Unidad 2:
Gestión de Procesos

Tema 6, Concurrencia:

Exclusión mutua y sincronización.

6.1 Principios generales de concurrencia.

6.2 EM: Soluciones software (Algoritmos de Dekker y Peterson).

6.3 EM: Soluciones hardware.

6.4 Semáforos (Problema del Productor / Consumidor).

6.1 Principios generales de concurrencia.

- Conceptos generales:
  - **Concurrencia:** Existencia simultánea de varios procesos en ejecución.
  - **IMPORTANTE: EXISTENCIA SIMULTÁNEA NO IMPLICA EJECUCIÓN SIMULTÁNEA.**
  - **Paralelismo:** Caso particular de concurrencia.
  - Necesidad de sincronización y comunicación.
    - **Comunicación:** Necesidad de **transmisión de información** entre procesos concurrentes.
    - **Sincronización:** Necesidad de que las ejecuciones de los procesos concurrentes se produzcan según una **secuenciación temporal**, conocida y establecida entre los propios procesos.

6.1 Principios generales de concurrencia.

- Cooperación entre procesos:
  - Dos técnicas de operación:
    - **Sistemas multiprogramados con un solo procesador:** los procesos se intercalan en el tiempo para dar la apariencia de ejecución simultánea.
    - **Sistemas multiprocesador:** además de intercalarse los procesos se superponen.
  - Estas técnicas son ejemplos de procesamiento concurrente y plantean varios problemas:
    - 1. Compartir recursos globales conlleva riesgos. Debemos de ser muy **cuidadosos en el orden** en que dos procesos modifican independientemente el valor de una variable global que ambos usan.
    - 2. Gestionar la asignación óptima de recursos. Debemos de **impedir que se produzcan situaciones de interbloqueo.**
    - 3. **Localizar errores de programación.**

6.1 Principios generales de concurrencia.

- Cooperación entre procesos:
  - Veamos un ejemplo sencillo de un **procedimiento compartido por varios procesos del mismo usuario:**

```
void echo(){
    ent = getchar();
    sal = ent;
    putchar(sal);
}
```

Implementa la función de entrada de un carácter desde teclado.

Cada carácter de entrada se almacena en la variable “ent”, después se transfiere a la variable “sal” y se envía a la pantalla. **Este procedimiento se comparte por varios procesos del mismo usuario de un sistema monoprocesador**, para leer caracteres de teclado y visualizarlos por pantalla.

6.1 Principios generales de concurrencia.

- Cooperación entre procesos:
  - Esta compartición de código permite la interacción entre procesos pero puede dar lugar a problemas, tales como aparecen en la secuencia:

```
/*Proceso P1*/
...
ent = getchar();
...
sal = ent;
putchar(sal);
...
...
```

```
/*Proceso P2*/
...
...
ent = getchar ();
sal = ent;
...
putchar(sal);
...
```

6.1 Principios generales de concurrencia.

- Cooperación entre procesos:
  - P1 llama a echo() y es interrumpido inmediatamente después de leer la entrada.
  - P2 se activa y llama a echo() y lo ejecuta hasta el final mostrando el carácter leído en la pantalla.
  - P1 se reanuda donde lo dejó, sin embargo “ent” contiene un valor diferente al tecleado originalmente, que se transfiere a la variable “sal” y se visualiza.
  - Resultado: Se pierde el primer carácter y el segundo se visualiza dos veces.

6.1 Principios generales de concurrencia.

- Cooperación entre procesos:
  - El **problema** es que **a las variables “ent” y “sal” tienen acceso todos los procesos que llaman al procedimiento echo().**
  - Impongamos la **restricción** de que aunque echo sea un procedimiento global **sólo puede estar ejecutándolo un proceso cada vez**. La secuencia anterior queda ahora:
    - P1 llama a echo() y es interrumpido inmediatamente después de leer la entrada.
    - P2 se activa y llama a echo(). Como P1 estaba usando echo(), P2 se bloquea al entrar al procedimiento y se le suspende mientras espera que quede libre echo().
    - P1 se reanuda y completa la ejecución de echo(). Se visualiza el carácter correcto.
    - Cuando P1 abandona echo() se retira el bloqueo sobre P2. Cuando más tarde se reanude P2 la llamada a echo() se ejecutará con éxito.

6.1 Principios generales de concurrencia.

- Labores del sistema operativo:
  - 1. **Seguir la pista de los distintos procesos activos.**
  - 2. **Asignar y retirar los recursos:**
    - Tiempo de procesador.
    - Memoria.
    - Archivos.
    - Dispositivos de E/S.
  - 3. **Proteger los datos y los recursos físicos.**
  - 4. **Resultados de un proceso independientes** de la velocidad relativa a la que se realiza la ejecución de procesos concurrentes.

6.1 Principios generales de concurrencia.

- Interacción entre procesos:
  - La interacción entre procesos se clasifica de acuerdo al nivel de conocimiento que cada proceso tiene de los demás:
    - **Los procesos no tienen conocimiento de la existencia de los demás:** Independientes, no van a operar juntos, entre ellos se establece una **competencia por los recursos** (disco, archivos, impresora, ...).
    - **Los procesos tienen un conocimiento indirecto de los otros:** No se conocen específicamente unos a otros pero comparten el acceso a algunos objetos como el buffer de E/S. Relación de **cooperación** para compartir.
    - **Los procesos tienen un conocimiento directo de los otros:** Diseñados para **cooperar** trabajando en conjunto en alguna actividad y comunicándose entre ellos.

6.1 Principios generales de concurrencia.

- Competencia entre procesos por los recursos:
  - **Exclusión mutua:** Para que el **acceso a ciertos recursos sea exclusivo de un proceso cada vez**. A la parte del programa que los utiliza se le llama sección crítica.
  - **Sección crítica:**
    - Cada proceso tiene un segmento de código llamado sección crítica.
    - **No está permitido que más de un proceso estén simultáneamente en su sección crítica.**
    - Un protocolo rige la forma de entrar y salir de la sección crítica.

6.1 Principios generales de concurrencia.

- Interbloqueo e inanición:
  - Hacer que se cumpla la exclusión mutua puede dar lugar a dos problemas adicionales:
    - **Interbloqueo:** Considera dos procesos y dos recursos. Supón que cada proceso necesita acceder a ambos recursos para llevar a cabo una parte de su función. Puede suceder que el sistema operativo asigne R1 a P1 y R2 a P2. **Cada proceso está esperando uno de los dos recursos. Ninguno liberará el recurso que posee hasta que adquiera el otro y realice su tarea.**
    - **Inanición:** Supón tres procesos que acceden periódicamente a un recurso. Considera que P1 posee el recurso y que P2 y P3 están esperando. Cuando P1 haya ejecutado su sección crítica tanto P2 como P3 podrán solicitar el recurso. Supón que le le concede el acceso a P3 y que, antes de que termine su sección crítica, P1 solicita acceso de nuevo y así sucesivamente, **se puede llegar a una situación en la que P2 nunca accede al recurso.**

6.1 Principios generales de concurrencia.

- El problema de la sección crítica:
  - Cualquier solución al problema de la sección crítica debe satisfacer los tres requisitos:
    - **Exclusión Mutua:** Sólo un proceso ejecuta simultáneamente su sección crítica.
    - **Progreso:** Cuando ningún proceso ejecuta su sección crítica, algún proceso que lo solicite podrá entrar utilizando un protocolo, que impida la entrada simultánea de varios. La decisión de quién entra no se puede posponer indefinidamente.
    - **Espera limitada:** Ningún proceso debe esperar ilimitadamente la entrada en la sección crítica.

6.1 Principios generales de concurrencia.

- Requisitos para la Exclusión Mutua:
  - Cualquier solución que de soporte a la EM debe de cumplir los requisitos:
    - 1. **Sólo un proceso** debe tener permiso para **entrar en la sección crítica** por un recurso en un instante dado.
    - 2. Un **proceso que se interrumpe** en una sección no crítica debe hacerlo **sin interferir con los otros procesos.**
    - 3. No pueden permitirse **el interbloqueo o la inanición.**
    - 4. Cuando ningún proceso está en su sección crítica, **cualquier proceso que solicite entrar en la suya debe poder hacerlo sin dilación.**
    - 5. **No** hacer **suposiciones sobre la velocidad relativa** de los procesos o el número de procesadores.
    - 6. Un proceso permanece en su sección crítica sólo por un **tiempo finito.**

6.1 Principios generales de concurrencia.

- Requisitos para la Exclusión Mutua:
  - Hay tres categorías de soluciones para implementar la Exclusión Mutua:
    - **Soluciones por software** en los procesos.
    - **Soluciones por hardware.**
    - **Soluciones por software en el sistema operativo**.
  - La estructura general de cualquier mecanismo para implementar la sección crítica es la siguiente:

entrada de la sección crítica;
código (programa) de la sección crítica;
salida de la sección crítica;

6.2 Exclusión Mutua: Soluciones software.

- Algoritmo de Dekker (alternancia estricta):
  - **Idea de partida:** No se puede acceder simultáneamente (desde varios procesadores) a ninguna localidad de memoria.
  - **Espera activa:**
    - Se utiliza una **variable global "turno" para controlar la entrada** en la sección crítica.
    - Cualquier proceso que desee entrar en su sección crítica **comprobará primero el valor de la variable "turno":**
      - Mientras sea distinto al de su identificador de proceso debe esperar.
      - Cuando coincida entrará en la sección crítica.
    - Cuando un proceso sale de la sección crítica, **actualiza "turno"** con el código de otro proceso en espera.

6.2 Exclusión Mutua: Soluciones software.

- Algoritmo de Dekker (1er intento):

```
Proceso 0                    Proceso 1
...                          ...
/*esperar*/                  /*esperar*/
while (turno!=0);            while (turno!=1);
/*sección crítica*/          /*sección crítica*/
...                          ...
turno=1;                     turno=0;
...                          ...
```

Diseñados para poder **pasar el control de ejecución entre ellos**, no es una técnica apropiada para dar soporte al procesamiento concurrente.

Cuando hay diferencias grandes de velocidad es el proceso más lento el que marca el ritmo.

6.2 Exclusión Mutua: Soluciones software.

- Algoritmo de Dekker (2º intento):
  - Verifico la sección crítica del otro proceso y **coloco una señal al entrar y salir en su sección crítica**:

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

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

6.2 Exclusión Mutua: Soluciones software.

- Algoritmo de Dekker (2º intento):
  - Se solucionan los problemas anteriores, sin embargo, ahora surgen otros nuevos:
    - Si **uno de los procesos falla** dentro de su sección crítica **el otro quedará bloqueado permanentemente.**
    - **No se garantiza la Exclusión Mutua** como vemos en la secuencia:
      - 1. P0 ejecuta el while y encuentra señal[1] a falso.
      - 2. P1 ejecuta el while y encuentra señal[0] a falso.
      - 3. P0 pone señal[0] a cierto y entra en su sección crítica.
      - 4. P1 pone señal[1] a cierto y entra en su sección crítica.
    - **Ambos procesos están en su sección crítica**, esto se debe a que esta solución no es independiente de la velocidad de ejecución relativa de los procesos.

6.2 Exclusión Mutua: Soluciones software.

- Algoritmo de Dekker (3er intento):

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

Una vez que un proceso ha puesto su **señal en cierto**, el otro **no puede entrar a su sección crítica** hasta que el primero salga de ella.

Se garantiza la EM, sin embargo, **se generará interbloqueo** si ambos procesos ponen su señal a cierto antes de ambos hayan ejecutado el while.

Además, **si un proceso falla** en su sección crítica, **el otro queda bloqueado permanentemente.**

6.2 Exclusión Mutua: Soluciones software.

- Algoritmo de Dekker (4º intento):
  - El **interbloqueo** se había originado porque **cada proceso establecía su señal sin conocer el estado del otro proceso**. Para solucionarlo haremos que los procesos activen y desactiven su señal:

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

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

6.2 Exclusión Mutua: Soluciones software.

- Algoritmo de Dekker (4º intento):
  - Esta solución **garantiza la EM y practicamente evita el IB**, aunque podría darse la secuencia:
    - P0 pone señal[0] a cierto.
    - P1 pone señal[1] a cierto.
    - P0 comprueba señal[1].
    - P1 comprueba señal[0].
    - P0 pone señal[0] a falso.
    - P1 pone señal[1] a falso.
    - P0 pone señal[0] a cierto.
    - P1 pone señal[1] a cierto.
  - Esta situación de bloqueo puede prolongarse mucho tiempo.

6.2 Exclusión Mutua: Soluciones software.

- Algoritmo de Dekker (solución final):
  - **Combinación entre:**
    - **Primer intento**, donde se usa la variable turno (ahora usamos **turno** para **indicar quien tiene prioridad** para entrar a su sección crítica).
    - **Cuarto intento**.
  - **Funcionamiento:**
    - Cuando P0 quiere entrar en su sección crítica pone señal[0]=cierto; y comprueba la señal de P1.
      - Si está a falso, entra inmediatamente.
      - Si está a cierto, consulta turno:
        - Si turno==0; sigue verficando la señal de P1.
        - Si turno==1; desactiva su señal y espera hasta que turno==0;

6.2 Exclusión Mutua: Soluciones software.

- Algoritmo de Dekker (Solución Final):

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

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

6.2 Exclusión Mutua: Soluciones software.

- Algoritmo de Peterson:
  - Más simple que el de Dekker, **garantiza la exclusión mutua**:
    - Cada proceso tiene un turno para entrar en la sección crítica.
    - Si un proceso desea entrar en la sección crítica, debe activar su señal y puede que tenga que esperar a que llegue su turno.
  - **Impide el interbloqueo** ya que si un proceso se encuentra esperando en el “while”, entonces la señal y el turno del otro proceso están activadas. El proceso que está esperando entrará en su sección crítica cuando la señal o el turno del otro se desactiven.

6.2 Exclusión Mutua: Soluciones software.

- Algoritmo de Peterson:

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

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

6.3 Exclusión Mutua: Soluciones hardware.

- Inhabilitación de interrupciones:
  - Un servicio continuará ejecutándose hasta que solicite un servicio del SO o hasta que sea interrumpido.
  - Para garantizar la EM es suficiente con impedir que un proceso sea interrumpido.
  - Se limita la capacidad del procesador para intercalar programas.
  - Multiprocesador:
    - Inhabilitar las interrupciones de un procesador no garantiza la Exclusión Mutua.

6.3 Exclusión Mutua: Soluciones hardware.

- Instrucciones especiales de máquina:
  - Se realizan en un único ciclo de instrucción.
  - No están sujetas a injerencias por parte de otras instrucciones.
  - Ejemplos:

Instrucción Comparar y Fijar:

```
booleano TS (int i){
    if(i==0){
        i=1;
        return cierto;
    }
    else{
        return falso;
    }
}
```

Instrucción Intercambiar:

```
void intercambiar(int registro, int memoria){
    int temp;
    temp=memoria;
    memoria=registro;
    registro=temp;
}
```

6.3 Exclusión Mutua: Soluciones hardware.

- Instrucciones de máquina y exclusión mutua:
  - **Ventajas:**
    - Es **aplicable a cualquier nº de procesos en sistemas con memoria compartida,** tanto de monoprocesador como de multiprocesador.
    - Es **simple y fácil de verificar.**
    - Puede usarse para disponer de **varias secciones críticas**.
  - **Desventajas:**
    - La **espera activa consume tiempo del procesador.**
    - Puede producirse **inanición** cuando un proceso abandona la sección crítica y hay más de un proceso esperando.
    - **Interbloqueo:** de procesos de baja prioridad frente a otros de prioridad mayor.

6.4 Semáforos.

- Introducción:
  - Mecanismo **frente a problemas de concurrencia**.
  - Funcionamiento basado en la **cooperación** entre procesos.
  - **Uso de señales**, podemos obligar a un proceso a detenerse hasta que reciba una señal.
  - Para esta señalización se usan unas variables especiales llamadas semáforos:
    - Para **transmitir una señal** por el semáforo s, los procesos ejecutan la primitiva **signal(s).**
    - Para **recibir una señal** del semáforo s, los procesos ejecutan la primitiva **wait(s).** Si la señal aún no se ha emitido, el proceso es suspendido hasta que se emite la señal.

6.4 Semáforos.

- Introducción:
  - Los **semáforos** se pueden considerar **variables enteras** sobre las que se definen las siguientes operaciones:
    - Un semáforo **puede iniciarse con un valor no negativo**.
    - La operación **wait() reduce el valor del semáforo**. Si el valor se hace negativo, el proceso que la ejecuta se bloquea.
    - La operación **signal() incrementa el valor del semáforo**. Si el valor no es positivo, se desbloquea un proceso.

6.4 Semáforos.

Semáforos:

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

void signal(semaforo s){
    s.contador++;
    if(s.contador<=0){
        quitar el proceso de s.cola;
        pasar el proceso a la cola listos;
    }
}
```

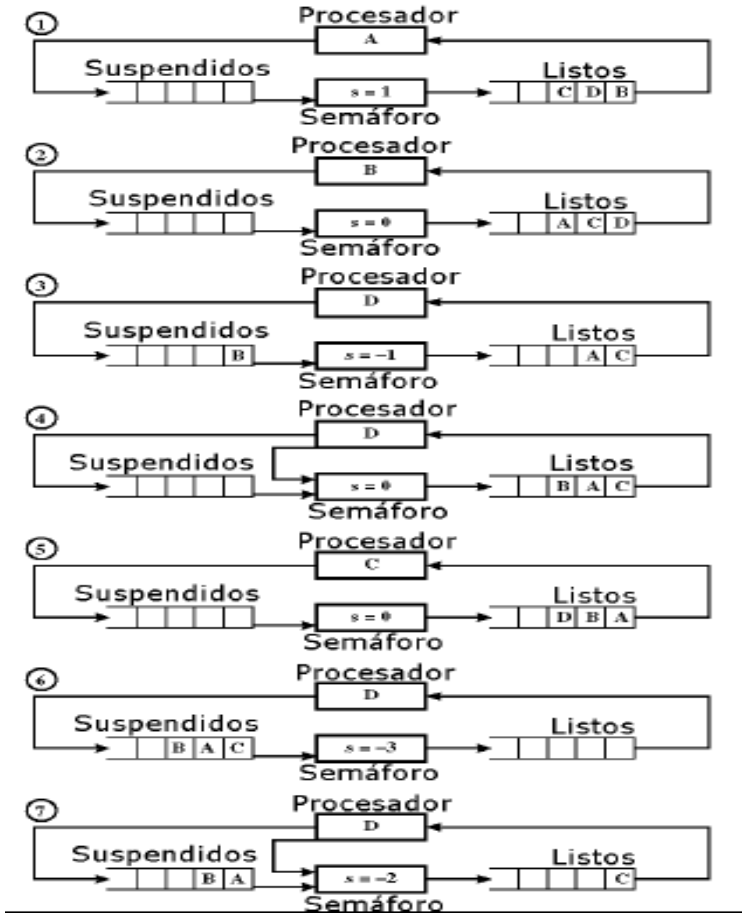
Semáforos binarios:

```
struct semaforoB{
    enum(cero,uno)valor;
    tipoCola cola;
}
void wait(semaforoB s){
    if(s.valor==1)
        s.valor=0;
    else{
        poner este proceso en s.cola;
        bloquear este proceso;
    }
}
void signalB(semaforoB s){
    if(s.cola.esvacia())
        s.valor=1;
    else{
        quitar el proceso de s.cola;
        pasar el proceso de s.cola a la cola de listos;
    }
}
```

6.4 Semáforos.

- En ambos tipos de semáforos **existe una cola que mantiene a los procesos esperando**. El orden en que se retiran los procesos de la cola define **dos categorías de semáforos:**
  - a) **Semáforos robustos:** funcionan como una cola FIFO, el que ha estado bloqueado más tiempo es el que sale de la cola.
  - b) **Semáforos débiles:** no se especifica el orden en el que se retiran los semáforos.

6.4 Semáforos.



6.4 Semáforos.

- Funcionamiento de un semáforo robusto:
  - En la figura se muestra el funcionamiento de un semáforo robusto en una situación en la que los procesos A,B y C dependen de un resultado provisto por el proceso D:
    - 1. A ejecuta , efectúa un wait(), termina y pasa a listos.
    - 2. Entra B a ejecutar, ejecuta un wait() y se suspende.
    - 3. Entra D a ejecutar y efectúa un signal().
    - 4. Debido al signal(), B pasa a listos y s vale cero.
    - 5. D termina y entra C a ejecutar en s=0.
    - 6. C, A y B ejecutan wait() y pasan a suspendidos.
    - 7. D ejecuta, efectúa un signal() y pasa C a listos.

6.4 Semáforos.

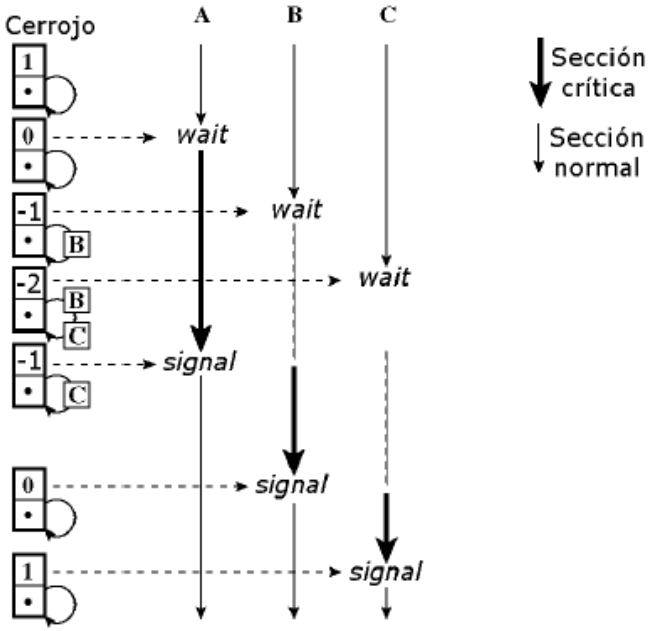
- Exclusión mutua mediante semáforos:

```
/*Programa Exclusión Mutua*/

const int n = /*numero de procesos*/
semaforo s=1;
void P(int i){
   while(cierto){
      /*entrada a la sección crítica*/
      wait(s);
      /*sección crítica*/
      ...
      /*salida de la sección crítica*/
      signal(s);
      /*resto del código*/
   }
}
void main(){
   parbegin(P(1), P(2), ..., P(n));
}
```

6.4 Semáforos.

- Exclusión mutua mediante semáforos:
  - En la siguiente figura se ilustra el funcionamiento de la exclusión mutua mediante un semáforo con tres procesos que intentan acceder a un recurso compartido.



6.4 Semáforos: Problema del Productor/Consumidor.

- Problemas clásicos de sincronización:
  - Problema del productor consumidor.
  - Problema de los lectores escritores.
  - Problema de los filósofos comensales.

- Problema del productor/consumidor:
  - Planteamiento:
    - Uno o más productores generan datos y los sitúan en un buffer.
    - Un único consumidor saca elementos del buffer de uno en uno.
    - Sólo un productor o consumidor puede acceder al buffer en un instante dado.

6.4 Semáforos: Problema del Productor/Consumidor.

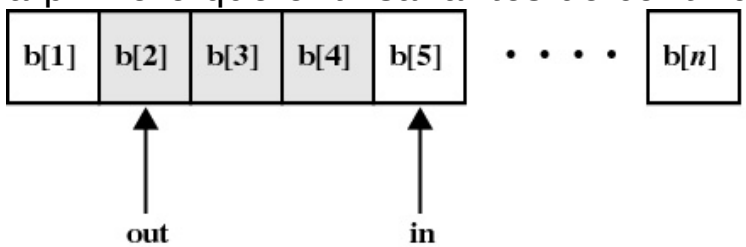
- Problema del productor consumidor (buffer ilimitado):
  - Analizaremos varias soluciones para ilustrar tanto la potencia como los riesgos de los semáforos.
  - Supongamos primero que el **buffer es ilimitado** y que consiste en un vector lineal de elementos. Se pueden definir las funciones productor y consumidor según:

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

```
/*Consumidor*/
while(cierto){
    while(ent<=sal);
    w=b[sal];
    sal++;
    /*consumir el elemento*/
}
```

6.4 Semáforos: Problema del Productor/Consumidor.

- Problema del productor consumidor (buffer ilimitado):
  - El consumidor los puede utilizar sólo cuando están disponibles; es por ello que se comprueba primero que ent>sal antes de continuar.



  - Se debe **garantizar que no accedan al buffer dos agentes simultáneamente**, sobre todo si son productores, pues podrían sobreescribir un dato en el mismo sitio.
  - Esta situación podría producirse si, por ejemplo, un producto es interrumpido después de ejecutar b[ent]=v y otro productor entra a ejecutar.
  - Se puede proponer una solución por medio de semáforos binarios. En este caso, se usa un contador n de elementos en el buffer que reemplaza a ent y sal (n=ent-sal).
  - El semáforo s se usa para hacer cumplir la exclusión mutua.
  - El semáforo retraso se usa para obligar al consumidor a esperar si el buffer está vacío.

6.4 Semáforos: Problema del Productor/Consumidor.

- Problema del productor consumidor (buffer ilimitado):
  - Una solución incorrecta por medio de semáforos binarios.

```
int n;
semaforoB s=1; /*para exclusion*/
semaforoB retraso=0; /*para cons*/
void productor(){
   while(cierto){
      producir(); /*el dato a consumir*/
      waitB(s); /*verificar la EM*/
      añadir();
      n++; /*incrementar cont de dato*/
      if(n==1)
         signalB(retraso); /*desbloq*/
      signalB(s); /*desbloq prod o cons*/
   }
}
```

```
void consumidor(){
   waitB(retraso); /*esperar dato*/
   while(cierto){
      waitB(s); /*si s=0 ponerlo a 1*/
      coger(); /*coger el dato*/
      n--; /*decrementar datos a coger*/
      signalB(s); /*si cola vacia s=1*/
      consumir(); /*consumir el dato cogido*/
      if(n==0)
         waitB(retraso); /*cotejar n*/
   }
}
void main(){
   n=0;
   parbegin(productor, consumidor);
}
```

6.4 Semáforos: Problema del Productor/Consumidor.

- Problema del productor consumidor (buffer ilimitado):
  - El primer waitB ejecutado por el consumidor sirve para evitar que se entre a consumir cuando aún no hay dato.
  - El código if(n==0) waitB(retraso) en el consumidor suspende al proceso cuando se ha vaciado el buffer.
  - El código if(n==1) signalB(retraso) en el productor sirve para desbloquear al consumidor al poner un elemento.

  - Nota: Esta **solución no es correcta** ya que puede ocurrir que cuando un productor genera un nuevo dato mientras el consumidor ha terminado su sección crítica y está ejecutando consumir(), es decir, antes de ejecutar la comprobación sobre n $\Rightarrow$ **el consumidor se ejecuta dos veces para un solo dato generado** (puesto que no hubo correspondencia entre los signalB(retraso) ejecutados por el productor y los waitB(retraso) ejecutados por el consumidor) $\Rightarrow$ n es negativo, lo que indicaría que **se consume un dato que aún no se ha generado.**

6.4 Semáforos: Problema del Productor/Consumidor.

- Problema del productor consumidor (buffer ilimitado):

	Productor	Consumidor	s	n	retraso
0			1	0	0
1	producir()		1	0	0
2	waitB(s)		0	0	0
3	añadir()		0	0	0
4	n++		0	1	0
5	if (n==1) (signalB(retraso))		0	1	1
6	signalB(s)		1	1	1
7		waitB(retraso)	1	1	0
8		waitB(s)	0	1	0
9		coger()	0	1	0
10		n- -	0	0	0
11		SignalB(s)	1	0	0
12	producir()		1	0	0
13	waitB(s)		0	0	0
14	añadir()		0	0	0
15	n++	consumir()	0	1	0
16	if (n==1) (signalB(retraso))		0	1	1
17	signalB(s)		1	1	1
18		if (n==0) (waitB(retraso))	1	1	1
19		waitB(s)	0	1	1
20		coger()	0	1	1
21		n- -	0	0	1
22		signalB(s)	1	0	1
23		if (n==0) (waitB(retraso))	1	0	0
24		waitB(s)	0	0	0
25		coger()	0	0	0
26		n- -	0	-1	0
27		signalB(s)	1	-1	0

6.4 Semáforos: Problema del Productor/Consumidor.

- Problema del productor consumidor (buffer ilimitado):
  - Solución: Introducir una **nueva variable auxiliar que guarde el valor de n en la sección crítica**, para luego compararla fuera de ella.

```
int n;
/*datos sin consumir*/
semaforoB s=1; /*para exclusion*/
semaforoB retraso=0; /*para cons*/
void productor(){
   while(cierto){
      producir();
      /*producir el dato a consumir*/
      waitB(s); /*verificar la EM*/
      añadir();
      n++; /*incrementar cont de dato*/
      if(n==1)
         signalB(retraso); /*desbloq*/
      signalB(s); /*desbloq prod o cons*/
   }
}
```

```
void consumidor(){
    int m; /*variable auxiliar local*/
    waitB(retraso); /*esperar dato*/
    while(cierto){
       waitB(s); /*si s=0 ponerlo a 1*/
       coger(); /*coger el dato*/
       n--; /*decrementar datos a coger*/
       m=n;
       signalB(s); /*si cola vacia s=1*/
       consumir(); /*consumir el dato cogido*/
       if(m==0)
          waitB(retraso); /*cotejar m*/
    }
}
void main(){
    n=0;
    parbegin(productor, consumidor);
  }
```

6.4 Semáforos: Problema del Productor/Consumidor.

- Problema del productor consumidor (buffer ilimitado):
  - Una **solución** más simple y elegante **usando semáforos generales**. La variable **n es un semáforo** y su valor sigue siendo igual al del número de elementos del buffer.

```
semaforo n=0;
semaforo s=1;

void productor(){
   while(cierto){
      producir(); /*el dato a consumir*/
      wait(s); /*verificar la EM*/
      añadir();
      signal(s); /*desbloq proceso*/
      signal(n); /*desbloq consum*/
   }
}
```

```
void consumidor(){
   while(cierto){
      wait(n); /*si no hay dato bloquearse*/
      wait(s); /*entrar en EM*/
      coger(); /*tomar el dato*/
      signal(s); /*salir de zona de exclusion*/
      consumir(); /*consumir el dato cogido*/
   }
}
void main(){
   parbegin(productor, consumidor);
}
```

6.4 Semáforos: Problema del Productor/Consumidor.

- Problema del productor consumidor (buffer limitado):
  - **Planteamiento:**
    - Si un **productor** intenta insertar cuando el **buffer está lleno**, hay que **bloquearlo**.
    - Si un **consumidor** intenta leer cuando el **buffer está vacío**, hay que **bloquearlo**.
    - Si un **productor** está **bloqueado**, hay que **desbloquearlo cuando se consuma un elemento.**
    - Si el **consumidor** está **bloqueado**, hay que **desbloquear al agregar un nuevo elemento.**

6.4 Semáforos: Problema del Productor/Consumidor.

- Problema del productor consumidor (buffer limitado):

```
int tamanyo_de_buffer= /*tamanyo del buffer*/;
semaforo n=0; /*para el cons*/
semaforo s=1; /*para la EM*/
semaforo e = tamanyo_de_buffer;
void productor(){
   while(cierto){
      producir(); /*el dato a consumir*/
      wait(e); /*bloq si no hay huecos*/
      wait(s); /*bloq si hay otro P_C*/
      añadir();
      signal(s); /*salir de la SC*/
      signal(n); /*desbloq consum*/
   }
}
```

```
void consumidor(){
   while(cierto){
      wait(n); /*si no hay dato bloquearse*/
      wait(s); /*bloq si hay otro P_C*/
      coger(); /*tomar el dato*/
      signal(s); /*salir de zona de exclusion*/
      signal(e); /*indicar hueco en el buffer*/
      consumir(); /*consumir el dato cogido*/
   }
}
void main(){
   parbegin(productor, consumidor);
}
```