# Tema 3. Primeras aproximaciones a la Programación Concurrente

## 1. Tipos de sincronización

- Entre procesos concurrente aparecen problemas de sincronización, p.e.:
  - Competencia por recursos no compartibles (e.g. impresora)
  - Cooperación compartiendo información (la información se almacena en recursos no compartibles y puede dar lugar a fallos en la coherencia de datos)
  - Cooperación mediante paso de mensajes (orden para el envío y recepción de mensajes)

- Necesitamos disponer de reglas de *sincronización* entre procesos
- Estas reglas de sincronización entre procesos concurrentes son de dos tipos:
  - *Exclusión mutua*
  - *Serialización* o *Condición de sincronización*.

- Además existen distintos tipos de mecanismos con los que se pueden implementar ambos tipos de sincronización.

### 1.1. Exclusión mutua

- Sincronización necesaria cuando los procesos desean utilizar un recurso no compartible
- Cuando un proceso está accediendo a este tipo de recursos se dice que está en su sección crítica
- Ejemplos:
  - acceso en modo escritura a una variable global compartida
  - impresora

- Garantizar la exclusión mutua en la ejecución de secciones críticas consiste en **diseñar un protocolo de entrada y uno de salida**

mediante el cual se sincronice la entrada de los procesos a su sección crítica.

- Los protocolos de E/S en la *sección crítica* deben cumplir:
  - En la exclusión mutua pura sólo un proceso puede estar en un determinado momento en su sección crítica
  - Cuando un proceso sale de la sección crítica debe permitir pasar a otro proceso que estaba esperando para acceder a la misma
  - Por tanto cuando un proceso quiera entrar en su *sección crítica* y ésta esté libre, podrá hacerlo

- Un esquema de procesos ejecutando su sección crítica sería:

```
1: while forever:
2:         # resto del codigo
3:         cs_entry()          # preprotocolo
4:         critical_section()
5:         cs_exit()           # posprotocolo
6:         # resto del codigo
```

- Los protocolos de entrada y salida deben ser rápidos y garantizar:
  1. **Exclusión mutua:** no pueden acceder dos procesos a la vez a la *sección crítica*
  2. **Progreso en la ejecución:** si varios procesos desean entrar a la *sección crítica* al menos uno de ellos debe poder hacerlo
  3. **Espera limitada:** cualquier proceso debe poder entrar a la *sección crítica* en un tiempo finito

- Además son deseables los cuatro requisitos de Dijkstra:
  1. La solución debe ser simétrica, la misma para todos los procesos.
  2. No se deben hacer suposiciones sobre la velocidad relativa de los procesos ni suponer que éstas son constantes.
  3. Entrada inmediata o no interferencia, si un proceso se interrumpe en el resto del código no debe bloquear a otros procesos.
  4. La decisión sobre qué proceso debe entrar a la *sección crítica* debe tomarse en un número finito de pasos

- Existen variantes a la exclusión mutua 'pura':
  **Exclusión múltiple:**

como mucho **n** procesos pueden estar en su sección crítica a la vez

**Exclusión mutua por categorías:**
en un momento determinado sólo un tipo de proceso puede estar en la *sección crítica* p.e. problema lectores/escritores: la presencia de un escritor en la *sección crítica* excluye a otros escritores y a todos los lectores (la de un lector no excluye a otros lectores)

### 1.2. Serialización de Procesos

- Alguna bibliografía lo llama: *Condición de sincronización*.
- Se produce cuando un objeto de datos compartido está en un estado inapropiado para ejecutar una operación determinada.
- Cualquier proceso que intente ejecutar esta operación debe ser retrasado hasta que el estado del dato cambie porque otro proceso haya emprendido una acción determinada
- **Ejemplo:** procesos *lector*, *gestor* e *impresor*.
  - El lector no podrá depositar una imagen en el buffer de entrada compartido si este está lleno.
  - El gestor no podrá tomar una imagen si el buffer está vacío, etc.

## 2. Soluciones para los dos tipos de sincronización

- Basada en el **hardware**:
  - Inhibición de las interrupciones

- Basadas en **memoria compartida**:
  - Espera ocupada
  - Semáforos
  - Regiones críticas
  - Regiones críticas condicionales
  - Monitores

- Basadas en **memoria distribuida**:
  - Paso de mensajes y tuberías
  - Llamadas a procedimientos remotos
  - Invocaciones remotas

## 2.1. Inhibición de interrupciones

- Tenemos que garantizar la indivisibilidad de una secuencia de acciones
- Como la CPU sólo pasa de un proceso a otro como resultado de una interrupción (de reloj u otras) podemos deshabilitarlas antes de entrar en la sección crítica y habilitarlas al salir de la misma
- **Problemas:**
  - ¿qué peligro tiene dejar que un proceso de usuario tenga poder de deshabilitar las interrupciones?
  - ¿qué pasa si hay dos o más CPU's?

## 2.2. Espera ocupada para exclusión mutua

- El proceso espera mediante la comprobación continua del valor de una variable compartida, manteniendo ocupada la CPU.
- Podemos tener dos tipos de soluciones:
  - **Basadas en hardware:** permiten operaciones atómicas más complejas, como intercambiar contenido de dos posiciones de memoria, leer, escribir, etc…
  - **Basadas en software:** las únicas operaciones atómicas son `load` y `store` para leer y escribir en direcciones de memoria. Si dos de estas instrucciones coinciden simultáneamente el resultado sería equivalente a una ejecución secuencial pero en orden desconocido.

- El esquema general de esta solución sería el siguiente:
  - Inicialización de variables globales

```
turn = 1
estados = [0, 0]
```

  - Programa que ejecuta cada proceso

```
while True:
        # resto del código
        #
        entry_critical_section()
        critical_section()
        exit_critical_section()
        #
        # resto del código
```

- Soluciones para **2** procesos:
  - Primeros intentos
  - Algoritmo de Dekker
  - Algoritmo de Peterson

- Soluciones para **n** procesos:
  - Algoritmo de Eisenberg-McGuire
  - Algoritmo de Lamport o *algoritmo de la panadería*
  - Algoritmo *rápido* de Lamport

- **¡Ojo! Problema** con los **sistemas multiprocesador** modernos

## 2.3. Primer intento

```
turn = 0

# Busy waiting ... SpinLocks
while turn != 0:
        pass

critical_section()

turn = 1
```

- Garantiza la exclusión mutua, pero…
- Provoca que el derecho a usar la *sección crítica* sea alternativo para los dos procesos: $P_0$, $P_1$, $P_0$, etc…→ no se satisface la condición de progreso en la ejecución pues para que entre $P_0$ tiene que entrar antes $P_1$ y si $P_1$ no lo hace…
- Procesos demasiado *acoplados*, si uno de los procesos falla el otro se quedará detenido

## 2.4. Segundo intento

```
1: states = [False, False] # is_in[False, False]
2:
3: while states[1]:
4:         pass
5: states[0] = True
6:
7: critical_section()
8:
9: states[0] = False
```

- Problema fundamental: no asegura la exclusión mútua

```
P0                          P1
¿states[1]? → False
                            ¿states[0]? → False
                            states[1] = True
                            ...
states[0] = True
...
                ## BOOM ##
```

## 2.5. Tercer intento

```
1: states = [False, False]  # resto del codigo
2:
3: states[0] = True
4: while states[1]:
5:         pass
6:
7: critical_section()
8:
9: states[0] = False
```

- Problema: posible interbloqueo

```
P0                          P1
states[0] = True
                            states[1] = True
                            ¿states[0]? → True
                            ...
¿states[1]? → True
```

```
...
                ## DEADLOCK! ##
```

## 2.6. Cuarto intento

```
 1: states = [False, False]  # resto del codigo
 2:
 3: states[0] = True
 4: while states[1]:
 5:         states[0] = False
 6:         states[0] = True
 7:
 8: critical_section()
 9:
10: states[0] = False
```

- Garantiza la exclusión mutua
- No se produce espera ilimitada
- Pero no asegura que se pueda entrar en la *sección crítica* en un tiempo finito porque los procesos pueden estar continuamente cediéndose el paso para entrar.

## 2.7. Algoritmo de Dekker (1963)

```
 1: cs_ready = [False, False]
 2: turn = 0                # 0 ó 1 , es indiferente
 3:
 4: cs_ready[0] = True
 5: while cs_ready[1]:
 6:         if turn == 1:
 7:                 cs_ready[0] = False
 8:                 while turn != 0:  # (1)
 9:                         pass
10:                 cs_ready[0] = True
11:
12: critical_section()
13:
14: cs_ready[0] = False
15: turn = 1              # (2)
```

- En **(1)** $P_0$ espera si no es su turno, su estado se mantendrá en falso y $P_1$ podrá entrar a la *sección crítica*

- En **(2)** cuando un proceso sale de su *sección crítica* cede el turno al otro

- Es correcto ya que:
  - Asegura la exclusión mutua
  - Cumple la condición de progreso en la ejecución
  - Satisface la limitación de espera

- Está orientado a entornos centralizados por la variable `turn` y a dos procesos
- Problema: es complejo y difícil de seguir para más de dos procesos

## 2.8. Algoritmo de Peterson (1981)

- Simplifica el de Dekker, más sencillo de entender.
- Mismas variables, cambia el orden de las instrucciones.
- Ahorra unos ciclos de proceso.

```
 1: cs_ready = [False, False]
 2:
 3: cs_ready[i] = True
 4: turn = j                          # (1)
 5: while cs_ready[j] and turn == j:  # (2)
 6:         pass
 7:
 8: critical_section()
 9:
10: cs_ready[i] = False
11:
12: # Resto del codigo
```

- En **(1)** cede el turno al otro proceso
- En **(2)** espera si el estado del otro es `True` y es su turno

- Verifica la exclusión mutua
- Verifica la condición de progreso en la ejecución
- La espera es limitada
- Solución más sencilla y elegante que Dekker

## 2.9. Algoritmo *incorrecto* de Hyman

```
states = [False, False]
turn = 0    # 0 ó 1 , es indiferente

states[0] = True
while turn != 0:
        while states[1]:
                pass
        turn = 0

critical_section()

states[0] = False
```

- Parece correcto, pero fíjate:

```
P0                      P1
                        states[1] = True
                        ¿turn != 1? → True
                        ¿states[0]? → False
states[0] = True
¿turn == 0? → True
                        turn = 1
                        ...
              ## BOOM! ##
```

## 2.10. Algoritmo de Lamport

- Este algoritmo es adecuado para ejecutarse en entornos distribuidos
- Verifica la exclusión mutua, el progreso de ejecución y no hay espera ilimitada
- También se conoce como **Algoritmo de la Panadería**

Estructuras de datos utilizadas:

`choosing`:
: Array de valores booleanos.

`choosing[i] = True`:
: Indica que el proceso $P_i$ esta cogiendo numero.

`choosing[i] = False`:
: Indica que el proceso $P_i$ ha terminado de cogerlo.

Inicialmente todos los `choosing[i] = False`.

**`number` :**

Array de enteros. Cada elemento indica el número que ha cogido el proceso. Inicialmente todas las componentes valen `0`.

**El par `(choosing[i], number[i])` :**

Pertenece al proceso $P_i$

```
1: choosing = [False, ..., False]       #(1)
2: number = [0, ..., 0]
3:
4: choosing[i] = True                   #(2)
5: number[i] = 1 + max(number)
6: choosing[i] = False                  #(3)
7: for j in range(0,N):
8:                   while choosing[j]:             #(4)
9:                                       pass
10:                  while number[j] > 0 and
11:                                          (number[j] < number[i] or
12:                                          (number[j] == number[i] and j
13:                                          pass
14:
15: critical_section()
16:
17: number[i] = 0
```

- En **(1)** el array tiene la misma dimensión que `number`.
- En **(2)** se indica que está por entrar a la *sección de selección de número*.
- En **(3)** se indica que ya acabó la selección.
- En **(4)** si el proceso **j** está cogiendo número se le espera porque podría corresponderle el turno.

Nos garantiza que:

**Exclusión mutua :**

Ya que cuando un proceso $P_i$ está en su sección crítica, éste ha pasado por el segundo `while` al poseer el número menor o el menor identificador entre los de menor número, y si otro proceso intenta pasar a su sección crítica tendría que cumplir lo mismo, lo que no es

posible debido a que el número que extraiga será siempre mayor que el que posee el proceso que está dentro de la sección crítica.

**Progreso en la ejecución :**
Según el orden en que se toma el número.

**Limitación en la espera :**
Por la entrada a la sección crítica según el orden.

**Tiene un inconveniente :**
Los números que cogen los procesos pueden aumentar a lo largo de la ejecución superando la capacidad de cualquier tipo de datos.

# 3. Conclusiones

- La solución basada en *espera activa* satisface la exclusión mutua, junto con las condiciones de *no inanición* y *progreso de la ejecución*.
- Sin embargo, la *espera activa* es ineficiente:
  - Los procesos ocupan el procesador incluso aunque no puedan avanzar

- El uso de *Variables Compartidas* genera esquemas:
  - Demasiado complejos
  - Muy artificiales
  - Propensos a errores: No hay una separación entre las variables de los cómputos de los procesos y las utilizadas para sincronización.
  - Inadecuados para grandes esquemas de protección.
  - Fuertemente acoplados: los procesos deben de *saber* unos de otros.
  - No funcionan en los sistemas multiprocesador modernos (los modelos eficientes de consistencia de memoria no garantizan el reordenamiento adecuado de los accesos a memoria)
  - Para tratar estos problemas se puede recurrir a las *barreras de memoria* o al uso de primitivas hardware (instrucciones específicas para tratar la concurrencia en ensamblador)

# 4. Instrucciones atómicas complejas en ensamblador

**Exchange :**

Intercambia el contenido de dos posiciones de memoria de forma atómica

- Espera ocupada: (ver código en la sección de ejemplos en Unix)

**TestAndSet :**

Guarda `1` o `true` en el parámetro que tiene y devuelve el valor que tenía este parámetro originalmente:

```
bool function test_and_set(bool* p) {
    bool old_p = *p;
    *p = true;
    return *old_p;
}
```

- Espera ocupada:

```
// lock == true significa que NO podemos entrar en la s.c.
//         el candado esta cerrado.
while (test_and_set(&lock));
critical_section();
lock = false;
```

**CompareAndSwap :**

Compara el contenido de una posición de memoria con un valor dado y, sólo si es el mismo, modifica el valor de la memoria por un nuevo valor dado

```
bool function compare_and_swap(int* value,
                               int expected, int new_v) {
     int tmp = *value;

     if (*value == expected)
         *value = new_v;

     return tmp;
}
```

- Espera ocupada:

```
// lock == 0 significa que SI podemos entrar en la s.c.
//         el candado esta cerrado.
while (compare_and_swap(&lock, 0, 1) != 0);
critical_section();
lock = 0;
```

# 5. Anexo: memoria compartida en Unix/System V

- Con pseudocódigo utilizar variables compartidas es tan sencillo como declararlas como variables globales.
- En general, con procesos, en la vida real, las cosas no son tan sencillas...

# 6. Ejemplos en Unix

- Veamos el código de los programas:
  - `indeterminismo.c` (System V)
  - `Dekker_con_procesos.c` (System V)
  - `counter_peterson.c` (barreras gcc)
  - `exchange.c` (ensamblador e hilos posix)

- Por sencillez, utilizaremos principalmente *hilos Posix* en **C** o *Threads de Java o Python* para las prácticas.
- En GNU/Linux se puede simular un sistema monoprocesador con la instrucción taskset (`man taskset`)

# 7. Cuestiones breves

- Indicar las condiciones exigidas por los protocolos de E/S a una correcta solución al problema de la exclusión mutua
- Formular varios problemas donde se requieran condiciones de sincronización
- ¿A qué hace referencia el término *espera ocupada*?
- ¿Cuál es el principal inconveniente de las soluciones basadas en espera ocupada?

- Explicar por qué el algoritmo de Lamport no presenta espera ilimitada

# 8. Problema

- Discutir la corrección del algoritmo de exclusión mutua siguiente (si es incorrecto encuentra una traza que viole la exclusión mutua, si es correcto pruébalo):

```
1: states = [1, 1]
2:
3: states[0] = 1 - states[1]
4: while states[1] != 0:
5:         states[0] = 1 - states[1]
6:
7: critical_section()
8:
9: states[0] = 1
```

- Lo mismo con el algoritmo:

```
 1: states = [1, 1]
 2:
 3: states[0] = 0
 4: while states[1] == 0:
 5:         states[0] = 1
 6:         while states[1] == 0:
 7:                 pass
 8:         states[0] = 0
 9:
10: critical_section()
11:
12: states[0] = 1
```

- Adapta al código del programa `counter.py` del *tema 2* de modo que satisfaga la exclusión mutua en el acceso a la variable compartida `counter` mediante el algoritmo de Peterson

```
1: import threading
2:
3: THREADS = 2
4: MAX_COUNT = 10000000
5:
6: counter = 0
7:
8: def thread():
9:         global counter
10:
11:         print("Thread {}".format(threading.current_thread().name))
12:
13:         for i in range(MAX_COUNT//THREADS):
14:                 counter += 1
15:
16: def main():
17:         threads = []
18:
19:         print("Starting...")
20:
21:         for i in range(THREADS):
22:                 # Create new threads
23:                 t = threading.Thread(target=thread)
24:                 threads.append(t)
25:                 t.start() # start the thread
26:
27:         # Wait for all threads to complete
28:         for t in threads:
29:                 t.join()
30:
31:         print("Counter value: {} Expected: {}\n".format(counter, MA
32:
33: if __name__ == "__main__":
34:         main()
```

## 8.1. Aclaraciones

- **En ningún caso estas transparencias son la bibliografía de la asignatura**, por lo tanto debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.