Programación Sobre Redes

T10: Sincronización entre procesos II

Nicolás Mastropasqua

September 17, 2020

Instituto Industrial Luis A. Huergo

Contenidos

1. Repaso

2. El problema de la sección crítica

Repaso

’Bondades’ de la concurrencia

Vimos que tener procesos (o threads) que colaboren entre ellos puede resultar útil en varios escenarios. ¿Qué dificultades trae esto?

- Grado de paralelización
- Race conditions
- ...

El problema de la sección crítica

Intuición

Sabemos que el baño es una sección crítica, y nos gustaría **garantizar** que nunca hay mas de dos personas usándolo

Sección crítica: Idea general

- Consideremos un sistema con n procesos $\{p_0, ..., p_{n-1}\}$.
- Cada proceso tiene un segmento de código, llamado sección crítica, en el cual puede modificar variables compartidas.
- Debemos encontrar un protocolo de comunicación que garantice que:

propiedades sección crítica

- Si un proceso está ejecutando su sección crítica, ningún otro puede ejecutar la suya.
- En otras palabras, no hay dos procesos en la sección crítica.

Si tenemos esta garantía, ¿le decimos chau a las race conditions?

Requerimientos sección crítica

El problema consiste en encontrar una solución que cumpla:

Propiedades

- **Exclusión mutua:** Si P_i está ejecutando en su sección crítica, ningún otro proceso puede estar ejecutando la suya.
- **Progreso:** Si no hay procesos en la sección crítica y algunos desean entrar, entonces solo aquellos que están fuera de la ”remainder section” van a participar en la elección. Tal elección no se pospondrá indefinidamente.
- **Espera acotada:** Existe un límite en la cantidad de veces que otros procesos pueden entrar a su sección crítica despues de que un proceso haya requerido entrar a la suya y antes de que esta petición sea otorgada

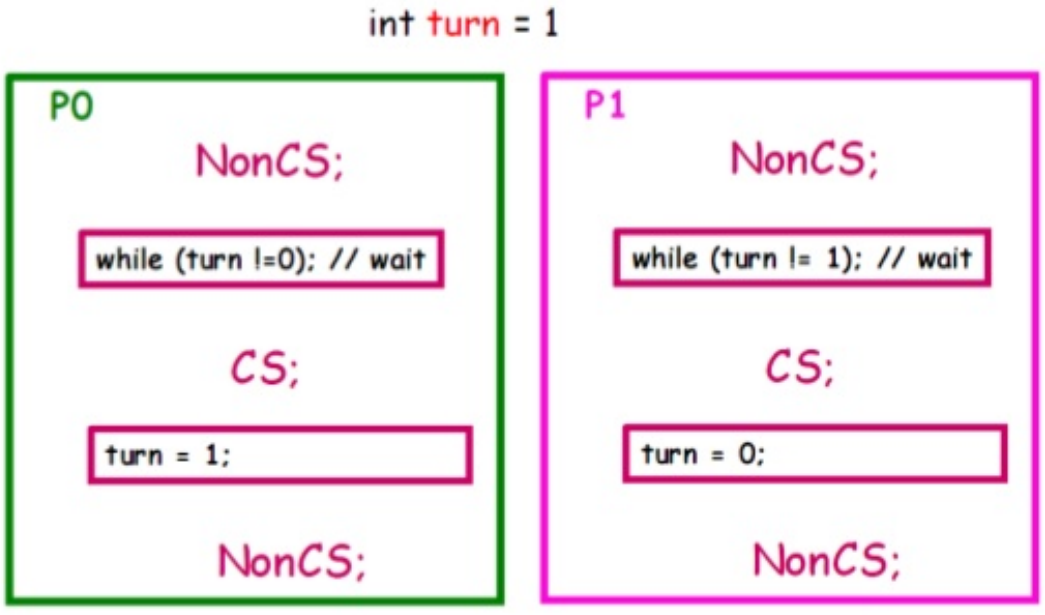
De alguna manera, las dos últimas, indican que no exista starvation.

Explorando soluciones

Determinar si el siguiente algoritmo resuelve el problema de la sección crítica. Asumir solo dos procesos, **i, j**.
La variable flag intenta codificar la intención de entrar a la sección crítica.

```
do {
    flag[i] = TRUE;
    while (flag[j] );
        critical section
    flag[i] = FALSE;
        remainder section
} while (TRUE);
```

Algoritmo de alternación estricta



Garantiza exclusión mutua? Sí!

Garantiza progreso? No!

Hace busy waiting!

Nos quedamos sin ideas...

Desactivamos interrupciones

- ¿Resolvemos el problema? Discutible...
- ¿Ventajas y desventajas?

Algoritmo de Peterson

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

- Funciona para **dos procesos** (se puede extender)
- Hace **busy waiting**
- En arquitecturas modernas, **podría no garantizar exclusión mutua**

Test And Set

- El Hardware suele proveer una instrucción que permite establecer atomicamente el valor de una variable entera en 1.
- Una operación es atómica a **nivel hardware** si es **indivisible**, el sitema no puede distinguir estados intermedios.

```
// Pseudocódigo.
// TAS suele ser una instrucción de Assembler
bool  TestAndSet(bool *destino)
{
    bool  resultado = *destino;
    *destino = TRUE;
    return  resultado;
}
```

TAS: Ejemplo para sección crítica

```
boolean lock;

void  main()
{
    while (TRUE)
    {
        while (TestAndSet (&lock));

        critical_section();

        lock = FALSE;

        remainder_section();
    }
}
```

A tener en cuenta

- Algunos lenguajes utilizan esta **primitiva de sincronización**, para crear objetos atómicos. Por ejemplo, en el TP1 utilizamos atomic<int> .
- Utilizando TAS como en el ejemplo, estamos haciendo busy waiting. Una política muy agresiva con el CPU.
- Una alternativa para esto, son los semáforos.