

TEMA 1 PCD

Programación Concurrente->Multiprocesador

Comunicación: colaborar para un determinado fin

Sincronización: compitan por los recursos del sistema

Sistemas monoprocesadores(multiprogramación)

Aprovecha procesos de entrada y salida para liberar al procesador

Para varios usuarios

Los procesos comparten la misma memoria, utilizan para sincronizarse variables compartidas

Sistemas multiprocesador

Sistemas fuertemente acoplados: variables en la memoria compartida(multiproceso)

Sistemas fuertemente acoplados: no existe memoria compartida, comunicación por paso de mensajes(sistemas distribuidos)

Tipos de programación dependientes del hardware

Programación concurrente. Independiente de arquitectura. Define un conjunto de acciones que pueden ser ejecutadas simultáneamente.

Incluye a las dos siguientes.

Programación paralela. Es un tipo de programa concurrente, diseñada para un sistema multiprocesador. Incluye a la siguiente.

Programación distribuida programa paralelo, en sistemas distribuidos.



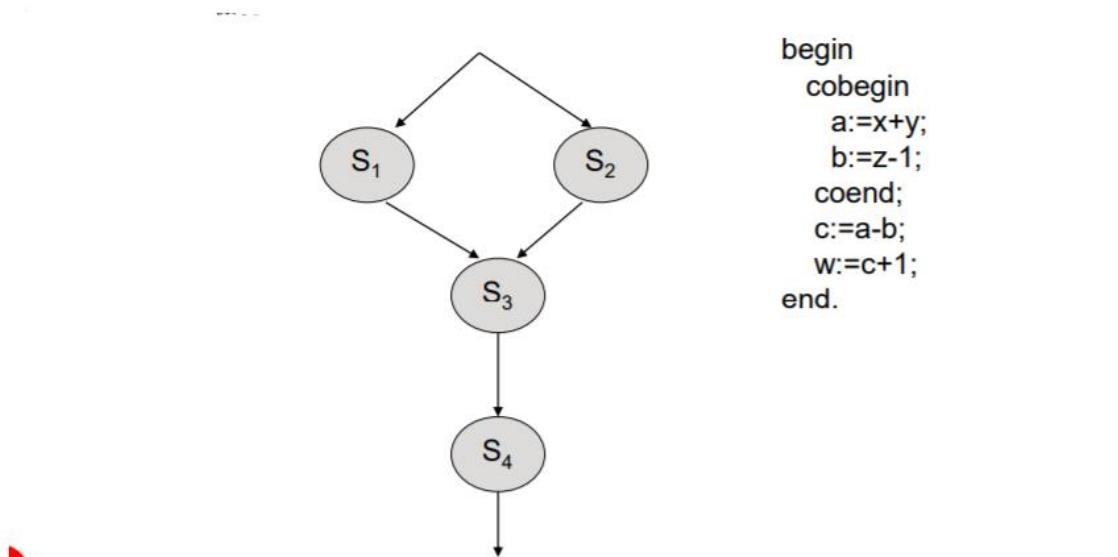
Condiciones de Bernstein: determina si un conjunto de instrucciones pueden ejecutarse concurrentemente

Para que dos conjuntos de instrucciones S_i y S_j se ejecuten concurrentemente se tiene que cumplir que:

- $L(S_i) \cap E(S_j) = \emptyset$
- $E(S_i) \cap L(S_j) = \emptyset$
- $E(S_i) \cap E(S_j) = \emptyset$

• Grafos de precedencia

Sentencias COBEGIN- COEND



Características Sistemas concurrentes

Mayor velocidad de ejecución y CPU

Problemas inherentes al pc:

De seguridad

Exclusión mutua(sección crítica: código que se ejecuta de forma indivisible, no en dos procesadores)

Condición de sincronización: cuando un recurso compartido se queda esperando a que otro proceso lo libere (cambiar estado)

Interbloqueo pasivo :deadlock p1 bloquea a p2 y p2 bloquea a p1

De vivacidad

Interbloqueo activo ejecuta instrucciones pero no hace los procesos

Inanición: el proceso espera demasiado tiempo que sea ejecutado y se muere(elimina)

Sistemas Distribuidos:

- Los componentes se encuentran en distintos ordenadores conectados en red que se comunican con el paso de mensajes
- Concurrencia de los componentes
- No tienen un reloj global (no sincronización perfecta de los nodos)
- Fallos independientes en cada componente

Ventajas:

Aprovechar la potencia de varios nodos

Compartir recursos

Ejemplos

Internet,intranet, Paso de mensajes,Cliente servidor,RPC(procedimientos remotos),RMI(invocar métodos desde un intermediario)

PCD Tema 2

Procesos e Hilos

Vida del Proceso->Se crea, espera a ser ejecutado (se puede bloquear), se ejecuta ,(se puede suspender) termina , se destruye

Bloque de control del proceso BCP, monitoriza el proceso

Pascal FC:

```
bucle:repeat-forever
```

Hilos: cada secuencia de control dentro de un proceso que ejecuta sus instrucciones de forma independiente. Requieren ejecución. ej: En Word(proceso) puede haber un hilo(método) chequeando la gramática mientras que otro guarda el documento

Hilos en Java : Clase Thread

Varios hilos: varios flujos

Puede ceder voluntariamente en control (por abandono, bloqueo ,espera) o ser desalojado. Calculándose el siguiente por prioridad

Uso en java: O se hereda de la clase Thread (extends Thread), o se implementa la interface Runnable (implements Runnable)

Coche **extends Thread**{

```
public void run(){  
    try {  
        avanzar();  
    } catch (InterruptedException e) { return; }  
}
```

Coche c=new Coche(); también se puede definir así: Thread c= new Coche()

c.start(); Método que se llama en el main o donde lo vallas a lanzar que llama al metodo run() que se ejecuta a la vez que otros t.start (ej: que varios coches avancen a la vez)

Para cuando **Coche implements Runnable** :

```
Coche c1 = new Coche ();
Coche c2 = new Coche ();
Thread t1 = new Thread (c1);
Thread t2 = new Thread (c2);
t1.start();
t2.start();
```

Se utiliza mas **implements Runnable** es mas apropiada porque en java no hay herencia múltiple para no usar esa baza

Finalización de un Thread:

Cuando acaba el run

Se hace un t1.stop() //lanza excepción

Llamada System.exit() o Runtime.exit() //acaban todos los Thread

Que el run lance una excepción

Suspensión de un Thread: no para terminar, si no para pausar o desactivar

t1.suspend(); y para volver a activarlo t1.resume();

Pueden causar problemas de deadlocks porque para que otro Thread quiera entrar en la sección critica debe esperar a que se reanude y termine, porque a lo mejor el otro Thread necesita entrar a la sección critica para resumirlo y DEADLOCK

Java puede hacer que un solo procesador lance varios Thread a la vez

Podemos dormir a Thread utilizando Thread.sleep(10000) para 10 segundos

Prioridades

Rango de prioridades 1-10 por defecto prioridad 5 los hilos de prioridad inferior se ejecutaran cuando estén bloqueados los de prioridad superior.

setPriority(10) getPriority()

El método **t1.yield()** hace que el hilo actualmente en ejecución ceda el paso de modo que puedan ejecutarse otros hilos listos para ejecución

El método **t1.join()** para mantener la ejecución del hilo que se está ejecutando actualmente hasta que el hilo especificado esté muerto, usado para establecer orden

Hilos Daemon: Los demonios mueren o finalizan cuando todos los hilos que

no sean daemon hayan terminado su ejecución, no tienen capacidad para terminar ellos mismos el programa y tienen la prioridad más baja.

isDaemon() true o false para saber si es

setDaemon() para activarlo o desactivarlo

Ej: *Garbage collector* es un hilo Daemon de la MVJ que libera memoria de objetos

Planificador:

Preventivo: le da a cada hilo un tiempo de ejecución en el sistema

No preventivo: ejecuta un hilo hasta que termina

Tema 3

Problemas de la programación concurrente

Reglas de sincronización necesarias entre procesos concurrentes

Exclusión mutua: procesos desean utilizar un recurso no compatible.

Sección Crítica :es la parte de código que utiliza un proceso en el acceso a un recurso no compatible y que debe ejecutarse en EM

Un proceso concurrente debe cumplir :

-Exclusión mutua. No pueden acceder dos procesos a la vez a la sección crítica.

-LIMITACIÓN EN LA ESPERA. Que ningún proceso espere de forma indefinida para entrar en la sección crítica.

-PROGRESO EN LA EJECUCIÓN. Que cuando un proceso quiere ejecutar su sección crítica pueda hacerlo si esta está libre.

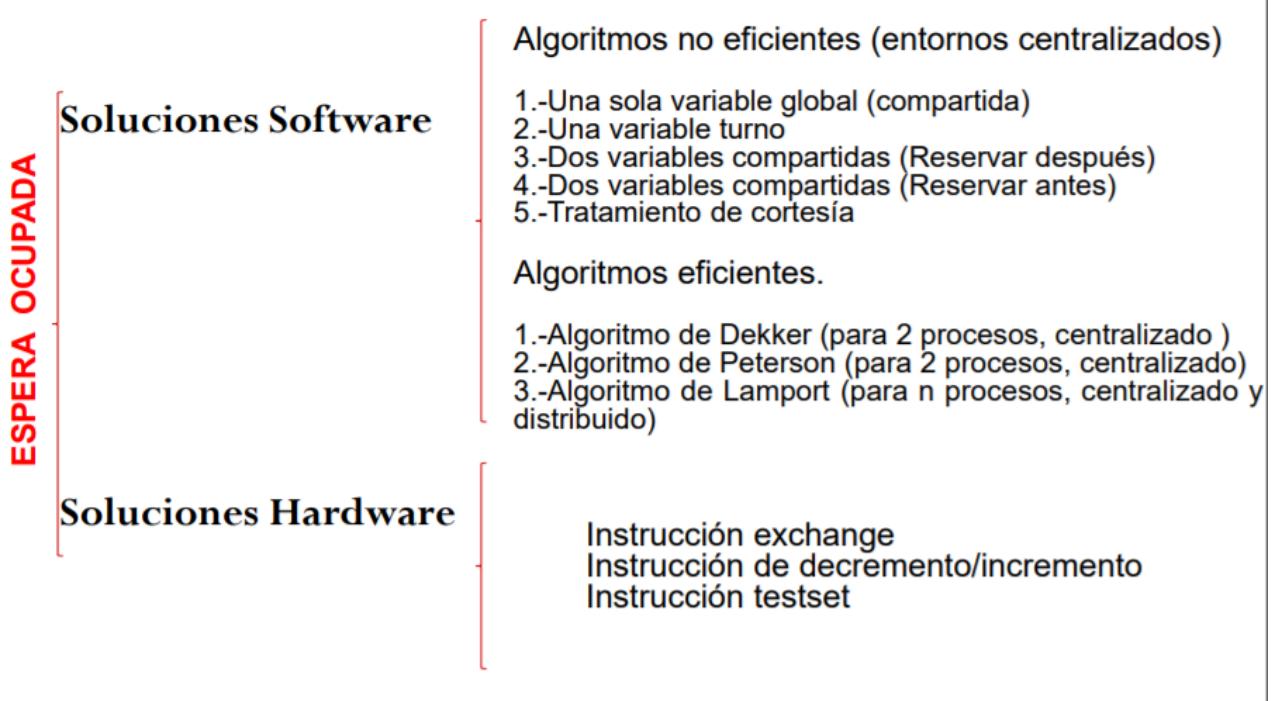
CONDICIONES DE SINCRONIZACIÓN: condiciones para acceder a un proceso ej(El proceso lector no podrá depositar una imagen en el buffer si éste está Lleno)

Problemas de sincronización concurrente

Inhibición de interrupciones

Sistemas Monoprocesador: no correcta porque el usuario puede deshabilitar procesos. Solución lenta, no garantiza secciones críticas

Espera Ocupada: esperar comprobando una variable todo el rato. Consumo tiempo de procesador, no siendo productiva, difíciles de diseñar y complejos



Soluciones Software

No eficientes

4. Algoritmos **no** eficientes

Estos Algoritmos no eficientes demostrarán errores típicos en algoritmos concurrentes.

- **Usar una sola variable global (compartida)** que informe del estado de la sección crítica.

V=slibre sección crítica libre

V=scocupada sección crítica está siendo ejecutada por otro proceso.

```
process p0
repeat
    while v=scocupada do;
    v:=scocupada;
    Sección Crítica
    v:=slibre
    resto0
forever
```

```
process p1
repeat
    while v=scocupada do;
    v:=scocupada;
    Sección Crítica
    v:=slibre
    resto1
forever
```

Problema: Los dos procesos entran en while al mismo tiempo, y ejecutan la sección crítica a la vez. Este algoritmo no garantiza la exclusión mutua.

4. Algoritmos no eficientes

- **Usar una variable “turno”:** esta variable contendrá el número (identificador) del proceso que puede entrar en la sección crítica.

```
process p0
repeat
    while turno=1 do;
        Sección Crítica
        turno:=1;
        resto de proceso 0;
forever
```

```
process p1
repeat
    while turno=0 do;
        Sección Crítica
        turno:=0;
        resto de proceso 1;
forever
```

Garantiza la exclusión mutua

Problema: El derecho de usar la sección crítica es alternativa entre los procesos, por lo que no se satisface la condición de **progreso en la ejecución**. Si turno vale 0 y P1 llega antes que P0, P1 tiene que esperar porque no es su turno, si un proceso quiere entrar dos veces seguidas, no podría.

Si P1 ejecuta la SC y pone turno a 0, supongamos que resto1 es corto y resto0 es largo, P1 no puede ejecutarse hasta que P0 no termine con resto0, entre en su sección crítica y ponga turno a 1. Poco tolerante a fallos.

14

Tema 3. Primeras aproximaciones

4. Algoritmos no eficientes

El problema de la alternancia se produce porque no se conserva suficiente información sobre el estado de cada proceso, sólo se recuerda cuál es el proceso que puede entrar en la sección crítica.

- **Usar dos variables compartidas**, para cada proceso. Los dos leen las dos variables, pero solo escriben sobre una de las variables. (**Reservar después**)

enSC: en sección crítica

restoproceso: no está en sección crítica

Se inicializan c1 y c0 a restoproceso.

```
process p0
repeat
    while c1=enSC do;
        c0:=enSC;
        Sección Crítica
        c0:=restoproceso;
        resto 0
forever
```

```
process p1
repeat
    while c0=enSC do;
        c1:=enSC;
        Sección Crítica
        c1:=restoproceso;
        resto 1
forever
```

Problema: Los dos procesos entran en while al mismo tiempo, y ejecutan la sección crítica a la vez. **Este algoritmo no garantiza la exclusión mutua.**

15

Tema 3. Primeras aproximaciones

- **Tratamiento de cortesía.** Ceder el turno.

Cuando un proceso comprueba que el otro quiere entrar a la SC le cede el turno cortésmente.

```
process p0
repeat
    c0:=quiereentrar;
    while c1=quiereentrar do
        begin
            c0:=restoproceso;
            Hacer_algo;
            c0:=quierentrar;
        end;
    Sección Crítica
    c0:=restoproceso;
    resto 0;
forever
```

```
process p1
repeat
    c1:=quiereentrar;
    while c0=quiereentrar do
        begin
            c1:=restoproceso;
            Hacer_algo;
            c1:=quierentrar;
        end;
    Sección Crítica
    c1:=restoproceso;
    resto 1;
forever
```

Garantiza la exclusión mutua

Problema: Los dos procesos se pueden quedar de manera indefinida cediéndose el paso, con solo suponer que el while dura lo mismo en ambos. No se produce espera ilimitada porque existe una esperanza de que se salga de esta situación, pero no se asegura que se acceda a la SC en un tiempo finito.

17 Tema 3. Primeras aproximaciones

4. Algoritmos **no** eficientes

- **Tratamiento de cortesía.** Ceder el turno.

Cuando un proceso comprueba que el otro quiere entrar a la SC le cede el turno cortésmente.

```
process p0
repeat
    c0:=quiereentrar;
    while c1=quiereentrar do
        begin
            c0:=restoproceso;
            Hacer_algo;
            c0:=quierentrar;
        end;
    Sección Crítica
    c0:=restoproceso;
    resto 0;
forever
```

```
process p1
repeat
    c1:=quiereentrar;
    while c0=quiereentrar do
        begin
            c1:=restoproceso;
            Hacer_algo;
            c1:=quierentrar;
        end;
    Sección Crítica
    c1:=restoproceso;
    resto 1;
forever
```

Garantiza la exclusión mutua

Problema: Los dos procesos se pueden quedar de manera indefinida cediéndose el paso, con solo suponer que el while dura lo mismo en ambos. No se produce espera ilimitada porque existe una esperanza de que se salga de esta situación, pero no se asegura que se acceda a la SC en un tiempo finito.

17

Tema 3. Primeras aproximaciones

Eficientes

5.Algoritmos eficientes

- **Algoritmo de Dekker** (para 2 procesos centralizados). Si hay conflicto comprobar el turno. Combina la idea de los turnos del segundo algoritmo y el tratamiento de cortesía del quinto. Cada proceso comparte dos variables.

C0 y C1 = restoproceso, turno = 0 ó 1 (es indiferente)

```
process p0
repeat
    C0:=quiereentrar;
    while C1=quiereentrar do
        If turno=1 then
            begin
                C0:=restoproceso;
                while turno=1 do;
                C0:=quiereentrar;
            end;
        Sección Crítica;
        turno:=1;
        C0:=restoproceso;
        resto 0
    forever
```

```
process p1
repeat
    C1:=quiereentrar;
    while C0=quiereentrar do
        If turno=0 then
            begin
                C1:=restoproceso;
                while turno=0 do;
                C1:=quiereentrar;
            end;
        Sección Crítica;
        turno:=0;
        C1:=restoproceso;
        resto 1
    forever
```

Garantiza la exclusión mutua, el progreso en la ejecución y la limitación en la espera.

18

Tema 3. Primeras aproximaciones

5.Algoritmos eficientes

- **Algoritmo de Peterson**. Reservar y ceder.

C0 y C1 = restoproceso, turno = 0 ó 1 (es indiferente)

```
process p0
repeat
    C0:=quiereentrar;
    turno:=1;
    while c1=quiereentrar and turno=1
        do;
    Sección Crítica;
    C0:=restoproceso;
    resto 0;
forever
```

```
process p1
repeat
    C1:=quiereentrar;
    turno:=0;
    while c0=quiereentrar and turno=0
        do;
    Sección Crítica;
    C1:=restoproceso;
    resto 1;
forever
```

Garantiza la exclusión mutua, el progreso en la ejecución y la limitación en la espera.

5. Algoritmos eficientes

- **Algoritmo de Lamport.** Para n procesos, adecuado para entornos distribuidos. (**algoritmo de la panadería**)

- C: array[0..n-1]. Coge numero (cognum, nocognum)
- Numero: array[0.. N-1]. Número cogido por cada proceso.
- $[(a,b) \leq (c,d)]$ si $[a \leq c \text{ o } (a=c \text{ y } b \leq d)]$

$(C[i],\text{numero}[i])$ pertenece al proceso P_i , que puede leerlas y modificarlas, un proceso P_j sólo puede leerlas.

```
process pi
repeat
    C[i]:=cognum; //Pi está cogiendo numero
    Numero[i]:=1+max(Numero[0],...,Numero[n-1]);
    C[i]:=nocognum; //Pi ha terminado de coger numero
    for j:=0 to n-1 do
        begin
            while (c[j]=cognum) do;
                while ((Numero[j]!>0) and ((Numero[i],i)>(Numero[j],j))) do;
            end;
            Sección Crítica
            Numero[i]:=0;
            restoi
    forever
```

Garantiza la exclusión mutua, el progreso en la ejecución y la limitación en la espera.

Inconveniente: los números que cogen los procesos **puede aumentar tanto que sobrepase la capacidad de cualquier tipo de datos con el que se representen éstos.**

Entras a la panadería y pides turno para ser atendido(sección critica) cuando entras compras y cedes al siguiente.

Soluciones Hardware

6.Soluciones Hardware

Estas instrucciones se ejecutan indivisiblemente.

- **Instrucción exchange(r,m):** intercambia los contenidos de las direcciones r y m de forma atómica.

Usan una variable compartida por todos los procesos, m inicializada a 1, que indica SC libre, si está a 0 la SC está ocupada y una variable local ri para cada proceso Pi inicializada a 0.

```
process P0
repeat
    repeat exchange(r0,m) until r0=1;
    Sección Crítica;
    exchange(r0,m);
    resto 0;
forever
```

```
process P1
repeat
    repeat exchange(r1,m) until r1=1;
    Sección Crítica;
    exchange(r1,m);
    resto 0;
forever
```

Garantiza la exclusión mutua y el progreso en la ejecución.

Problema: No satisface la limitación en la espera para n procesos, ya que un proceso puede ser retrasado de forma infinita en su entrada a la sección crítica.

6.Soluciones Hardware

- **Instrucción de decremento/incremento.**

- **subc(r,m)** decrementa en 1 el contenido de m y copia el resultado en r de forma atómica.
- **addc(r,m)** incrementa en 1 el contenido de m y copia el resultado en r de forma automática.

Si usamos subc, se inicializa m a 1, la SC está libre, si usamos addc se inicializa m a -1, indicando que SC está libre.

```
process p0
repeat
    repeat subc(r0,m) until r0=0;
    Sección Crítica;
    m:=1;
    resto 0;
forever
```

```
process p1
repeat
    repeat subc(r1,m) until r1=0;
    Sección Crítica;
    m:=1;
    resto 1;
forever
```

Garantiza la exclusión mutua

Inconvenientes: la variable m puede aumentar o disminuir tanto que puede producir un desbordamiento y un proceso puede ser retrasado de forma infinita en su entrada a la sección crítica.

6.Soluciones Hardware

- **Instrucción testset (m)** realiza la siguiente secuencia de operaciones de forma indivisible:

- Comprueba el valor de la variable m
- Si el valor=0 lo cambia por 1 y devuelve el resultado de verdad.
- En otro caso no cambia el valor de m y devuelve falso.

El protocolo de exclusión mutua requiere una variable compartida **m inicializada a 0 => sección crítica libre, m=1 sección crítica ocupada.**

```
process P0
repeat
    repeat until testset(m);
    Sección Crítica;
    m:=0;
    resto 0;
forever
```

Garantiza la exclusión mutua

Inconvenientes: Esta solución no satisface la condición de espera limitada, para n procesos.

```
process P1
repeat
    repeat until testset(m);
    Sección Crítica;
    m:=0;
    resto1;
forever
```

Si se implementa con una rray de booleana y una llave booleana Garantiza la exclusión mutua, el progreso en la ejecución y la limitación en la espera

TEMA 4 Semáforos

Dos o mas procesos cooperan por medio de señales simples de modo que se puede obligar a detener a un proceso en una posición determinada hasta que reciba una señal específica.

signal(s) En java nombre_semaforo.release() Desbloquea procesos bloqueados , si no hay procesos bloqueados incrementa el valor de s

wait(s) En java nombre_semaforo.acquire () Decrementa el valor de s si llega a 0 el proceso se bloquea

Espera no ocupada (solución a bajo nivel)

Tipos de Semáforo

-Semáforo binario admite 0 y 1

-Semáforo general

Solucionar problema Exclusión Mutua(semáforo binario)

Antes de la sección critica un wait() y al terminar la sección critica un signal() usando semáforo inicializado a 1

Solucionar problema Condición de sincronización(semáforo general)

Supongamos los dos procesos concurrentes siguientes y que P2 no puede ejecutar d hasta que P1 haya ejecutado a.

```
process P1;  
begin  
  a;  
  b;  
end;
```

```
process P2;  
begin  
  c;  
  d;  
end;
```

Si quisieramos que los dos procesos se

Solución: usar un semáforo s inicializado a 0.

```
process P1;  
begin  
  a;  
  signal(s);  
  b;  
end;
```

```
process P2;  
begin  
  c;  
  wait(s);  
  d;  
end;
```

Problemas Clásicos Semaforos

Productor/Consumidor: conjunto de procesos que producen información que otros procesos consumen.(a diferentes velocidades

Tenemos un bufer b donde se van almacenando los elementos(producir) y de donde se extraen (consumir).

3 semaforos:

-mutex: para controlar el acceso en exclusión mutua al buffer, inicializada a 1

-vacio: cuenta el numero de espacios libres if vacios =0 bloquea prod

-Lleno: para contar el número de posiciones llenas del buffer, if lleno =0 bloquea consumidores

- Productor/Consumidor.

Possible solución . buffer , frente y cola =0, mutex =1, vacios =n, llenos=0

```
process productor;
begin
    repeat
        producir item;
        wait (vacios);
        wait(mutex);
        buffer[frente]:=item;
        frente:=(frente+1) mod n;
        signal(mutex);
        signal(llenos)
    forever
end;
```

```
process consumidor;
begin
    repeat
        wait(llenos);
        wait(mutex);
        item:=buffer[cola];
        cola:=(cola+1)mod n;
        signal(mutex);
        signal(vacios);
        consumir item;
    forever
end;
```

Lectores y Escritores: Un grupo de personas que leen y escriben en un mismo ficheros. Condiciones: puede haber n numero de lectores leyendo a la vez ,pero solo un escritor escribiendo ,no se admite escritura y lectura a la vez.

Resoluciones en función de la prioridad.

Prioridad en la lectura: ningún lector esperará salvo que un escritor haya obtenido permiso para usar el recurso.

4. Problemas Clásicos. (16 Octubre)

- **Lectores y Escritores. Prioridad en la lectura.**

Variables a usar: **nl** entera inicializada a 0, número de lectores que hay usando el recurso en un momento dado. Un semáforo **mutex** inicializado a 1 para asegurar exclusión mutua cuando se actualiza nl. Un semáforo **wrt** inicializado a 1 común a lectores y escritores, funciona como semáforo de **exclusión mutua** para escritores y lo usan el primer/último lector para entrar/salir de la SC.

```
process lector;
begin
    ...
    wait(mutex);
    nl := nl+1;
    if(nl=1) then wait(wrt);
    signal(mutex);
    ... leer del recurso...
    wait(mutex);
    nl:=nl-1;
    if (nl=0) then signal(wrt);
    signal(mutex);
end;
```

```
process escritor;
begin
    wait(wrt);
    escribir_recurso;
    signal(wrt);
end;
```

La clave está en **wait(wrt)** que realiza el primer lector que consigue acceder al recurso, ya que pone el semáforo a 0, impidiendo que los escritores accedan al recurso.

Prioridad en la escritura: si un escritor está esperando, ningún lector nuevo debe iniciar su lectura.

Variables:

int **nl** = 0, cuenta nº de lectores.

int **nel** = 0, cuenta nº de lectores esperando.

int **nee**=0 nº de escritores esperando.

esc=false para saber si alguien esta escribiendo.

mutex=1 semaforo binario(exclusión mutua)

lector y **escritor** semáforos binarios inicializados a 0

Si hay un while no es eficiente porque es espera ocupada

```

process type lector;
begin
    wait(mutex);
    if(esc or nee>0) then
        begin
            nle:=nle+1;
            signal(mutex);
            wait(lector);
            nle:=nle-1;
        end;
    nl:=nl+1;
    if (nle>0) then signal(lector);
    else signal (mutex); ← El último lector libera la
    leer_recurso;
    wait(mutex);
    nl:=nl-1;
    if (nl=0 and nee>0) then
        signal(escritor)
    else signal (mutex);
end;

```

Desbloqueo encadenado

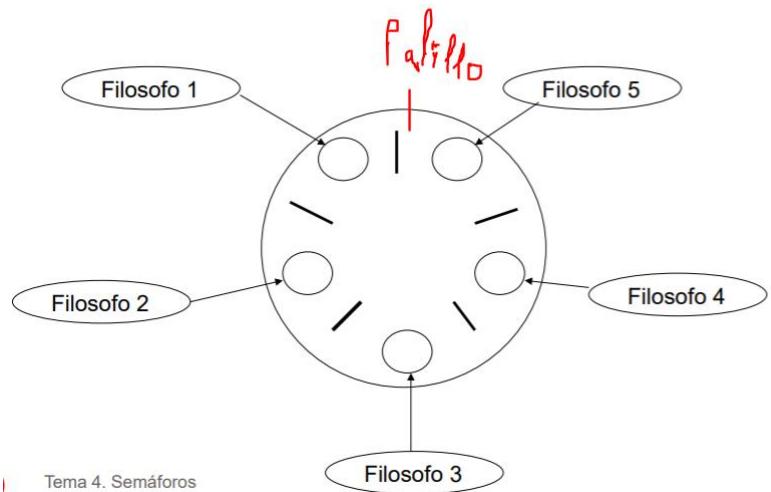
```

process type escritor;
begin
    wait(mutex);
    if (esc or nl>0) then begin
        nee:=nee+1
        signal(mutex);
        wait(escriotor);
        nee:=nee-1;
    end;
    esc:=true;
    signal(mutex);
    escribir_recurso;
    wait(mutex);
    esc:=false;
    if (nee>0) then signal(escriotor)
    else if (nle>0) then signal(lector)
    else signal(mutex)
end;

```

Comida de los filósofos:

Un filosofo para comer necesita 2 palillos que tienen que estar libres. Se tienen que turnar.



Tema 4. Semáforos

Soluciones:

Un array de semáforos ,cada palillo un semáforo. Problema si todos cogen el palillo de la izq a la vez interbloqueo

While estar libre Problema: espera ocupada

Buena: Se crea un array de 5 semáforos palillos inicializados a 1 y cada filosofo bloquea sus palillos cuando está comiendo . Si es filosofo impar toma primero su palillo de la izquierda y luego el de la derecha, mientras que un filósofo par, elige primero su palillo de la derecha y luego el de la izquierda.

También debería tener un semaforo por cada sitio para que no se sienten 2 en el mismo xd

Inconvenientes Semáforos

De bajo nivel.

SC comienza en wait finaliza en signal.

Necesita revisión de código para entenderlo,difícil de mantener y programar

TEMA 5 Monitores

Solucionan la complejidad de entender los algoritmos de Semáforos

Procesos Pasivos: implementan los monitores

Procesos Activos :resto de procesos (lector , escritor)

Programa con Monitores

-Monitor

-Variables locales: tamaño cola.

-Procedimientos interno: manipulan variables locales

-Procedimientos externos :pueden ser llamados desde fuera del monitor

-Código de inicialización begin end

-Variables de condición: controlan las condiciones de sincronización (ej si esta lleno vacío no puede insertar)

Exclusión Mutua: proporcionada por el monitor

Cola del monitor: FIFO .El monitor bloquea y almacena los procesos activos quieren ejecutar un procedimiento cuando ya hay otro proceso dentro del monitor(ejecutándose) cuando un proceso activo deja el monitor(termina de ejecutarse) el monitor desbloquea el proceso que este al frente de la cola ejecutándolo

Condiciones de sincronización: Variables de condición

Contienen colas FIFO de procesos pudiendo:

Bloquearlos **delay(c)** en java **await(c)**

Desbloquearlo **resume** en java **signal**

Empty(c) devuelve un boolean=true si la cola de la variable c esta vacía

Tipos de resume(c)

Desbloquear y continuar (DC):

Retorno forzado (RF): Libera al monitor y ejecuta el proceso sin prioridad de cola

Desbloquear y Esperar (DE): espera en la cola.

Desbloquear y espera urgente (DU): Desbloquea y espera en una cola de cortesía de mayor prioridad

Problemas Clásicos Monitores

Productor consumidor

```
monitor buffer;
const n:=10;      Tam maximo del buffer
var tam,frente,cola:integer;
lleno,vacio: condition;
recurso: array[0..n-1] of item;
export insertar, extraer;
procedure insertar(elemento:item)
begin
  if tam=n then delay(lleno); Si el buffer esta lleno bloqueo al productor hasta que se libere con resume(lleno)
  recurso[cola]:=elemento; Una vez desbloqueado almacenamos el elemento en el buffer
  cola:=(cola+1) mod n;   Actualizamos el puntero del ultimo elemento
  tam:=tam+1;             Incrementamos la variable contador de elementos
  resume(vacio);
end;                  Desbloqueamos al consumidor porque ya hay un elemento en el buffer para ser extraido
procedure extraer (var elemento:item);
begin
  if tam=0 then delay(vacio); Si el buffer esta vacio bloqueamos al consumidor hasta que resume(vacio)
  elemento:=recurso[frente]; Extraemos un elemento del inicio de la cola
  frente:=(frente+1) mod n;   Actualizamos el puntero
  tam:=tam-1;               Decrementamos el tamaño
  resume(lleno);            Desbloqueamos al productor dado a que ya puede insertar un elemento por que sobra un
                            espacio
end;
begin
  frente:=0;cola:=0;tam:=0;
end;
```

```
process productor;
var elemento:item;
begin
  repeat
    elemento:=producir_elemento;
    buffer.insertar(elemento);
  forever
end
```

```
process consumidor;
var elemento:item;
begin
  repeat
    elemento:=buffer.extraer;
    consumir_elemento(elemento);
  forever
end
```

Lectores y escritores:prioridad lectores

- Los lectores pueden acceder siempre al recurso a no ser que exista un escritor escribiendo.

```
monitor L_E;
var nl:integer; Numero lectores
escribiendo:boolean;
lector,escritor: condition;
export abrir_lectura, cerrar_lectura,
abrir_escritura, cerrar_escritura;
```

```
procedure abrir_lectura;
begin
  if (escribiendo) then delay(lector); Bloqueamos lector
  nl:=nl+1; Incrementamos numero lectores
  resume(lector); Liberamos Lector
end;
procedure cerrar_lectura;
begin
  nl:=nl-1; Decrementamos numero lectores
  if (nl=0) then resume (escritor) Liberamos escritor
end;
```

```
procedure abrir_escritura;
begin
  if (nl>>0) or escribiendo then delay(escritor); Bloqueamos al escritor porque hay un
  escribiendo:=true;
end;
procedure cerrar_escritura;
begin
  escribiendo:=false;
  if empty(lector) then resume (escritor) Si no hay ningun lector se
  else resume(lector) desbloquea al escritor
end;
begin
  nl:=0;
  escribiendo:=false;
end;
```

```
process type lector;
begin
  ...
  L_E.abrir_lectura;
  Leer del recurso;
  L_E.cerrar_lectura;
  ...
end;
```

```
process type escritor;
begin
  ...
  L_E.abrir_escritura;
  Escribir en el recurso;
  L_E.cerrar_escritura;
  ...
end;
```

Inconvenientes

Si el flujo de lectores es constante, los escritores pueden sufrir de inanición(proceso muere).

Para evitar este problema se utiliza la política de dar prioridad a los escritores

Lectores y escritores:prioridad Escritores

```
monitor L_E;
var nl:integer;
escribiendo:boolean;
lector,escritor: condition;
export abrir_lectura, cerrar_lectura,
abrir_escritura, cerrar_escritura;

procedure abrir_lectura;
begin
  if (escribiendo) or not empty(escritor)
  then delay(lector);
  nl:=nl+1;
  resume(lector);
end;
procedure cerrar_lectura;
begin
  nl:=nl-1;
  if (nl=0) then resume (escritor)
end;
```

```
procedure abrir_escritura;
begin
  if (nl>>0) or escribiendo then delay(escritor);
  escribiendo:=true;
end;
procedure cerrar_escritura;
begin
  escribiendo:=false;
  if not empty(lector) then resume (lector)
  else resume(escritor)
end;
begin
  nl:=0;
  escribiendo:=false;
end;
```

Diferencias con prioridad lectores

- Si hay algún escritor escribiendo o en espera se bloquea al lector cuando quiere leer
- Si hay algún lector se le da paso, si no se libera al escritor

Comida de filosofos

```
monitor comida_filosofos;
const N=5;
var estado:array[0..N-1] of
(pensando, comiendo, hambriento);
dormir:array[0..N-1] of condition;
i: integer
export coger_palillos, soltar_palillos;

procedure coger_palillos(i:integer);
begin
    estado[i]:=hambriento;
    test(i);
    if estado[i]<>comiendo
    then delay(dormir[i]);
end;
procedure soltar_palillos(i:integer);
begin
    estado[i]:=pensando;
    test((i+4) mod N);
    test((i+1) mod N);
end;
```

```
procedure test (k:integer);
begin
    If (estado[(k+N-1) mod N]<>comiendo)and
    (estado[(k+1) mod N]<>comiendo) and
    (estado[k]= hambriento) then
    begin
        estado[k]:=comiendo;
        resume (dormir[k]);
    end;
begin
    for i:=0 to N-1 do
        estado[i]:=pensando;
end;
```

test: Permite que un filósofo cambie su estado a comiendo en el caso de que esté hambriento y sus dos vecinos no estén comiendo

```
procedure type filosofo (i:integer);
begin
repeat
    piensa;
    comida_filosofos.coger_palillos(i);
    come;
    comida_filosofos.soltar_palillos(i);
forever
end;
```

Todos los filósofos pensando.

1 quiere comer ,coger_palillos(1)

se pone hambriento ,hace test como el fil 5 y fil 2

esta comiendo=false y fil 1=hambriento se pone

comer, luego se libera durmiendo.



Monitores en Java

Lock m=new ReentrantLock(); m.lock(); m.unlock();

Condition c=m.newCondition(); c.await(); c.signal();

ReentrantReadWriteLock x=new ReentrantReadWriteLock(); Sirve para 2 locks: 1 de lectura concurrente y otro escritura excluyente con los métodos x.readLock() x.writeLock()

El método Lock() Cierra la instancia de la clase Lock para que todos los hilos que llamen a lock() estén bloqueados hasta que se ejecute unlock().

```
import java.util.concurrent.locks.*;
public class Buffer {
    Lock lock = new ReentrantLock();
    Condition lleno = lock.newCondition();
    Condition vacio = lock.newCondition();

    public void insertar (int elemento) {
        lock.lock();
        try { while (cuantos == N) lleno.await();
            /* s.c. INSERTAR */
            vacio.signal();
        } finally
            lock.unlock();
    }
    public int extraer () {
        lock.lock();
        try { while (cuantos == 0) vacio.await();
            /* s.c. EXTRAER */
            lleno.signal();
        } finally
            lock.unlock();
    }
}
```

Siempre debe haber un finally para el unlock();

await() lanza una excepción InterruptedException que hay que capturar o relanzar

TEMA 6 Paso de mensajes

Ventajas:

Uso en cualquier sistema de hardware

No existe exclusión mutua

Desventajas:

Los canales de comunicación son vulnerable a fallos(perdida del mensaje latencia...)

Comunicación Directa:

Comunicación Directa simetrica:

Se conoce al receptor y emisor send(a,mensaje) send(b,mensaje)

Comunicación Directa asimetrica:

el receptor no conoce al emisor send(a,mensaje) recibe(Id,mensaje)

Comunicación Indirecta: Receptor y emisor sin Identificar

La comunicación se realiza depositando el mensaje en un almacén(buzón)

send(buzonA,mensaje) receive(buzonA,mensaje)

Sincronización

Comunicación Asincrono buffer,email ,foros

Comunicación Sincrona canal,chats,telefono

Características

Flujo de Datos: Unidireccional->Correo Bidireccional->chat

Capacidad del canal: Cero->no buffer, Infinita , finita

Tamaño del msg: fija o variable controlada por el programador

Canales con tipo o sin: Tipo de dato que contienen

Paso por referencia: enviar al receptor la dirección del mensaje(**memoria común**)
no sistemas distribuidos mas eficiencia por la no duplicidad de datos

Paso por copia: enviar al receptor una copia del mensaje,mas seguro no modifiable

Sintaxis

Si envías **send(consumidor,msg)** se queda bloqueado hasta que recibas **receive(productor,msg)** recibes del productor y mandas al consumidor el mismo msg

Select

Se selecciona una de manera aleatoria

```
select
  RECEIVE(buzon1, mensaje);
  sentencias;
or
  RECEIVE(buzon2, mensaje);
  sentencias;
or
  ...
or
```

Select con guardas con condiciones

```
select
  when condicion1 =>
    receive(buzon1, mensaje);
    sentencias;
or
  when condicion2 =>
    receive(buzon2, mensaje);
    sentencias;
or
  ...
```

Productor Consumidor Paso de Mensajes

```
program Prod_Cons_Buffer_Lim;
const TAM_BUFFER = 4;
process Productor;
var elemins: integer;
begin
  for mensaje:=1 to 20 do
    SEND (Buffer, elemins)
end;
process Consumidor;
var elemcons: integer;
begin
  repeat
    RECEIVE (Buffer, elemcons);
    writeln (elemcons)
  until mensaje=20
end;
```

```
begin
  cobegin
    Productor; Consumidor; Buffer;
  coend
end;
```

18

```
process Buffer;
var
  Almacen: array[1..TAM_BUFFER of integer];
  Contador,entrada,salida:integer;
  Elemens:integer;
begin
  contador:=0; entrada:=1; salida:=1;
  repeat
    select
      when contador <= TAM_BUFFER
        RECEIVE (Productor,elemens);
        almacen[entrada]:=elemens;
        entrada:=(entrada mod TAM_BUFFER) +1;
        contador:=contador+1;
      or
      when contador >> 0
        SEND (Consumidor, almacen[salida]);
        salida:=(salida mod TAM_BUFFER)+1;
        contador:=contador -1;
    end
  forever
end;
```

TEMA 7 Paso de mensajes Asíncrono

En la conexión cliente/servidor uno a muchos el buffer se le denomina puerto

Buzón Pascal-FC FiFO

Nombre **mailbox of** tipo tam ilimitado

Nombre **mailbox[1...n] of** tipo tam limitado(si esta lleno se bloquea)

send(b,m) enviar m al buzón b (bloquea si esta lleno)

receive(b,m) el buzón b recibe un mensaje y lo almacena en m(bloquea si esta vacío)

empty (b) para saber si el buzón esta vacío en ese intante t

Operaciones atómicas(indivisibles)

Prioridad lectura Mensajes Asíncronos

Un lector sólo
esperará cuando
exista un escritor que
ya tenga permiso para
escribir.

La solución propuesta
es muy parecida a la
solución con
semáforos binarios.

nl: numero de lectores
leyendo o con
intención de leer. =0
mutex: buzón para
E.M.en el acceso a nl.

wrt: buzón para E.M.
en el acceso al
recurso compartido
por escritores y
lectores.

```
program Lectores_Escritores;
type
Item=...
const NUMLEC=10; NUMESC=10;
var
    mutex,wrt: mailbox of integer;
    testigo: item;
    nl: integer;
process lector;
var
    testigo: item;
begin
repeat
    receive(mutex, testigo);
    nl:=nl+1;
    if (nl=1) then receive (wrt, testigo);
    send(mutex, testigo);
    leer del recurso
    receive (mutex, testigo);
    nl:=nl-1;
    if (nl=0) then send (wrt, testigo);
    send(mutex,testigo);
forever
end;
```

```
process escritor;
var
    testigo: item;
begin
repeat
    ...
    receive(wrt, testigo);
    escribir en el recurso;
    send(wrt,testigo);
    ...
forever
end;

begin
    nl:=0;
    send(mutex,testigo);
    send(wrt,testigo);
    cobegin
        (*lanzar lectores y
        Escritores*)
    coend;
end.
```

Prioridad escritura Mensajes Asíncronos

```
Program lectores_escritores;
type
    buzon_res= mailbox of item;
var
    a_lec: mailbox of integer;
    a_esc: mailbox of integer;
    c_lec: mailbox of integer;
    buzon: array[1..nlec+nesc]of buzon_res;
    contador: integer;
process type lector (ident : integer);
var
    mensaje: item;
begin
    repeat
        ...
        mensaje.id:=ident;
        send(a_lec, mensaje);
        receive (buzon[ident], mensaje)
        ...
        leer del recurso
        ...
        send(c_lec,mensaje);
        ...
    forever
end;
```

Ningún lector nuevo debe iniciar la lectura si un escritor está esperando. Se introduce un nuevo **proceso controlador** que se encargará de dar permiso a las operaciones de lectura y escritura.

buzon[i]: vector de buzones, **un buzón por cada proceso lector o escritor**, lo usa el controlador para enviar las confirmaciones a los procesos lectores o escritores .

```
process type escritor (ident : integer);
var
    mensaje: item;
begin
    repeat
        ...
        mensaje.id=ident;
        mensaje.men:=//lo que se quiere escribir:
        send (a_esc,mensaje)
        receive(buzon[ident], mensaje);
        ...
    forever
end;
```

```
process controlador;
begin
    nl:=0;
repeat
select
    when empty(a_esc) =>
        receive(a_lec, mensaje);
        nl:=nl+1;
        send(buzon[mensaje.id], mensaje)
    or
        receive(c_lec, mensaje);
        nl:=nl-1;
    or
        when (nl=0) =>
            receive (a_esc, mensaje);
            escribir en el recurso
            send (buzon[mensaje.id], mensaje);
end
forever
end;
```

Tema 7. Paso de mensaje asíncrono

Filósofos Mensajes Asíncronos

```
process type filosofo (id : integer);
var
    mensaje: item;
begin
    repeat
        ...
        mensaje.id:=id;
        send(pido_palillos[id], mensaje);
        receive (palillos_concedidos[id], mensaje)
        ...
        Filósofo comiendo
        ...
        send(suelto_palillos[id],mensaje);
        ...
    forever
end;
```

Un filósofo que quiere comer cogerá los dos palillos o esperará si alguno o los dos están cogidos por los filósofos vecinos, cada filósofo que quiera comer lo solicitará al proceso controlador .

Estructuras de datos:

pido_palillos: vector de buzones, para solicitar sus dos palillos.

palillos_concedidos: vector de buzones para indicar concesión de palillos.

suelto_palillos: vector de buzones para indicar al controlador de que ha terminado de comer.

palillos: vector para indicar si los palillos están cogidos o no.

```

process controlador;
var
    mensaje: item; palillos: array[0..N-1] of integer;
    cont: integer;
begin
    repeat
        select
            for cont=0 to N-1 replicate
                when (palillo[cont]=1) and (palillo[(cont+1) mod n]=1) =>
                    receive (pido_palillos[cont], mensaje);
                    palillo[cont]=0;
                    palillo[(cont+1) mod n]=0;
                    send(palillo_concedidos[cont], mensaje);
            or
            for cont=0 to N-1 replicate
                receive (suelto_palillos[cont], mensaje);
                palillos[cont]=1;
                palillos[(cont+1) mod n]=1;
        end
    forever

```

16 d;

Tema 7 Paso de mensajes asíncronos

Paso de Mensajes Asíncronos Java

MailBox Mail=new MailBox();

Object=Mail.receive()

Mail.send(Object)

Encontrar Proveedor de correo compatible con SSL o TLS para enviar mensajes a través de un puerto y un protocolo

SSL(capa de conexión segura) para cuando se usa SMTP , establecer una conexión segura puerto:465

TLS (Seguridad de la capa de transporte) para cuando se usa SMTP , comienza con una comunicación no segura para luego establecerla como segura para transmitir datos puertos:25,587

SI NINGUNO ES COMPATIBLE SE OFRECE EL PUERTO 25 COMO
COMUNICACIÓN NO SEGURA&NO ENcriptada)

TEMA 8 Paso de mensajes síncrono

Envío y recepción son bloqueantes:

El emisor envía un mensaje y el receptor todavía no ha realizado la operación de recepción. En este caso el emisor queda bloqueado a la espera

de que el receptor realice la operación de recepción.

El receptor realiza la operación de recepción cuando el emisor todavía no ha realizado la operación de envío. En este caso el receptor queda bloqueado hasta que el emisor realiza la operación de envío.

No es Necesario **Buffer** , utilizaremos **Canales con tipo**

-Flujo de datos unidireccionales

-Comunicación 1-1

ch channel of tipo

ch ! s Envía el valor de la expresión s al canal ch

ch ? r Recibe del canal ch un valor y lo asigna a la variable r

Si el canal es solo para sincronizar (sin msg) var **ch: channel of synchronous** **ch ! any** **ch ? any**

Espera selectiva: el proceso se queda esperando hasta que o envie o reciba.

Uso de select

- Con guardas(condiciones) **when** true
- or **terminate** para terminar y no queda bloqueado si no existen llamadas pendientes
- else** Si ninguna de las alternativas de la sentencia select se puede atender de inmediato, entonces se ejecutará la alternativa else
- or **timeout n** :Si pasados n segundos desde que se ejecutó la sentencia select no ha sido posible ejecutar alguna de las alternativas, entonces se ejecutará la alternativa timeout

Otro tipo de select

- pri select** or ...prioridad por el orden de aparición

```
pri select
    ch1 ? mensaje1;
    /* más sentencias */
or
    ch2 ? mensaje2;
    /* más sentencias */
or
    ...
else
    /* más sentencias */
end;
```

Se repite el bucle

hasta n , cambiando el canal y el mensaje

```
select
    for cont:=1 to N replicate
        begin
            ch[cont] ? mensaje[cont];
            (* más sentencias *)
        end;
```

```
select
    when condition1 =>
        ch1 ? mensaje1;
        (* más sentencias *)
    or
    ...
    or
        when condicionN =>
            chN ? mensajeN
            (* más sentencias *)
    end;
```

```
select
    ch1 ? mensaje1;
    (* más sentencias *)
or
    terminate
end
```

```
select
    ch1 ? mensaje1;
    (* más sentencias *)
or
    ch2 ? mensaje2;
    (* más sentencias *)
or
    ...
else
    (* más sentencias *)
end;
```

```
select
    ch[cont] ? mensaje[cont];
    (* más sentencias *)
or
    timeout n;
    (*más sentencias*)
end
```

Productor/Consumidor Mensajes Síncronos

Productor/Consumidor

```

program productor-consumidor
type
    canal = channel of integer;
const
    tam= 5;N=5;M:=5;
var
    insertar: array [1..N] of canal;
    extraer: array [1..M] of canal;
    cont1,cont2 : integer;
process type consumidor (ident: integer);
var
    elemento: integer;
begin
    repeat
        ...
        extraer[ident] ? elemento
        ...
    forever
end;

```

12

Tema 8. Paso de mensaje síncrono

```

process type productor (ident: integer);
var
    elemento: integer;
begin
    repeat
        ...
        insertar[ident] ! elemento;
        ...
    forever
end;

```

Es necesario añadir un proceso controlador que se encargue de gestionar el buffer. Es necesario tener **tantos canales como procesos productores y consumidores existan.**

Estructuras empleadas:

insertar: vector de canales, uno asociado a cada productor, productor i con canal insertar[i]

extraer: vector de canales, uno por cada consumidor.

buffer: vector donde se almacenan los elementos producidos, variable local del proceso controlador.

cola,cabeza: variables para controlar la posición donde se insertan y se extraen los elementos.

Productor/Consumidor

```

process controlador_buffer;
var
    buffer: array[0..tam] of integer;
    cola, cab, numelem,
    cont1, cont2: integer;
begin
    cola:=0;
    cab:=0;
    numelem:=0;
    repeat
        select

```

Este proceso **controlador_buffer**, una vez inicializadas las variables queda a la espera de peticiones para insertar o extraer elementos del buffer.

Las alternativas de la sentencia select llevan guardas, si se quiere introducir elementos en el buffer (**numelem < tam**) y si se quiere consumir elementos (**numelem > 0**). Además se ha incorporado la alternativa **terminate**, si todos los procesos consumidores y productores han terminado, el controlador puede terminar y no quedar bloqueado indefinidamente.

13

Tema 8. Paso de mensaje síncrono

```

for cont1:=1 to N replicate
    when numelem < tam =>
        insertar[cont1]?buffer[cab];
        cab:=(cab+1)mod tam;
        numelem:=numelem +1;
    or
    for cont2:=1 to M replicate
        when numelem > 0 =>
            extraer[cont2]!buffer[cola];
            cola:=(cola+1) mod tam;
            numelem:=numelem-1;
        or
            terminate;
    end;
    forever
end;
var
    pro: array [1..N] of productor;
    con: array [1..M] of consumidor;
begin
    cobegin
        for cont1:=1 to N do
            pro[cont1](cont1);
        for cont2:=1 to M do
            con[cont2](cont2);
            controlador_buffer;
    coend;
end;

```

Prioridad Lectores Mensajes Síncronos

abrir_lectura y cerrar_lectura: vector de canales de sincronización, para sincronización de lectores con el controlador.

abrir_escritura y cerrar_escritura: vector de canales de sincronización empleado por los procesos escritores para indicar que quieren realizar una escritura sobre la variable compartida.

escribiendo: variable boolean que indica si un proceso escritor está realizando una operación de escritura.

```
process type lector(ident : integer);
begin
    repeat
        ...
        abrir_lectura[ident] ! any
        /*leer del recurso*/
        cerrar_lectura[ident] ! any
        ...
    forever
end;
```

```
process type escritor(ident : integer);
begin
    repeat
        ...
        abrir_escritura[ident] ! any
        /*escribir en el recurso*/
        cerrar_escritura[ident] ! any
        ...
    forever
end;
```

```
process controlador;
var ...
begin
    nl:=0;
    escribiendo:=false;
    repeat
        select
            for cont1:=1 to n_lec replicate
                when escribiendo=false =>
                    abrir_lec[cont1] ? any
                    nl:=nl+1;
            or for cont2:=1 to n_lec replicate
                cerrar_lectura[cont2] ? any
                nl:=nl-1;
            or for cont3:=1 to n_esc replicate
                when nl=0 and escribiendo=false =>
                    abrir_esc[cont3] ? any
                    escribiendo:=true;
            or for cont4:=1 to n_esc replicate
                cerrar_esc[cont4] ? any
                escribiendo:=false;
            or terminate;
        end;
    forever
end;
```

Prioridad Escritores Mensajes Síncronos

peticion: vector de canales de tipo char, tantos como lectores y escritores haya, para indicar que tipo de operación se va a realizar (L:lectura o E:escritura)
n_peticionesE: número de peticiones de operaciones de escritura.

```
process type lector (ident : integer);
begin
    repeat
        ...
        petucion[ident] ! 'L';
        abrir_lec[ident] ! any
        /*leer del recurso*/
        cerrar_lec[ident] ! any
        ...
    forever
end;
```

```
process type escritor(ident : integer);
begin
    repeat
        ...
        petucion[ident+n_lec] ! 'E';
        abrir_escritura[ident] ! any
        /*escribir en el recurso*/
        cerrar_escritura[ident] ! any
        ...
    forever
end;
```

```
process controlador;
var ...
begin
    nl:=0;
    escribiendo:=false;
    n_peticionesE:=0;
    repeat
        select
            for cont0:=1 to n_lec+n_esc
            replicate
                petucion[cont0] ? tipopetucion;
                if tipopetucion='E' then
                    n_peticionesE:=n_peticionesE+1
            or for cont1:=1 to n_lec
            replicate
                when n_peticionesE=0 =>
                    abrir_lec[cont1] ? any
                    nl:=nl+1;
```

```
or for cont2:=1 to n_lec
replicate
    cerrar_lec[cont2] ? any
    nl:=nl-1;
or for cont3:=1 to n_esc
replicate
    when nl=0 and escribiendo=false =>
        abrir_esc[cont3] ? any
        escribiendo:=true;
or for cont4:=1 to n_esc
replicate
    cerrar_esc[cont4] ? any
    escribiendo:=false;
    n_peticionesE:=n_peticionesE-1;
or terminate;
end;
forever
end;
```

Comida filosofos Mensajes Síncronos

pido_palillos: vector de canales, uno por cada filósofo para indicar al controlador su deseo de comer. Son canales de sincronización.

suelto_palillos: vector de canales de sincronización para indicar que han terminado de comer y desean liberar los palillos.

palillos: vector de enteros para saber si los palillos están o no libres (1:libre 0:cogido)

```
18
process type filosofo (id : integer);
begin
repeat
    writeln('filosofo ', id, 'pensando');
    writeln('filosofo ', id, 'quiere comer');
    pido_palillos[id] ! any;
    writeln('filosofo ', id, 'comiendo');
    suelto_palillos[id] ! any;
    writeln('filosofo ', id, 'ha terminado de comer');
forever
end;
```

```
process controlador;
var
    palillos: array[0..N-1] of integer;
    cont: integer;
begin
for cont:=0 to N-1 do
    palillos[cont]:=1;
repeat
select
    for cont:=0 to N-1 replicate
        when (palillos[cont]=1) and (palillos[(cont+1) mod n]=1)) =>
            pido_palillos[cont] ? any;
            palillos[cont]:=0;
            palillos[(cont+1) mod n]:=0;
    or
        for cont:=0 to n replicate
            suelto_palillos[cont] ? any;
            palillos[cont]:=1;
            palillos[(cont+1) mod n]:=1;
    or
        terminate
    end
forever
end;
```

Tema 8. Paso de mensaje síncrono

TEMA 9. Invocación remota (IR) y llamada a procedimiento remoto (RPC)

Invocación remota(llamada a otro procesado): comunicación síncrona. El emisor queda bloqueado esperando por una respuesta del proceso receptor. Cuando esta respuesta tiene lugar, el proceso emisor continúa normalmente.

El flujo de datos puede ser bidireccional. Un proceso puede recibir invocaciones desde cualquier proceso (cliente /servidor).

Comunicación asimétrica(emisor no conoce al receptor, pero el receptor si al emisor)

Llamada Procedimiento Remoto RPC:

Misma idea pero es una llamada a un procedimiento de otra maquina. Soporta comunicación síncrona(fácil) y asíncrona (mas compleja)

Puntos de entrada **entry pe**: llamadas a otros procesos , suele implementarse en colas de prioridad

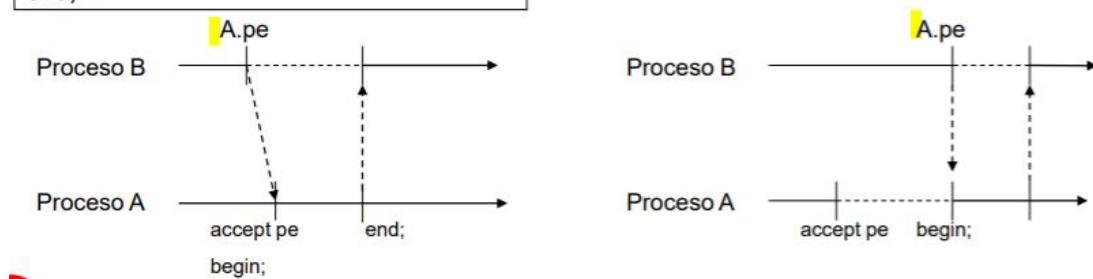
accept :Debe existir al menos una sentencia accept por cada punto de entrada

Si accept contienen null es un mecanismos de sincronizacion

El proceso que realiza la llamada al pe (B) queda bloqueado hasta que el proceso llamada llegue y complete la cita. Si el proceso A llega a su sentencia accept y no se ha efectuada ninguna llamada, también quedara bloqueado.

```
process A;
// se declara un punto de entrada pe
entry pe (a:integer; var b:integer);
begin
...
accept pe(a:integer; var b:integer) do
begin
...
end;
...
end;
```

```
process B;
var
cont:integer;
begin
//llamada al punto de entrada pe del proceso A
A.pe(3,cont);
end;
```



Usos de Select

- No se puede hacer replicate ni llamadas a puntos de entrada
- Con guardas(when),con **terminate**,con **timeout n** ,y con **else** mirar pag 32 de aquí

Productor/Consumidor Invocación Remota

```
program productor-consumidor

process consumidor (ident: integer);
var
    elemento: integer;
begin
repeat
    ...
    Buffer.extraer(elemento);
    ...
forever
end;
```

```
process productor (ident: integer);
var
    elemento: integer;
begin
repeat
    ...
    Buffer.insertar(elemento);
    ...
forever
end;
```

```
process Buffer;
entry extraer(var elemento: integer);
entry insertar(elemento: integer);
var
    buffer: array[0..tam] of integer;
    i,cola, cab, numelem,
    cont1, cont2: integer;
begin
    cola:=0;
    cab:=0;
    numelem:=0;
repeat
select
    when numelem < tam =>
        accept insertar (elemento:
integer) do
            buffer[cab]:=elemento;
            cab:=(cab+1)mod tam;
            numelem:=numelem +1;
    or
```

```
or
when numelem > 0 =>
accept extraer (var elem: integer) do
    elem:=buffer[cola];
    cola:=(cola+1) mod tam;
    numelem:=numelem-1;
or
    terminate;
end;
forever
end;
```

Prioridad en la lectura Invocación Remota

```
process lector(ident : integer);
begin
    repeat
        ...
        controlador.peticion('L');
        controlador.abrir_leitura;
        /*leer del recurso*/
        controlador.cerrar_leitura;
        ...
    forever
end;
```

```
process controlador;
entry ...
var ...
begin
    nl:=0;
    escribiendo:=false;
    repeat
        select
            accept peticion(tipo: char) do
                begin
                    if tipo='L' then
                        nl:=nl+1;
                end;
            or when escribiendo=false =>
                accept abrir_lec do
                    null;
            or accept cerrar_leitura do
                nl:=nl-1;
        end;
```

```
process escritor(ident : integer);
begin
    repeat
        ...
        controlador.peticion('E');
        controlador.abrir_escritura;
        /*escribir en el recurso*/
        controlador.cerrar_escritura;
        ...
    forever
end;
```

```
or when nl=0 and escribiendo=false =>
    accept abrir_esc do
        escribiendo:=true;
    or accept cerrar_esc do
        escribiendo:=false;
    or terminate;
    end;
forever
end;
```

Prioridad en la escritura Invocación Remota

```
process lector(ident : integer);
begin
    repeat
        ...
        controlador.peticion('L');
        controlador.abrir_leitura;
        /*leer del recurso*/
        controlador.cerrar_leitura;
        ...
    forever
end;
```

```
process escritor(ident : integer);
begin
    repeat
        ...
        controlador.peticion('E');
        controlador.abrir_escritura;
        /*escribir en el recurso*/
        controlador.cerrar_escritura;
        ...
    forever
end;
```

```
process controlador;
entry ...
var ...
begin
    nl:=0;escribiendo:=false;nesc:=0;
repeat
    select
        accept peticion(tipo: char) do
            begin
                If tipo='E' then
                    nesc:=nesc+1;
                end;
        or when nesc=0 =>
            accept abrir_lec do
                nl:=nl+1;
        or accept cerrar_lectura do
            nl:=nl-1;
        or when nl=0 and
escribiendo=false =>
            accept abrir_esc do
                escribiendo:=true;
```

```
or accept cerrar_esc do
begin
    escribiendo:=false;
    nesc:=nesc-1;
end;
or terminate;
end;
forever
end;
```

Filósofos Invocación Remota

```
process type filosofo (id : integer);
var
    palillo1, palillo2: integer;
begin
    palillo1:=id;
    palillo2:= (id+1) mod n_filosofos;
repeat
    //pensar
    evitainterbloqueo.entrada;
    palillo[palillo1].coger;
    palillo[palillo2].coger;
    //comer
    palillo[palillo1].soltar;
    palillo[palillo2].soltar;
    evitainterbloqueo.sale;
forever
end;
```

Las llamadas Pe son antendidas en orden de llegada FIFO

```
process type control_palillos;
entry coger;
entry soltar;
begin
repeat
    accept coger do
        null;
    accept soltar do
        null;
forever
end;
```

```
process EvitalInterbloqueos;
entry entra;
entry sale;
var comiendo: integer;
begin
repeat
    select
        when comiendo < N_filosofos =>
            accept entra do
                Comiendo:=comiendo+1;
        or
            accept sale do
                comiendo:=comiendo-1;
        or
            terminate;
    end;
forever
end;
```

En java

```
import messagepassing.*;

public class ServidorPE extends RemoteServer implements Runnable {

    //La clase Selector modela el concepto de espera selectiva.
    Selector Selector1;
    //Me declaro dos puntos de entrada
    EntryPoint EP ;
    EntryPoint ES;
    int arg;

    public ServidorPE () {
        Selector1= new Selector();
        //Creamos un punto de entrada de nombre "Hola"
        EP = new EntryPoint("Hola");
        ES = new EntryPoint("Adios");

        //registramos el punto de entrada
        registerEntryPoint(EP);
        registerEntryPoint(ES);

        //añadimos los puntos de entrada al selector
        //sender - valor booleano indicando que el canal s será utilizado para enviar (true)
        //o para recibir (false)
        Selector1.addSelectable(EP,false);
        Selector1.addSelectable(ES,false);
    }

    public void run(){
        int alternativa;
        while (true)
        {
            //alternativa = Selector1.selectOrBlock();
            //se simula la seleccion aleatoria
            alternativa =Selector1.selectOrTimeout(2);
            System.out.println ("Valor de alternativa "+alternativa);
            if (alternativa ==1)
            {
                System.out.println ("Antes de accept EP");
                EP.accept(new Action()
                {
                    public Object doWork(Object args) {
                        System.out.println ("Ejecutando codigo del servidor ");

                        arg = 2;
                        return "Hola";
                    }
                });
            }
            else
            {
                if (alternativa ==2)
                {
                    System.out.println ("Antes de accept ES");
                    ES.accept();
                    System.out.println ("Antes de replay ES ");
                    arg=2;
                    ES.replay("Adios");
                }
                if (alternativa ==0)
                {
                    System.out.println ("alternativa cero");
                }
            }
        }
    }
}
```

```
import messagepassing.*;  
  
public class ClientePE implements Runnable  
{  
    String salida;  
    String salida1;  
    RemoteServer RS;  
    int id;  
  
    public ClientePE (RemoteServer RS, int id)  
    {  
        this.RS = RS;  
        this.id = id;  
    }  
  
    public void otrastareas()  
    {  
        System.out.println ("Cliente "+id+" Haciendo otras tareas");  
        try {  
            Thread.sleep(100);  
        } catch (InterruptedException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
    }  
  
    public void run()  
    {  
        for (int i=0;i<10;i++)  
        {  
            try {  
                if (id==0)  
                {  
                    try {  
                        Thread.sleep(100);  
                    } catch (InterruptedException e) {  
                        // TODO Auto-generated catch block  
                        e.printStackTrace();  
                    }  
                }  
                System.out.println ("Cliente "+id+" Antes de llamar al PE Hola");  
                salida = (String) RS.call("Hola",id);  
                System.out.println ("Cliente "+id+" despues de llamar al PE Hola "+salida);  
            } catch (RemoteServerException e) {  
                // TODO Auto-generated catch block  
                e.printStackTrace();  
            }  
            otrastareas();  
            try {  
                System.out.println ("Cliente "+id+" Antes de llamar al PE Adios");  
                salida1= (String) RS.call("Adios", id);  
                //RS.call("Adios", id);  
                System.out.println ("Cliente "+id+" despues de llamar al PE Adios "+salida1);  
            } catch (RemoteServerException e) {  
                // TODO Auto-generated catch block  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Main

```
public class Main {  
    public static void main(String[] args) {  
        ClientePE clientes [] = new ClientePE [3];  
        ServidorPE SPE = new ServidorPE();  
        new Thread (SPE).start();  
  
        for (int i=0;i<2;i++) {  
            clientes [i]=new ClientePE (SPE,i);  
  
        }  
        /* Creamos los hilos */  
        for (int i=0;i<2;i++) {  
            new Thread (clientes [i]).start();  
  
        }  
    }  
}
```

TEMA 10 Java RMI

Middleware: software para comunicarse con otros softwares, app, redes...

Modelos de infraestructuras para sistemas distribuidos

Sockets :nivel abstracción bajo, para generar dinámicamente canales de comunicación

Remote Procedure Call (RPC): invocación de procedimientos. Para programación estructurada

Invocación remota de objetos....

RMI (Remote Method Invocation) es la solución Java para la comunicación de objetos Java distribuidos. Tanto el servidor como el cliente.

RMI Ventajas: sin recodificar, más eficientes que peticiones http

Desventajas:El paso de parámetros requiere serialización (lento)

Componentes RMI

Clientes: Localizan e invocan métodos remotos por los servidores. Interactúan con los objetos remotos a través de interfaces remotas

Servidores: Conjunto de objetos que ofrecen interfaces remotas públicas cuyos métodos pueden ser invocados por clientes de cualquier procesador de la plataforma.

Registro: Servicio estático que se establece en cada nodo, en el que se registran los servidores con un nombre, y donde los clientes los localizan por él.

Servicios

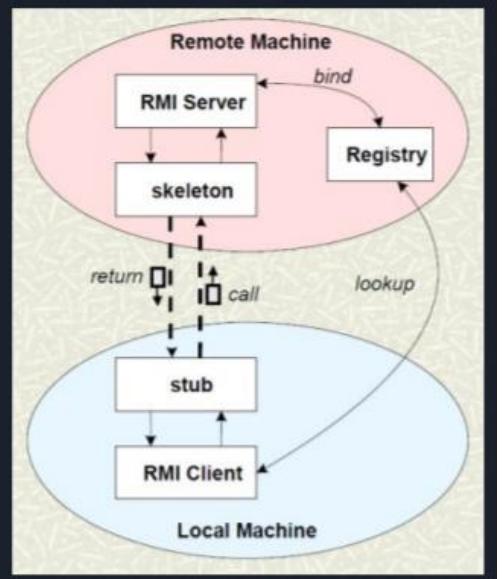
Localizar objetos: **rmiRegistry** para registrar objetos con nombre y luego buscarlos

Comunicación entre objetos remotos : un objeto con referencia remota(stub) de un objeto con interfaz remota puede invocar sus métodos

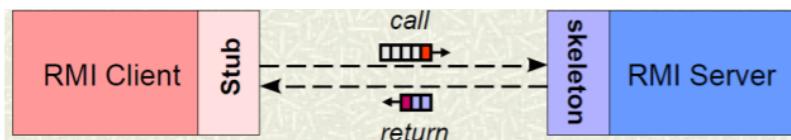
Descargar objetos remotos: RMI ofrece mecanismos para transferir por valor los objetos que se pasan como parámetro de los métodos que se invocan.

Funcionamiento

- **BIND:** Registro del **Servidor** en Registry.
- **LOOKUP:** Localización del **Servidor** en el Registry por parte del **Cliente**.
- **Cliente** crea **Stub** referenciando a **Skeleton**.
- **Cliente** invoca localmente a **Stub** y este la transfiere como mensaje al **Skeleton**.
- **Skeleton** invoca localmente el método del **Servidor** y transfiere al **Stub**, como mensaje, los resultados.
- **Stub** finaliza retornando los resultados al **Cliente**.



Funcionamiento interno Stub-Skeleton



- 1. Un cliente invoca un método remoto invocando localmente el mismo método en el stub.
- 2. El stub genera un mensaje que contiene: la referencia al método y un stream de bytes que resulta de secuenciar los parámetros del método.
- 3. El stub crea dinámicamente un socket y establece la conexión con el skeleton.
- 4. El skeleton recibe el mensaje los decodifica y delega en un thread la invocación del método del servidor. Quedando dispuesto de nuevo a la recepción de un nuevo mensaje.
- 5. El thread genera un mensaje con el stream de bytes que corresponde a la secuenciarización de los resultados de la invocación.
- 6. El thread envía por el socket abierto el mensaje de retorno.
- 7. El stub decodifica el mensaje y concluye la invocación inicial retornando los resultados al cliente.

Pasos para desarrollar una aplicación distribuida RMI

1. Se define la interfaz remota
2. Se desarrolla el servidor que implementa la interfaz remota.
3. Se desarrolla el cliente.

En java 1.8 set path="C:\Program Files\Java\jdk1.8.0_221\bin"

4. Se compilan los ficheros Java fuentes. javac * java
5. Se ejecuta el RMI Registry en el procesador remoto. rmiregistry
6. Se ejecuta el servidor en el procesador remoto. Java nombreclase
7. Se ejecuta el cliente en el procesador local. java nombreclase

Para generar los archivos stub y skeleton utilizar el compilador ->**rmic** nombreclase