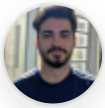


CarlosGarSil98
www.wuolah.com/student/CarlosGarSil98

Tema-2.pdf

Tema 2

3° Programación Concurrente y Distribuida

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingeniería
UHU - Universidad de Huelva

TEMA 2 PRIMERAS APROXIMACIONES A PROBLEMAS DE CONCURRENCIA

- Los procesos concurrentes necesitan reglas de sincronización para controlar sus relaciones.
- Recursos no compatibles ➔ Los procesos compiten (EXCLUSIÓN MUTUA).
- Condición de sincronización: los procesos cooperan compartiendo información.

TIPOS DE SINCRONIZACIÓN Y SU SOLUCIÓN

EXCLUSIÓN MUTUA: Acción para la sincronización entre 2 procesos o más, para compartir uso de un recurso.

SECCIÓN CRÍTICA: Parte del código de un proceso que hace uso del recurso no compartible, ejecución en exclusión mutua.

Para garantizar la exclusión mutua se debe cumplir:

EXCLUSIÓN MUTUA: No puede haber 2 o más procesos a la vez en la sección crítica.

LIMITACIÓN DE ESPERA: No se puede esperar de forma indefinida.

PROGRESO EN LA EJECUCIÓN: Si la sección crítica está libre un proceso podrá entrar.

CONDICÓN DE SINCRONIZACIÓN: Propiedad para que un proceso no realiza una acción hasta que otro proceso realice antes otra acción.

• Espera ocupada • Semáforos • Regiones Críticas Condicionales • Monitores	Memoria compartida
• Operaciones de paso de mensajes • Invocaciones remotas	Paso de Mensajes

SOLUCIONES DE ESPERA OCUPADA

Implementa la sincronización con un proceso de espera que comprueba constantemente una variable, manteniendo ocupada a la CPU. Hay dos tipos de soluciones:

- **Soluciones SOFTWARE:** Instrucciones atómicas de leer y escribir (ENSAMBLADOR).
- **Soluciones HARDWARE:** Instrucciones especiales.

SOLUCIONES SOFTWARE

Estructura de un programa en pascal FC:

La exclusión mutua está asegurada gracias a dos secuencias de instrucciones:

PROTROCOLO DE ENTRADA: Comprueba la condición para autorizar la entrada.

PROTOCOLO DE SALIDA: Indica que se ha terminado de ejecutar la S.C.

Las condiciones indispensables para que se cumpla la exclusión mutua son:

1. El proceso no puede pararse durante la ejecución de los protocolos o en S.C.
2. No debe bloquearse cuando dos procesos intenten entrar en sus S.C.
3. Cada proceso debe tener éxito en la entrada de la S.C.
4. Un único proceso desee entrar en su S.C tendrá éxito y entrará en ella.

```
process P1
begin
    [**********************]
    (*Protocolo de entrada*)
    --SECCION CRITICA
    [**********************]
    (*Protocolo de salida*)
end;
```

```
program EM
    var
        v:integer;
        (*VARIABLES GLOBALES*)

    process P1;
    var
        (*VARIABLES*)
    begin
        (*PROCEDIMIENTOS*)
    end;

    process Pn;
    var
        (*VARIABLES*)
    begin
        (*PROCEDIMIENTOS*)
    end;

var
    (*VARIABLES LOCALES*)
begin
    (*PROCEDIMIENTO DEL MAIN*)
end.
```

ALGORITMO NO EFICIENTE Primer intento

```
process P0;
begin
    repeat
        --Protrocolo de entrada
        while seccionCriticaOcupada = true do
            (*Espera Ocupada*) ;
        seccionCriticaOcupada := true;

        (*SECCION CRITICA*)

        --Protocolo de salida
        seccionCriticaOcupada := false;
    until(true);
    Resto0;
end;
```

Puede ser concurrente

```
process P1;
begin
    repeat
        --Protrocolo de entrada
        while seccionCriticaOcupada = true do
            (*Espera Ocupada*) ;
        seccionCriticaOcupada := true;

        (*SECCION CRITICA*)

        --Protocolo de salida
        seccionCriticaOcupada := false;
    until(true);
    Resto1;
end;
```

Si alternamos las acciones, ambos procesos encuentran la sección crítica libre y entran a la vez. Por tanto, hay falta de exclusión.

ALGORITMO NO EFICIENTE Segundo intento

```
process P0;
begin
    repeat
        --Protrocolo de entrada
        while turno = 1 do
            --No hace nada;

        (*SECCION CRITICA*)

        --Protocolo de salida
        turno := 1;
    until(true);
    Resto0;
end;
```

```
process P1;
begin
    repeat
        --Protrocolo de entrada
        while turno = 0 do
            --No hace nada;

        (*SECCION CRITICA*)

        --Protocolo de salida
        turno := 0;
    until(true);
    Resto1;
end;
```

No satisface la condición de progreso. Cuando alguno de los procesos acaba antes que el otro o falla, se produce una espera ilimitada. No cumple el "Progreso en la ejecución".

ALGORITMO NO EFICIENTE Tercer intento

```
process P0;
begin
    repeat
        --Protrocolo de entrada
        while c1 = enSeccionCritica do
            (*Espera Ocupada*) ;
        c0 := enSeccionCritica;

        (*SECCION CRITICA*)

        --Protocolo de salida
        c0 := fueraSeccionCritica;

        Resto0;
    until(true);
end;
```

```
process P1;
begin
    repeat
        --Protrocolo de entrada
        while c0 = enSeccionCritica do
            (*Espera Ocupada*) ;
        c1 := enSeccionCritica;

        (*SECCION CRITICA*)

        --Protocolo de salida
        c1 := fueraSeccionCritica;

        Resto1;
    until(true);
end;
```

Cuando se van alternando las instrucciones, ambos procesos entran a la vez en la sección crítica. No se garantiza la exclusión mutua.

ALGORITMO NO EFICIENTE **Cuarto intento**

```
process P0;
begin
    repeat
        --Protrocolo de entrada
        c0 := quiereEntrar;
        while c1 = quiereEntrar do
            (*Espera Ocupada*) ;

        (*SECCION CRITICA*)

        --Protocolo de salida
        c0 := fueraSeccionCritica;

        Resto0;
    until(true);
end;
```

```
process P1;
begin
    repeat
        --Protrocolo de entrada
        c1 := quiereEntrar;
        while c0 = quiereEntrar do
            (*Espera Ocupada*) ;

        (*SECCION CRITICA*)

        --Protocolo de salida
        c1 := fueraSeccionCritica;

        Resto1;
    until(true);
end;
```

Cuando se van alternando las instrucciones, ambas quieren entrar en la S.C pero ambas se quedan dentro del WHILE, generando un bloqueo (LIVEBLOCK).

ALGORITMO NO EFICIENTE **Quinto intento (Solución parcial)**

```
process P0;
begin
    repeat
        --Protrocolo de entrada
        c0 := quiereEntrar;
        while c1 = quiereEntrar do
        begin
            c0 := fueraSeccionCritica;
            (*Realiza una espera*)
            c0 := quiereEntrar;
        end;

        (*SECCION CRITICA*)

        --Protocolo de salida
        c0 := fueraSeccionCritica;

        Resto0;
    until(true);
end;
```

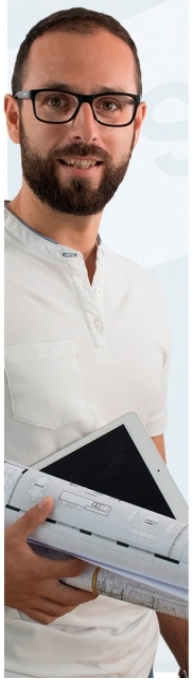
```
process P1;
begin
    repeat
        --Protrocolo de entrada
        c1 := quiereEntrar;
        while c0 = quiereEntrar do
        begin
            c1 := fueraSeccionCritica;
            (*Realiza una espera*)
            c1 := quiereEntrar;
        end;

        (*SECCION CRITICA*)

        --Protocolo de salida
        c1 := fueraSeccionCritica;

        Resto1;
    until(true);
end;
```

No garantiza el acceso de tiempo finito. Por tanto, es imposible conocer su eficiencia.

ALGORITMO DE DEKKER (1965) **Mezcla de la segunda y cuarta opción.**

```
process P0;
begin
    repeat
        c0 := quiereEntrar;
        while c1 = quiereEntrar do begin
            if turno = 1 them begin
                c0 = fueraSeccionCritica;
                while turno = 1 do
                    (*Espera ocupada*);
                c0 := quiereEntrar;
            end;
        end;

        (*SECCION CRITICA*)

        turno := 1;
        c0 := fueraSeccionCritica;

        Resto0;
    until(true);
end;
```

```
process P1;
begin
    repeat
        c1 := quiereEntrar;
        while c0 = quiereEntrar do begin
            if turno = 0 them begin
                c1 := fueraSeccionCritica;
                while turno = 0 do
                    (*Espera ocupada*);
                c1 := quiereEntrar;
            end;
        end;

        (*SECCION CRITICA*)

        turno := 0;
        c1 := fueraSeccionCritica;

        Resto1;
    until(true);
end;
```



ALGORITMO DE PETERSON (1981)

```
process P0;
begin
    repeat
        c0 := quiereEntrar;
        turno := 1;
        while (c1 = quiereEntrar) and (turno = 1) do
            (*Espera ocupada*);
        end;

        (*SECCION CRITICA*)

        c0 := fueraSeccionCritica;

        Resto0;
    until(true);
end;
```

```
process P1;
begin
    repeat
        c1 := quiereEntrar;
        turno := 0;
        while (c0 = quiereEntrar) and (turno = 0) do
            (*Espera ocupada*);
        end;

        (*SECCION CRITICA*)

        c1 := fueraSeccionCritica;

        Resto1;
    until(true);
end;
```

ALGORITMO INCORRECTO DE HYMAN (1966)

```
process P0;
begin
    repeat
        c0 := quiereEntrar;
        while turno <> 0 do begin
            while c1 = quiereEntrar do begin
                (*Espera ocupada*);
            end;
            turno := 0;
        end;

        (*SECCION CRITICA*)

        c0 := fueraSeccionCritica;

        Resto0;
    until(true);
end;
```

```
process P1;
begin
    repeat
        c1 := quiereEntrar;
        while turno <> 1 do begin
            while c0 = quiereEntrar do begin
                (*Espera ocupada*);
            end;
            turno := 1;
        end;

        (*SECCION CRITICA*)

        c1 := fueraSeccionCritica;

        Resto1;
    until(true);
end;
```

ALGORITMO DE EISENBERG–McGUIRE (1972)

```
process PI;
var
    i,j:integer;
begin
    repeat
        repeat
            indicador[i] := quiereEntrar;
            j := indice;
            while j <> i do begin
                if incador[j] <> fueraSeccionCritica then
                    j := indice;
                else
                    j := (j+1) mod N;
                end;
            end;
            indicador[i] := enSeccionCritica;
            j := 0;
            while (j<n) AND ((j=i) OR (indicador[j] <> enSeccionCritica)) do j := j+1;
        until((j>n) AND ((indice=i) OR (incador[indice] = fueraSeccionCritica)));
        indice := i;

        (*SECCION CRITICA*)

        j := (indice+1) mod N;
        while indicador[j] = fueraSeccionCritica do j := (j+1) mod N;
        indice := j;
        indicador[i] := fueraSeccionCritica;
        RESTOi;
    until(true);
end;
```

Índice es el id del proceso que tiene el turno

Busca de forma circular los que quieren entrar. Solo sale si ningún otro quiere entrar

Asegura que no hay nadie en la sección crítica

Busca a otro que quiera entrar

ALGORITMO DE LAMPORT (1974) **Algoritmo de la Panadería**

```
process Pi;
var
    i,j: integer;
begin
    repeat
        c[i] := cogeNumero;
        numero[i] := 1+max(numero[0],...,numero[n-1]);
        c[i] := nocogeNumero;
        for j=0 to n-1 do begin
            while c[j] = cogeNumero do
                (*Espera a que el anterior acabe de coger numero*);
            while (numero[j] <> 0) AND ((numero[i],i) > (numero[j],j)) do
                (*Espera Ocupada*);
        end;

        (*SECCION CRITICA*)

        numero[i] := 0;
        RESTOi;
    until(true);
end;
```

Dos hilos con el mismo numero

Solución

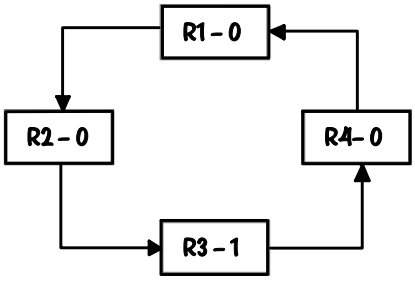
Comprueba el número, si no, prioridad al menor indice

SOLUCIONES HARDWARE

EXCHANGE **La instrucción Exchange(r,m) intercambia de forma atómica el contenido de las posiciones de memoria m y r.**

```
process P0;
var
    r0 := 0;
    m := 1;
begin
    repeat
        repeat
            exchange(r0,m);
        until(r0=1);

        (*SECCION CRITICA*)

        exchange(r0,m);
        Resto0;
    until(true);
end;
```

```
process P1;
var
    r1 := 0;
    m := 1;
begin
    repeat
        repeat
            exchange(r1,m);
        until(r1=1);

        (*SECCION CRITICA*)

        exchange(r1,m);
        Resto1;
    until(true);
end;
```



DECREMENTO La instrucción subc(r,m) decrementa en 1 el contenido de m y copia el resultado en r, de forma atómica.

```
process P=0;
begin
    repeat
        repeat
            subc(r0,m);
        until(r0=0);

        (*SECCION CRITICA*)

        m := 1;
        Resto0;
    until(true);
end;
```

```
process P=1;
begin
    repeat
        repeat
            subc(r1,m);
        until(r1=0);

        (*SECCION CRITICA*)

        m := 1;
        Resto1;
    until(true);
end;
```

INCREMENTO La instrucción addc(r,m) incrementa en 1 el contenido de m y el resultado lo copia en r, de forma atómica.

```
process P0;
begin
    repeat
        repeat
            addc(r0,m);
        until(r0=0);

        (*SECCION CRITICA*)

        m := -1;
        Resto0;
    until(true);
end;
```

```
process P1;
begin
    repeat
        repeat
            addc(r1,m);
        until(r1=0);

        (*SECCION CRITICA*)

        m := -1;
        Resto1;
    until(true);
end;
```

TESTSET La instrucción testset(m) realiza la siguiente secuencia de acciones de forma atómica:

- Comprueba el valor de la variable m.
- Si el valor es 0 lo cambia por 1 y devuelve como resultado true.
- En otro caso no modifica el valor y devuelve false.

```
process P0;
begin
    repeat
        repeat
            (*Espera Ocupada*);
        until(testset(m));

        (*SECCION CRITICA*)

        m := 0;
        Resto0;
    until(true);
end;
```

```
process P1;
begin
    repeat
        repeat
            (*Espera Ocupada*);
        until(testset(m));

        (*SECCION CRITICA*)

        m := 0;
        Resto1;
    until(true);
end;
```