Programación Avanzada(TC2025)

Tema 5. Programación concurrente

Instituto Tecnológico y de Estudios Superiores de Monterrey. Campus Santa Fe
Departamento de Tecnologías de Información y Electrónica
Dr. Vicente Cubells (vcubells@itesm.mx)

Temario

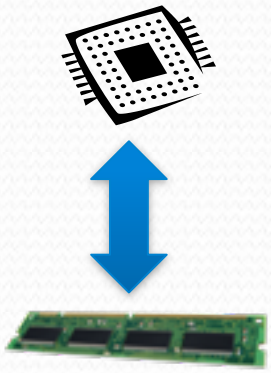
- Fundamentos de la programación concurrente
- Mecanismos de sincronización
  - Exclusión mutua
  - Variables de condición
  - Semáforos y mutexes
  - Monitores
  - Paso de mensajes
  - Barreras

¿Qué es la programación concurrente?

- Notaciones y técnicas de programación usadas para expresar paralelismo potencial y solucionar los problemas de sincronización y comunicación entre procesos
- Concurrencia: Existencia simultánea, no implica ejecución simultánea
- Paralelismo: Existencia simultánea, sí implica ejecución simultánea

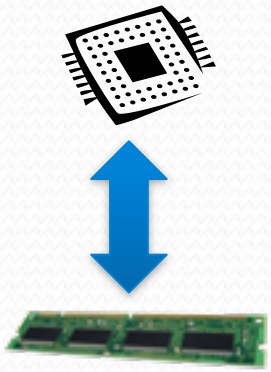
Arquitecturas básicas de programación concurrente

Arquitecturas básicas de programación concurrente

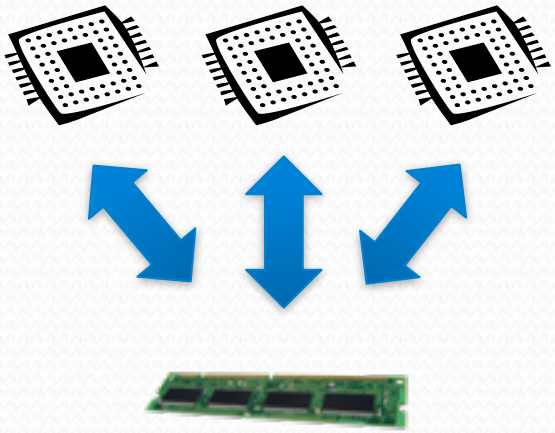


Monoprocesador
CPU alterna instrucciones de distintos procesos

Arquitecturas básicas de programación concurrente

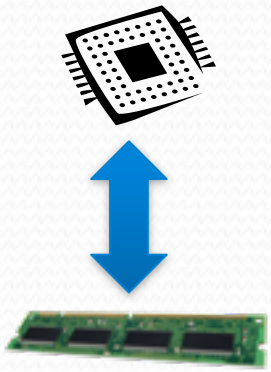


Monoprocesador
CPU alterna instrucciones de distintos procesos

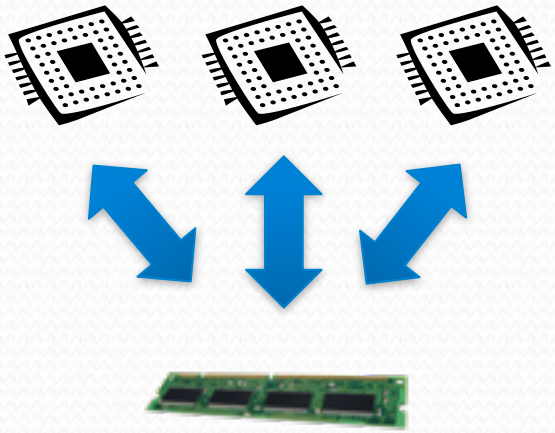


Multiprocesador de memoria compartida
Datos comunes

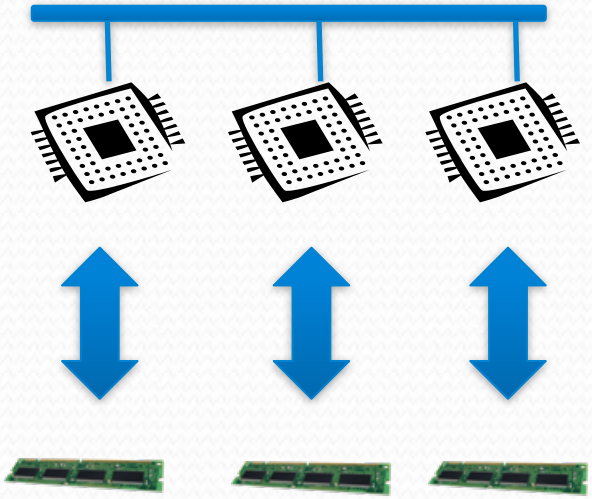
Arquitecturas básicas de programación concurrente



Monoprocesador
CPU alterna instrucciones de distintos procesos

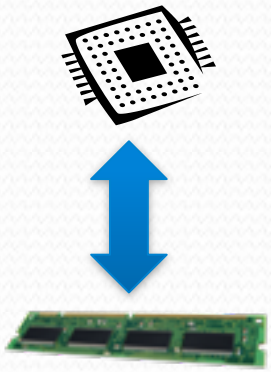


Multiprocesador de memoria compartida
Datos comunes

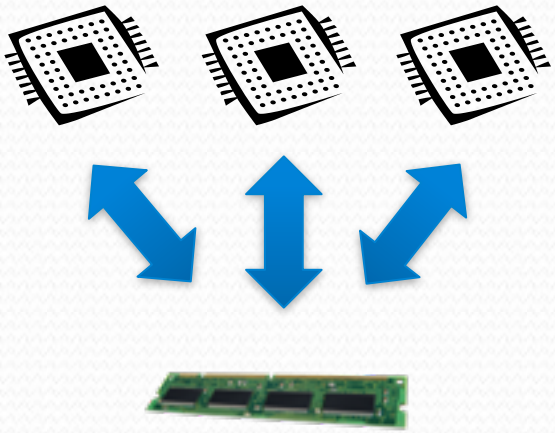


Sistema distribuido
No memoria común
Red de interconexión

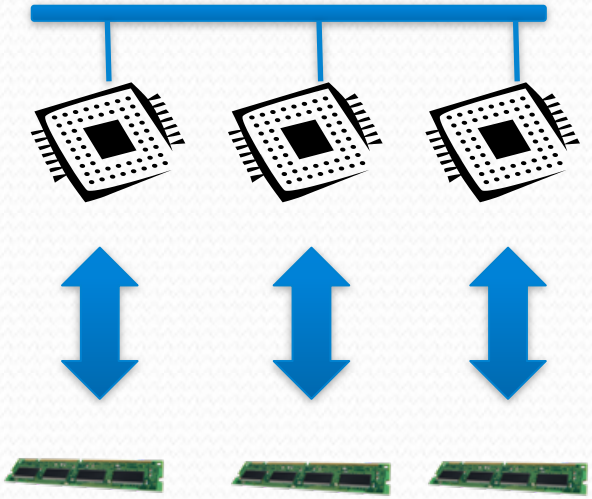
Arquitecturas básicas de programación concurrente



Monoprocesador
CPU alterna instrucciones de distintos procesos



Multiprocesador de memoria compartida
Datos comunes



Sistema distribuido
No memoria común
Red de interconexión

Paralelismo Virtual

Paralelismo Real (o híbrido)

¿Qué se puede ejecutar concurrentemente?

Condiciones de Bernstein

- Sea
  - $L(S_k)=\{a_1,a_2,...,a_n\}$ el conjunto de lectura del conjunto de instrucciones S_k formado por todas las variables cuyos valores son referenciados (se leen) durante la ejecución de las instrucciones en S_k
  - $E(S_k)=\{b_1,b_2,...,b_n\}$ el conjunto de escritura del conjunto de instrucciones S_k formado por todas las variables cuyos valores son actualizados (se escriben) durante la ejecución de las instrucciones en S_k

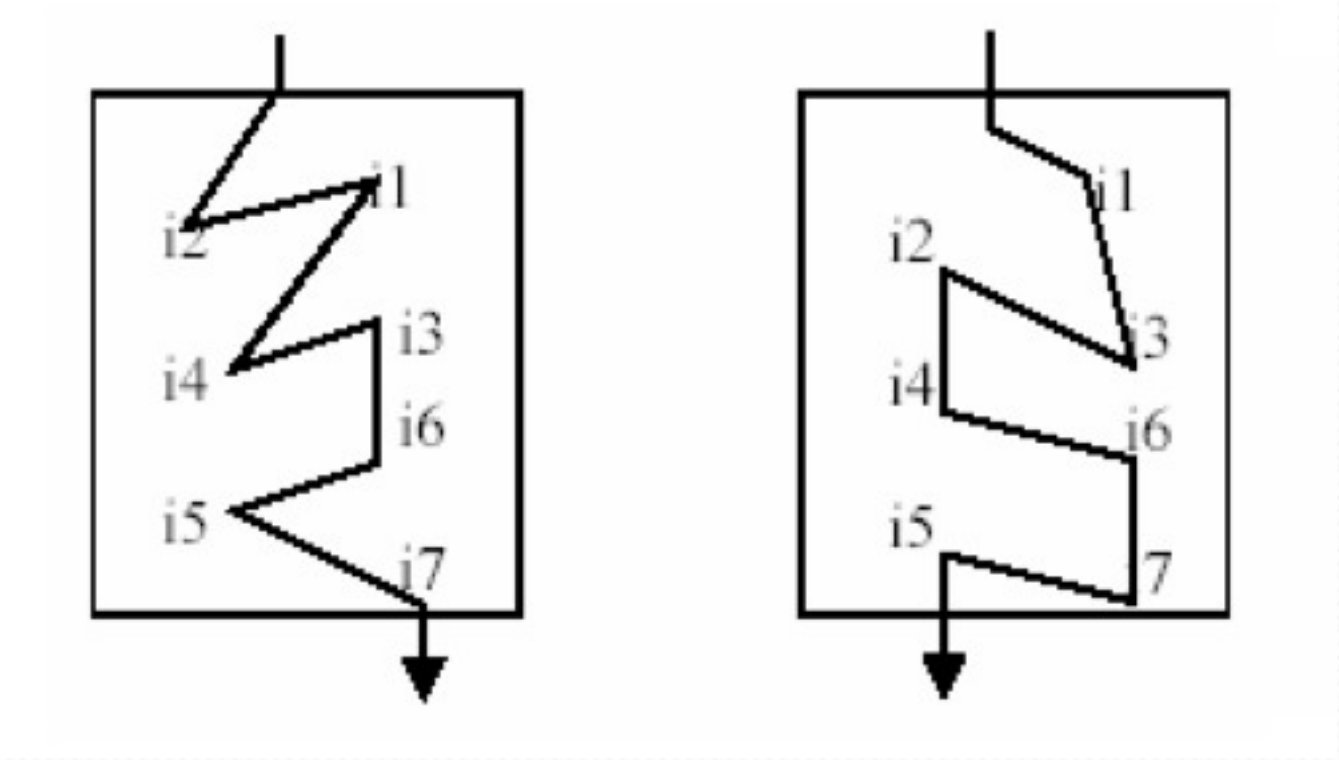
¿Qué se puede ejecutar concurrentemente?

Condiciones de Bernstein

- Para que dos conjuntos de instrucciones S_i y S_j se puedan ejecutar concurrentemente, se tiene que cumplir que:
  - 1.- $L(S_i) \cap L(S_j) = \emptyset$
  - 2.- $E(S_i) \cap L(S_j) = \emptyset$
  - 3.- $E(S_i) \cap E(S_j) = \emptyset$

Características de los sistemas concurrentes

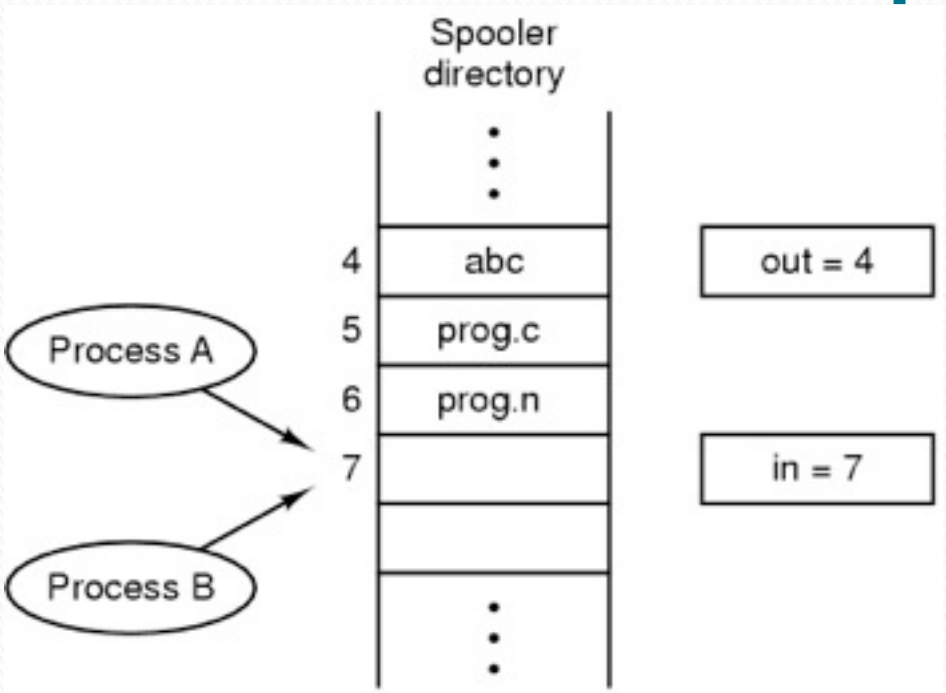
- Indeterminismo



Verificación de programas concurrentes

- **Propiedades de seguridad (safety)**
  - Exclusión mutua
  - Condición de sincronización
  - Interbloqueo (pasivo) – deadlock
- **Propiedades de viveza (liveness)**
  - Interbloqueo (activo) – livelock
  - Inanición – starvation

Condiciones de competencia

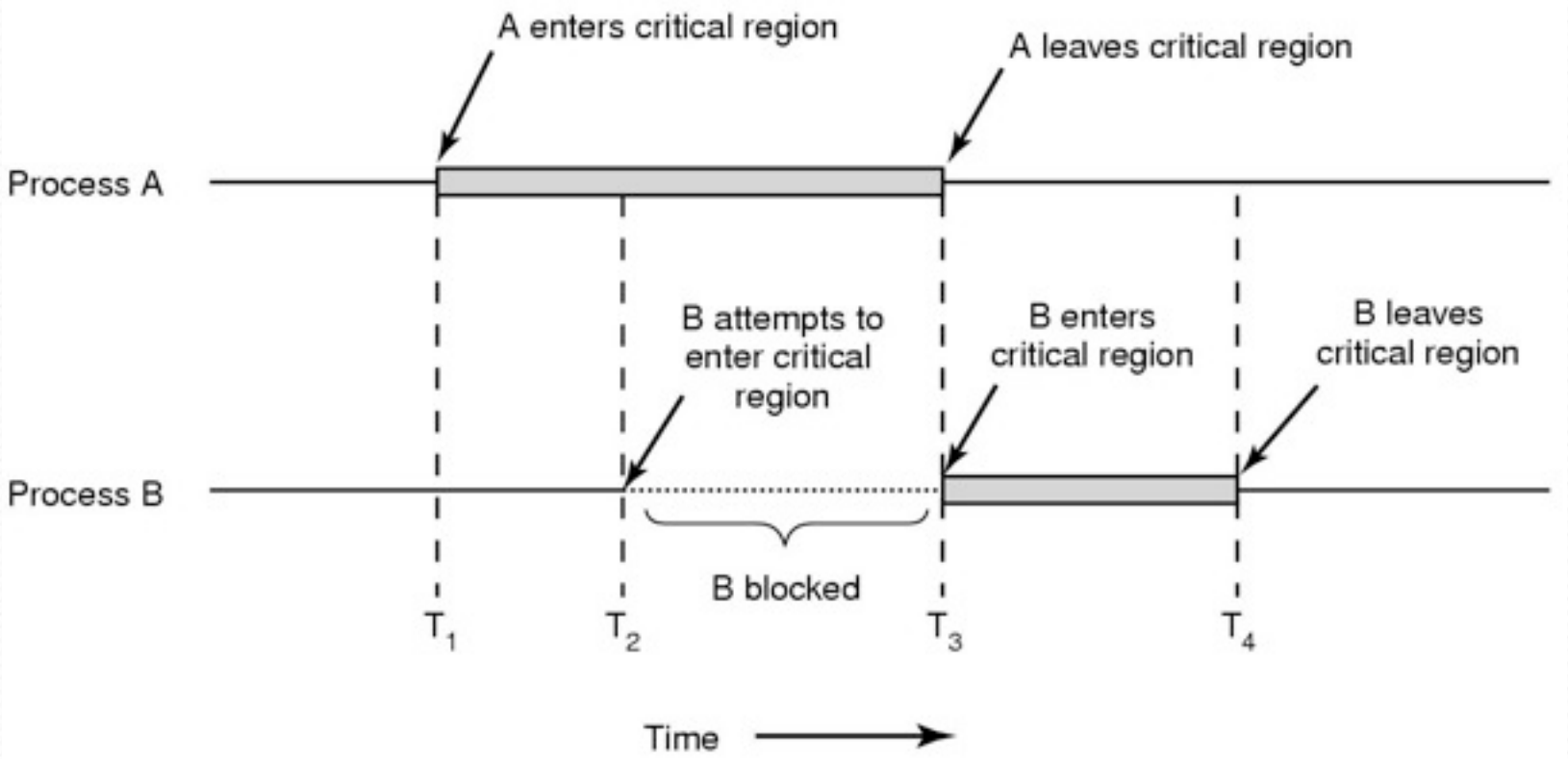


Dos procesos quieren acceder a la memoria compartida simultáneamente

Regiones críticas…

- Exclusión mutua
- Cuatro condiciones para proveerla:
  - Dos procesos no pueden estar al mismo tiempo dentro de sus regiones críticas
  - No se deben hacer asunciones sobre la velocidad o el número de CPU
  - Ningún proceso ejecutándose fuera de su región crítica puede bloquear a otro proceso
  - Ningún proceso debe esperar eternamente para entrar en su región crítica

Regiones críticas



Exclusión mutua usando regiones críticas

Exclusión mutua con espera ocupada…

- Varias soluciones
  - Desactivación de interrupciones
    - Problemas:
      - Control a procesos del usuario
      - Múltiples procesadores
  - Variables de bloqueo
    - Solución por software
    - Una variable compartida que se conmuta su valor
    - Mismo problema que la cola de impresión
      - Ambos procesos entran a su región crítica
    - ¿Incluir un segundo chequeo antes de almacenar el valor de lock soluciona el problema?

Exclusión mutua con espera ocupada…

- Varias soluciones
  - Alternancia estricta
    - Problema: Procesos difieren en velocidad de ejecución
    - Se viola la condición tres

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

(a) Proceso 0

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

(b) Proceso 1

Exclusión mutua con espera ocupada…

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Solución de Peterson para lograr la exclusión

Exclusión mutua con espera ocupada…

- Varias soluciones
  - La instrucción TSL
    - TEST AND SET LOCK
    - Lee el contenido de una palabra de memoria a un registro, almacena un valor <> 0 en dicha dirección de memoria
    - Son operaciones indivisibles
    - Ninguna otra CPU puede acceder

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was non zero, lock was set, so loop
        RET | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET | return to caller
```

Dormir y despertar…

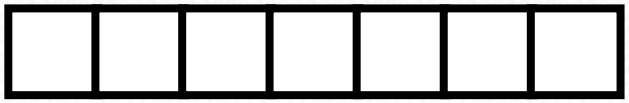
- Peterson y TSL correctas espera ocupada
  - Desperdician tiempo de CPU
  - Imagine dos procesos
    - H de mayor prioridad y L de menor prioridad.
    - H se ejecuta siempre que esté listo
    - En cierto momento, con L en su región crítica, H está listo para ejecutarse (terminó una operación de E/S que lo tenia bloqueado)
    - H comienza una espera ocupada, pero como L nunca es planificado cuando H está en ejecución, L no tiene oportunidad de salir de su región crítica, por lo que H cae en un ciclo infinito
    - Este fenómeno se le conoce como problema de inversión de prioridad
- Una solución: primitivas sleep() y wakeup()

Problema del productor–consumidor...

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Problema del productor–consumidor

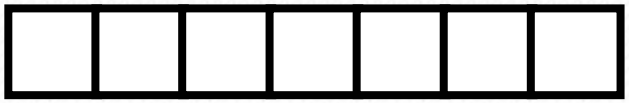
- ¿Dónde está el problema?
  - count



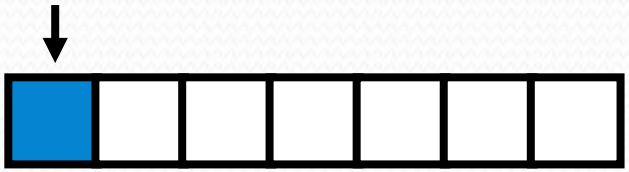
Consumidor lee **count** = 0
Planificador cambia de proceso

Problema del productor–consumidor

- ¿Dónde está el problema?
  - count



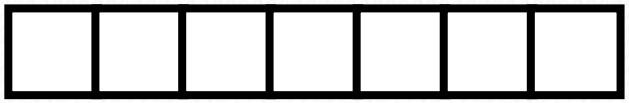
Consumidor lee **count** = 0
Planificador cambia de proceso



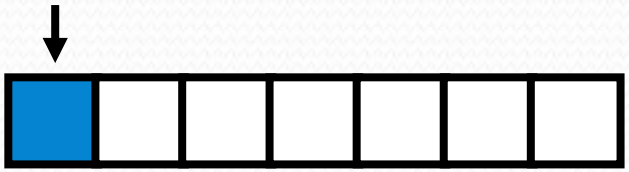
Productor escribe un elemento
count += 1 => **1**
Si **count** = 0 entonces ***wakeup (consumidor)***
Consumidor no dormido, ignora *wakeup*
Planificador cambia de proceso

Problema del productor–consumidor

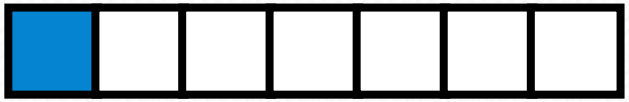
- ¿Dónde está el problema?
  - count



Consumidor lee **count** = 0
Planificador cambia de proceso



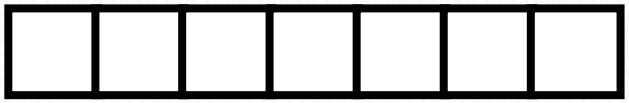
Productor escribe un elemento
count += 1 => **1**
Si **count** = 0 entonces ***wakeup (consumidor)***
Consumidor no dormido, ignora *wakeup*
Planificador cambia de proceso



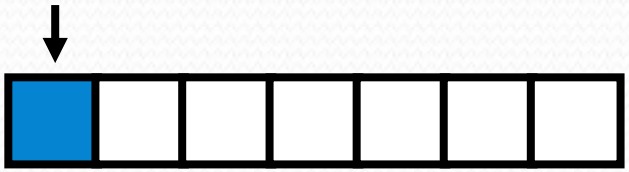
Consumidor había leído **count** = 0, va a ***sleep()***
Planificador cambia de proceso

Problema del productor–consumidor

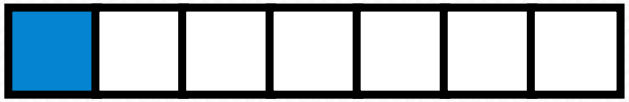
- ¿Dónde está el problema?
  - count



Consumidor lee **count** = 0
Planificador cambia de proceso



Productor escribe un elemento
count += 1 => **1**
Si **count** = 0 entonces ***wakeup (consumidor)***
Consumidor no dormido, ignora *wakeup*
Planificador cambia de proceso

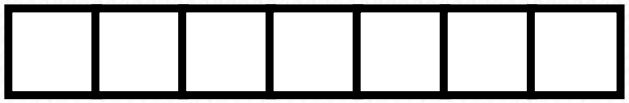


Consumidor había leído **count** = 0, va a ***sleep()***
Planificador cambia de proceso

Como **count** <> 0
Productor llenará almacén, va a ***sleep()***

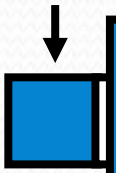
Problema del productor–consumidor

- ¿Dónde está el problema?
  - count

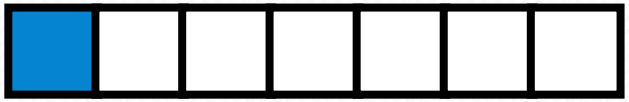


Consumidor lee **count** = 0
Planificador cambia de proceso

Productor escribe un elemento



Solución con un *bit de espera despertar*
Para muchos procesos…



Consumidor había leído **count** = 0, va a ***sleep()***
Planificador cambia de proceso

Como **count** <> 0
Productor llenará almacén, va a ***sleep()***

Semáforos…

- Variable entera para contar el número de despertares almacenados para su uso posterior. Propuesta por E. W. Dijkstra (1965)
- Dos operaciones sobre los semáforos
  - DOWN
    - Si S > 0 entonces S-=1 sino sleep()
    - La operación anterior es una sola operación atómica
    - Necesario para solucionar los problemas de sincronización y evitar condiciones de competencia
  - UP
    - S+=1
    - Si hay durmientes se selecciona uno aleatoriamente para que termine el DOWN

Implementando los semáforos

- Implementando UP y DOWN como llamadas al sistema
  - Es esencial que sean indivisibles

Implementando los semáforos

- Implementando UP y DOWN como llamadas al sistema
  - Es esencial que sean indivisibles
- Dos maneras de garantizar lo anterior son:
  - Desactivar las interrupciones,
    - El código pertenece al sistema
    - No hay riesgo de que se “olvide reactivarlas”
    - Tiempo de ejecución de las mismas es ínfimo
  - Si se usan varias CPU, utilizar la instrucción TSL para garantizar la exclusión mutua,
    - Espera ocupada no es preocupante
      - El tiempo de ejecución de las operaciones es predecible e ínfimo

Solución al problema del productor-consumidor utilizando semáforos…

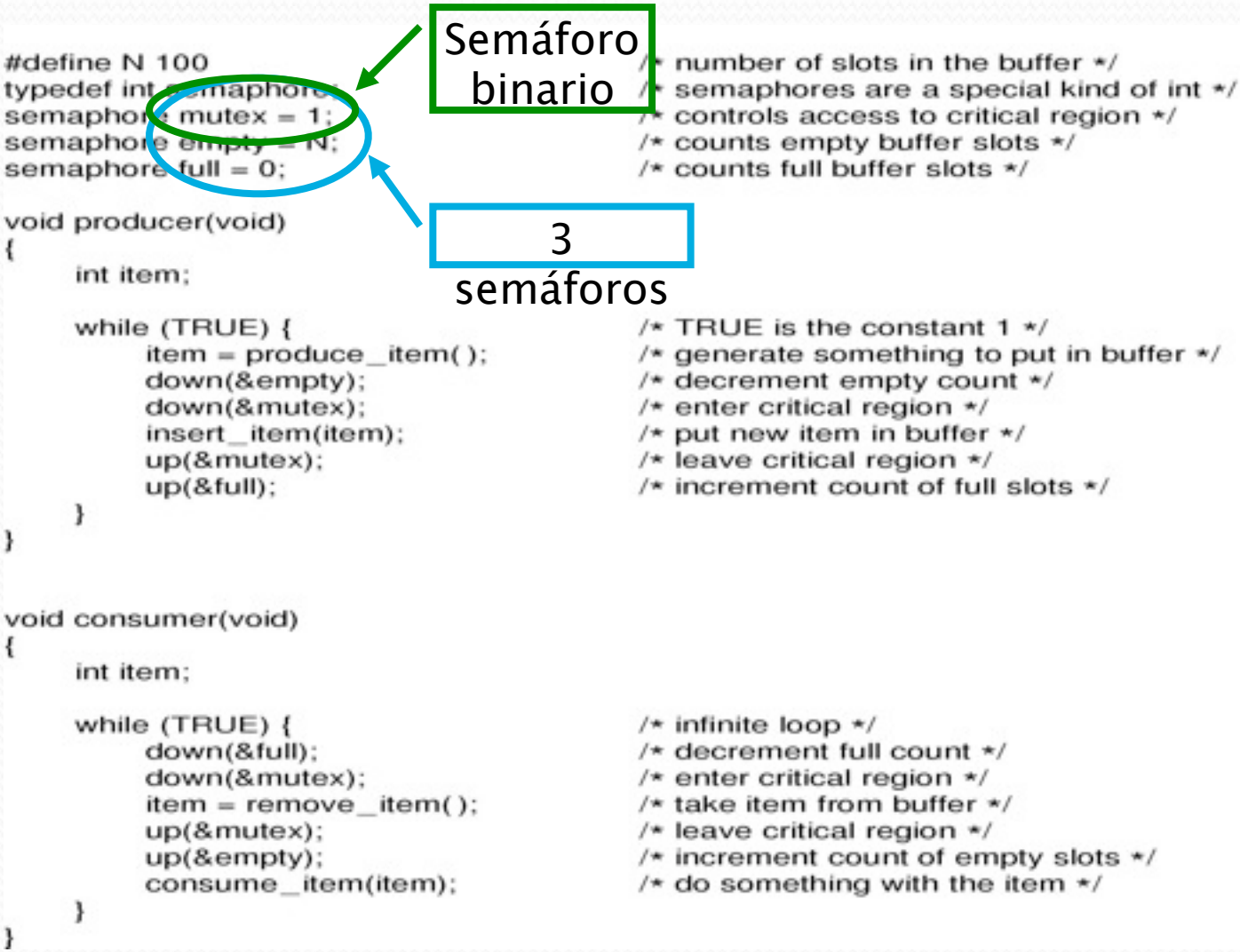
```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

Solución al problema del productor-consumidor utilizando semáforos…

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

3 semáforos

Solución al problema del productor-consumidor utilizando semáforos…



Solución al problema del productor-consumidor utilizando semáforos

Solución al problema del productor-consumidor utilizando semáforos

- Note que la solución emplea los semáforos con dos propósitos diferentes:
  - El semáforo `mutex` se emplea para garantizar la exclusión mutua
  - Los semáforos `full` y `empty` se utilizan para la sincronización
    - Garantizan que el productor se detenga si el buffer está lleno y
    - que se detenga el consumidor si el buffer está vacío

Mutex…

- Versión simplificada de un semáforo
  - Cuando no se necesita contar
  - Solo para exclusión mutua
  - Puede estar en dos estados
    - Bloqueado
    - Desbloqueado
  - Se usan con dos procedimientos
    - mutex_lock: Para entrar en la región crítica
    - mutex_unlock: Al salir de la región crítica

Mutex

```
mutex_lock:
        TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
        CMP REGISTER,#0             | was mutex zero?
        JZE ok                      | if it was zero, mutex was unlocked, so return
        CALL thread_yield           | mutex is busy; schedule another thread
        JMP mutex_lock              | try again later
ok:     RET | return to caller; critical region entered


mutex_unlock:
        MOVE MUTEX,#0               | store a 0 in mutex
        RET | return to caller
```

Implementación de ambos procedimientos usando TSL

Monitores...

Monitores…

- Dificultad al utilizar semáforos
  - Invirtiendo las operaciones DOWN en el productor
  - Bloqueo

Monitores…

- Dificultad al utilizar semáforos
  - Invirtiendo las operaciones DOWN en el productor
  - Bloqueo
- Primitiva de sincronización de alto nivel
  - Solo uno de los procesos puede estar activo en cada momento
  - Son construcciones del lenguaje de programación

Monitores

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Ejemplo de un monitor

Paso de mensajes

- Se puede implementar tanto en sistemas distribuidos como en sistemas de memoria compartida de uno o más CPUs
- Normalmente se definen un par de primitivas para lograr la transferencia de mensajes
  - send (destino, mensaje)
  - receive (origen, mensaje)

Pase de mensajes. Aspectos de diseño

Pase de mensajes. Aspectos de diseño

- Retos que enfrenta:
  - Procesos en diferentes máquinas
    - Pérdida de mensajes en la red
    - Mensajes de confirmación (ACK)
  - Direccionamiento e identificación del destino
    - proceso@máquina.dominio
  - Autenticación
    - ¿Me estoy comunicando con el extremo verdadero?
  - Formato de los mensajes
    - Copiar mensajes más lento que semáforos y monitores
    - Limitar tamaño de los mensajes para que quepa en los registros

Pase de mensajes. Sincronización

- El receptor no puede recibir un mensaje hasta que no sea enviado por el transmisor

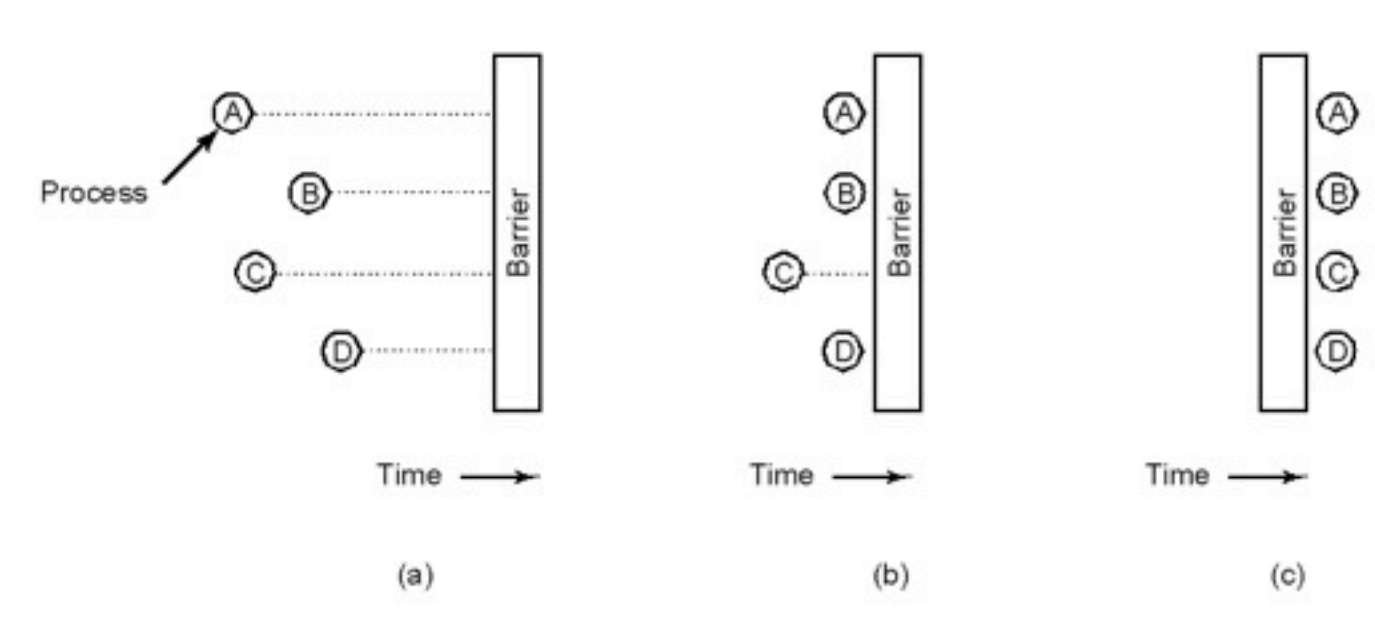
Pase de mensajes. Sincronización

- El receptor no puede recibir un mensaje hasta que no sea enviado por el transmisor
- ¿Qué pasa con un proceso luego de que invoca una primitiva send o receive?
  - enviar bloqueado, recibir bloqueado: Tanto el emisor como el receptor se bloquean hasta que el mensaje es recibido
  - enviar no bloqueado, recibir bloqueado: El emisor puede mandar uno o más mensajes a una variedad de receptores tan rápido como le es posible
  - enviar no bloqueado, recibir no bloqueado

Solución al problema del productor-consumidor con paso de mensajes

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Barreras



- Aplicaciones divididas en fases
- Todos los procesos deben terminar una fase para pasar a la siguiente
- Uso de la primitiva barrier

Resumiendo

- Los programas paralelos dependen completamente del programador
- Concurrencia implica competencia por los recursos
  - Sincronización
- Hay que evitar condiciones de competencia garantizando exclusión mutua
- Varios mecanismos, algunos no son válidos
- Semáforos y mutexes son la solución que emplean muchos SO