

Programación Concurrente

Sistemas Operativos - ITIG

Álvaro Polo Valdenebro

apoloval@gsync.es

Febrero 2009



©2009 GSyC
Algunos derechos reservados.
Este trabajo se distribuye bajo la licencia
Creative Commons Attribution Share-Alike
disponible en <http://creativecommons.org/licenses/by-sa/2.1/es>



- 1 Introducción
- 2 El problema de la sincronización
- 3 Herramientas de sincronización
 - Semáforos
 - Monitores
 - Paso de mensajes
- 4 Transacciones

Contenidos

- 1 Introducción
- 2 El problema de la sincronización
- 3 Herramientas de sincronización
 - Semáforos
 - Monitores
 - Paso de mensajes
- 4 Transacciones

Procesos

- Un proceso es una instancia de un programa en ejecución
- En un sistema **multitarea**, puede haber varios ejecutándose al mismo tiempo
- Esto nos permite...

- Un proceso es una instancia de un programa en ejecución
- En un sistema **multitarea**, puede haber varios ejecutándose al mismo tiempo
- Esto nos permite...
 - ...varios usuarios al mismo tiempo
 - ...varias tareas/servicios al mismo tiempo

Paralelismo y Pseudoparalelismo

- ¿A la vez? ¿Y si solo hay una CPU?

Paralelismo y Pseudoparalelismo

- ¿A la vez? ¿Y si solo hay una CPU?
- Hacemos trampas: **pseudoparalelismo**
 - Cada proceso se ejecuta como si tuviera la CPU para él solo
 - La CPU se reparte en rodajas de tiempo entre los procesos
 - Todo esto es completamente transparente para el código del proceso

Paralelismo y Pseudoparalelismo

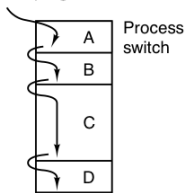
- ¿A la vez? ¿Y si solo hay una CPU?
- Hacemos trampas: **pseudoparalelismo**
 - Cada proceso se ejecuta como si tuviera la CPU para él solo
 - La CPU se reparte en rodajas de tiempo entre los procesos
 - Todo esto es completamente transparente para el código del proceso
- ¿Y si hay varias CPUs?

Paralelismo y Pseudoparalelismo

- ¿A la vez? ¿Y si solo hay una CPU?
- Hacemos trampas: **pseudoparalelismo**
 - Cada proceso se ejecuta como si tuviera la CPU para él solo
 - La CPU se reparte en rodajas de tiempo entre los procesos
 - Todo esto es completamente transparente para el código del proceso
- ¿Y si hay varias CPUs?
 - Lo mismo. Puede haber más procesos que CPUs.

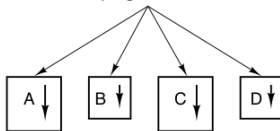
Paralelismo y Pseudoparalelismo

One program counter

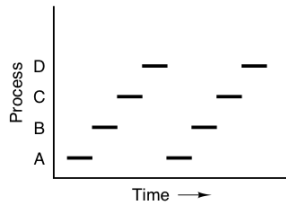


(a)

Four program counters



(b)



(c)

Modelo de procesos

Una vez más, nos abstraemos y olvidamos la CPU

- Un proceso es un programa o tarea en ejecución
- Ejecuta secuencialmente
- No es posible hacer suposiciones sobre el tiempo de ejecución
- El proceso tiene metadatos
 - Contador de programa
 - Puntero de pila
 - Algunos registros de propósito general
 - Otros recursos: memoria, ficheros abiertos, ...

Comunicación de procesos

Los procesos

- Se comunican entre sí
- Se sincronizan

¿Por qué?

Comunicación de procesos

Los procesos

- Se comunican entre sí
- Se sincronizan

¿Por qué?

- Compartir información entre usuarios
- Computación paralela
- Otras razones

Contenidos

- 1 Introducción
- 2 El problema de la sincronización
- 3 Herramientas de sincronización
 - Semáforos
 - Monitores
 - Paso de mensajes
- 4 Transacciones

Ejemplo: productor / consumidor

Comunicación de procesos estilo cliente-servidor

- Un proceso produce datos
- Otro proceso los consume

Declaraciones comunes

```
#define BUFFER_SIZE 64

typedef struct { ... } item_t;

item_t buffer[BUFFER_SIZE];

unsigned int in = 0, out = 0, counter = 0;
```

Ejemplo: productor / consumidor

Productor

```
while (true) {  
    item_t item;  
  
    item = produce_item();  
  
    // Wait until buffer is not full  
    while (counter == BUFFER_SIZE) {}  
  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```



Ejemplo: productor / consumidor

Consumidor

```
while (true) {
    item_t item;

    // Wait until buffer is not empty
    while (counter == 0) {}

    item = buffer[out];
    consume_item(item);

    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```



Ejemplo: productor / consumidor

Supongamos que `counter = 5` y productor y consumidor ejecutan a la vez `counter++` y `counter--`, respectivamente.

`counter++` en ensamblador

```
reg1    = counter
reg1    = reg1 + 1
counter = reg1
```

`counter--` en ensamblador

```
reg2    = counter
reg2    = reg2 - 1
counter = reg2
```

¿Cuanto vale counter?

Ejemplo: productor / consumidor

Podría ocurrir esto

Tiempo	Proceso	Acción	Resultado
T_0	<i>productor</i>	reg1 = counter	reg1 = 5
T_1	<i>productor</i>	reg1 = reg1 + 1	reg1 = 6
T_2	<i>consumidor</i>	reg2 = counter	reg2 = 5
T_3	<i>consumidor</i>	reg2 = reg2 - 1	reg2 = 4
T_4	<i>productor</i>	counter = reg1	counter = 6
T_5	<i>consumidor</i>	counter = reg2	counter = 4

Condiciones de carrera

- Si el resultado de la ejecución depende de quién sea más rápido o más lento tenemos **condiciones de carrera**
- Estas situaciones violan el modelo de procesos
- ¿Soluciones?

El problema de la sección crítica

Pretendemos

- Disponer de n procesos P_0, P_1, \dots, P_{n-1}
- Cada proceso tiene un segmento de código llamado **sección crítica** en el que puede modificar recursos compartidos
- Dos o más procesos no pueden ejecutar su sección crítica al mismo tiempo

```
while (true) {  
    enter_critical_section();  
    { ... } // update shared data  
    exit_critical_section();  
    { ... } // anything else  
}
```

El problema de la sección crítica

Pretendemos proporcionar operadores o protocolos de entrada y salida de la sección crítica con los siguientes requisitos

- **Exclusión mutua.** Si P_i está ejecutando su sección crítica, nadie más ejecutará la suya.
- **Progreso.** Si ningún proceso está en su sección crítica, los procesos que se encuentren en la entrada de sus regiones tendrán la oportunidad de entrar sin esperas indefinidas
- **Espera limitada.** Desde que un proceso solicita entrar en su sección crítica hasta que entra, el resto de procesos solo pueden haber entrado en las suyas un número limitado de veces.

Solución I: alternancia

Alternancia

```
// id and turn are values between 0 and 1
static int turn = 0;

void process (int id) {
    // Wait until my turn
    while (turn != id) {}

    { ... } // critical section

    turn = 1 - id;
}
```

Solución I: alternancia

- ¿Exclusión mutua?
- ¿Progreso?
- ¿Espera limitada?

Solución I: alternancia

- ¿Exclusión mutua? **Sí**
- ¿Progreso? **No, el acceso se alterna**
- ¿Espera limitada? **Sí**

Solución I: alternancia

- ¿Exclusión mutua? **Sí**
- ¿Progreso? **No, el acceso se alterna**
- ¿Espera limitada? **Sí**

Más problemas

- Espera activa
- Solo dos procesos

Solución II: flags

Versión con flags

```
// id is a value between 0 and 1
static bool flag[2];

void process (int id) {
    flag[id] = true;

    // Wait until my turn
    while (flag[1 - id]) {}

    { ... } // critical section

    flag[id] = false;
}
```

Solución II: flags

- ¿Exclusión mutua?
- ¿Progreso?
- ¿Espera limitada?

Solución II: flags

- ¿Exclusión mutua? **Sí**
- ¿Progreso? **No, ¡tenemos interbloqueos!**
- ¿Espera limitada? **Sí**

Solución II: flags

- ¿Exclusión mutua? **Sí**
- ¿Progreso? **No, ¡tenemos interbloqueos!**
- ¿Espera limitada? **Sí**

Más problemas

- De nuevo espera activa
- De nuevo solo dos procesos

Solución III: otra vez flags

Otra versión con flags

```
// id is a value between 0 and 1
static bool flag[2];

void process (int id) {
    flag[id] = true;

    // Wait until my turn
    while (flag[1 - id]) {
        flag[id] = false;
        flag[id] = true;
    }

    { ... } // critical section

    flag[id] = false;
}
```

Solución III: otra vez flags

- ¿Exclusión mutua?
- ¿Progreso?
- ¿Espera limitada?

Solución III: otra vez flags

- ¿Exclusión mutua? **Sí**
- ¿Progreso? **No, ¡ahora tenemos livelocks!**
- ¿Espera limitada? **Sí**

Solución III: otra vez flags

- ¿Exclusión mutua? Sí
- ¿Progreso? No, ¡ahora tenemos livelocks!
- ¿Espera limitada? Sí

Más problemas

- Ooooootra vez espera activa
- Ooooootra vez solo dos procesos

Solución IV: solución de Peterson

Solución de Peterson

```
// id and turn are values between 0 and 1
static bool flag[2];
static int turn;

void process (int id) {
    flag[id] = true;
    turn = 1 - id;

    // Wait until my turn
    while (flag[1 - id] && turn == (1 - id)) {}

    { ... } // critical section

    flag[id] = false;
}
```

Solución IV: solución de Peterson

- ¿Exclusión mutua?
- ¿Progreso?
- ¿Espera limitada?

Solución IV: solución de Peterson

- ¿Exclusión mutua? Sí
- ¿Progreso? ¡Sí, eureka!
- ¿Espera limitada?

Solución IV: solución de Peterson

- ¿Exclusión mutua? Sí
- ¿Progreso? ¡Sí, eureka!
- ¿Espera limitada?Sí, gracias a la cesión del turno

Solución IV: solución de Peterson

- ¿Exclusión mutua? **Sí**
- ¿Progreso? **¡Sí, eureka!**
- ¿Espera limitada?**Sí, gracias a la cesión del turno**

Pero una vez más...

- ¡¡Maldita espera activa!!
- ¡¡Malditos solo dos procesos!!

Cerrosos

Todas estas soluciones tratan de proporcionar lo mismo: **cerrosos**.

Cerrosos

```
void process (int id) {  
    lock();  
    { ... } // critical section  
    unlock();  
}
```

Cerros

- El primero en invocar `lock()` continúa. El resto se bloquean.
- Cuando ese primero invoca `unlock()`, pueden pasar dos cosas
 - Si había procesos bloqueados en `lock()`, se desbloquea uno
 - Si no hay nadie bloqueado, el próximo que llegue continuará sin bloquearse

¿Cómo hacer que `lock()` solo permita un proceso a la vez?

Operadores atómicos

- Necesitamos leer (saber si podemos entrar) e inmediatamente escribir (marcar que hemos entrado)
- Entre la lectura y la escritura, puede haber un cambio de contexto
- Necesitamos que sean una misma **operación atómica**

Inhibición de interrupciones

- ¿Cuándo abandona un proceso la CPU?

Inhibición de interrupciones

- ¿Cuándo abandona un proceso la CPU?
 - Cuando se duerme
 - Su rodaja de tiempo acaba (interrupción hardware)

Inhibición de interrupciones

- ¿Cuándo abandona un proceso la CPU?
 - Cuando se duerme
 - Su rodaja de tiempo acaba (interrupción hardware)
- ¿Y si inhibimos las interrupciones?

Inhibición de interrupciones

- ¿Cuándo abandona un proceso la CPU?
 - Cuando se duerme
 - Su rodaja de tiempo acaba (interrupción hardware)
- ¿Y si inhibimos las interrupciones?
 - Un solo proceso ejecutando en modo kernel (kernel no apropiativo)
 - Nadie le interrumpirá, con lo que lectura+escritura serán atómicas
 - ¿Qué pasa con los sistemas SMP (*Symmetric Multiprocessing*)?

Test-and-Set-Lock

Proporcionamos una operación **atómica**

Test-and-Set-Lock

```
bool testAndSetLock(bool *lock) {  
    bool l = *lock;  
    *lock = true;  
    return l;  
}
```

- En i386, se puede implementar con la instrucción máquina `xchg[bwl]`
- Pertenece al grupo de instrucciones **read-modify-write**

Test-and-Set-Lock

Y la aplicamos para implementar lock()

En la sección crítica

```
bool lock;  
  
void process(int pid) {  
    // Wait until unlocked  
    while(testAndSetLock(&lock)) {}  
  
    { ... } // critical section  
  
    lock = false;  
}
```

Test-and-Set-Lock

En ensamblador i386

```
lock:
    .long $0

testAndSetLockLoop:
    movl    %eax, $1
    xchgl   %eax, lock
    test    %eax, %eax
    jnz     testAndSetLockLoop
    ret

.process
    call    testAndSetLockLoop

    { ... } // critical section

    movl    lock, $0
```

Test-and-Set-Lock

- ¿Exclusión mutua?
- ¿Progreso?
- ¿Espera limitada?

Test-and-Set-Lock

- ¿Exclusión mutua? Sí
- ¿Progreso? Sí
- ¿Espera limitada?

Test-and-Set-Lock

- ¿Exclusión mutua? Sí
- ¿Progreso? Sí
- ¿Espera limitada?¡¡No!!

Test-and-Set-Lock

- ¿Exclusión mutua? Sí
- ¿Progreso? Sí
- ¿Espera limitada?¡¡No!!

Ahora...

- Seguimos teniendo espera activa :-)
- ¡¡Tenemos n procesos!!

Test-and-Set-Lock mejorado

Probamos con...

Entrada en la sección crítica

```
int n; // number of processes
bool waiting[n]; // who is waiting for entering
bool lock; // somebody is inside critical section

void process(int pid) {
    bool key = true;
    int other;

    waiting[pid] = true;
    while (waiting[pid] && key)
        key = testAndSetLock(&lock);
    waiting[pid] = false;

    { ... } // critical section
```

Test-and-Set-Lock mejorado

Salida de la sección crítica

```
{ ... } // critical section

other = (pid + 1) % n;
while (other != pid && !waiting[other])
    other = (other + 1) % n;
if (other == pid)
    lock = false;
else
    waiting[other] = false;
}
```

Test-and-Set-Lock

- ¿Exclusión mutua?
- ¿Progreso?
- ¿Espera limitada?

Test-and-Set-Lock

- ¿Exclusión mutua? Sí
- ¿Progreso? Sí
- ¿Espera limitada?

Test-and-Set-Lock

- ¿Exclusión mutua? Sí
- ¿Progreso? Sí
- ¿Espera limitada?¡¡Sí!!

Test-and-Set-Lock

- ¿Exclusión mutua? Sí
- ¿Progreso? Sí
- ¿Espera limitada?¡¡Sí!!

Finalmente...

- Más espera activa
- Tenemos n procesos

Contenidos

- 1 Introducción
- 2 El problema de la sincronización
- 3 Herramientas de sincronización**
 - Semáforos
 - Monitores
 - Paso de mensajes
- 4 Transacciones

Herramientas de sincronización

- Las soluciones vistas anteriormente son complejas para el programador
- En su lugar, usamos herramientas que ofrecen abstracciones más simples
 - Semáforos
 - Monitores
 - Paso de mensajes

Semáforos

- Variable entera
- Se inicializa a un valor positivo, incluyendo el cero
- Es accedida con dos operadores **atómicos**
 - `wait()`
 - `signal()`

Semáforos

wait()

```
void wait(semaphore s) {  
    while (s <= 0) {}  
    s--;  
}
```

signal()

```
void signal(semaphore s) {  
    s++;  
}
```

Semáforos

Si el valor oscila entre 0 y 1, hablamos de **mutex**

Mutex en sección crítica

```
semaphore s = 1;

void process(int pid) {
    // Wait until unlocked
    wait(s);

    { ... } // critical section

    signal(s);
}
```

Sincronización con semáforos

Ejercicio: sincronizar procesos

- Un proceso A que realiza una tarea que produce un resultado
- Otro proceso B hace algo con dicho resultado
- B tiene que esperar a que A acabe

Sincronización con semáforos

Sincronización con semáforos

```
semaphore s = 0;
```

```
void A(int pid) {  
    produce();  
    signal(s);  
}
```

```
void B(int pid) {  
    wait(s);  
    consume();  
}
```

Semáforos y atomicidad

- `wait()` y `signal()` deben ser operadores atómicos
- ¿Cómo?

Semáforos y atomicidad

- `wait()` y `signal()` deben ser operadores atómicos
- ¿Cómo?
 - Inhibición de interrupciones
 - Read-modify-write

Semáforos y espera activa

- Todas las soluciones que hemos visto hacen **espera activa**
- ¿Cómo eliminarla?

```
wait()
```

```
void wait(semaphore s) {  
    while (s <= 0) {} // spinlock  
    s--;  
}
```

Semáforos y espera activa

Solución: encolamos los procesos

Nuevo tipo semáforo

```
typedef struct {  
    int val;  
    struct process *plist; // FIFO list  
} semaphore;
```

Semáforos y espera activa

wait()

```
void wait(semaphore s) {  
    s->val--;  
    if (s->val < 0)  
        block(s); // block current process  
}
```

signal()

```
void signal(semaphore s) {  
    s->val++;  
    if (s->val <= 0)  
        wakeup(s); // wakeup a process  
}
```

Semáforos y espera activa

Con esto eliminamos la espera activa, pero...

Semáforos y espera activa

Con esto eliminamos la espera activa, pero... ¿qué ha pasado con la atomicidad?

- Antes podíamos recurrir a *read-modify-write*
- Ahora tenemos demasiado código que duerme y despierta procesos
- La lista de procesos bloqueados debe protegerse del acceso concurrente
- El propio código de `wait()` se convierte...

Semáforos y espera activa

Con esto eliminamos la espera activa, pero... ¿qué ha pasado con la atomicidad?

- Antes podíamos recurrir a *read-modify-write*
- Ahora tenemos demasiado código que duerme y despierta procesos
- La lista de procesos bloqueados debe protegerse del acceso concurrente
- El propio código de `wait()` se convierte... ¡en una sección crítica!

Semáforos y espera activa

¿Estamos ante un problema recursivo sin solución?

Semáforos y espera activa

¿Estamos ante un problema recursivo sin solución?

- ¿Por qué descartamos los **spinlocks**?

Semáforos y espera activa

¿Estamos ante un problema recursivo sin solución?

- ¿Por qué descartamos los **spinlocks**?
 - Malgastamos CPU

Semáforos y espera activa

¿Estamos ante un problema recursivo sin solución?

- ¿Por qué descartamos los **spinlocks**?
 - Malgastamos CPU
- ¿Y si es durante poco tiempo (como lo que tardamos en ejecutar `wait()`)?

Semáforos y espera activa

¿Estamos ante un problema recursivo sin solución?

- ¿Por qué descartamos los **spinlocks**?
 - Malgastamos CPU
- ¿Y si es durante poco tiempo (como lo que tardamos en ejecutar `wait()`)?
 - Malgastamos poca CPU

Semáforos y espera activa

¿Estamos ante un problema recursivo sin solución?

- ¿Por qué descartamos los **spinlocks**?
 - Malgastamos CPU
- ¿Y si es durante poco tiempo (como lo que tardamos en ejecutar `wait()`)?
 - Malgastamos poca CPU

Solución:

- El acceso concurrente a la sección crítica de `wait()` lo hacemos con espera activa
- El acceso concurrente a la sección crítica del programa lo hacemos con espera bloqueante

Semáforos

Por fin tenemos

- Exclusión mutua
- Progreso
- Espera limitada
- Espera casi bloqueante
- n procesos

Buffer acotado con semáforos

Ejercicio: implementar el buffer acotado con semáforos

- Un buffer de N posiciones
- Un productor mete datos en el buffer. Si esta lleno, espera.
- Un lector saca datos del buffer. Si está vacío, espera.

Buffer acotado con semáforos

buffer acotado

```
semaphore empty = BUFFER_SIZE, full = 0, mutex = 1;
void producer() {
    wait(empty);
    wait(mutex);
    { ... } // insert an item into buffer
    signal(mutex);
    signal(full);
}
void consumer() {
    wait(full);
    wait(mutex);
    { ... } // extract an item from buffer
    signal(mutex);
    signal(empty);
}
```

Problema de los lectores-escriptores

Ejercicio: implementar un mecanismo de sincronización de tal forma que

- Proteja una sección crítica del acceso de lectores y escritores
- Pueden entrar N lectores a la vez si no hay ningún escritor dentro
- Puede entrar 1 escritor a la vez si no hay ningún lector y ningún escritor dentro

Problema de los lectores-escriptores

Lectores/escriptores

```
sem mutex = 1;
sem wrt = 1;
int readcount = 0;

void writer() {
    wait(wrt);

    // write in critical section
    { ... }

    signal(wrt);
}
```

Lectores/escriptores

```
void reader() {
    wait(mutex);
    readcount++;
    if (readcount == 1)
        wait(wrt);
    signal(mutex);

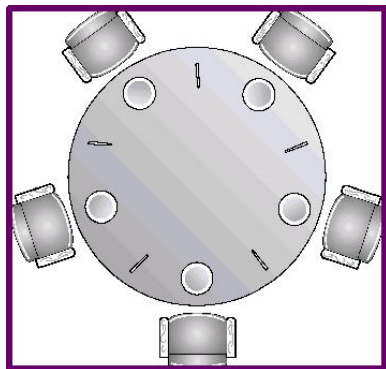
    // read from critical section
    { ... }

    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
}
```

Problema de la cena de los filósofos

Ejercicio: tenemos cinco filósofos cenando pasta alrededor de una mesa

- Solo hay cinco tenedores para comer para comer
- Cuando un filósofo desea comer, ha de usar los dos tenedores adyacentes
- Solo se puede coger un tenedor cada vez
- Si un tenedor está ocupado, no se puede coger



Problema de la cena de los filósofos

La cena de los filósofos

```
semaphore fork[5];

void philosopher(int id) {
    while (true) {
        wait(fork[id]);
        wait(fork[(id + 1) % 5]);

        { ... } // eat

        signal(fork[id]);
        signal(fork[(id + 1) % 5]);
    }
}
```

Problema de la cena de los filósofos

La cena de los filósofos

```
semaphore fork[5];

void philosopher(int id) {
    while (true) {
        wait(fork[id]);
        wait(fork[(id + 1) % 5]);

        { ... } // eat

        signal(fork[id]);
        signal(fork[(id + 1) % 5]);
    }
}
```

!!!No funciona!!!

Problema de la cena de los filósofos

Otro intento

```
enum { HUNGRY, EATING, THINKING } state[5];
semaphore mutex = 1, ready[5] = {0, 0, 0, 0, 0};

void philosopher(int id) {
    while (true) {
        take(id);

        { ... } // eat

        release(id);
    }
}
```

Problema de la cena de los filósofos

Comprobamos si un filósofo puede continuar comiendo

Otro intento

```
void check(int id) {
    if (state[id] == HUNGRY &&
        state[prev(id)] != EATING &&
        state[next(id)] != EATING) {
        state[id] = EATING;
        signal(ready[id]);
    }
}
```

Problema de la cena de los filósofos

Cogemos los tenedores

Otro intento

```
void take(int id) {  
    wait(mutex);  
    state[id] = HUNGRY;  
    check(id);  
    signal(mutex);  
    wait(ready[id]);  
}
```

Problema de la cena de los filósofos

Soltamos los tenedores

Otro intento

```
void release(int id) {  
    wait(mutex);  
    state[id] = THINKING;  
    check(prev(id));  
    check(next(id));  
    signal(mutex);  
}
```

Problema de la cena de los filósofos

Existen más soluciones

- **Usar un camarero** que regule el acceso a los tenedores.
- **Tenedores jerárquicos**, de manera que siempre se comience por el tenedor con el identificador más bajo
- **Solución Chandy-Misra**, basada en paso de mensajes

Monitores

- El uso de semáforos, aunque efectivo, puede conllevar problemas
- El programador puede equivocarse

Errores

```
signal(mutex);  
{ ... } // critical section  
wait(mutex);
```

Errores

```
wait(mutex);  
{ ... } // critical section  
wait(mutex);
```

Monitores

Solución: monitores

- Tipos abstractos de datos
 - Datos internos (no pueden accederse desde fuera del monitor)
 - Operaciones (pueden acceder a los datos internos del monitor)
 - Código de inicialización
- Las operaciones garantizan la exclusión mutua dentro del monitor
 - No habrá dos procesos ejecutando operaciones en el monitor, aunque sean operaciones distintas

Monitores

Son estructuras de alto nivel, proporcionadas por los lenguajes

Esquema de un monitor

```
monitor class Account {
    private int balance;

    public method init() {
        balance := 0;
    }
    public method int withdraw(int amount) {
        balance := balance - amount;
        return balance;
    }
    public method int deposit(int amount) {
        balance := balance + amount;
        return balance;
    }
}
```

Variables de condición

- Por si solos, los monitores no son capaces de modelar todas las estructuras de sincronización
- Para ayudarnos, usamos **variables de condición**
 - Asociado a una condición lógica
 - Operador **wait()**, se bloquea hasta que alguien invoque **signal()**
 - Operador **signal()**, desbloquea a alguien que invocó **wait()**
 - Semántica distinta a la de los semáforos: no hay contador

Variables de condición

Ejemplo: problema del buffer acotado

Buffer acotado implementado con monitor

```
monitor class Buffer {
    private item_t data[BUFFER_SIZE];
    private int in, out, count;
    private condition not_empty, not_full;

    public method init() {
        in = out = count = 0;
    }
    public method item_t get() { ... }
    public method void put(item_t item) { ... }
}
```

Variables de condición

Buffer acotado implementado con monitor - get()

```
public method item_t get() {
    item_t item;
    if (count == 0)
        not_empty.wait();
    item = data[in];
    in = (in + 1) % BUFFER_SIZE;
    count--;
    not_full.signal();
    return item;
}
```

Variables de condición

Buffer acotado implementado con monitor - put()

```
public method void put(item_t item) {  
    if (count == BUFFER_SIZE)  
        not_full.wait();  
    data[out] = item;  
    out = (out + 1) % BUFFER_SIZE;  
    count++;  
    not_empty.signal();  
}
```

Variables de condición

Pregunta: ¿podríamos haber hecho esto?

¿Es posible?

```
public method item_t get() {
    item_t item;
    if (count == 0)
        not_empty.wait();
    not_full.signal();
    item = data[in];
    in = (in + 1) % BUFFER_SIZE;
    count--;
    return item;
}
```

Variables de condición

Pregunta: ¿podríamos haber hecho esto?

¿Es posible?

```
public method item_t get() {
    item_t item;
    if (count == 0)
        not_empty.wait();
    not_full.signal();
    item = data[in];
    in = (in + 1) % BUFFER_SIZE;
    count--;
    return item;
}
```

¿Quién pasa a ejecutar después de signal()?

Variables de condición

Dos disciplinas

- **signal and wait**. Tras `signal()`, si hay alguien bloqueado en la variable condición, este se despierta y nosotros nos dormimos.
- **signal and continue**. Tras `signal()`, continuamos ejecutando en el monitor; cuando acabemos, podrá entrar aquel a quien despertamos.

Variables de condición

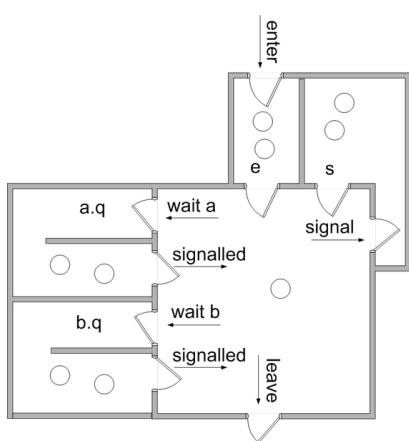


Figura: Signal and wait

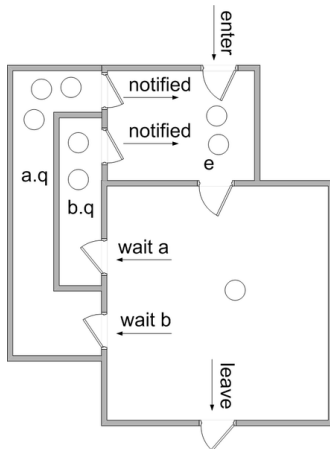


Figura: Signal and continue

Semáforo con monitores

Ejercicio: implementar un semáforo con monitores

- Valor del semáforo de tipo entero
- Operación `wait()` decrementa el semáforo. Si es cero, nos bloqueamos
- Operación `signal()` incrementa el semáforo. Si había alguien durmiendo, lo desbloqueamos.

Semáforo con monitores

Semáforo con monitores

```
monitor class Semaphore {
    private int value;
    private condition is_positive;

    public method void wait() {
        if (value == 0)
            is_positive.wait();
        value--;
    }

    public method void signal() {
        value++;
        is_positive.signal();
    }
}
```

Variables de condición

Las variables de condición en ocasiones se asocian a mutex

- Se usan entre el `wait()` y el `signal()` del mutex
- Si se invoca `wait()` en una variable de condición asociada a un mutex, éste se libera.
- Este esquema se usa en los threads (procesos ligeros) de POSIX

Paso de mensajes

- Las soluciones vistas anteriormente emplean memoria compartida...

Paso de mensajes

- Las soluciones vistas anteriormente emplean memoria compartida...
- ... pero en un sistema distribuido **¡no existe!**
- Ideamos otra alternativa: el **paso de mensajes**
- Los procesos se envían mensajes los unos a los otros
 - Puede ser a través de memoria compartida
 - Puede ser a **través de la red** (pérdida, duplicados, ...)

Paso de mensajes

Primitivas de comunicación

- Dos operaciones
 - `send(proc, msg)`, envía un mensaje `msg` a un proceso `proc`
 - `receive(proc, msg)`, recibe un mensaje `msg` de un proceso `proc`; si no hay mensajes, el receptor se bloquea
- Los mensajes que no son inmediatamente recibidos, se encolan en un **mailbox**
 - Problema: el mensaje a menudo se copia varias veces de buffer en buffer - bajo rendimiento
- Tanto el envío como la recepción se pueden hacer a un grupo de procesos

Paso de mensajes

Ejemplo: buffer acotado

Productor

```
void producer() {
    while (1) {
        item_t data;

        data = produce();
        receive(consumer, NULL);
        send(consumer, data);
    }
}
```

Consumidor

```
void consumer() {
    int i;
    item_t data;

    for (i = 0; i < BUFFER_SIZE; i++)
        send(producer, NULL);
    while (1) {
        receive(producer, data);
        consume(data);
        send(producer, NULL);
    }
}
```

Paso de mensajes

Ejercicio: semáforo con paso de mensajes

- Valor del semáforo de tipo entero
- Operación `wait()` decrementa el semáforo. Si es cero, nos bloqueamos
- Operación `signal()` incrementa el semáforo. Si había alguien durmiendo, lo desbloqueamos.

Paso de mensajes

Semáforo con mensajes

```
void wait() {
    send(semaphore, "wait");
    receive(semaphore, NULL);
}

void signal() {
    send(semaphore, "signal");
}
```

Paso de mensajes

Semáforo con mensajes

```
void semaphore() {
    int value = 0;
    message_t msg;
    process proc;

    while (1) {
        receive_from_any(&proc, &msg);
        if (msg == "wait") {
            if (value > 0) send(proc, NULL);
            else enqueue(proc);
        }
        else if (msg == "signal") {
            if (queued_count() > 0) {
                proc = dequeue();
                send(proc, NULL);
            }
            else value++;
        }
    }
}
```

Paso de mensajes

- El paso de mensajes no es solo un modelo de sincronización de procesos
- Es un fundamento básico de los sistemas distribuidos/**middleware**
 - MPI
 - CORBA
 - Java RMI
 - .Net Remoting

Contenidos

- 1 Introducción
- 2 El problema de la sincronización
- 3 Herramientas de sincronización
 - Semáforos
 - Monitores
 - Paso de mensajes
- 4 Transacciones**

Transacciones

La exclusión mutua nos ofrece cierta atomicidad: una vez entramos nadie nos detiene

- ¿Nadie?

Transacciones

La exclusión mutua nos ofrece cierta atomicidad: una vez entramos nadie nos detiene

- ¿Nadie?
- ¿Y si se va la luz?
- ¿Y si el equipo se avería?
- ¿Y si...?

Transacciones

Operación bancaria

```
void move_money(account_t *acc1,
                account_t *acc2,
                int amount) {
    acc1->balance -= amount;
    acc2->balance += amount;
}
```

¿Qué ocurre si el sistema falla habiendo comenzado pero antes de terminar?

Transacciones

Ofrecemos un nuevo mecanismo: la **transacción**

- Atomicidad completa de un conjunto de instrucciones que forma una función lógica
- La operación, una vez comenzada, o se **aborta** o se **confirma**
 - `abort()` operador para cancelar la transacción (al detectar problemas)
 - `commit()` operador para dar por finalizada la transacción (todo ha ido bien)

Transacciones

Recuperación basada en registro de escritura anticipada (**write-ahead logging**)

- Todas las operaciones se almacenan secuencialmente en un medio persistente
 - Entradas de comienzo y fin de transacción (*abort* o *commit*)
 - Entradas de escritura
 - Identificador de transacción
 - Elemento de datos
 - Valor antiguo
 - Valor nuevo
- Antes de escribir los datos, se añade la entrada correspondiente en el registro

Registro de transacción - operación bancaria

<T-46f124c8, start>

<T-46f124c8, write, acc1, 1200, 1000>

<T-46f124c8, write, acc2, 600, 800>

<T-46f124c8, commit>

Transacciones

Proceso de recuperación

- Operador `undo()`, deshace una transacción
- Operador `redo()`, rehace una transacción
- Ambas deben ser **idempotentes**

Al recuperar

- Se recorre el registro buscando transacciones que hacer y deshacer
- Si aparece la marca *start* pero no *commit*, se deshace
- Si aparecen tanto *start* como *commit*, se rehace
- Empleamos puntos de comprobación (*checkpoints*) para simplificar

Transacciones

Utilidad

- Sistemas gestores de bases de datos (DBMS)
- En sistemas operativos...

Transacciones

Utilidad

- Sistemas gestores de bases de datos (DBMS)
- En sistemas operativos...
- ... se usa para implementar sistemas de ficheros fiables (*journaling*)
 - ReiserFS y Reiser4 de Linux
 - Ext3 y Ext4 de Linux
 - XFS de Linux e IRIX
 - JFS de Linux, OS/2 y AIX
 - UFS de Solaris
 - HFS+ de Apple
 - NTFS (a partir de Windows Vista) de Microsoft