Programación Sobre Redes

T11: Sincronización entre procesos III

Nicolás Mastropasqua

September 28, 2020

Instituto Industrial Luis A. Huergo

Contenidos

1. Repaso
2. Semáforos
3. Ejercicios

Repaso

Single-thread

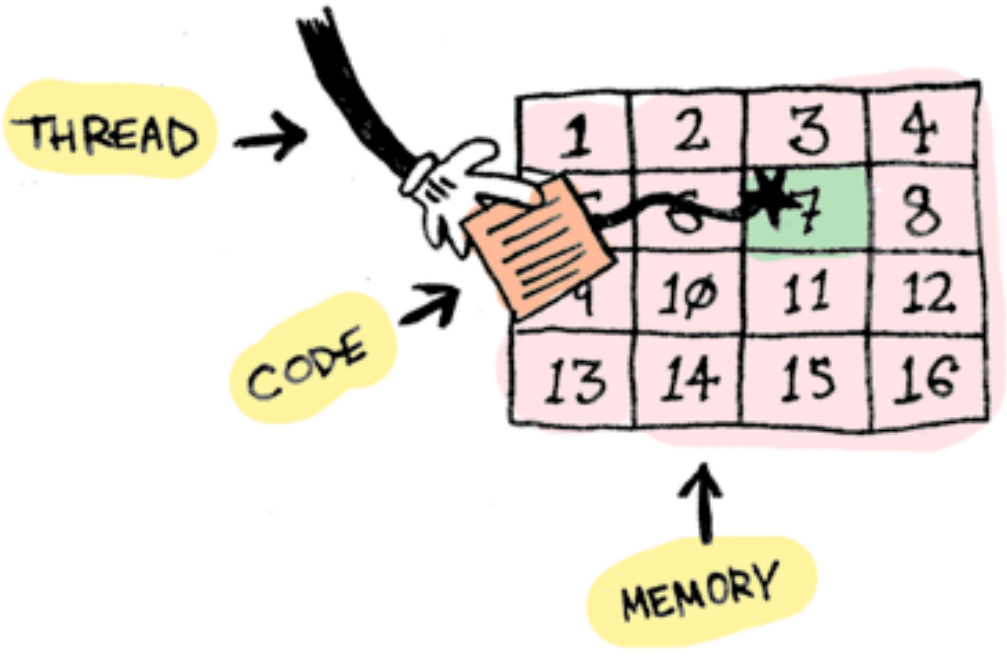


Figure 1: Source

Multi-thread, empezaban los problemas

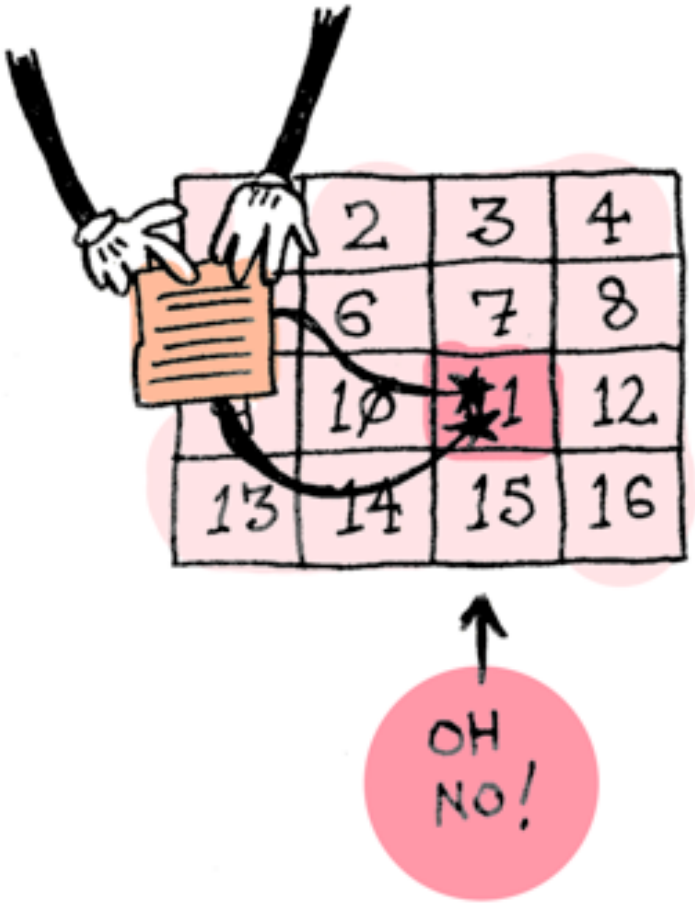


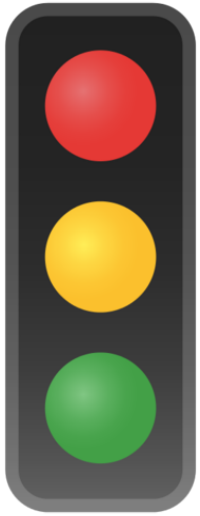
Figure 2: Source

¿Cómo resolvíamos problema de la sección crítica?

- Deshabilitar interrupciones. ¿Problemas?
- Algoritmo de Peterson . ¿Problemas?
- TAS. Objetos atómicos. ¿Problemas?

Semáforos

¿Qué es un semáforo?



- Un semáforo es un sistema para comunicar algo visualmente, con banderas, luces u otro mecanismo.
- Para nosotros, va a ser una estructura de datos que nos va a ayudar a resolver problemas de sincronización.

Gracias Dijkstra

Figure 3: Source

Intuición

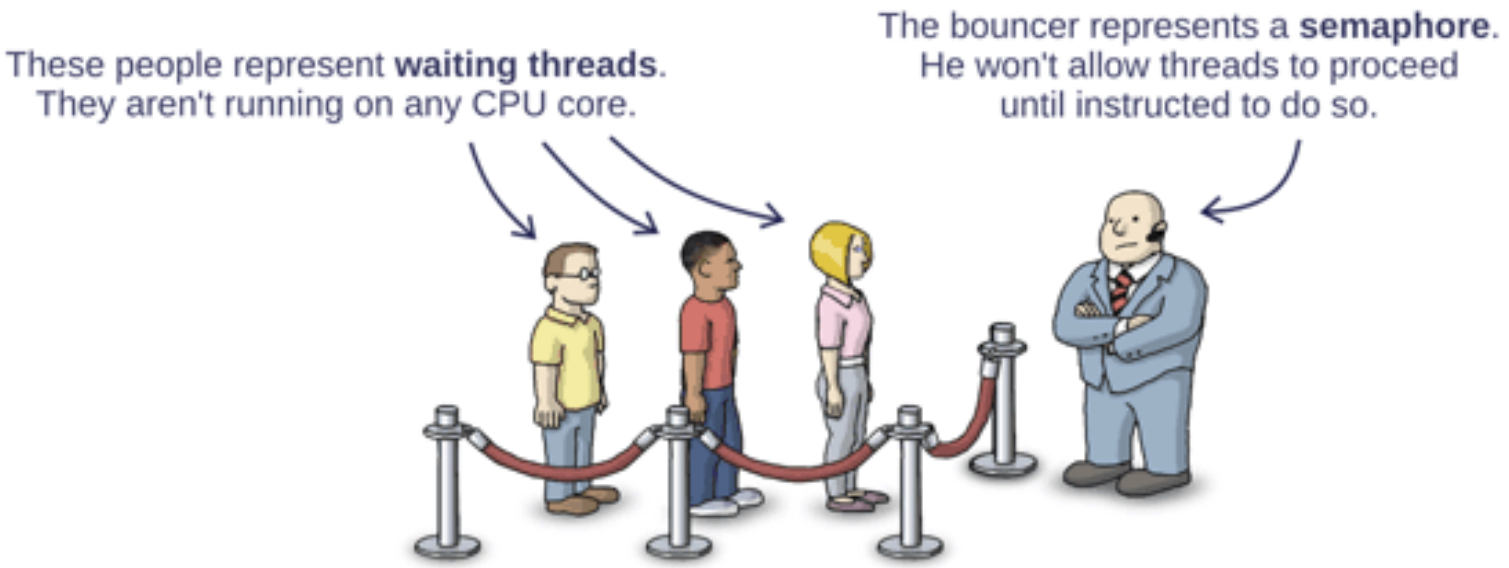
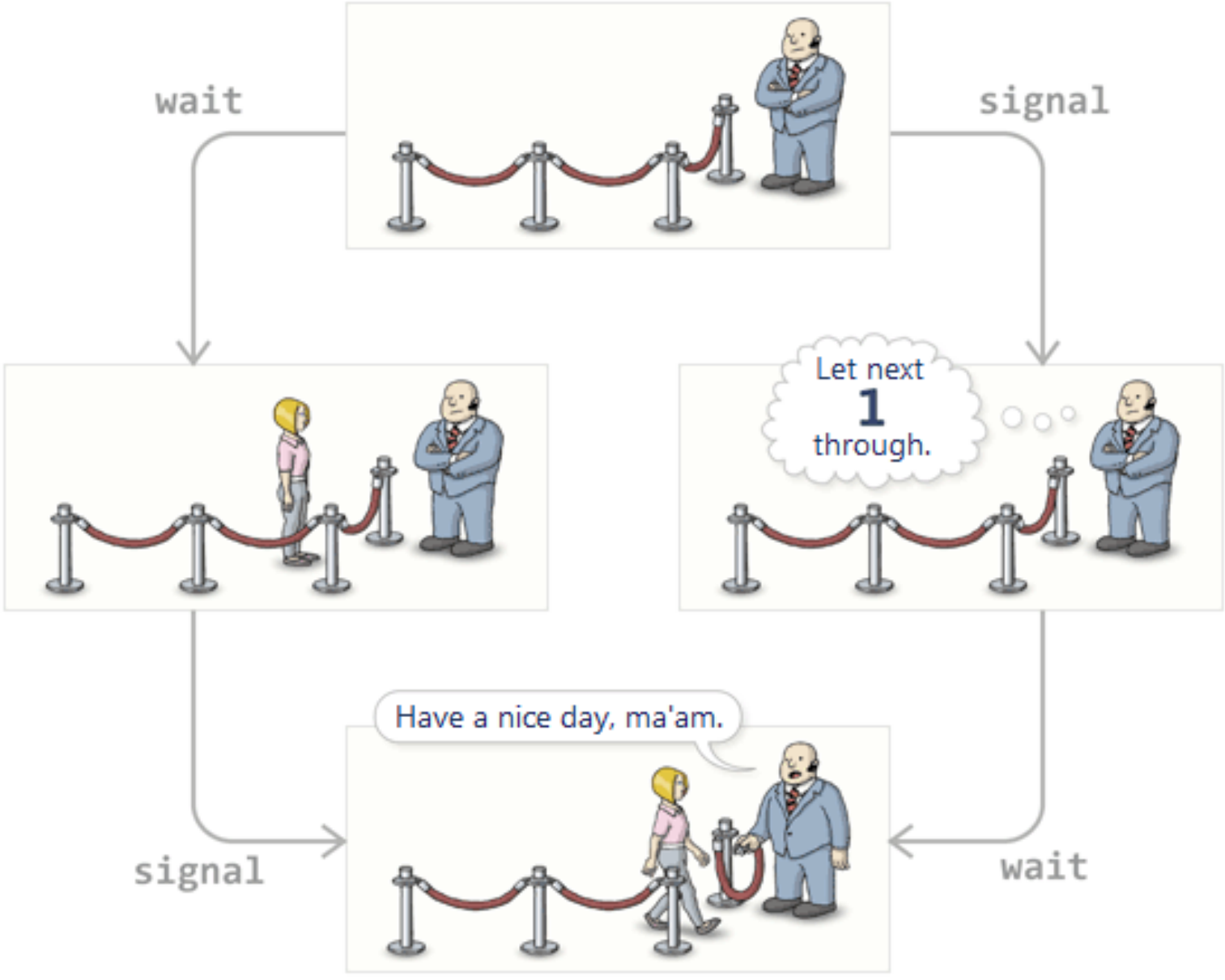


Figure 4: Source

Intuición


wait
signal
Let next
1
through.
signal
Have a nice day, ma'am.
wait

¿Qué hago con un semáforo?

- Lo puedo crear, inicializándolo con un entero.
- Puedo hacer wait(). Esto va a decrementar el valor del entero en uno, **solo si al hacer esto no queda negativo**.
- Puedo hacer signal() e incrementar el valor en uno.

...

- ¿Dónde está lo interesante de esto?
- ¿Por qué algo que se comporta como un entero me va a ayudar con la sincronización?
- ¿Al menos se pueden evitar data races?

Implementación de semáforos

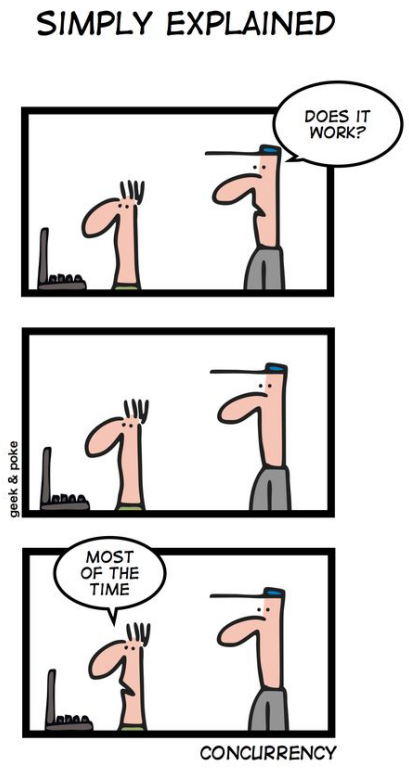
Las funciones wait y signal tienen **efectos secundarios** interesantes

```
wait(){
    while(S <=0 )
        block();
    S++;
}
signal(){
    S++;
    if (hay bloqueados?)
        desbloquear_alguno();
}
```

Notar que en esta caso S nunca puede ser negativo. Sin embargo, hay implementaciones donde esto podría ocurrir.

En algunos casos wait puede llamarse **down, decrement, o P**. Lo mismo para signal (**up, increment o V**)

Ay, la concurrencia...



Para que todo funcione bien, hay que pedir que las funciones anteriores se ejecuten de manera atómica

Algunas observaciones inmediatas

- No es posible chequear el valor del entero! Eso es parte de la representación interna del objeto
- Por lo anterior, no es posible saber si un proceso va a bloquearse antes de decrementar el semáforo
- Luego de que un proceso incremente un semáforo y se despierte otro, ambos ejecutan concurrente. No es posible saber cual de los dos continuará de forma inmediata.
- Podemos interpretar el valor de un semáforo como la capacidad, es decir la cantidad de procesos que pueden decrementarlo sin bloquearse

¿Por qué semáforos?

Analizando semáforos

- Con los semáforos anteriores evitamos **busy waiting**. ¿Por qué?
- No siempre es conveniente utilizalos. Tener en cuenta el overhead que provoca hacer un **context switch**.
- En algunos casos, hacer **busy waiting** puede convenir. Es por eso que existen los **SpinLocks** (o TASlocks, basados en TestAndSet)
- Las soluciones que utilizan semáforos pueden resultar bastante declarativas y útiles para demostrar propiedades
- Están implementados en muchos sistemas de manera eficiente. Soluciones portables.

Ejercicios

Sintáxis

En general, durante esta etapa, vamos a pensar los problemas utilizando pseudocódigo:

Funciones de un semáforo

- *Sema = Semaphore(n)* :Constructor de semaforo. En este caso, inicializado en *n*.
- Sema.signal(), Sema.Wait()

Ejercicios

Resolver los ejercicios de la sección introductoria de la práctica 4. Entregarlos en un doc por Classroom

Resolver las secciones **Tutorial, Unsynchronized code, locks y sempahores** de The deadlock empire

Deadlock

- Implementar semáforos con una cola de espera puede llevar a una situación donde dos o más procesos están esperando indefinidamente por un evento que solo puede ser causado por alguno de los procesos que está esperando.
- Cuando se alcanza este estado, se dice que hay deadlock

Ejemplo deadlock

Supongamos que tenemos dos semáforos S, Q inicializados en uno y dos procesos P0 y P1.

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Deadlock!

En este caso, consideramos el **scheduling** en el que P0 hace wait(S) y luego P1 hac wait(Q). Luego, cuando P0 quiera hacer wait(Q) se bloqueará, esperando un signal(Q) de P1. Sin embargo, P1 hará wait(S) y se bloqueará, esperando un signal(S) de P0. Por lo tanto, el sistema entra en **deadlock**.