
while (1) {
   while (ent <= sal) { /* nada */ };
   w = buffer[sal];
   sal = sal + 1;
   consume(w);
}
```

Las coincidencias en acceso no están resueltas

Productor / Consumidor - Semáforos

Solución 1: semáforos binarios

- Se gestiona el total de elementos en una variable n para llevar la cuenta del número de elementos que hay en el buffer:

$$n = ent - sal$$

- Se utiliza el semáforo binario exmut para hacer cumplir la exclusión mutua sobre n.

- Se utiliza el semáforo binario retraso para que el consumidor espere la llegada de elementos al buffer (esperará si el buffer está vacío).

- Suponemos que al menos un productor se ejecutará antes que todos los consumidores. Sin esta suposición, un consumidor podría extraer elementos de un buffer vacío.

Productor / Consumidor - Semáforos

Solución 1: semáforos binarios

```
#define numProd 10;                 // 10 productores
#define numCons 10;     // 10 consumidores

// Variables compartidas
int n = 0;                          // Elementos del buffer
semaforo exmut;                     // Ex. mutua
semaforo retraso;                   // Evita consumir si buffer vacío

void productor() {
   int elemento;
   while(1) {
      elemento = producir();
     waitB(exmut);                  //-- Inicio de Sección Crítica.
     aniadir(elemento);     // Añade elemento al buffer.
     n += 1;
     if (n == 1)                    // El único elemento lo acabamos de
       signalB(retraso);    // añadir, despierta 1er. bloqueado
     signalB(exmut);        //----- Fin de Sección Crítica.
   }
}
```

Productor / Consumidor - Semáforos

Solución 1: semáforos binarios

```
void consumidor() {
   int elemento;
   while(1) {
       waitB(exmut);                    //----- Inicio de Sección Crítica.
      elemento = extraer();             // Saca elemento del buffer.
      n -= 1;
      signalB(exmut);        //----- Fin de Sección Crítica.
      consumir(elemento);
      if (n == 0)                       // Si se consume último elemento
         waitB(retraso);      // hay que notificar al semáforo.
   }
}

void main() {
   exmut.contador = 1;
   retraso.contador = 0;
   for (int i = 1; i <= numProd; i++) crearProductor();
   for (int j = 1; j <= numCons; j++) crearConsumidor();
}
```

Productor / Consumidor - Semáforos

Solución 1: semáforos binarios

	n	retraso
Inicio	0	0
Productor ejecuta su sección crítica y añade dato	1	1
Consumidor1 entra en su sección crítica	0	0
Productor ejecuta su sección crítica y añade dato	1	1
Consumidor1, sale de sección crítica, pero no decrementa `retraso` (*)	0	1
Consumidor2 ejecuta sección crítica y consume un dato inexistente. No decrementa retraso porque n no es 0.	-1	1
Consumidor1 (`if n == 0`) fue TRUE, entonces decrementa retraso	-1	0

(*) Consumidor1 se queda tras la instrucción "`if (n==0)`", sin ejecutar la siguiente. Como no está en sección crítica en ese momento, otro proceso puede ejecutar código de su sección crítica.

Productor / Consumidor - Semáforos

Solución 2: semáforos generales

- El semáforo entero n llevará la cuenta de los elementos del buffer.
- El semáforo binario exmut servirá para acceder en exclusión mutua al buffer.

Solución correcta

```
// Variables compartidas:
semaforo n;
semaforo exmut;
n.contador = 0;
exmut.contador = 1;

void productor() {
   int elemento;
   while(1) {
      elemento = producir();
      waitB(exmut);
      aniadir(elemento);
      signalB(exmut);
      signal(n);
   }
}
```

```
void consumidor() {
   int elemento;
   while(1) {
      wait(n);
      waitB(exmut);
      elemento = extraer();
      signalB(exmut);
      consumir(elemento);
   }
}
```

Problema de la Barbería - Semáforos

Enunciado (Libro Stallings):

La barbería tiene **3 sillas** de peluquería, **3 barberos**, una zona de espera donde se pueden acomodar **4 clientes en un sofá**, y una **sala de espera** en la que se puede alojar al resto de clientes de pie.

Las medidas de seguridad limitan a 20 el número de clientes que puede haber en el establecimiento. En el ejemplo, procesaremos 50 clientes.

Restricciones:

- Los clientes no entran a la tienda si ésta está al completo (20 clientes).
- Si el sofá está completo, se espera de pie.
- Cuando el barbero está libre, atiende al cliente que más tiempo lleva en el sofá, y el que lleve más tiempo de pie, se sienta en el sofá.
- Cuando finaliza un corte de pelo, el barbero cobra (sólo se puede cobrar uno a la vez porque sólo hay una caja registradora); los barberos reparten el tiempo entre **CORTAR_PELO, COBRAR, DORMIR** en su silla a la espera de clientes.

Problema de la Barbería - Semáforos

Solución NO equitativa:

Se soluciona el problema mediante los siguientes semáforos:

Semáforo	Wait	Signal	Valor Inicial
max_capacidad	Cliente espera para entrar	Cliente que sale de la barbería avisa a un cliente a la espera de entrar.	20
sofa	Cliente espera a sentarse en sofá	Cliente se levanta del sofá, avisa a otro para sentarse.	4
silla_barbero	Cliente espera una silla de barbero vacía.	Barbero avisa cuando se queda libre una silla.	3
cliente_listo	Barbero espera hasta que el cliente esté en la silla.	Cliente avisa al barbero de que ya está sentado en la silla.	0
terminado	Cliente espera al corte de pelo completo.	Barbero avisa del fin de corte de pelo.	0
dejar_silla	Barbero espera hasta que se levante el cliente.	Cliente avisa al Barbero cuando se levanta de la silla.	0
pago	Cajero espera a que pague el cliente	Cliente avisa de que ya ha pagado.	0
recibo	Cliente espera el recibo de haber pagado.	Cajero entrega el recibo al cliente.	0
coord	Espera a que un Barbero esté libre para cortar el pelo o para cobrar.	Indica que el Barbero está libre.	3

Problema de la Barbería - Semáforos

Solución NO equitativa:

```
// Inicialización:
semaforo max_capacidad;   max_capacidad.contador = 20;
semaforo sofa;            sofa.contador = 4;
semaforo silla_barbero;   silla_barbero.contador = 3;
semaforo coord;           coord.contador = 3;
semaforo cliente_listo;   cliente_listo.contador = 0;
semaforo terminado;                terminado.contador = 0;
semaforo dejar_silla;     dejar_silla.contador = 0;
semaforo pago;            pago.contador = 0;
semaforo recibo;          recibo.contador = 0;

// Programa principal:
void main() {
   int i;
   for (i = 1, i <= MAX_CLIENTES, i++)
         insertar_cliente();
   for (i = 1, i <= NUM_BARBEROS, i++)
         insertar_barbero();
   insertar_cajero();
}
```

Problema de la Barbería - Semáforos

Solución NO equitativa:

```
void cliente() {
   wait(max_capacidad);
   entrar_en_tienda();
   wait(sofa);
   sentarse_en_sofa();
   wait(silla_barbero);
   levantarse_del_sofa();
   signal(sofa);
   sentarse_en_silla_barbero();
   signal(cliente_listo);
   wait(terminado);
   levantarse_de_silla();
   signal(dejar_silla);
   pagar();
   signal(pago);
   wait(recibo);
   salir_tienda();
   signal(max_capacidad);
}
```

```
void barbero() {
   while(1) {
         wait(cliente_listo);
         wait(coord);
         cortar_pelo();
         signal(coord);
         signal(terminado);
         wait(dejar_silla);
         signal(silla_barbero);
   }
}
```

```
void cajero() {
   while(1) {
         wait(pago);
         wait(coord);
         aceptar_pago();
         signal(coord);
         signal(recibo);
   }
}
```

Problema de la Barbería - Semáforos

Solución NO equitativa:

Esta **solución no es equitativa** porque puede ocurrir la siguiente situación:

- Tres clientes están esperando a que se les corte el pelo =>□ los tres estarán bloqueados en la cola del semáforo terminado en orden de llegada.
- Cuando a un cliente se le termina de cortar el pelo => el barbero hace signal(terminado).
- Si suponemos que el último que llegó es el primero al que se termina de cortar el pelo (barbero rápido o alopecia galopante), al ejecutar el barbero signal(terminado), **se sacará de la cola al primero que entró**, ¡¡ aunque no se le haya terminado de cortar el pelo !!. Mientras que el que terminó deberá esperar a que los otros dos clientes sean sacados de la cola.

Problema de la Barbería - Semáforos

Solución Equitativa:

- Se asigna un **número único a cada cliente** en su llegada:
- Variable compartida **contador**, cuyo acceso está protegido por el semáforo binario exmut1.
- **terminado** es un vector de semáforos, uno para cada cliente.
- Cliente ejecuta wait(terminado[numCliente]) cuando se sienta para ser atendido -> espera en su propio semáforo.
- Barbero ejecuta signal(terminado[numCliente]) para liberar al cliente apropiado cuando termina de atenderle.
- Para que los barberos conozcan el número del cliente que van a atender, cada cliente inserta su número en la cola cola1 justo antes de avisar al barbero a través del semáforo cliente_listo. Cuando el barbero puede cortar el pelo, saca el número más bajo de la cola (el primero que entró) y lo coloca en su variable local cliente_b.

Problema de la Barbería - Semáforos

Solución Equitativa:

```
// Inicialización:
semaforo max_capacidad;       max_capacidad.contador = 20;
semaforo sofa;                sofa.contador = 4;
semaforo silla_barbero;       silla_barbero.contador = 3;
semaforo coord;               coord.contador = 3;
semaforo exmut1;              exmut1.contador = 1;
semaforo exmut2;              exmut2.contador = 1;
semaforo cliente_listo;       cliente_listo.contador = 0;
semaforo dejar_silla;         dejar_silla.contador = 0;
semaforo pago;                pago.contador = 0;
semaforo recibo;              recibo.contador = 0;
semaforo terminado[MAX_CLIENTES];
    for (int k = 0; k < MAX_CLIENTES; k++) terminado[k].contador = 0;
int contador = 0;

// Programa principal:
void main() {
   for (int i = 1, i <= MAX_CLIENTES, i++) insertar_cliente();
   for (int j = 1, j <= NUM_BARBEROS, j++) insertar_barbero();
   insertar_cajero();
}
```

Problema de la Barbería - Semáforos

Solución Equitativa:

```
void barbero() {
    int cliente_b;
    while(1) {
        wait(cliente_listo);
        wait(exmut2);
        sacar_cola(cliente_b);
        signal(exmut2);
        wait(coord);
        cortar_pelo();
        signal(coord);
        signal(terminado[cliente_b]);
        wait(dejar_silla);
        signal(silla_barbero);
    }
}
void cajero() {
    while(1) {
        wait(pago);
        wait(coord);
        aceptar_pago();
        signal(coord);
        signal(recibo);
    }
}
```

```
void cliente() {
    int numCliente;
    wait(max_capacidad);
    entrar_en_tienda();
    wait(exmut1);
    contador++; numCliente = contador;
    signal(exmut1);
    wait(sofa);
    sentarse_en_sofa();
    wait(silla_barbero);
    levantarse_del_sofa();
    signal(sofa);
    wait(exmut2);
    poner_cola1(numCliente);
    signal(exmut2);
    signal(cliente_listo);
    sentarse_en_silla_barbero();
    wait(terminado[numCliente]);
    levantarse_de_silla();
    signal(dejar_silla);
    pagar();
    signal(pago);
    wait(recibo);
    salir_tienda();
    signal(max_capacidad);
}
```

4.1.4. REGIONES CRÍTICAS.

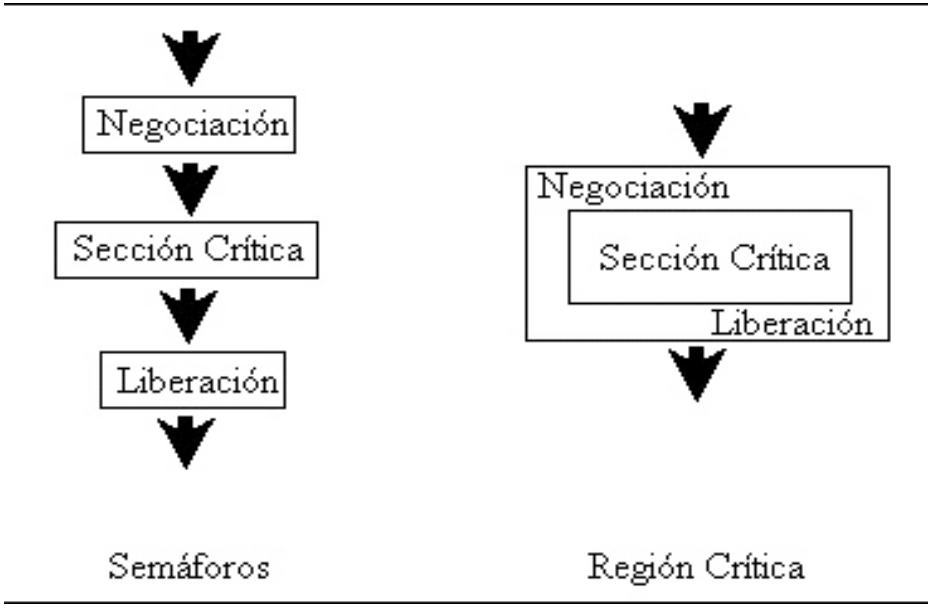
Regiones Críticas

“Problemas” de los semáforos:

- Alta probabilidad de equivocación por parte del programador.
- Los compiladores no ayudan al programador en su uso eficiente.
- Un uso incorrecto de los semáforos puede dar lugar a errores de temporización difíciles de detectar (falta *semántica* que ayude)

El concepto de **región crítica (RC)** fue introducido por Hansen en 1972:

- Una RC es un **“constructor”** que resuelve el problema de exclusión mutua estableciendo un mecanismo de acceso a los recursos compartidos.



Regiones Críticas

Características de las RC:

- El compilador implementa automáticamente los mecanismos necesarios para sincronizar el acceso a cada RC.
- Cada recurso compartido debe ser definido como *shared*.
- Existirá una RC para cada recurso compartido.
- No está permitido el acceso al recurso compartido fuera de la RC.
- Cada RC tiene asociada una cola para aquellos procesos que esperan a que la RC sea liberada.

Declaración de recurso compartido (sintaxis tipo “Pascal Concurrente”)

```
var v: shared T;    (* Variable compartida v de tipo T *)
```

Constructor:

```
region v do
  begin
    <sección crítica>
  end;
```

Regiones Críticas

Problema de las RC:

- La sincronización entre procesos que aguardan por una condición puede necesitar **espera activa**.

  ***Ejemplo*:**

```
...
repeat                      (* Hay espera activa en repeat *)
  region v do               (* Solo un proc. ejecuta la RC *)
    begin
      ok := (v > 0);
      if ok then
        v := v - 1;
    end;
until ok;
...
```

- **Solución**: incluir una condición de “guarda” para cada RC. Son las regiones críticas condicionales (RCC).

Regiones Críticas Condicionales

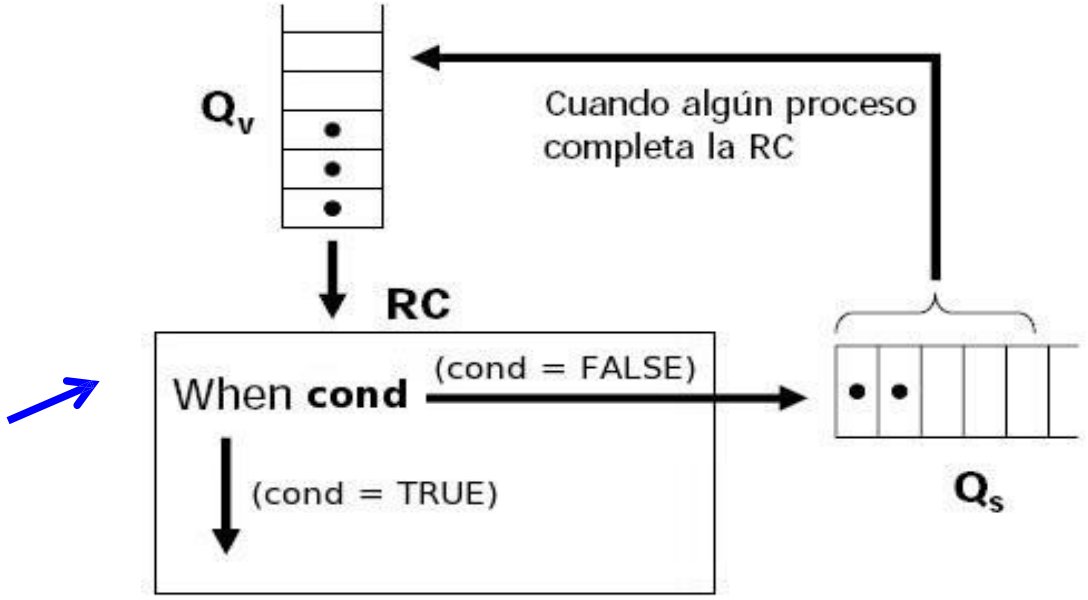
Características de las RCC:

- La idea es similar a las RC, con la diferencia de que se añade una condición de guarda en la región crítica.
- La evaluación de la condición se considera parte de la región crítica.
- Un proceso ejecuta el código limitado por la RCC sólo si la condición es cierta.

Sintaxis:

```
region v when cond do
  begin
    <sección crítica>
  end;
```

Hansen propuso la siguiente implementación con dos colas



Productor / Consumidor - RCCs

```
PROGRAM ProdCons;        (* Productor / Consumidor BUFFER LIMITADO *)
CONST MAX=50;
TYPE
  TBuffer = RECORD
    sigent, sigsal, cont: INTEGER;
    elems: ARRAY [1..MAX] OF INTEGER;
  END;
VAR buffer : SHARED TBuffer;


PROCESS Productor;
VAR pdato: INTEGER;
BEGIN
   ...
   pdato := producir();
   REGION buffer WHEN buffer.cont<MAX DO
     BEGIN
       buffer.elems[buffer.sigent] := pdato;
       buffer.sigent := (buffer.sigent MOD MAX) + 1;
       buffer.cont := buffer.cont + 1;
     END;
   ...
END;
```

OJO: el registro protegido se compone de buffer + índices

Productor / Consumidor - RCCs

```
PROCESS Consumidor;
VAR cdato: INTEGER;
BEGIN
  ...
  REGION buffer WHEN buffer.cont<>0 DO
    BEGIN
      cdato := buffer.elems[buffer.sigsal];
      buffer.sigsal := (buffer.sigsal MOD MAX) + 1;
      buffer.cont := buffer.cont - 1;
    END;
  ...
END;

BEGIN (* Principal *)
  (* Inicialización del buffer *)
  REGION buffer WHEN true DO
  BEGIN
      buffer.sigent:=1; buffer.sigsal:=1; buffer.cont := 0;
    END;
  COBEGIN
    Productores; Consumidores;    (* Generaría varios de cada uno *)
  COEND;
END.
```

4.1.5. MONITORES.

Monitores

Estructuras con funcionalidad equivalente a la de los semáforos:

- Resuelven el acceso en exclusión mutua a recursos compartidos.
- Permiten sincronización entre procesos evitando espera activa.
- Más sencillos de programar que los semáforos.
- Las operaciones que contienen proporcionan *semántica*.
- Se han usado en diversos lenguajes de programación:
  - PASCAL FC
  - Modula-2
- JAVA (synchronized)

http://www.artima.com/insidejvm/ed2/threadsynchP.html

Dos tipos de monitores:

- Monitores con señales (Hoare).
- Monitores con notificación y difusión (Lampson y Redell).

Monitores con Señales

Propuestos por primera vez por Hoare en 1974

- **Biografía de Tony Hoare: http://es.wikipedia.org/wiki/C._A._R._Hoare**
- **Inventor del algoritmo Quicksort, profesor emérito en Oxford, actualmente vinculado a Microsoft**

http://research.microsoft.com/users/thoare/

Un monitor con señales es un **módulo SW** que **consta de** uno o más **procedimientos**, una **secuencia de inicialización** y unos **datos locales**.

Características básicas:

- Las variables locales sólo pueden ser accedidas por los procedimientos del monitor.
- Los procesos "entran" en el monitor invocando a uno de los procedimientos contenidos en él.
- Sólo un proceso puede ejecutar código del monitor en cada instante, otros procesos que hayan invocado al monitor quedarán suspendidos.

Monitores con Señales

Exclusión Mutua:

- El acceso a una estructura de datos (o recurso) compartida puede protegerse declarándola dentro de un monitor y programando accedentes como procedimientos del monitor.

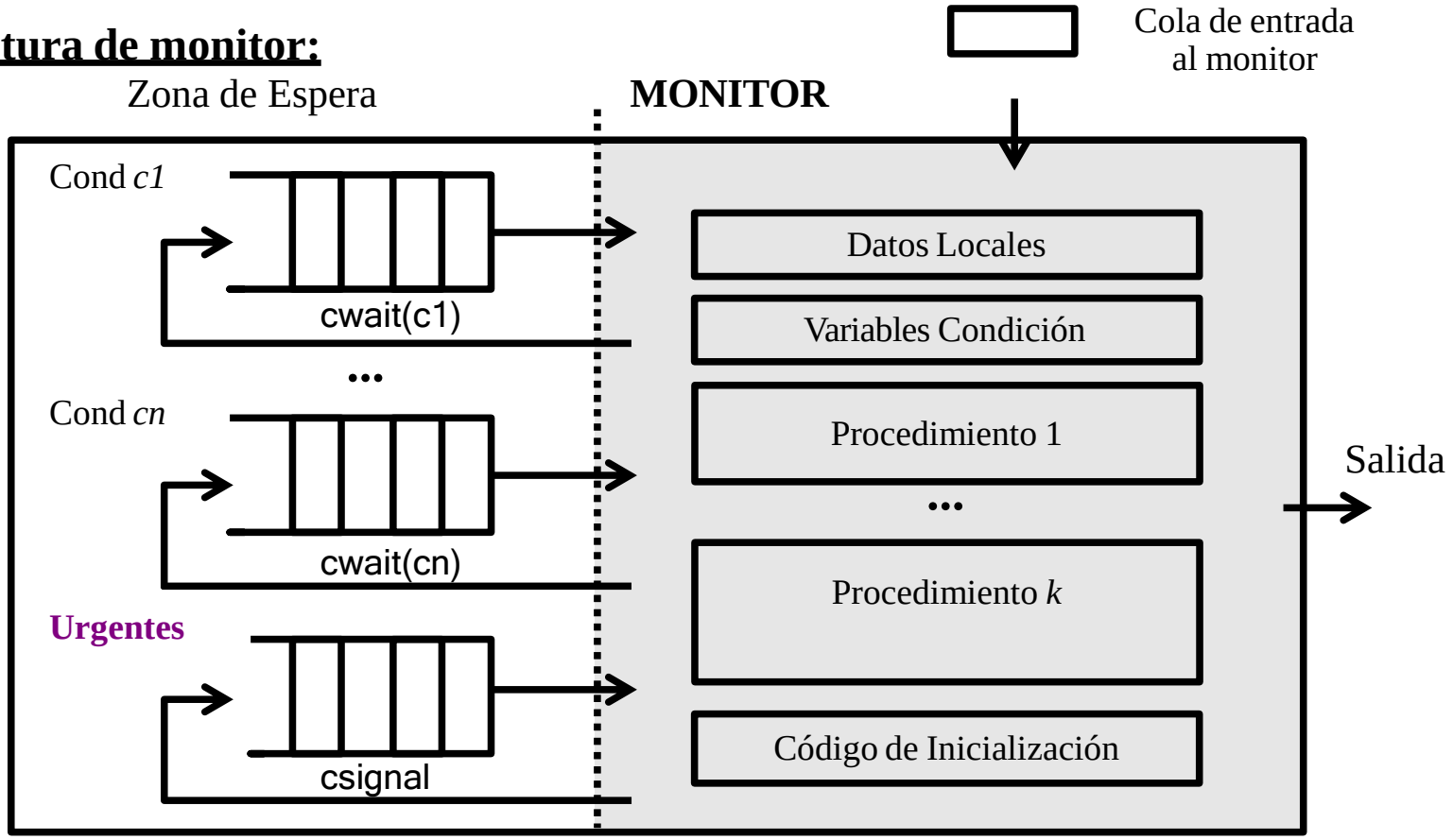
Sincronización:

- Un monitor proporciona sincronización a través de **variables de condición** que se colocan dentro del monitor. Las **operaciones** que permiten trabajar con esas variables son:

cwait(c): suspende la ejecución del proceso que llama bajo la variable de condición *c* y lo coloca en una cola de procesos vinculada esa condición. El monitor pasa a estar disponible.

csignal(c): reanuda la ejecución de algún proceso suspendido después de un *cwait* bajo la misma condición. Si hay varios procesos, elige uno de ellos; si no hay ninguno, no hace nada.

Monitores con Señales



Los procesos van a "Urgentes" si no invocan csignal como última instrucción del procedimiento.

Prod. / Cons. - Monitores Señales

Productor / Consumidor Buffer Limitado.

```
Monitor buffer_acotado;        // Inicio del bloque monitor.
   int buffer[1..N];           // Vars. del monitor (buffer   limitado)
   int sigent, sigsal;
   int contador;
   condition no_lleno, no_vacio;

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

   int tomar() {
      int y;
      if (contador == 0)
         cwait(no_vacio);
      y = buffer[sigsal];
      sigsal = (sigsal + 1) % N;
      contador--;
      csignal(no_lleno);
      return y;
   }

{
   sigsal = 0;   /* Código de inicialización del monitor. */
   sigent = 0;
   contador = 0;
} // Fin del monitor
```

Monitores con Notificación y Difusión

Propuestos por Lampson y Redell en 1980 para el lenguaje Mesa.

Inconvenientes de la solución de Hoare:

- Si hay al menos un proceso en una cola de condición, el proceso que ejecuta csignal sobre esa cola debe salir inmediatamente del monitor o suspenderse en él (cola de urgentes):
- S**i el p**roceso que ejecuta csignal no terminó de ejecutar instrucciones del monitor, se necesitan dos cambios de contexto.
- La planificación debe ser estricta porque la condición podría cambiar.

Solución de Lampson y Redell:

- Se sustituye el csignal por **cnotify**, con la siguiente interpretación:
- **cnotify(c)**: cuando un proceso ejecuta cnotify(c), se notifica al primer proceso de la cola *c* que se reanudará en breve, pero como no le asegura la reanudación inmediata, éste habrá de comprobar de nuevo la condición antes de poder seguir ejecutándose.

Monitores con Notificación y Difusión

Productor / Consumidor con notificación y difusión:

```
void aniadir(int x) {
   while (contador == N)
      cwait(no_lleno);
   buffer[sigent] = x;
   sigent = (sigent + 1) % N;
   contador++;
   cnotify(no_vacio);
}
```

```
int tomar() {
   int y;
   while (contador == 0)
      cwait(no_vacio);
   y = buffer[sigsal];
   sigsal = (sigsal + 1) % N;
   contador--;
   cnotify(no_lleno);
   return y;
}
```

- Se sustituyen los if por while para que los procesos vuelvan a comprobar las condiciones (cnotify no garantiza que al retomar la ejecución la condición se cumpla).
- En la mayoría de los casos, cnotify evita los dos cambios de contexto.

Monitores con Notificación y Difusión

Mejoras en monitores con notificación y difusión:

- Asociar un temporizador a las colas de condición que limite el tiempo de bloqueo.
- Evita la inanición si un proceso falla antes de enviar la señal.
- **Primitiva extra** para monitores con notificación y difusión:

  cbroadcasting(c): ocasiona que todos los procesos en espera de que se cumpla una condición, pasen a estar en la cola de listos.

Ventaja fundamental de los monitores con notificación y difusión:

- Es más complicado que los errores de programación afecten al sistema porque, aún mandando una señal equívoca, los procesos deben comprobar la condición al volver a ejecutar.

4.1.6 PASO DE MENSAJES.

Paso de Mensajes

El mecanismo de paso de mensajes permite:

- **Sincronizar procesos** (exclusión mutua).
- **Intercambiar información** entre los procesos que lo necesiten.

Método **fácilmente implementable** en sistemas distribuidos, multiprocesador y monoprocesador de memoria compartida.

Hay diversas implementaciones del mecanismo de paso de mensajes, aunque habitualmente se ofrece su soporte a través de dos **primitivas**:

  - send(destino, mensaje)
  - receive(origen, &mensaje)

- Éste es el **conjunto mínimo de operaciones** para que dos procesos puedan dedicarse al paso de mensajes. destino y origen son identificadores de los procesos destinatario y fuente del mensaje.

Paso de Mensajes

Características de la implementación del paso de mensajes:

Sincronización: un proceso receptor no puede obtener un mensaje hasta que no sea enviado por otro proceso.

- Se debe especificar qué ocurre después de que un proceso ejecute send ó receive.
  Posibilidades: detenerse (bloquearse) o no.
- Envío bloqueante, recepción bloqueante (*rendezvous*).
- Fuerte sincronización entre procesos.
- Envío no bloqueante, recepción bloqueante.
- Útil en programación, evitando emisores “desbocados”.
- Envío no bloqueante, recepción no bloqueante.
- Nadie debe esperar.
- “Peligroso” en programación.
- Envío bloqueante, recepción no bloqueante.

Paso de Mensajes

- **Direccionamiento**: especificación de emisor y receptor de los mensajes. Tipos de direccionamiento:

- Direccionamiento **directo**:
- send: incluye la identificación del proceso destino del mensaje.
- receive: hay dos maneras de gestionarlo.
- Designación *explícita* del emisor.
- Éste no siempre es conocido de antemano
- Muy recomendable para cooperación entre procesos.
- Designación ***implícita***: el parámetro origen tomará un valor de retorno cuando ya se haya completado la operación.

- Direccionamiento **indirecto** (*buzones*). Ventajas:
  - Se desacopla la relación emisor – receptor. Relaciones diversas:
  - Uno a uno: impide errores y da privacidad a la comunicación.
  - Uno a muchos: difusión
  - Muchos a uno: cliente/servidor, el buzón pasa a ser un ***puerto***.

Paso de Mensajes

Direccionamiento **indirecto** (*buzones*), ***continúa***:

La **asociación de buzones a procesos** puede ser:

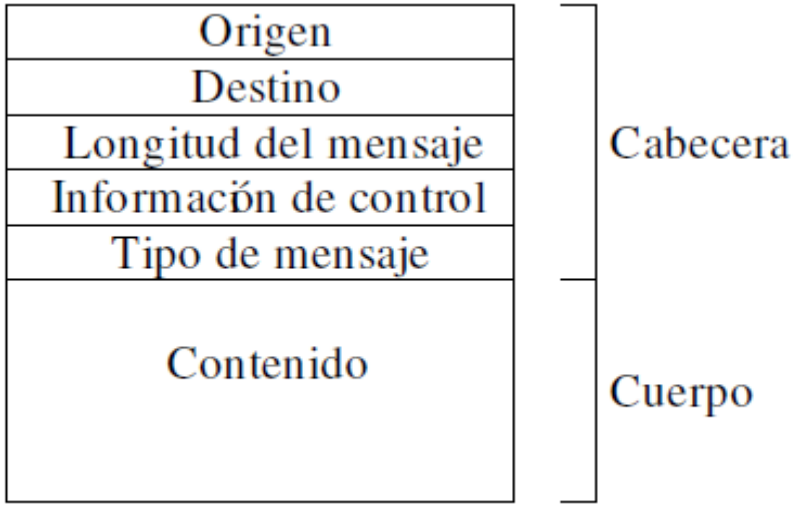
- **Estática**: los puertos suelen estar asignados estáticamente. Un puerto se crea y se asigna a un proceso definitivamente.
- **Dinámica**: cuando hay varios emisores, la asociación de un emisor a un buzón puede realizarse dinámicamente.
- Necesarias primitivas de conexión y desconexión del buzón.

Propietario de un buzón:

- Un puerto normalmente pertenece y es creado por el receptor.
- Un buzón es propiedad del proceso que lo crea (sea emisor o receptor).
- Cuando un proceso se destruye, también se destruyen los buzones que poseyera.

Paso de Mensajes

- **Formato de mensajes**: el formato de los mensajes depende de los objetivos del sistema de mensajería.
- Mensajes cortos y de tamaño fijo:
- Mejor procesamiento y ahorro en coste de almacenamiento.
- El mensaje puede ser la referencia a un archivo.
- Formato típico de un mensaje:
- **Cabecera**, con información relativa al mensaje. Longitud fija.
- **Cuerpo**, contenido real del mensaje. Longitud variable.



Paso de Mensajes

- **Disciplina de cola del buzón**:
- Habitualmente FIFO:
- Insuficiente para mensajes urgentes.
- Alternativas a FIFO:
- Asociación de prioridades en el emisor.
- Permitir al receptor examinar los mensajes de la cola y elegir uno en función de su criterio.

- **Exclusión mutua**: recepción bloqueante, *token*.
- Antes de acceder a la sección crítica, todo proceso debe recibir un mensaje que le indique la entrada.
- Tras recibirlo, ejecuta la sección crítica
- Al terminar la sección crítica envía un mensaje para los próximos procesos.

Paso de Mensajes

- **Exclusión mutua**: implementación.

```
const int N = 20;                    // Numero de procesos.

void P() {
   mensaje msg;
   while(1) {
      receive(exmut, &msg);          // Recepción BLOQUEANTE
      <Sección crítica>
      send(exmut, msg);                          // Envío no bloqueante
      // Resto del código de P.
   }
}

void main() {
   crear_buzon(exmut);
   /* Envío de mensaje vacío para que el primer proceso pueda
      entrar en su sección crítica. */
   send(exmut, NULL);
   // Genera N procesos P().
   genera_P(N);
}
```

Prod. / Cons. - Paso de Mensajes

Productor / Consumidor Buffer Limitado.

```
const int capacidad = N;

void productor() {
   mensaje msg;
   while(1) {
      receive(p_producir,&msg);
      msg = producir();
      send(p_consumir,msg);
   }
}

void main() {
   crear_buzon(p_producir); crear_buzon(p_consumir);
   for (int i = 1; i <= N; i++)
      send(p_producir,NULL);
   pid_t pid = fork();
   switch(pid) {
      case -1: printf("Error, no funcionó fork"); break;
      case 0: consumidor(); break;     // Código del hijo.
      default: productor(); break;     // Código del padre.
}
```

```
void consumidor() {
   mensaje msg;
   while(1) {
      receive(p_consumir,&msg);
      consumir(msg);
      send(p_producir,msg)
   }
}
```

4.1.7. LLAMADA A PROCEDIMIENTOS REMOTOS (RPC).

RPC

Los procedimientos conforman fronteras naturales para la división de los programas en procesos:

- Interfaces propios, puntos de entrada y puntos de salida.
- RPC hace transparentes al programador los sistemas distribuidos.

Aspectos a tener en cuenta:

- Transferencia de control
- Vinculación ó enlace
- Flujo de datos

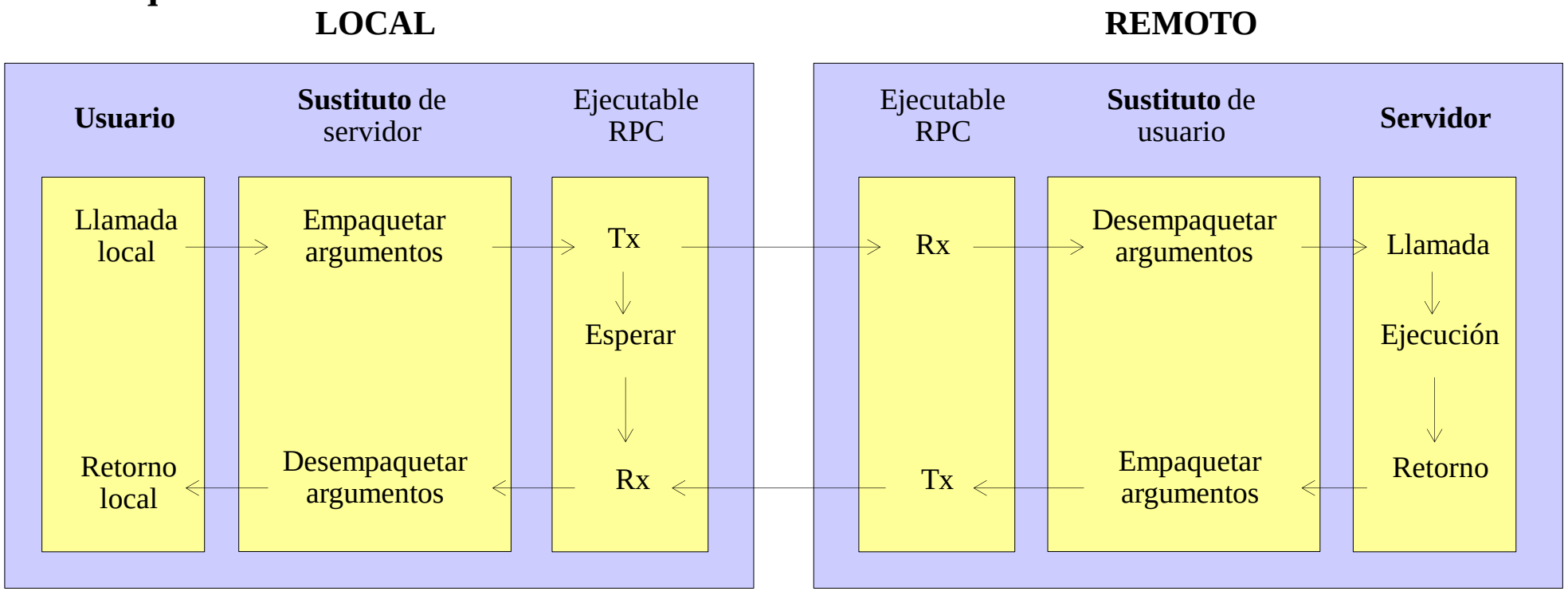
Transferencia de control:

- Suspensión del proceso invocante
- Transferencia de datos al sistema invocado.
- Ejecución del procedimiento remoto
- Devolución de los resultados
- Reanudación del proceso invocante

RPC

Transferencia de control: (sigue)

- **Incorpora el concepto de *sustituto* (stub) de usuario para representar al procedimiento al que se llama de modo remoto.**



RPC

Vinculación:

- Proceso consistente en dos pasos:
- Localización de módulos de componentes
- Resolución de referencias a direcciones para producir la imagen

Flujo de Datos:

- Paso de parámetros:
- El problema es la homogeneidad de los datos. Estándares !!

RPC es un protocolo de comunicación implementado en diversos SSOO y lenguajes de programación