Sincronización 4

Sistemas Operativos y Distribuidos

Prof. Javier Echaiz
D.C.I.C. – U.N.S.
http://cs.uns.edu.ar/~jechaiz
je@cs.uns.edu.ar

Sincronización de procesos

Fundamentos

- El acceso concurrente a datos compartidos puede resultar en inconsistencias.
- Mantener la consistencia de datos requiere mecanismos para asegurar la ejecución ordenada de procesos cooperativos.
- Supóngase que se modifica el código del productor-consumidor agregando una variable *contador*, inicializado a 0 e incrementado en cada momento que un nuevo item es agregado al buffer.

Buffer Limitado

- **Datos compartidos** **type** *item* = … ;
  var *buffer* **array** [0..*n*-1] **of** item;
  in, *out*: 0..*n*-1;
  contador: 0..*n*;
  in, *out*, *contador* := 0;
- Proceso Productor

```
repeat
   …
   produce un item en nextp
   …
   while contador = n do no-op;
   buffer [in] := nextp;
   in := in + 1 mod n;
   contador := contador +1;
until false;
```


Buffer Limitado (cont.)

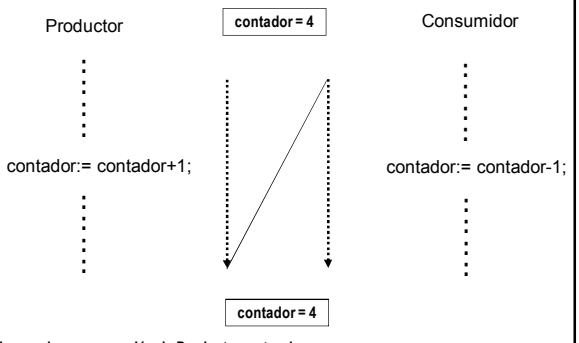
Proceso Consumidor

```
repeat
   while contador = 0 do no-op;
   nextc := buffer [out];
   out := out + 1 mod n;
   contador := contador – 1;
   …
   consume el item en nextc
   …
until false;
```

- Las sentencias:
  - *contador* := *contador* + 1;
  - *contador* := *contador* - 1;

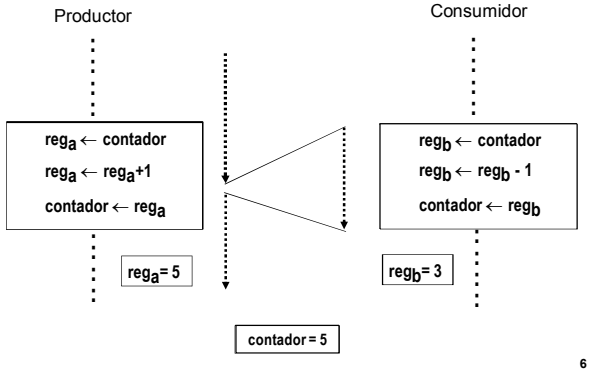
  deben ser ejecutadas *atómicamente*.

El Problema

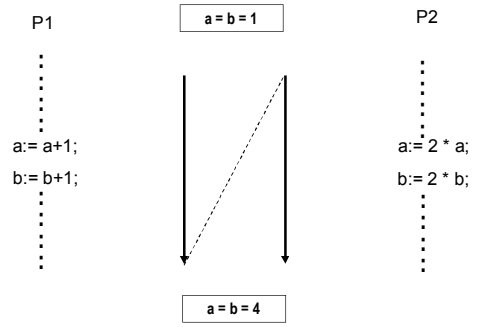


Luego de una operación de Productor y otra de Consumidor … contador permanece invariante

El Problema

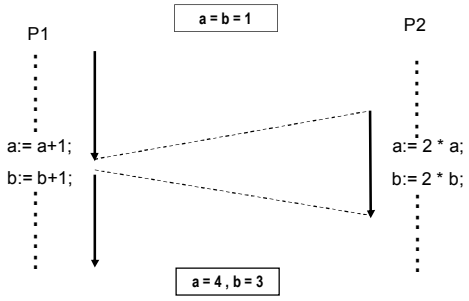


El Problema



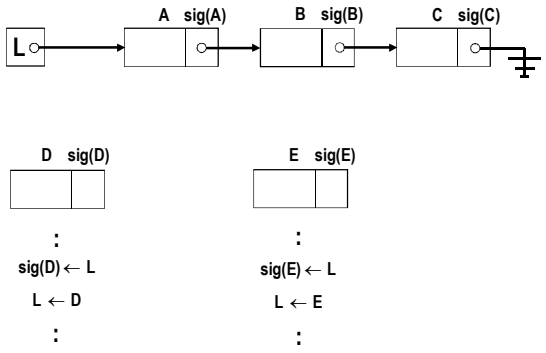
Luego de la ejecución de P1 seguida de P2 los valores de *a* y *b* deben ser iguales entre sí

El Problema

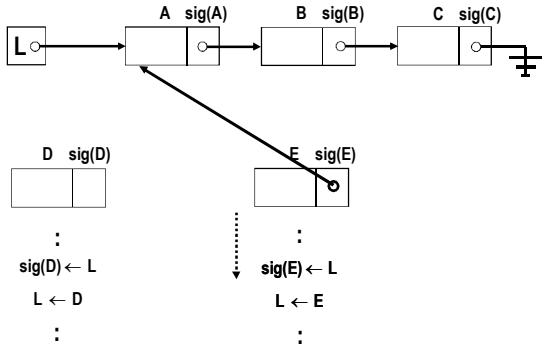


Luego de la ejecución de P1 y P2 los valores de *a* y *b* deberían ser iguales entre sí, *pero no lo son.*

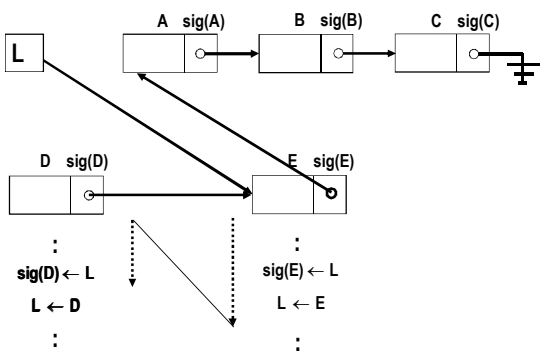
El Problema



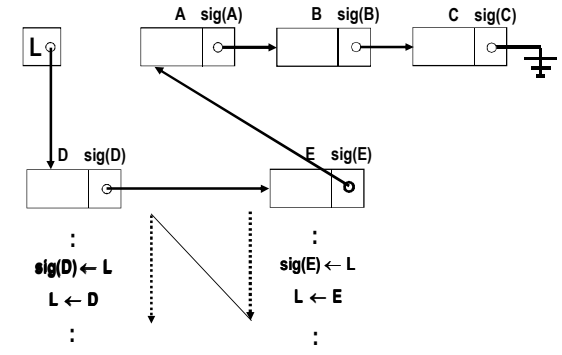
El Problema



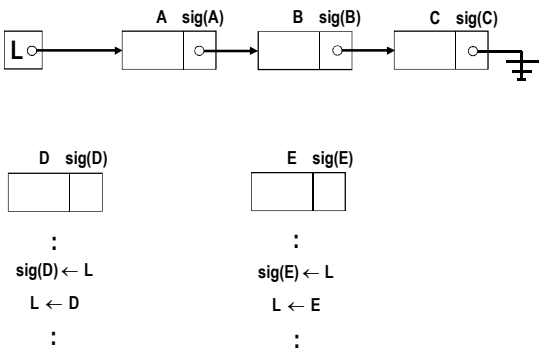
El Problema



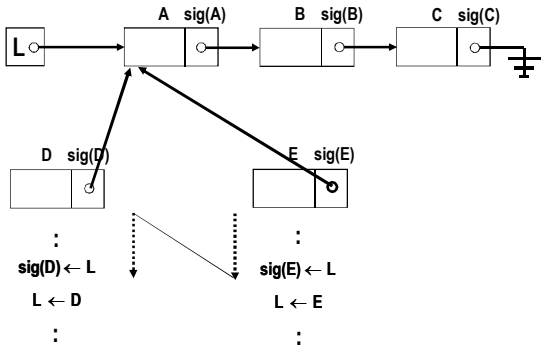
El Problema



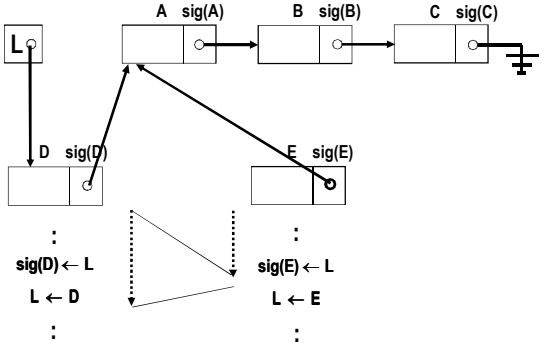
El Problema



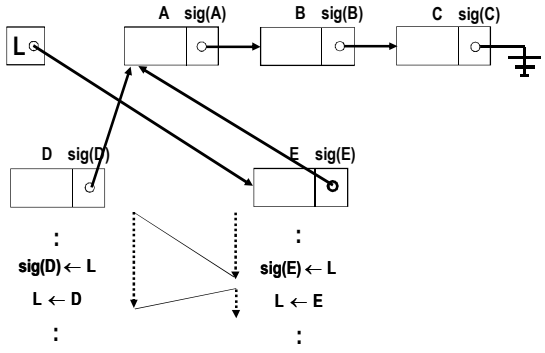
El Problema



El Problema



El Problema



Condición de Carrera

- **Condición de carrera** – Es la situación donde varios procesos acceden y manejan datos compartidos concurrentemente. El valor final de los datos compartidos depende de que proceso termina último.
- Para prevenir las condiciones de carrera, los procesos concurrentes cooperativos deben estar **sincronizados**.

Problema de la Sección Crítica

- *n* procesos todos compitiendo para usar datos compartidos.
- Cada proceso tiene un segmento de código llamado ***sección crítica***, en el cual los datos compartidos son accedidos.
- Problema – asegurar que cuando un proceso está ejecutando en su sección crítica, no se le permite a otro proceso ejecutar en su respectiva sección crítica.

Solución al Problema de la Sección Crítica

1. **Exclusión Mutua**. Si el proceso P_j está ejecutando en su sección crítica, entonces ningún otro proceso puede estar ejecutando en sus secciones críticas.
2. **Progreso**. Si ningún proceso está ejecutando en su sección crí-tica y existen procesos que desean entrar en sus secciones criticas, entonces la selección de procesos que desean entrar en la sección crítica no puede ser pospuesta indefinidamente.
3. **Espera Limitada**. Debe existir un límite en el número de veces que a otros procesos les está permitido entrar en sus secciones críticas después que un proceso ha hecho un requerimiento para entrar en su sección crítica y antes que ese requerimiento sea completado.

- Asuma que cada proceso ejecuta a velocidad distinta de cero.
- No se asume nada respecto a la velocidad relativa de los *n* procesos.

Intentos Iniciales para resolver el Problema

- Sólo 2 procesos, P_0 y P_1
- Estructura general del proceso P_i

repeat
protocolo de entrada
sección crítica
protocolo de salida
resto de sección
until *falso*;

- Los procesos pueden compartir algunas variables comunes para sincronizar sus acciones.

Algoritmo

- Combina las variables compartidas de los algoritmos 1 y 2.
- Proceso P_i

repeat
flag [*i*] := *true*;
turn := *j*;
while (*flag* [*j*] and *turn* = *j*) **do** *no-op*;
sección crítica
flag [*i*] := *false*;
resto de sección
until *false*;

- Alcanza los tres requerimientos; resuelve el problema de la sección crítica para dos procesos.

Algoritmo del "Panadero"

Sección crítica para *n* procesos

- Antes de entrar en su sección crítica, el proceso recibe un número. El poseedor del número mas chico entra en su sección crítica.
- Si los procesos P_i y P_j reciben el mismo número , si $i < j$, entonces P_i es servido primero; sino lo es P_j .
- El esquema de numeración siempre genera números en orden incremental de enumeración;

p.e., 1,2,3,3,3,3,4,5...

Algoritmo del "Panadero"(Cont.)

- Notación orden lexicográfico (ticket #,id proceso #)
  - (a,b) < c,d) si a < c o si a = c y b < d
  - max (a_0,…, a_{n-1}) es un número k, tal que $k \geq a_i$ para i = 0, …, n – 1
- Dato compartido

var *choosing*: **array** [0..*n* – 1] **of** *boolean*;
number: **array** [0..*n* – 1] **of** *integer*,

Las estructuras de datos son inicializadas a *false* y 0 respectivamente.

Algoritmo del "Panadero"(Cont.)

repeat
choosing[*i*] := *true*;
number[*i*] := *max*(*number*[0], *number*[1], …, *number* [*n* – 1])+1;
choosing[*i*] := *false*;
for *j* := 0 **to** *n* – 1
do begin
while *choosing*[*j*] **do** *no-op*;
while *number*[*j*] ≠ 0
and (*number*[*j*],*j*) < (*number*[*i*], *i*) **do** *no-op*;
end;
sección crítica
number[*i*] := 0;
resto de sección
until *false*;

Sincronización por Hardware

- Verifica y modifica el contenido de una palabra atómicamente.

función *Test-and-Set* (**var** target: *boolean*): *boolean*;
begin
Test-and-Set := *target*;
target := *true*;
end;

Exclusión Mutua con Test-and-Set

- Dato compartido : **var** *lock*: *boolean* (*inicialmente false*)
- Proceso P_i

repeat
while *Test-and-Set* (*lock*) **do** *no-op*;
sección crítica
lock := *false*;
resto de sección
until *false*;

Semáforos

- Es una herramienta de sincronización.
- Semáforo *S* – variable entera
- Puede ser accedido solo por dos operaciones indivisibles (atómicas):

wait (*S*): **while** $S \leq 0$ **do** *no-op*;
$S := S - 1$;

signal (*S*): $S := S + 1$;

Ejemplo: Sección Crítica de *n* Procesos

- Variables compartidas
  - **var** *mutex* : *semaphore*
  - inicialmente *mutex* = 1
- Proceso P_i

repeat
wait(*mutex*);
sección crítica
signal(*mutex*);
resto de sección
until *false*;

Implementación del Semáforo

- Se define un semáforo como un registro

type *semaphore* = **record**
valor: *integer*
L: **list of** *procesos*;
end;

- Se supone dos operaciones simples:
  - **block**: suspende el proceso que la invoca.
  - **wakeup(*P*)**: reactiva la ejecución de un proceso bloqueado P.

Implementación (Cont.)

- Las operaciones del semáforo se definen como:

***wait*(*S*):** *S.value* := *S.value* – 1;
if *S.value* < 0
then begin
agregue este proceso a *S*.L;
block;
end;
***signal*(*S*):** *S.value* := *S.value* + 1;
if *S.value* ≤ 0
then begin
remueva un proceso *P* de *S.L*;
wakeup(*P*);
end;

El Semáforo como Herramienta General de Sincronización

- Ejecute *B* en P_j solo después que *A* ejecute en P_i
- Use el semáforo *flag* inicializado a 0
- Código:

P_i	P_j
A	*wait*(*flag*)
signal(*flag*)	*B*

Dos Tipos de Semáforos

- *Semáforos de Cuenta (o contador)* – valor entero sobre un rango de dominio irrestricto.
- *Semáforo Binario* – valor entero que puede valer sólo 0 o 1; suele ser simple de implementar.
- Se puede implementar un semáforo de cuenta *S* como un semáforo binario. Cómo?

Problemas Clásicos de Sincronización

- Problema del Buffer Limitado
- Problema de Lectores y Escritores
- Problema de los Filósofos Cenando

Ver slides adicionales:
Sincronización Extras y ExtrasBook

Monitores

- Es un constructor de sincronización de alto nivel que permite compartir en forma segura un tipo de dato abstracto entre procesos concurrentes.

```
type monitor-name = monitor
    declaraciones de variables
    procedure entry P1 :(…);
        begin … end;
    procedure entry P2(…);
        begin … end;
        
    procedure entry Pn (…);
        begin…end;
    begin
        código de inicialización
    end
```


Monitores (Cont.)

- Se debe declarar una variable de condición para permitir que un proceso espere dentro del monitor:

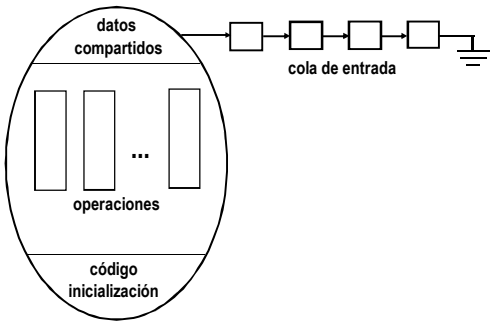
  var *x, y: condition*
- La variable de condición puede ser solamente usada con las operaciones *wait* y *signal*.
  - La operación

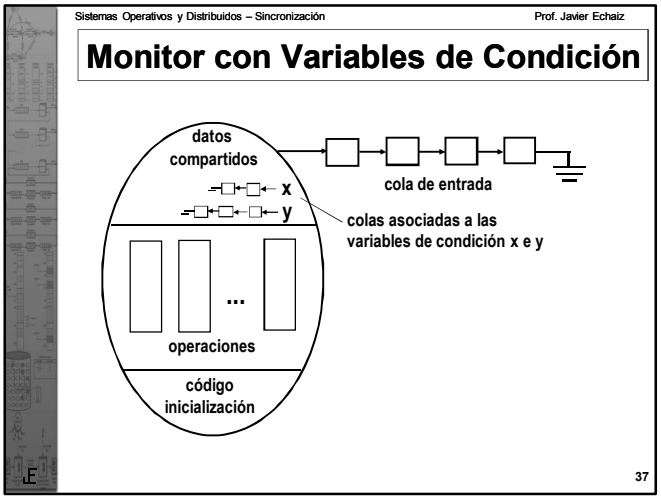
    x.wait;

    significa que el proceso que invoca esta operación es suspendido hasta que otro proceso invoque

    x.signal;
  - La operación *x.signal* reactiva exactamente un proceso suspendido. Si no hay procesos suspendidos, entonces la operación no tiene efecto.

Vista esquemática de un Monitor




Sistemas Operativos y Distribuidos – Sincronización
Prof. Javier Echaiz
Monitor con Variables de Condición
datos compartidos
x
y
cola de entrada
colas asociadas a las variables de condición x e y
...
operaciones
código inicialización
37


Sistemas Operativos y Distribuidos – Sincronización
Prof. Javier Echaiz
Coming Next
Deadlocks
38