Concurrencia y Exclusión Mutua

Dr. Roberto Gómez Cárdenas

ITESM-CEM

rogomez@itesm.mx

http://homepage.cem.itesm.mx/rogomez

Programas secuenciales vs concurrentes

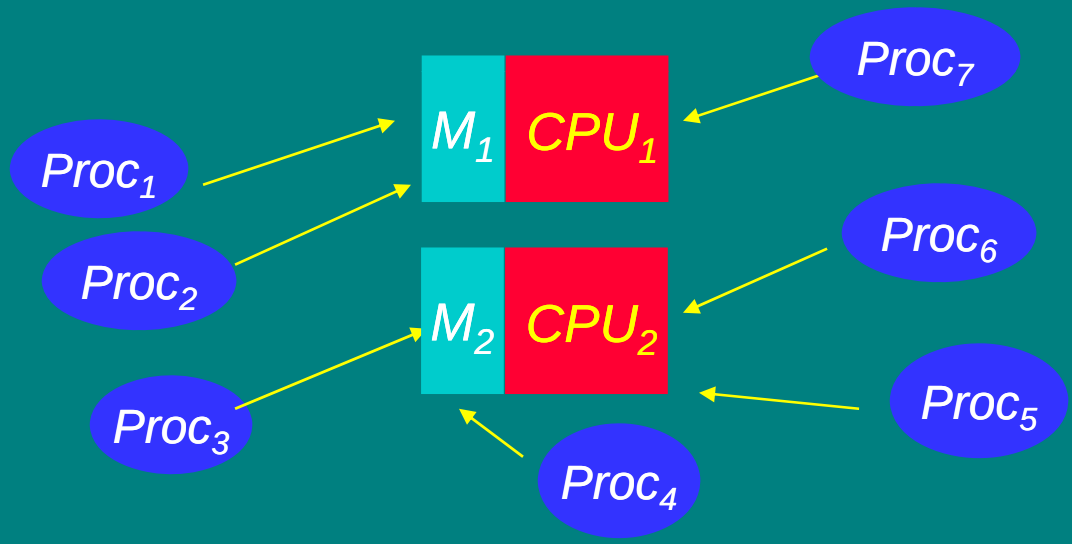
Características procesos concurrentes

- *Los procesos son concurrentes si existen simultáneamente*
- *Pueden funcionar en forma totalmente independiente, unos de otros*
- *Pueden ser asincronos lo cual significa que en ocasiones requieren cierta sincronización y cooperación*

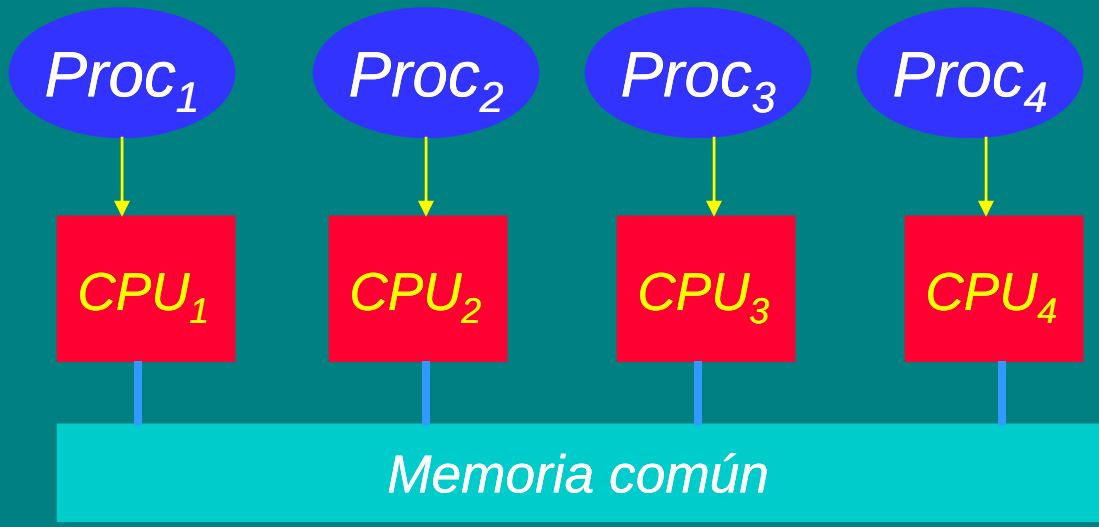
Clasificación procesos concurrentes

- *Multiprogramación*
- *Multiprocesamiento*
- *Procesamiento distribuido*

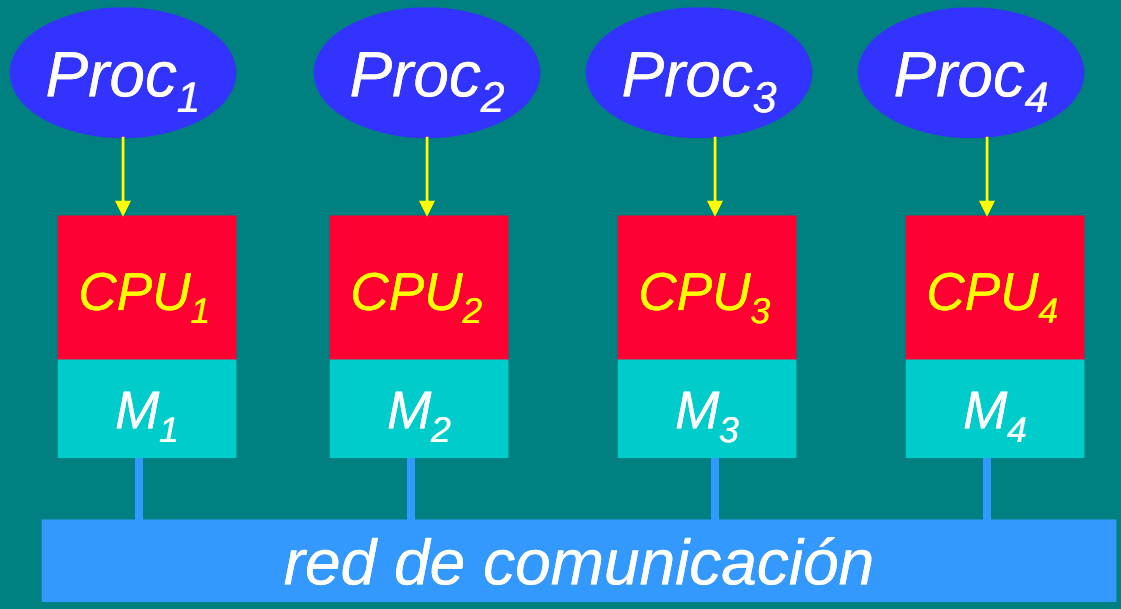
Multiprogramación



Multiprocesamiento



Procesamiento distribuido



Interacción entre procesos

- *Variables compartidas.*
- *Envío de mensajes*

La sincronización

Notación expresar concurrencia

- *¿Cómo indicar una ejecución concurrente?*
- *¿Qué modo de comunicación entre procesos se va a utilizar?*
- *¿Cuál mecanismo de sincronización se va a usar?*

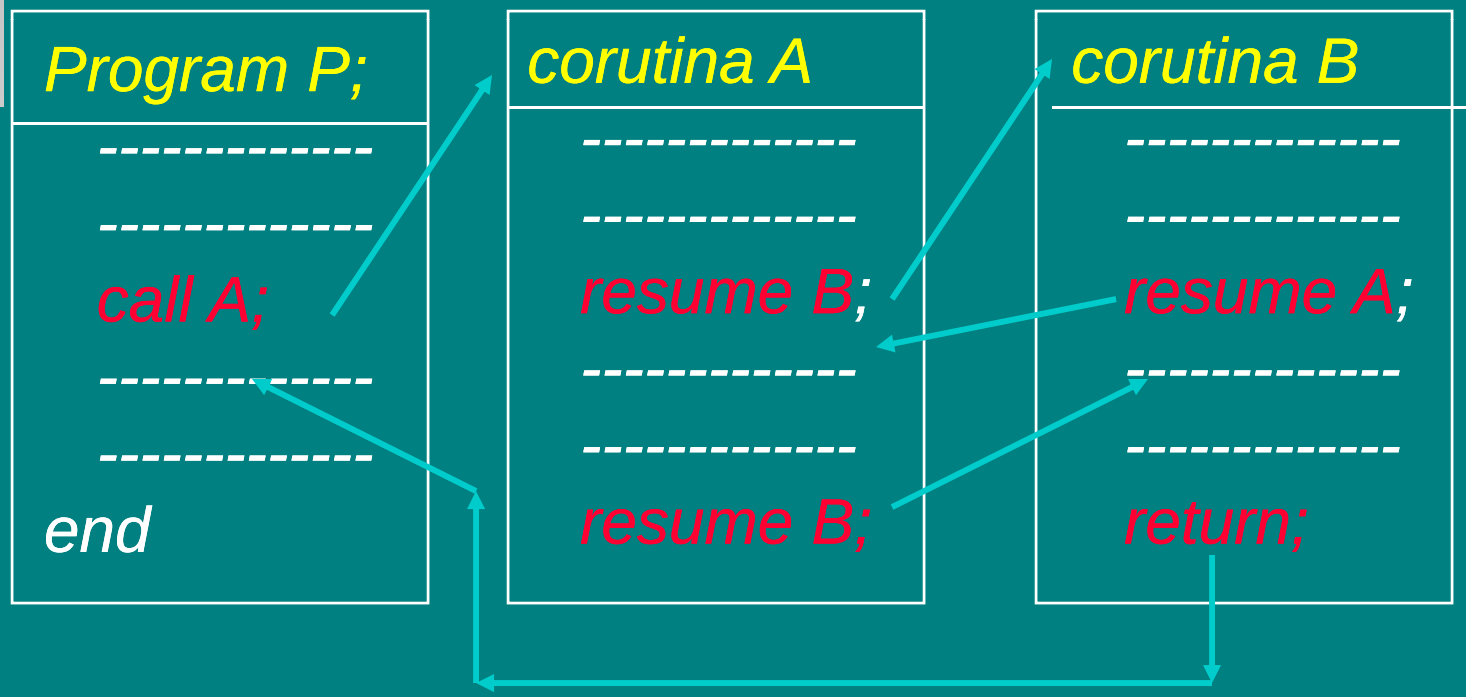
Especificando ejecución concurrente

- *Existen varias notaciones*
  - *corutinas*
  - *fork-join*
  - *el enunciado co-begin*
- *Importante diferenciar:*
  - *la definición del proceso*
  - *de la sincronización del proceso*

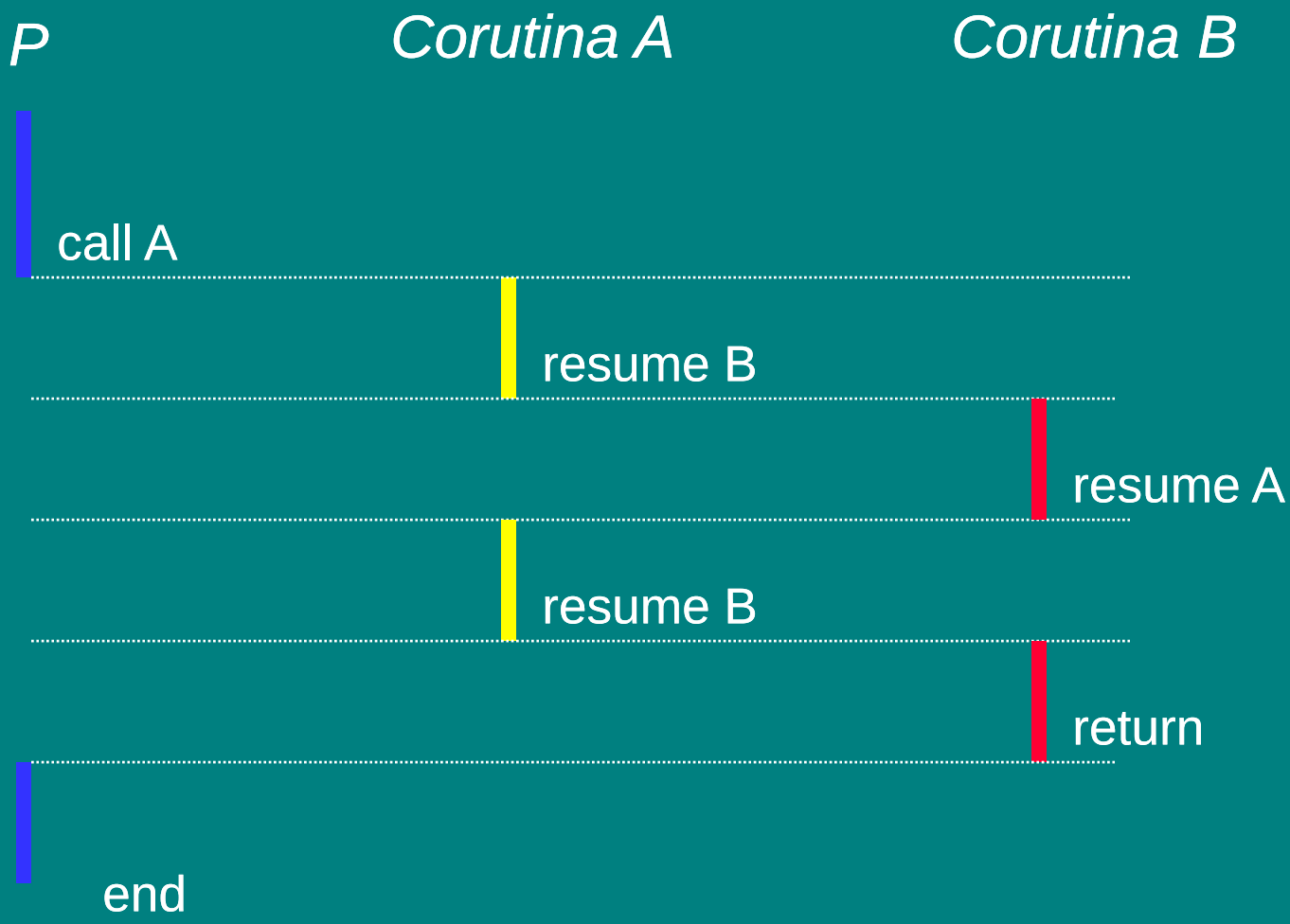
Elementos corutinas

- *El enunciado resume*
  - *transfiere control a la corutina mencionada*
  - *guarda información necesaria para controlar la ejecución de regreso*
- *El enunciado call*
  - *inicializa el cálculo de la corutina*
- *El enunciado return*
  - *transfiere el control de regreso al procedimiento que realizó un call*

Ejemplo corutinas



La ejecución



Enunciados fork-join

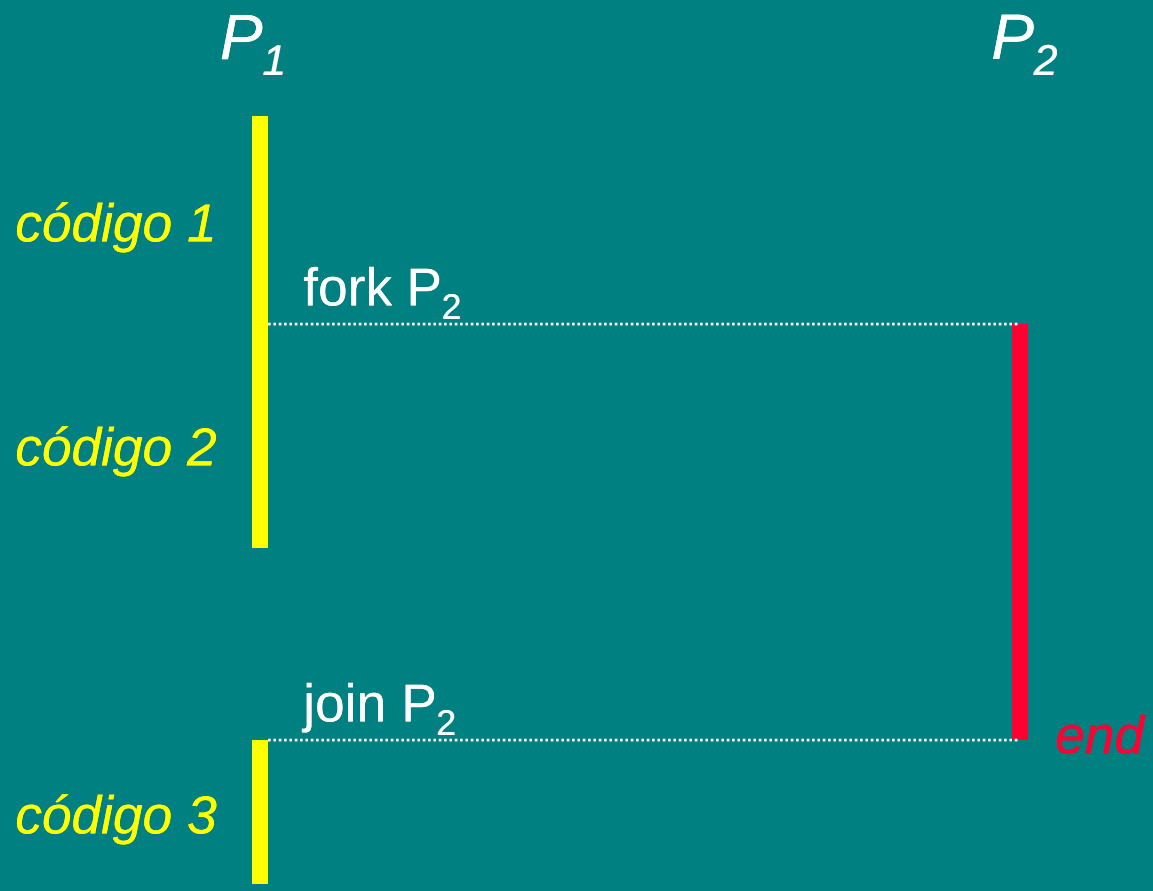
- *Enunciado fork especifica que una rutina puede empezar su ejecución*
- *La rutina invocada y la rutina invocadora proceden concurrentemente*
- *Para sincronizar invocada e invocadora, esta última puede ejecutar un join*
- *Enunciado join retrasa ejecución rutina invocadora hasta que la rutina invocada termine*

Ejemplo fork-join

```
program P1
    ------------
    <codigo 1>
    fork P2;
    ------------
    <codigo 2>
    join P2;
    ------------
    <codigo 3>
```

```
program P2
    ------------
    ------------
    ------------
    ------------
    ------------
    ------------
end
```

La ejecución



El enunciado cobegin

- *Es una forma estructurada de denotar una ejecución concurrente*
- *Por ejemplo:*

  cobegin S_1 || S_2 || … || S_n coend

  - *denota una ejecución concurrente de S_1, S_2, … S_n*
  - *cada uno de los S_i's puede ser cualquier enunciado incluyendo un cobegin o un bloque con declaraciones locales*

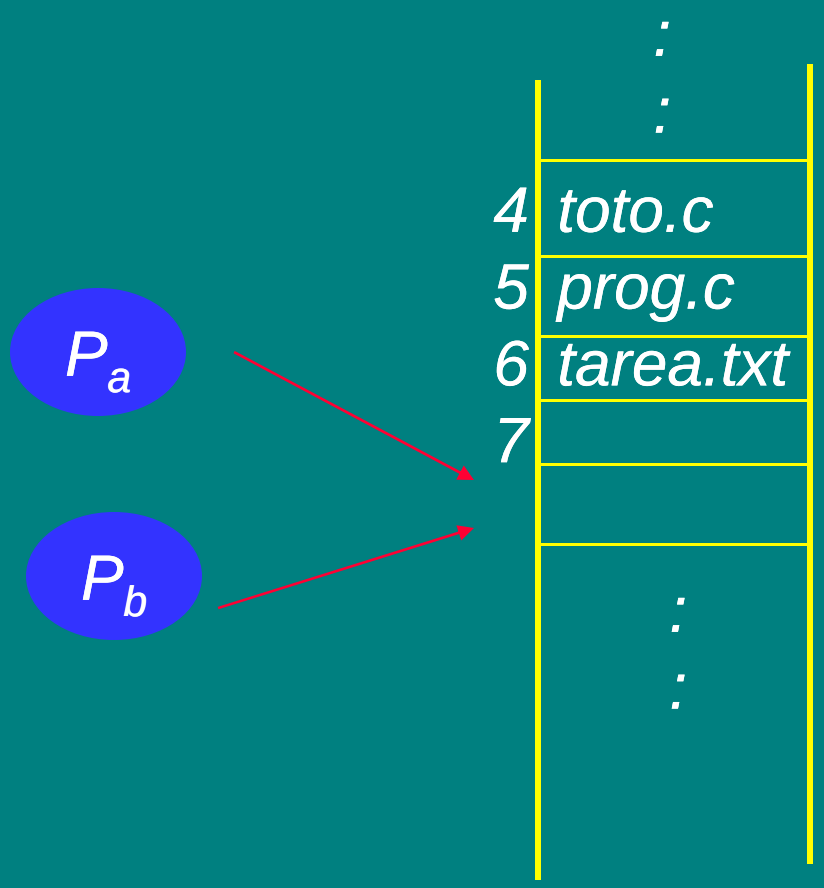
Comunicación y sincronización procesos

- *Procesos requieren comunicación entre ellos.*
- *Procesamiento distribuido*
  - *envío/recepción de mensajes*
- *Multiprocesamiento:*
  - *pipes: salida proceso es entrada de otro*
  - *uso de variables compartidas o de algún espacio en común donde puedan leer o escribir los procesos*

Ejemplo: spooler impresión

- *Cuando un proceso desea imprimir un archivo escribe el nombre de este en un directorio*
- *Otro proceso, demonio impresión (lpd) verifica si existen archivos por imprimir, los imprime y elimina sus nombres del directorio*

Elementos spooler impresión



out = 4

in = 7

P_a *desea imprimir* t_1

P_b *desea imprimir* t_2

Pasos en impresión archivos

Proceso A

lee valor in y almacena en next
next := 7

termina quantum

Proceso B

lee valor in y almacena en sig
sig := 7
actualiza valor in (in = 8)
escribe t_2 en localidad 7
sale y hace otras cosas

actualiza valor in (in = 8)
escribe t_1 en localidad 7
sale y hace otras cosas

termina quantum

Conceptos sincronía

- *Condiciones de competencia.*
- *La exclusión mutua.*
- *Sección crítica.*

Asignación recursos

- *Sean n procesos que entran en conflicto por el acceso a un recurso único no compartible*
- *Recurso a usar en sección crítica*
- *Necesario usar un protocolo formado de tres partes:*

protocolo de adquisición
<uso del recurso en sección crítica>
protocolo de liberación

Posibles problemas

- *Interbloqueo: los procesos se bloquean entre si y nadie tiene acceso al recurso, no hay actividad en ninguno de los procesos*
  - *si estoy solo paso*
  - *si no: dejo pasar a los demás*
- *Hambruna (starvation)*
  - *sean tres procesos: P_1 P_2 y P_3*
  - *si el recurso siempre se le asigna, alternativamente, a P_1 y P_2, P_3 no tendrá acceso a él*

Condiciones solución mutex

- *Dos procesos no deben encontrarse al mismo tiempo dentro de sus secciones críticas*
- *No se deben hacer hipótesis sobre la velocidad o el número de CPUs*
- *Ninguno de los procesos que estén en ejecución fuera de su SC puede bloquear a otros procesos*
- *Ningún proceso debe esperar eternamente para entrar a su SC*

Posibles soluciones

- *Des-activación de interrupciones*
- *Variables de cerradura*
- *Alternancia estricta*
- *Esperando que el otro termine*
- *Cediendo el lugar al otro*
- *Esperando un tiempo aleatorio*
- *Algoritmo de Dekker*
- *Algoritmo de Peterson*

Des-activación interrupciones

- *Solución más simple: cada proceso desactivará todas sus interrupciones justo antes de entrar en sección crítica y las activará al salir de ella*
- *Con las interrupciones desactivadas no puede ocurrir una interrupción de reloj*
- *Problema: un proceso (usuario) podría desactivarlas y nunca activarlas de nuevo*
- *Si hay más de dos CPUs, la desactivación sólo afecta a uno de ellos*

Variables de cerradura

- *Antes proceso entre a su sección crítica se hace prueba a una variable tipo cerradura:*
  - *si esta es 0: proceso la cambia a 1 y entra a su sección crítica*
  - *si esta es 1: espera hasta que la variable cambie su valor a 0*
- *Problema*
  - *proceso P1 lee un valor de 0*
  - *antes modifique el valor se le acaba el quatum*
  - *proceso P2 también va a leer un valor de 0*
  - *se van a tener dos procesos en S.C.*

Alternancia estricta

```
void proc1()
{
    while (TRUE) {
        while (turno != 0);
        seccion_critica(P1);
        turno=1;
        fuera_sec_crit(P1);
    }
}
```

```
void proc2()
{
    while (TRUE) {
        while (turno != 1);
        seccion_critica(P2);
        turno=0;
        fuera_sec_crit(P2);
    }
}
```

```
main(){
   turn=1;
   cobegin proc1() || proc2()  coend
}
```

Detalles alternacia estricta

- *La prueba de una variable, en espera de que tome cierto valor se conoce como espera activa*
  - *esto provoca un desperdicio del CPU*
- *¿Qué pasa cuando uno de los procesos es más lento que el otro?*
  - *en lo que proc1 ejecuta fue_sec_crit(p1) el proceso proc2 ejecuta sec_crit(p2), cambia valor a turno, ejecuta fue_sec_crit(p2) y regresa al while*
  - *se van a tener un proceso bloqueado por otro que no esta en su sección crítica*

Esperando que el otro termine

```
int p1in, p2in;
void proc1()
{
        while (TRUE) {
                while (p2in == 1);
                p1in = 1;
                sección_critica_P1();
                p1in = 0;
                out_sec_critica_P1();
        }
}
```

Los dos procesos

```
int   p1in = 0;
Int  p2in = 0;
```

```
void proc1()
{
   while (TRUE) {
        while (p2in == 1);
        p1in = 1;
        sección_critica_P1();
        p1in = 0;
        out_sec_critica_P1();
    }
}
```

```
void proc2()
{
   while (TRUE) {
        while (p1in == 1);
        p2in = 1;
        sección_critica_P2();
        p2in = 0;
        out_sec_critica_P2();
    }
}
```

Comentarios

- *Dos variables*
  - *p1in: proceso 1 en sección crítica*
  - *p2in: proceso 2 en sección crítica*
- *¿Qué pasa si se ejecuta en tandem?*
  - *se van a tener dos procesos en sección crítica*

Cediendo el lugar al otro

```
int p1deseaentar, p2deseaentrar;
void proc1()
{
        while (TRUE) {
                p1deseaentrar = 1;
                while (p2deseaentrar == 1);
                sección_critica_P1();
                p1deseaentrar = 0;
                out_sec_critica_P1();
        }
}
```

Los dos procesos

```
int    p1deseaentar  = 1;
Int    p2deseaentrar  =1;
```

```
void proc1()
{
    while (TRUE) {
        p1deseaentrar = 1;
        while (p2deseaentrar == 1);
        sección_critica_P1();
        p1deseaentrar = 0;
        out_sec_critica_P1();
    }
}
```

```
void proc2()
{
    while (TRUE) {
        p2deseaentrar = 1;
        while (p1deseaentrar == 1);
        sección_critica_P1();
        p2deseaentrar = 0;
        out_sec_critica_P2();
    }
}
```

Comentarios

- *El proceso indica su deseo de entrar a sección crítica*
- *Si el otro proceso también quiere entrar le cede el paso*
- *Problema ejecución tandem:*
  - *si cada proceso asigna un 1 a su bandera antes de verificar, cada proceso encontrará la bandera del otro con valor 1 los dos entrarán en un ciclo infinito*
  - *se da un bloqueo mutuo (interbloqueo o deadlock)*

Esperando un tiempo aleatorio

```
int p1deseaentar, p2deseaentrar;
void proc1()  {
        while (TRUE) {
                p1deseaentrar = 1;
                while (p2deseaentrar == 1) {
                        p1deseaentrar = 0;
                        retraso(aleatorio, algunosciclos);
                        p1deseaentrar = 1;
                }
                sección_critica_P1();
                p1deseaentrar = 0;
                out_sec_critica_P1();   }
}
```

Comentarios

- *Se obliga a cada proceso a asignar repetidamente el valor falso a su bandera por periodos cortos, calculados de forma aleatoria*
- *Problema: una ejecución tandem y que el valor aleatorio será el mismo para los dos procesos*
- *Situación muy poco probable pero posible*

Algoritmo Dekker

```
void proc1()
{
    while (TRUE) {
        p1_deseaentrar = TRUE;
        while (p2_deseaentrar == TRUE)
            if (proc_favorecido == SEGUNDO) {
                p1_deseaentrar = FALSE;
                while (proc_favorecido == SEGUNDO) ;
                p1_deseaentrar = TRUE;
            }
        seccion_critica_P1();
        proc_favorecido = SEGUNDO;
        p1_deseaentrar = FALSE;
        out_seccion_critica_P1();
    }
}
```

Algoritmo Petterson

```
void proc1()
{
    while (TRUE)  {
        p1_deseaentrar = TRUE;
        proc_favorecido = SEGUNDO;
        while (p2_deseaentrar == TRUE) and
                                    (proc_favorecido == SEGUNDO);
        seccion_critica_P1();
        p1_deseaentrar = FALSE;
        out_seccion_critica_P1();
    }
}
```