

Arquitectura y Sistemas Operativos

UTN MAR DEL PLATA

CONCURRENCIA

ACIERNO – GIMENEZ - ORELLANO

PROCESOS CONCURRENTES

Los temas centrales del diseño de sistemas operativos están relacionados con la gestión de procesos e hilos:

- ✓ ***Multiprogramación.** Gestión de múltiples procesos dentro de un sistema mono procesador.*
- ✓ ***Multiprocesamiento.** Gestión de múltiples procesos dentro de un multiprocesador.*
- ✓ ***Multiprocesamiento distribuido.** Gestión de múltiples procesos que ejecutan sobre múltiples sistemas de cómputo distribuidos.*

PROCESOS CONCURRENTES

La concurrencia aparece en tres contextos:

- ✓ *Múltiples aplicaciones.* La multiprogramación fue ideada para permitir compartir dinámicamente el tiempo de procesamiento entre varias aplicaciones.
- ✓ *Aplicaciones estructuradas.* Algunas aplicaciones pueden ser programadas como un conjunto de procesos concurrentes.
- ✓ *Estructura del Sistema Operativo.* Los sistemas operativos son muchas veces implementados como un conjunto de procesos o hilos.

PROCESOS CONCURRENTES

Se plantean las siguientes dificultades:

- 1. La compartición de recursos globales está llena de peligros.*
- 2. Para el S.O. es complicado gestionar la asignación de recursos de manera óptima.*
- 3. Llega a ser muy complicado localizar errores de programación porque los resultados son no deterministas y no reproducibles.*

ALGUNOS TÉRMINOS RELACIONADOS CON LA CONCURRENCIA



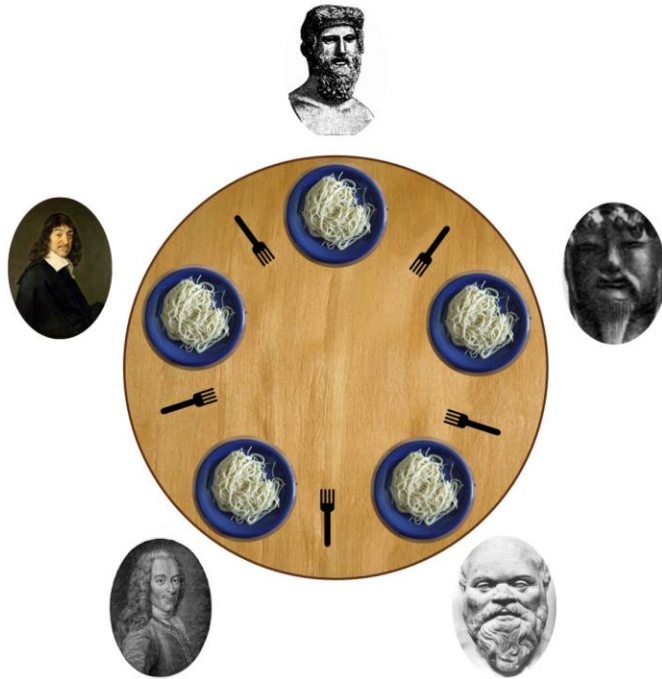
-
- ✓ *Sección Crítica (Critical section):* sección de código de un proceso que requiere acceso a recursos compartidos.
 - ✓ *Interbloqueo (Deadlock):* situación donde dos o mas procesos son incapaces de actuar porque cada uno está esperando que alguno haga algo.
 - ✓ *Círculo vicioso (Live lock):* dos o mas procesos cambian continuamente su estado en respuesta a cambios en los otros procesos, sin realizar trabajo útil.

ALGUNOS TÉRMINOS RELACIONADOS CON LA CONCURRENCIA



-
- ✓ **Exclusión mutua (mutual exclusion):** requisito fundamental que evita que si un proceso esta en su sección crítica, ningún otro proceso pueda entrar en su sección crítica.
 - ✓ **Condición de carrera (race condition):** situación donde múltiples hilos o procesos leen y escriben un dato compartido y el resultado final depende de la coordinación de sus ejecuciones.
 - ✓ **Inanición (Starvation).** Un proceso preparado para avanzar es soslayado indefinidamente por el planificador, aunque puede avanzar nunca se escoge.

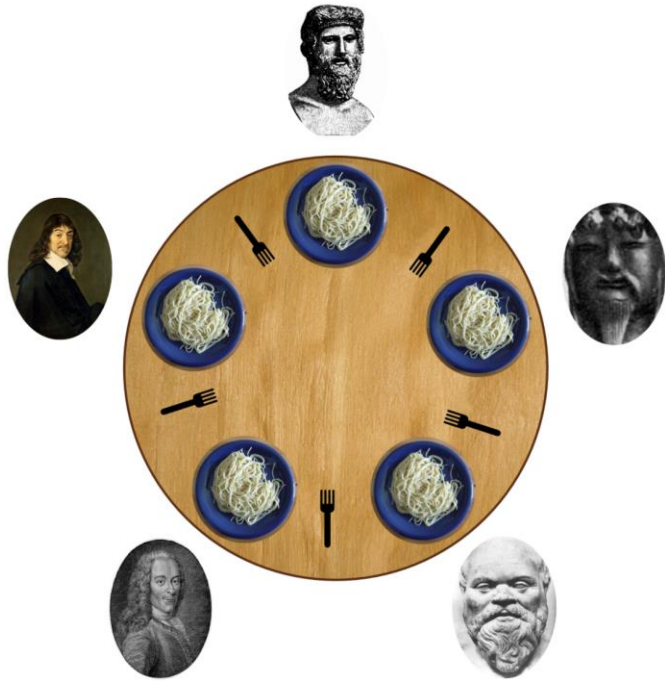
EJEMPLO: CENA DE LOS FILÓSOFOS



Enunciado:

Cinco filósofos se sientan alrededor de una mesa a comer y pensar. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato. Para comer los fideos son necesario dos tenedores y cada filósofo puede tomar los que están a su izquierda o derecha. Si un filósofo toma un tenedor y el otro esta ocupado, se quedará esperando con el tenedor en la mano hasta que pueda tomar el otro.

EJEMPLO: CENA DE LOS FILÓSOFOS



¿Qué pasa si dos filósofos intentan tomar el mismo tenedor a la vez?

- Se produce una **condición de carrera**: ambos compiten y uno de ellos se queda sin comer.

¿Y si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo?


- Se quedarán esperando eternamente, por lo tanto se produce un **interbloqueo o deadlock**.

EXCLUSIÓN MUTUA



Para entender la exclusión mutua, veamos este ejemplo:

```
int contador = 0;
if (evento)
    contador++;
    mov AX, contador //leer contador
    inc contador //incrementar contador
    mov contador, AX //actualizar
                      //contador
```

An orange diagram consisting of a vertical line with an arrow pointing right to the 'mov AX, contador' line, and a horizontal line with an arrow pointing left to the 'contador++;' line, illustrating a race condition where the update is lost.

Un proceso podría ser desalojado antes de actualizar el contador y por lo tanto se podría perder información.

EXCLUSIÓN MUTUA

Se debe garantizar la exclusión mutua. Cualquier mecanismo que proporcione la exclusión debe cumplir los siguientes requisitos:

- 1. Sólo se permite un proceso al mismo tiempo dentro de su sección crítica para el mismo recurso u objeto compartido.*
- 2. Un proceso que no está en su sección crítica no debe impedir que otro proceso acceda a la suya.*
- 3. No puede pasar que un proceso que solicite acceso a la S.C. sea postergado indefinidamente (ni inanición ni interbloqueo).*

EXCLUSIÓN MUTUA

¿Más requisitos? - Si: 

4. Cuando ningún proceso este en su S.C., cualquier proceso que solicite entrar, debe permitírsele sin demora.
5. No se hacen suposiciones sobre las velocidades relativas de los procesos ni sobre el número de procesadores.
6. Un proceso debe permanecer dentro de su sección crítica por un tiempo finito.

EXCLUSIÓN MUTUA

Un proceso se lo puede ver de la siguiente forma:

```
sección_residual  
entrada_sección_crítica  
seccion_crítica  
salida_sección_crítica
```

Dónde:

seccion_residual: es el fragmento del proceso que no necesita acceso exclusivo

entrada_sección_crítica y salida_sección_crítica: se debe garantizar el acceso exclusivo, por eso se los conoce como primitivas de exclusión mutua.

EXCLUSIÓN MUTUA PARA DOS PROCESOS – TÉCNICAS DE SOFTWARE

Veremos distintas tentativas de solución que surgieron para manejar la exclusión mutua, aunque son incorrectas sirvieron para poder llegar a una solución completa y sin errores:

Consideraremos 4 intentos de solución donde se planteará su algoritmo y posibles errores.

EXCLUSIÓN MUTUA PARA DOS PROCESOS – PRIMER INTENTO

Se utiliza una variable global *turno* que es compartida por dos procesos:

```
int turno = 0;
```

```
void proceso0(){
```

```
...
```

```
while(turno != 0);
```

```
/* sección  
   crítica*/
```

```
turno = 1;
```

```
...
```

```
}
```

*Espera
activa*



```
void proceso1(){
```

```
...
```

```
while(turno != 1);
```

```
/* sección  
   crítica*/
```

```
turno = 0;
```

```
...
```

```
}
```

EXCLUSIÓN MUTUA PARA DOS PROCESOS – PRIMER INTENTO

Garantiza la propiedad de exclusión mutua pero tiene dos desventajas:

- ✗ Los procesos deben alternarse en el uso de su sección crítica por lo tanto el ritmo de ejecución viene dictado por el proceso más lento.*
- ✗ Si un proceso falla, el otro quedará permanentemente bloqueado*

*Esta solución es una **corruptiva**. Están diseñadas para pasar el control de ejecución entre ellas mismas.*

EXCLUSIÓN MUTUA PARA DOS PROCESOS – SEGUNDO INTENTO

Cada proceso debe llevar su propia «llave» para entrar en la s.c., de modo que si uno falla, el otro puede continuar accediendo:

enum booleano{falso = 0, verdadero = 1}; booleano estado[2] = {0, 0};

void proceso0(){

...

while(estado[1]);

estado[0] = verdadero;

/ sección*

crítica/*

estado[0] = falso;

...

}

void proceso1(){

...

while(estado[0]);

estado[1] = verdadero;

/ sección*

crítica/*

estado[1] = falso;

...

}

EXCLUSIÓN MUTUA PARA DOS PROCESOS – SEGUNDO INTENTO

Si un proceso falla fuera de la s.c., el otro proceso no queda bloqueado. Sin embargo, si un proceso falla dentro de su s.c. o después de establecer su estado, el otro proceso quedará permanentemente bloqueado.

Esta situación es peor que la anterior ya que no garantiza la exclusión mutua, evaluemos la siguiente traza de ejecución:

Proceso0

```
0  while(estado[1]);
1
2  estado[0] = verdadero;
3
4  /* sección crítica */
5
```

Proceso1

```
while(estado[0]);
estado[1] = verdadero;
/* sección crítica */
```

EXCLUSIÓN MUTUA PARA DOS PROCESOS – TERCER INTENTO

El segundo intento falla debido a que un proceso puede cambiar su estado después de que otro proceso lo haya cambiado.

En este intento se trata de solucionar intercambiando dos sentencias:

```
void proceso0(){
```

```
...
```

```
    estado[0] = verdadero;
```

```
    while(estado[1]);
```

```
    /* sección
```

```
        crítica*/
```

```
    estado[0] = falso;
```

```
...
```

```
}
```

```
void proceso1(){
```

```
...
```

```
    estado[1] = verdadero;
```

```
    while(estado[0]);
```

```
    /* sección
```

```
        crítica*/
```

```
    estado[1] = falso;
```





```
...
```

```
}
```

EXCLUSIÓN MUTUA PARA DOS PROCESOS – TERCER INTENTO

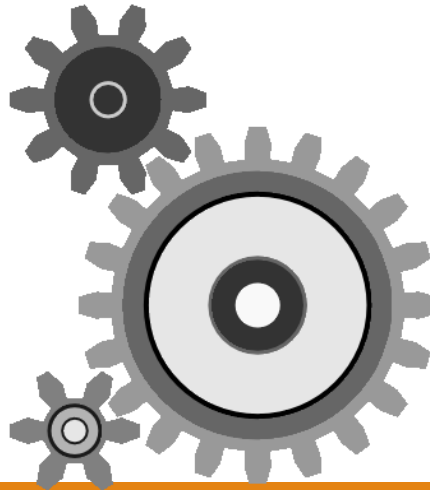
Esta intento de solución garantiza la exclusión mutua. Desde el punto de vista del P0, una vez que establece su estado a verdadero, P1 no puede entrar a su s.c. hasta que P0 lo haya abandonado.

Pero este intento genera un nuevo problema:

	Proceso0	Proceso1
0	estado[0] = verdadero;	
1		estado[1] = verdadero;
2	while(estado[1]); 	
3		while(estado[0]); 
4	INTERBLOQUEO (DEADLOCK)  	

EXCLUSIÓN MUTUA PARA DOS PROCESOS – CUARTO INTENTO

En esta tentativa cada proceso se vuelve mas «respetuoso». Cada uno establece si estado en verdadero pero está preparado a cambiar su estado si otro decide entrar. Veamos su código en la siguiente diapositiva (no entraba en esta).



Gif para ocupar lugar

EXCLUSIÓN MUTUA PARA DOS PROCESOS – CUARTO INTENTO

```
void proceso0(){
```

```
...
```

```
estado[0] = verdadero;
```

```
while(estado[1]){
```

```
    estado[0] = falso;
```

```
    //retraso
```

```
    estado[0] = verdadero;
```

```
}
```

```
/* sección crítica */
```

```
estado[0] = falso;
```

```
...
```

```
}
```

```
void proceso1(){
```

```
...
```

```
estado[1] = verdadero;
```

```
while(estado[0]){
```

```
    estado[1] = falso;
```

```
    //retraso
```

```
    estado[1] = verdadero;
```

```
}
```

```
/* sección crítica */
```

```
estado[1] = falso;
```

```
...
```

```
}
```

EXCLUSIÓN MUTUA PARA DOS PROCESOS – CUARTO INTENTO

Esta cercano a una solución correcta, pero todavía falla. La exclusión mutua está garantizada. Sin embargo siguiendo la siguiente secuencia de eventos:

Proceso0

0 estado[0] = verdadero;

1

2 while(estado[1]) 

3

4 estado[0] = falso;

5

6 estado[0] = verdadero;

7

8 while(estado[1]) 

Proceso1

estado[1] = verdadero;

while(estado[0]) 

estado[1] = falso;

estado[1] = verdadero;

CIRCULO VICIOSO (LIVE LOCK)  

EXCLUSIÓN MUTUA PARA DOS PROCESOS – ALGORITMO DE DEKKER

*Es necesario observar el estado de ambos procesos, que se consigue mediante la variable **estado**, pero no es suficiente. Se debe imponer un orden. Se puede utilizar la variable **turno** del primer intento, e indica que proceso tiene el derecho a insistir en entrar a su s.c.
Veamos su código (en la siguiente diapositiva).*

No pongan esta cara...



EXCLUSIÓN MUTUA PARA DOS PROCESOS – ALGORITMO DE DEKKER

booleano estado[2] = {0, 0}; int turno = 1;

```
void proceso0(){
    while(verdadero){
        estado[0] = verdadero;
        while(estado[1])
            if(turno==1){
                estado[0] = falso;
                while(turno == 1);
                estado[0] = verdadero;
            }
        /* sección crítica */
        turno = 1; estado[0] = falso;
    ... }}
```

```
void proceso1(){
    while(verdadero){
        estado[1] = verdadero;
        while(estado[0])
            if(turno==0){
                estado[1] = falso;
                while(turno == 0);
                estado[1] = verdadero;
            }
        /* sección crítica */
        turno = 0; estado[1] = falso;
    ... }}
```


EXCLUSIÓN MUTUA PARA DOS PROCESOS – ALGORITMO DE PETERSON

El algoritmo de Dekker resuelve el problema de exclusión mutua pero con un programa bastante complejo. Peterson proporcionó en 1981 una solución mas simple: Gracias a Dios!

```
void proceso0(){
    while(verdadero){
        estado[0] = verdadero;
        turno = 1;
        while(estado[1]&&turno==1);
        /* sección crítica */
        estado[0] = falso;
        ...}}
```

```
void proceso1(){
    while(verdadero){
        estado[1] = verdadero;
        turno = 0;
        while(estado[0]&&turno==0);
        /* sección crítica */
        estado[1] = falso;
        ...}}
```

EXCLUSIÓN MUTUA – SOPORTE HARDWARE

La solución de software es fácil que tenga una alta sobrecarga de procesamiento y el significativo el riesgo de errores lógicos. Consideraremos varias soluciones de hardware a la exclusión mutua.



DESHABILITAR INTERRUPCIONES

En una máquina monoprocesador, los procesos concurrentes no pueden solaparse, solo pueden entrelazarse. Un proceso continuará ejecutando hasta que se invoque un servicio del sistema operativo o hasta que sea interrumpido. Por tanto para garantizar la exclusión mutua, hay que impedir que un proceso sea interrumpido. Un proceso puede cumplir la exclusión mutua de este modo:

```
while(true){  
    /* deshabilitar interrupciones */  
    /* sección crítica */  
    /* habilitar interrupciones */  
    /* resto */  
}
```

DESHABILITAR INTERRUPCIONES

Dado que la sección crítica no puede ser interrumpida, se garantiza la exclusión mutua, aunque el precio de esta solución es alto. La eficiencia de ejecución podría degradarse notablemente porque se limita la capacidad del procesador. Otro problema es que esta solución no funcionará sobre una arquitectura multiprocesador.

INSTRUCCIONES MÁQUINA ESPECIALES

Los diseñadores de procesadores han propuesto varias instrucciones máquina que llevan a cabo dos acciones atómicamente, como leer o escribir o leer y comprobar, sobre una única posición de memoria con un único ciclo de búsqueda de instrucción. Durante la ejecución de la instrucción, se bloquea el acceso a toda otra instrucción que referencia esa posición. Veremos la instrucción `test and set` y la instrucción `exchange`.

INSTRUCCIONES MÁQUINAS ESPECIALES – TEST AND SET

La instrucción puede definirse de la siguiente manera:

```
boolean testset(int i){  
    if(i == 0){  
        i = 1;  
        return true;  
    }else{  
        return false;  
    }  
}
```

La instrucción comprueba el valor del argumento i. Si el valor es 0, entonces la instrucción reemplaza el valor por 1 y devuelve cierto. En caso contrario, el valor no se cambia y devuelve falso. Toda la instrucción se garantiza que se ejecute de manera atómica.

INSTRUCCIONES MÁQUINAS ESPECIALES— INSTRUCCIÓN *EXCHANGE*

La instrucción exchange (intercambio) puede definirse de la siguiente manera:

```
void exchange(int registro, int memoria){  
    int temp;  
    temp = memoria;  
    memoria = registro;  
    registro = temp;  
}
```

La instrucción intercambia los contenidos de un registro con los de una posición de memoria.

INSTRUCCIONES MÁQUINAS ESPECIALES– INSTRUCCIÓN *EXCHANGE*

Ejemplo de uso:

```
int cerrojo;
void P(int i){
    int llavei = 1;
    while(true){
        exchange(llavei, cerrojo);
        while(llavei != 0);
        /* seccion critica */
        exchange(llavei , cerrojo);
        ... }}
```

Una variable compartida **cerrojo** se inicializa en 0, cada proceso utiliza una variable local **llavei**, que se inicializa en 1. El único proceso que puede entrar en su s.c. es aquel que encuentra **cerrojo** igual a 0, y al cambiar **cerrojo** en 1 se excluyen a todos los otros procesos de la s.c.. Cuando un proceso abandona la s.c., se restaura **cerrojo** al valor 0.

INSTRUCCIONES MÁQUINAS ESPECIALES – PROPIEDADES

La utilización de una instrucción máquina especial para conseguir la exclusión mutua tiene ciertas ventajas:

- ✓ *Es aplicable a cualquier número de procesos sobre un procesador único o multiprocesador de memoria compartida*
- ✓ *Es simple, y por tanto, fácil de verificar*
- ✓ *Puede ser utilizado para dar soporte a múltiples secciones críticas: cada sección crítica puede ser definida por su propia variable.*

INSTRUCCIONES MÁQUINAS ESPECIALES – PROPIEDADES

Hay algunas desventajas serias:

- x Se emplea espera activa. Así, mientras un proceso está esperando para acceder a una s.c., continúa consumiendo tiempo de procesador.*
- x Es posible la inanición. Cuando un proceso abandona su s.c. y hay más de un proceso esperando, la selección del proceso en espera es arbitraria. Así a algún proceso podría denegársele indefinidamente acceso.*
- x Es posible el interbloqueo. Supongamos que el P1 ejecuta una instrucción especial y entra a su s.c.. Entonces el P1 es interrumpido para darle el procesador a P2, que tiene más prioridad. Si P2 intenta usar el mismo recurso que P1, se le denegara el acceso y caerá en una espera activa. P1 nunca será escogido por ser de menor prioridad que P2.*