Concurrencia entre Procesos.

Sistemas Operativos
Tema 3.

Procesamiento concurrente.

- Procesamiento concurrente: base de los sistemas operativos modernos (multiprogramados):
  - Un conjunto de procesos que potencialmente pueden ejecutarse concurrentemente (a la vez).
- Problemática:
  - Podemos querer que dos o más procesos cooperen.
  - Que accedan a datos comunes.
- Se debe proporcionar:
  - Mecanismos de sincronización y comunicación entre procesos.
  - Ejecución ordenada

Problemática del procesamiento concurrente.

- Problema básico:
  - Dos o más procesos quieren llevar a cabo una determinada tarea concurrentemente.
  - Pueden llegar a resultados incorrectos.
    - Debido a que no sabemos el orden de ejecución.
    - No sabemos cuando serán interrumpidos y su alternancia en la CPU.
- Ejemplo: Productor-Consumidor ejecutado concurrentemente.
  - ***Productor:*** Genera información y la acumula en un buffer.
  - ***Consumidor:*** Accede al buffer y usa la información.
  - Caso de:
    - Programa de impresión que produce caracteres y consume el manejador de la impresora.
    - Compilador produce código ensamblador y consume el ensamblador.

Problemática del procesamiento concurrente.

- Sincronización:
  - Productor: No debe acceder al buffer si está lleno.
  - Consumidor: No debe acceder al buffer si está vacío.
  - Usan variables comunes.
- Variables compartidas:

variable n;
tipo elemento= (elemento del buffer);
variable buffer: matriz[0..n-1] de elemento;
entrada, salida: 0..n-1;
contador: 0..n;

  - **Entrada, salida:** valor inicial 0.
  - **Entrada:** Siguiente elemento del buffer libre (productor).
  - **Salida:** Primer elemento buffer lleno (consumidor).
  - **Contador:** Número de elementos en el buffer.
  - **Buffer vacío:** Contador = 0.
  - **Buffer lleno:** Contador = n.

Problemática del procesamiento concurrente.

- Código del proceso productor:

```
mientras verdadero hacer:
    ...
    producir un elemento en productor_siguiente;
    ...
    mientras contador= n hacer nada;
    buffer[entrada]= productor_siguiente;
    entrada= (entrada + 1) modulo n;
    contador= contador + 1;
```

- Código del proceso consumidor:

```
mientras verdadero hacer:
    mientras contador= 0 hacer nada;
    consumidor_siguiente= buffer[salida];
    salida= (salida + 1) modulo n;
    contador= contador-1;
    ...
    consumir siguiente elemento en consumidor_siguiente;
    ...
```


Problemática del procesamiento concurrente.

- Por separado son correctas. Concurrentemente pueden no funcionar:
  - Ejemplo:
    - En un momento Contador = 5.
    - Ejecución concurrente productor-consumidor, enunciados "contador = contador +1", "contador = contador-1":

Resultado Contador = 4, 5 ó 6.

- Veamoslo:
  - Ejecución "contador = contador +1" en instrucciones máquina:

P1.- registro A = contador; (Esta en memoria);
P2.- registro A = registro A + 1;
P3.- contador = registro A; (Contenido registro a memoria).

  - Ejecución "contador = contador - 1" en instrucciones máquina:

C1.- registro B = contador; (Esta en memoria);
C2.- registro B = registro B - 1;
C3.- contador = registro B; (Contenido registro a memoria).

Problemática del procesamiento concurrente.

- El orden de esas instrucciones máquina puede ser cualquiera. Ejemplo:

1.-	*P1.-*	*registro A = contador;*	[registro A = 5]
2.-	*P2.-*	*registro A = registro A + 1;*	[registro A = 6]
3.-	*C1.-*	*registro B = contador;*	[registro B = 5]
4.-	*C2.-*	*registro B = registro B - 1;*	[registro B = 4]
5.-	*P3.-*	*contador = registro A;*	[contador = 6]
6.-	*C3.-*	*contador = registro B;*	[contador = 4]

  - O por ejemplo:

5.-	*C3.-*	*contador = registro B;*	[contador = 4]
6.-	*P3.-*	*contador = registro A;*	[contador = 6]

Problemática del procesamiento concurrente.

- Otro ejemplo:
  - Proceso que controla la impresora:

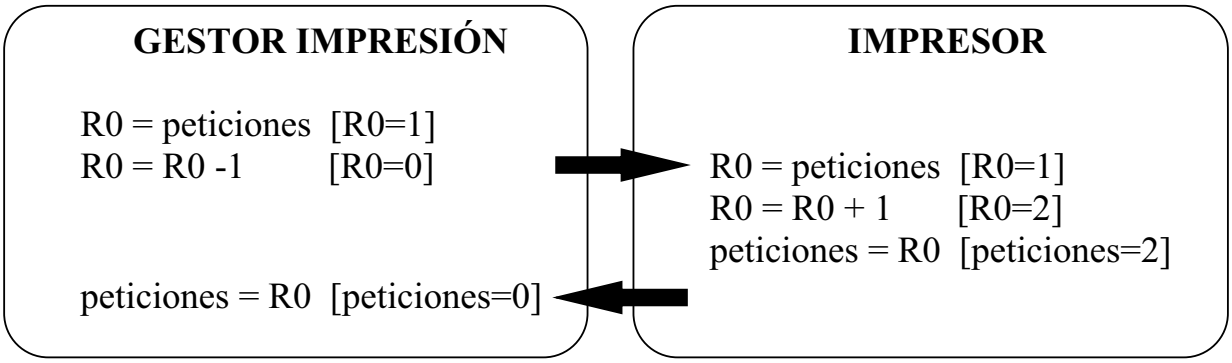
```
mientras verdadero hacer
    si (peticiones > 0)  entonces
        Extraer nombre de fichero de cola_peticiones;
        peticiones = peticiones -1;
```

  - Proceso que imprime un fichero:

```
Introducir nombre de fichero en cola_peticiones
peticiones = peticiones +1;
```

Problemática del procesamiento concurrente.

- Ejecución concurrente: "peticiones = peticiones + 1"
  "peticiones = peticiones - 1"
- Valor inicial peticiones = 1.



- Valor final Peticiones = 0. Debería ser 1.

Problemática del procesamiento concurrente.

- ¿ Dónde está el problema?.
  - Permitir que dos procesos accedan a la misma zona de memoria para cooperar:
    - Variable **contador**.
    - Variable **peticiones**.
  - No conocemos la secuencia de instrucciones.
- A esto se le denomina:
  - Problema de **la sección crítica**.
  - Es un **problema básico** de ejecución **concurrente** de procesos.

Sección crítica.

- Supongamos n procesos que pretenden ejecutarse concurrentemente $\{P_1, P_2, \ldots, P_N\}$.
- **Sección critica** de uno de ellos:
  - Segmento de código en el que un proceso ***modifica***:
    - Variables comunes.
    - Actualiza una tabla común.
    - Escribe en un archivo.
    - Etc.
- Consideraremos un proceso en general como:

mientras verdadero ejecutar

sección de entrada

sección crítica

sección de salida

sección restante

Sección crítica.

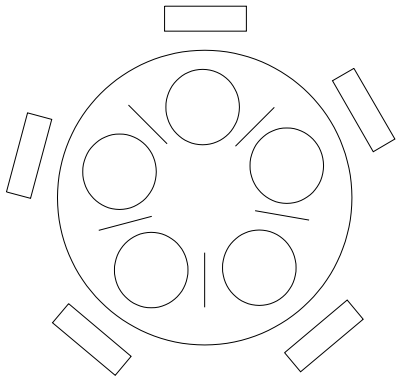
- Una solución debe cumplir tres requisitos:
  - **Exclusión mutua**:
    - No puede haber más de un proceso a la vez en sección crítica.
  - **Progreso:**
    - Ningún proceso esta en sección crítica, pero algunos desean entrar.
    - Sólo los procesos que no estén en sección restante participan en la decisión de quién entra en sección crítica.
    - La decisión debe realizarse en un tiempo finito.
  - **Espera limitada:**
    - Si un proceso quiere entrar en sección crítica, sólo debe esperar un número finito de veces a que otros entren antes que él.
  - **Además:**
    - No hacer suposiciones sobre velocidad relativa de los procesos.
    - No hacer suposiciones sobre el número de procesadores.

Problemas clásicos de sincronización de Procesos.

- Hay una serie de problemas clásicos:
  - Son problemas de sincronización.
  - Se usan para testar soluciones a problemas de control de concurrencia.
- Problema del productor consumidor:
  - Con y sin buffer limitado.
- Problema de los lectores y escritores:
  - Tenemos una **estructura de datos**: un archivo, registro etc.
  - Varios **procesos** quieren leer la estructura (lectores) o **escribir** en ella (escritores).
  - Los procesos se ejecutan de forma concurrente.
  - Variantes:
    - **Prioridad lectores:** Los lectores no deben esperar a que terminen otros lectores porque haya un escritor esperando.
    - **Prioridad escritores:** Si un escritor está esperando ningún lector puede comenzar a leer.
  - Evitar bloqueos indefinidos.

Problemas clásicos de sincronización de Procesos.

- Problema de los filósofos comensales:
  - Tenemos cinco filósofos alrededor de una mesa: **piensan y comen**.
  - Si un filósofo **piensa no molesta** a sus colegas.
  - Si **tiene hambre**:
    - **Coge** los **palillos**, de uno en uno.
    - De los que tiene a su derecha e izquierda.
    - **Si están libres**.
  - Cuando **termina de comer**:
    - Deja los palillos sobre la mesa.
    - Continua pensando.



Problemas clásicos de sincronización de Procesos.

- Posibles soluciones libres de bloqueos mutuos:
  - Sólo dos filósofos como máximo pueden comer al mismo tiempo.
  - Un filosofo coge sus palillos sólo si los dos están disponibles (en sección crítica).
  - Un filósofo impar coge primero el palillo de su izquierda, si es par coge primero el de su derecha.
- Una solución satisfactoria debe garantizar que ningún filosofo muera de inanición (además de bloqueo mutuo).

Soluciones.

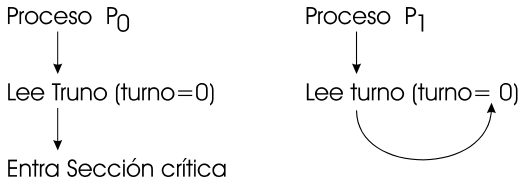
- Veremos algunas soluciones para el problema de sección crítica y de sincronización.
- Hay:
  - Soluciones software:
    - Algoritmos en sección de entrada y salida en cada programa.
  - Soluciones hardware:
    - Uso de instrucciones proporcionadas por algunos procesadores, en sección entrada y salida.
  - Herramientas proporcionadas por los sistemas operativos:
    - Semáforos.
    - Regiones críticas.
    - Regiones críticas condicionales.
    - Monitores.
    - Mensajes.

Soluciones software.

- Problema de la sección crítica con dos procesos P_0, P_1:
  - Alternancia estricta (no satisface el requisito progreso):
    - Uso de variable turno= 0, 1que indica proceso (P_0, P_1) que va a entrar en región crítica.
  - Algoritmo proceso P_0 (P_1):

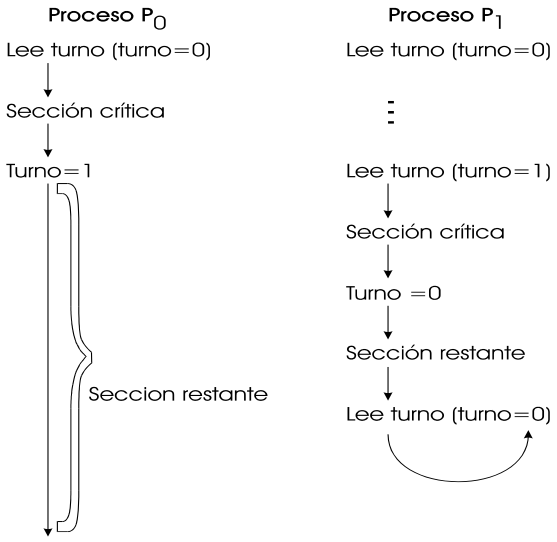
mientras verdadero hacer
mientras turno!= 0 (1) hacer nada
sección crítica
turno= 1 (0)
sección restante

  - Es un algoritmo de espera activa. Ejemplo P_1 esperando:



Soluciones software.

- No satisface el requisito progreso (problemas si un proceso es más rápido que otro):



Soluciones software.

- Otro algoritmo (no satisface el requisito de espera limitada):
  - Uso de variable indicador para saber si un proceso quiere entrar o está en sección crítica.
  - Algoritmo, proceso P_0 (P_1):

inicialmente indicador[0 y 1]=0

mientras verdadero hacer

indicador[0 (1)]=1
mientras indicador[1 (0)] hacer nada

sección crítica

indicador[0 (1)]=0

sección restante

  - No satisface requisito espera limitada, secuencia:

1.- P_0 pone indicador[0]=1
2.- P_1 pone indicador[1]=1 Bucle infinito P_0 y P_1 !!

Soluciones software.

- Algoritmo de Peterson:
  - Cambia las dos ideas anteriores, usa dos variables compartidas: turno e indicador.
  - Algoritmo, proceso P_0 (P_1):

inicialmente indicador[0 y 1]=0 , turno=1 ó 0

mientras verdadero hacer

indicador[0 (1)]=1
turno= 1 (0)
mientras (indicador[1 (0)] y turno=1 (0)) hacer nada

sección crítica

indicador[0 (1)]=0

sección restante

Soluciones software.

- Problema sección crítica para N-procesos P_0, P_1, ... P_N:
  - Algoritmo de la panadería:
    - Cuando llegan los procesos a sección de entrada:
      - Se le da un número.
      - Entran en sección crítica por número de orden.
    - Al principio todos esperan a entrar en sección crítica a que todos tengan número.
    - El algoritmo no garantiza que no hayan dos o más procesos con el mismo número:
      - En dicho caso se decide por el numero de proceso.
    - En el algoritmo están usadas las operaciones:
      - $(a,b) < (c,d)$ si $a < c$ ó si $a = c$ y $b < d$
      - $\max(a_0, a_1, \ldots, a_{n-1})$ numero k tal que $k >= a_i$ para $i=0, \ldots, n-1$

Soluciones software.

- Algoritmo, proceso P_i :

mientras verdadero hacer

```
elección[i]=1
numero[i]=max ((numero[0],numero[1], ..., numero[n-1]) +1)
elección[i]=0
   para  j=0  hasta  n-1  hacer
     mientras eleccion[j] hacer nada
     mientras (numero[j]!=0 y ((numero[j],j) < (numero[i],i))) hacer
                                                              nada
```

sección critica

```
numero[i]=0
```

sección restante

Hardware específico.

- Instrucción TSL: (Test and Set Lock, Evaluar y Asignar).
  - Lee en un registro el contenido de la palabra de una dirección de memoria.
  - Pone en esa dirección un contenido distinto de cero (1).
  - Ejemplo:
    - **tsl registro, indicador**
      - Indicador, variable que está en memoria.
      - Copia el contenido de indicador en registro.
      - Pone el contenido de indicador a 1.
  - **Importancia:**
    - Se trata de una operación hardware, todo lo hace sin interrupción.
    - El procesador que la ejecuta bloquea la ruta de comunicación a la memoria (otro procesador no puede acceder a ella mientras se ejecuta).

Hardware específico.

- **Ejemplo de solución problema sección crítica (no garantiza espera limitada):**
  - Hay una variable indicador compartida, si indicador=0 entra en sección crítica.

mientras verdadero hacer

```
sección_entrada:  tsl registro,indicador  ;indicador a registro, indicador=1
                  cmp registro, #0        ;cero en indicador?
                  jnz seccion_entrada     ;no cero espera
                  ret
```

sección crítica

```
sección_salida:   mov indicador, #0
                  ret
```

sección restante

Hardware específico.

- Instrucción Intercambiar.
  - Intercambia el contenido de dos posiciones de memoria.
  - Ejemplo:
    - **intercambiar hola, indicador**
      - Pone en hola el contenido de indicador.
      - El contenido que **tenía** hola lo sitúa en indicador.
  - **Importancia:**
    - Lo realiza como una única instrucción hardware.
    - Otro procesador no puede acceder a las posiciones de memoria mientras se ejecuta.

Hardware específico.

- **Ejemplo de solución problema seccion crítica (no garantiza espera limitada):**
  - Hasta que una variable cerradura=0 no se entra en sección crítica.

Inicialmente variable compartida cerradura=0,
mientras verdadero hacer

```
seccion_entrada: mov  clave, #1                  ;clave=1
                 intercambiar cerradura,clave    ;clave=cerradura ,
                                                 ;cerradura=1
                 cmp clave, #0                   ;Había 0 en cerradura?
                 jnz seccion_entrada             ;Si había 0 región critica
                 ret
```

sección critica

```
seccion_salida:   mov  cerradura, #0
                  ret
```

seccion restante

Herramientas del sistema operativo.

- El sistema operativo puede ofrecer algunas herramientas de sincronización:
  - Semáforos.
  - Mensajes.
  - Regiones críticas.
  - Regiones críticas condicionales.
  - Monitores.

Semáforos.

- Un semáforo S es una variable entera.
- Se accede a ella:
  - Para la asignación de valores iniciales.
  - Con la operación espera (wait):

    espera(S): mientras S<=0 hacer nada
    S=S - 1
  - Con la operación señal (signal):

    señal(S): S=S+1
  - Estas operaciones se realizan sin interrupción.
  - Se realiza una espera activa.

Semáforos.

- Ejemplo utilización de semáforos:
  - Problema sección crítica con N procesos (sin espera limitada):
    - Semáforo común hola inicialmente a 1.

mientras verdadero hacer

espera(hola)

sección crítica

señal(hola)

sección restante

  - Sincronización E_2 y E_1, código de dos procesos.
    - Semáforo común con valor inicial 0.

Proceso P_1:	Proceso P_2:
E_1	*espera(hola)*
señal(hola)	E_2

Semáforos.

- Semáforos sin espera activa:
  - Podemos definir al semáforo cómo:

```
tipo  semáforo=  registro
                 valor:  entero (valor del semáforo)
                 L:      lista de procesos
                 fin_registro
```

  - Nueva operación bloquear (sleep):
    - Si un proceso la ejecuta se bloquea a sí mismo (pasa a la cola L).
  - Nueva operación despertar(P) (wake-up):
    - Si un proceso la ejecuta cambia el estado del proceso P de bloqueado a listo (cola de procesos listos).

Semáforos.

- Nuevas operaciones espera y señal:

  espera(S): *S.valor= S.valor - 1*
  si S.valor < 0 hacer
  añadir el proceso a L de S
  bloquear

  señal(S): *S.valor= S.valor + 1*
  si S.valor<= 0 hacer
  sacar un proceso P de L de S
  despertar(P)

- Se debe **garantizar** que dos procesos no ejecuten señal y/o espera a la vez en el mismo S.

Semáforos.

- Bloqueo mutuo:
  - Un conjunto de procesos está en estado de bloqueo mutuo cuando cada uno esta esperando un suceso que sólo puede producir otro proceso del conjunto.
  - Ejemplo de bloqueo mutuo por uso de semáforos.
    - Q y S semáforos, sin espera activa inicialmente = 1)

Proceso P0	**Proceso P1**
espera(S)	espera(Q)
espera(Q)	espera(S)
. . .	. . .
señal(S)	señal(Q)
señal(Q)	señal(S)

Semáforos.

- Problema productor - consumidor:
  - Valores iniciales: *vacio = n* , *lleno = 0* , *mutex = 1*.
  - **Proceso productor:**

```
mientras verdadero hacer
        ...   producir un elemento en productor_siguiente
        espera(vacio)
        espera(mutex)
        ...        añadir productor_siguiente a buffer
        señal(mutex)
        señal(lleno)
```

  - **Proceso consumidor:**

```
mientras verdadero hacer
        espera(lleno)
        espera(mutex)
        ...        pasar un elemento de buffer a csiguiente
        señal(mutex)
        señal(vacio)
        ... consumir elemento en consumidor_siguiente
```


Semáforos.

- Problema lectores-escritores prioridad lectores:
  - Valores iniciales *mutex* y *escritura = 1*, *nlectores = 0*.
  - **Proceso escritor:**

```
        espera(escritura)
        ...        se realiza escritura
        señal(escritura)
```

  - **Proceso lector:**

```
        espera(mutex)
            nlectores = nlectores + 1
            si nlectores = 1 entonces espera(escritura)
        señal(mutex)
        ...        se realiza la lectura
        espera(mutex)
            nlectores = nlectores - 1
            si nlectores = 0  entonces señal(escritura)
        señal(mutex)
```

Mensajes.

- Problemas de sincronización:
  - Pueden verse como problemas de comunicación entre procesos.
  - Ejemplo: Productor-consumidor.
- El método de pasos de mensajes permite que los procesos intercambien mensajes.
  - Es una función del sistema operativo, sin utilizar variables compartidas por parte del programador.
  - Facilita dos operaciones:
    - *enviar(mensaje).*
    - *recibir(mensaje).*
  - El sistema operativo proporciona un enlace de comunicación entre dos procesos que se quieren comunicar.

Mensajes.

- Dependiendo de como este implementado:
  - Los mensajes pueden ser: de *longitud fija o variable*.
  - El enlace puede ser:
    - ***Unidirecccional:*** Los procesos conectados no pueden enviar y recibir a la vez.
    - ***Bidireccional.***
    - ***Comunicación directa:*** En enviar y recibir se facilita el nombre del otro proceso.
      - *enviar(P, mensaje) recibir(Q, mensaje)*
    - ***Comunicación indirecta:*** Se utilizan buzones (puertos)
      - Buzón: objeto en el que los procesos pueden colocar mensajes y del que pueden extraerlos.
      - Operaciones:
        - *enviar(A, mensaje)* ; Enviar un mensaje al buzón A.
        - *recibir(A, mensaje)* ; Recibir un mensaje del buzón A.

Mensajes.

- ***Comunicación simétrica:*** Las operaciones recibir y enviar tienen el mismo formato.
- ***Comunicación asimétrica:***
  - Ejemplo asimétrica y directa:
    - *enviar(P,mensaje)* ; Enviar un mensaje al proceso P.
    - *recibir(id, mensaje)* ; Recibir un mensaje de cualquier proceso. ; id variable con nombre del proceso.
- ***Capacidad del enlace:*** Número de mensajes que puede contener temporalmente.
  - *Capacidad cero:*
    - Emisor debe esperar a que el receptor reciba el mensaje.
  - *Capacidad limitada:*
    - Un número n máximo de mensajes acumulados.
    - El emisor no espera mientras que hay espacio para acumular mensajes.

Mensajes.

- Productor-consumidor con paso de mensajes:
  - ***Proceso Productor:***

    mientras verdadero hacer:

    ...
    producir un elemento en productor_siguiente;
    ...
    enviar(consumidor, productor_siguiente)

  - ***Proceso Consumidor:***

    mientras verdadero hacer:

    recibir(productor, consumidor_siguiente)
    ...
    consumir elemento en consumidor_siguiente;
    ...

Regiones críticas.

- Se declara un variable ***v*** de tipo ***T*** compartida:

  variable v: compartida T

- Sólo se accede a ***v*** con un enunciado de región crítica:

  región v hacer S

  – Mientras se ejecuta S ningún otro proceso tiene acceso a ***v*** .

  – Ejemplo:

Proceso P_0	***Proceso P_1***
región v hacer S_1	*región v hacer S_2*

  – La ejecución concurrente de los procesos sería:

  - Secuencialmente primero S_1 y después S_2 o viceversa.

Regiones críticas.

- Implementación región crítica con semáforos:
  – Si *variable v: compartida T.*
  – Un semáforo ***semáforo*** valor inicial 1.
  – Ante una sentencia *región v hacer S* :

  espera(semáforo)
  S
  señal(semáforo)

- Se pueden anidar.
  – Ejemplo: *región x hacer región y hacer S1*
  – Puede dar problemas de bloqueos mutuos:

  Proceso P0: *región x hacer región y hacer S1*
  Proceso P1: *región y hacer región x hacer S2*

  – Con semáforos:

  P0: espera(x1) P1: espera(y1) P0: espera(y1) P1:espera(x1)

Regiones críticas condicionales.

- Se declara un variable ***v*** de tipo ***T*** compartida:

  variable v: compartida T

- El enunciado de región crítica condicional sería:

  región v cuando B hacer S

  – B es una expresión booleana.

  – Cuando se entra en sección crítica el proceso evalúa B:

  - B verdadero:
    – Ejecuta S.
  - B falso:
    – Espera hasta que B sea verdadero y no haya otro proceso ejecutando instrucciones en región crítica. ***v*** .

Regiones críticas condicionales.

- Problema productor-consumidor:

  variable deposito: compartida registro
  buffer: matriz[0..n-1] de elemento
  contador, entrada, salida enteros

  – **Proceso productor:**

```
mientras verdadero hacer:
    ...
    producir un elemento en productor_siguiente;
    ...
    región deposito cuando contador< n hacer
            buffer[entrada]= productor_siguiente;
            entrada= (entrada + 1) modulo n;
            contador= contador + 1;
```

  – **Proceso consumidor:**

```
mientras verdadero hacer:
    región deposito contador> 0 hacer
            consumidor_siguiente= buffer[salida];
            salida= (salida + 1) modulo n;
            contador= contador-1;
    ...
    consumir siguiente elemento en consumidor_siguiente;
    ...
```

Regiones críticas condicionales.

- Otro enunciado región crítica condicional:

  región v hacer *S1*
  esperar (B)
  S2
  fin región

  – B es una condición booleana.
  – Un proceso entra en región y ejecuta S1.
  – Evalúa B:
    - Si B es verdadero ejecuta S2 y sale de región.
    - Si B es falso:
      – Se abandona la exclusión mutua.
      – Espera hasta que B sea verdadero y no haya otro proceso en región ***v*** .

Regiones críticas condicionales.

- Lectores-escritores prioridad escritores:

  variable v: compartida registro
  nlectores, nescritores: entero;
  ocupado: boolean;

  – **Código Lector:**

  región v hacer
  esperar(nescritores = 0)
  nlectores = nlectores + 1
  fin región
  ...
  leer archivo
  ...
  región v hacer nlectores = nlectores - 1

Regiones críticas condicionales.

- Código proceso escritor:

```
región v hacer
        nescritores = nescritores +1
        esperar(( not ocupado) y (nlectores = 0)
        ocupado = verdadero
        fin región
...
escribir archivo
...
región v hacer
        nescritores = nescritores -1
        ocupado = falso
```


Monitores.

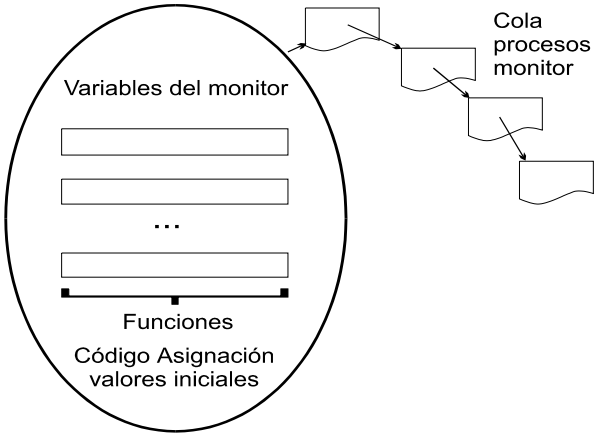
- Es un tipo especial definido por el programador en el que:
  - Hay unas variables locales.
  - Un conjunto de funciones.

```
tipo nombre-monitor = monitor
        declaración de variables
        función P1( ... )
            {  ...  }
        función P2( ... )
            {  ...  }
            ...
        { código de asignación inicial de variables }
```

  - Sólo las funciones definidas en el monitor tienen acceso a sus variables.
  - Un proceso usa el monitor haciendo una llamada a sus funciones.
  - Sólo puede haber un proceso activo en el tipo monitor.

Monitores.

- Esquema de un monitor:



Monitores.

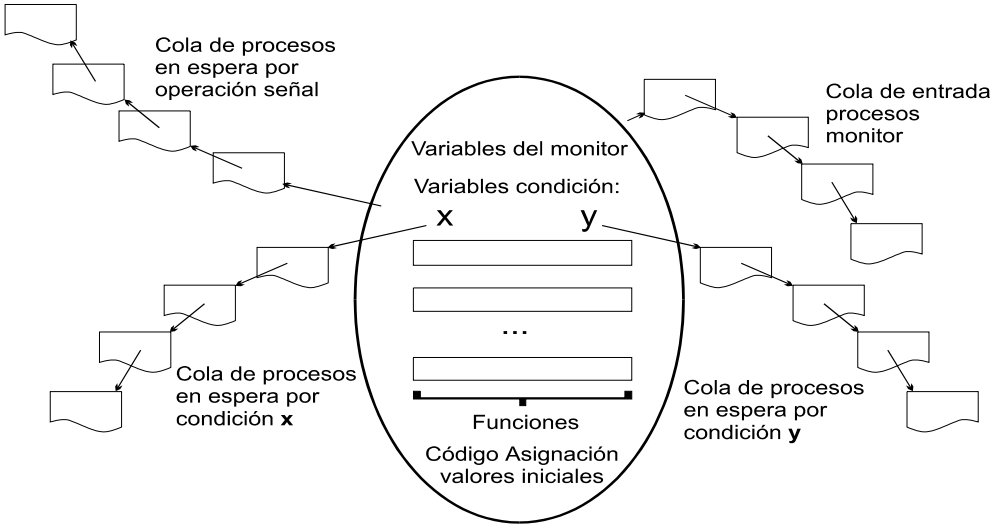
- Monitores con variables condición:
  - Se pueden definir variables de tipo condición:

    variable x, y : condición

  - Sólo se pueden realizar dos operaciones:
    - ***x.espera****:*
      - El proceso que la ejecuta suspende su ejecución.
    - ***x.señal:***
      - Reanuda la ejecución de un proceso Q esperando por *x.espera*.
      - Sino había ningún proceso esperando no tiene ningún efecto.
      - El proceso P que la ejecuta espera a que Q abandone el monitor o espere en otra condición.

Monitores.

- Esquema de un monitor con variables condición:



Monitores.

- Problema productor-consumidor:
  - Usaremos dos variable condición:
    - *lleno:* Para hacer esperar al productor si buffer lleno.
    - *vacío:* Para hacer esperar a consumidor si buffer vacío.
  - Usamos dos funciones:
    - Depositar_buffer: Introducirá un elemento en buffer.
    - Retirar_buffer: Obtendrá un elemento del buffer.
    - Se garantiza acceso en zona crítica.
  - **Código productor:**

    mientras verdadero hacer
    ... producir elemento en productor_siguiente ...
    productor_consumidor.depositar_buffer(productor_siguiente)
  - **Código consumidor:**

    mientras verdadero hacer
    productor_consumidor.retirar_buffer(consumidor_siguiente)
    ... consumir elemento en consumidor_siguiente ...

Monitores.

- Monitor usado:

```
tipo productor_consumidor = monitor
            variable lleno, vacio de condición
            variable contador, entrada, salida: entero
            variable buffer: matriz[0..n-1] de elemento
función depositar_buffer(dato)
            { si contador==n entonces lleno.espera
              buffer[entrada] = dato
              entrada =(entrada + 1) módulo n
              contador = contador +1
              si contador==1 entonces  vacio.señal  }
función retirar_buffer(dato)
            { si contador==0 entonces vacio.espara
              dato = buffer[salida]
              salida = (salida + 1) módulo n
              contador = contador - 1
              si contador==n - 1  entonces lleno.señal    }
{ contador = 0      entrada = 0       salida = 0   }
```


Monitores.

- Problema de los filósofos (sin espera limitada):
  - Se utiliza una variable para cada filósofo, indica su estado:
    variable *estado:* ***matriz[0..4] de*** *(pensando, hambriento, comiendo)*
  - En la solución:
    - El filósofo ***i*** sólo pasa al estado comiendo si sus vecinos no lo están:
      estado[(i+4) módulo 5] != comiendo.
      estado[(i+1) modulo 5] != comiendo.
  - Se utiliza otra variable tipo condición:
    variable *pobrecillo:* ***matriz[0..4] de condición***
    - Para suspender la ejecución de un filosofo hambriento que no puede conseguir los palillos.
  - El código del filósofo ***i*** cuando quiere comer sería:

```
filósofos_comensales.coger_palillos(i)
... comer ...
filósofos_comensales.dejar_palillos(i)
```

Monitores.

- Monitor usado:

```
tipo filósofos_orientales = monitor
    variable estado: matriz[0..4] de (pensando, hambriento, comiendo)
    variable pobrecillo: matriz[0..4] de condición
función coger_palillos(i)
            { estado[i]= hambriento
              evaluar(i)
              si estado[i] != comiendo entonces pobrecillo[i].espera   }
función dejar_palillos(i)
            { estado[i] = pensando
              evaluar((i+4) módulo 5)
              evaluar((i+1) módulo 5)   }
función evaluar(i)
            {si (estado[(i+4) módulo 5] != comiendo) y
                (estado[i] == hambriento) y
                (estado[(i+1) módulo 5] != comiendo) entonces
                                                {estado[i] = comiendo
                                                 pobrecillo[i].señal     }
{para i=0 hasta 4 hacer estado[i] = pensando}
```