Programación Sobre Redes

T4: Sincronización entre procesos 1

Nicolás Mastropasqua

August 3, 2020

Instituto Industrial Luis A. Huergo

Contenidos

1. Repaso: Concurrencia
2. Live Demo: ”The mind”
3. Sincronización de procesos

Repaso: Concurrencia

Concurrencia, ¿para qué?

- Eficiencia
- Modularidad
- Atender agentes ”en simultaneo”

La vieja y querida...concurrencia

- Contención y concurrencia son dos problemas fundamentales en un mundo donde cada vez se tiende más a la programación distribuida y/o paralela. Clusters, multicores, etc.
- Pero además, es importantísimo para los SO: Muchas estructuras compartidas, mucha contención

Bugs ”concurrentes”

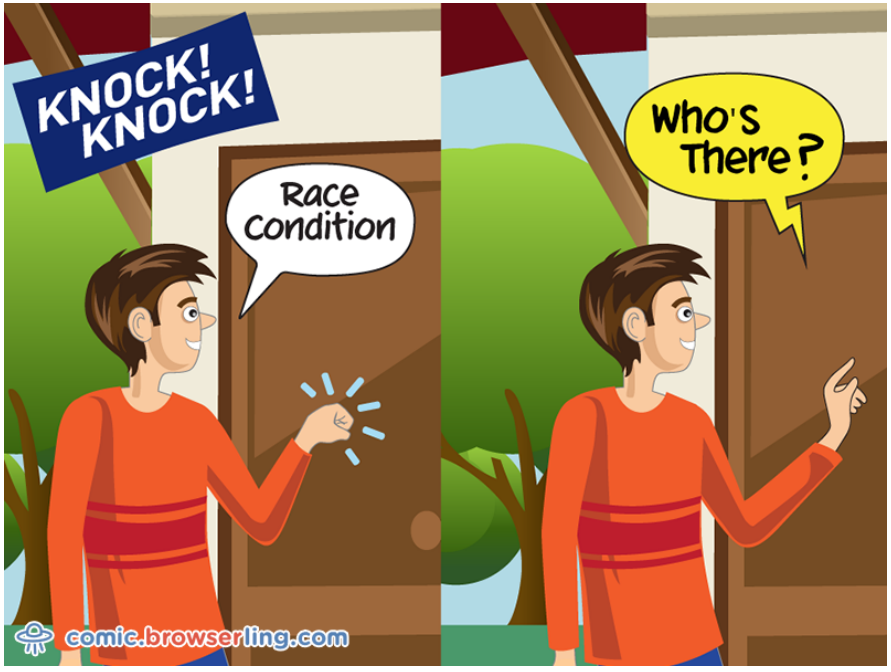
- Escribir programas concurrentes **correctos** es difícil.
- Como programadores, es fácil caer en la tentación de pensar ”secuencialmente”. Hay que lidiar con el no-determinismo
- Más aún, es complicado detectar los bugs, ya que son difíciles de reproducir.

No determinismo

- Recordemos que las ejecuciones están sujetas al no-determinismo. ¿Qué significa?
- Por ejemplo:
  - Dos threads. Uno imprime ”Hola” y el otro ”Chau”
  - ¿Puedo asumir algo sobre el orden de ejecución? No! Depende del scheduler
- En general, podría pasar que para el mismo input, mi programa tenga comportamientos distintos dependiendo de la ejecución que ”tocó”

Live Demo: ”The mind”

Race condition



- Para el mismo input, el programa da resultados distintos dependiendo del orden en el que se ejecutan sus diferentes partes.
- Una ”carrera” donde cada parte del programa (en ejecución) compite. El resultado depende de quien gana.
- ¿Se puede poner peor? :D

Caso de estudio

Situación

Se tiene un fondo de donaciones. Cada usuario que lo haga, obtiene un ticket para un sorteo. Consideremos el caso en el que dos usuarios quieren donar.

Implementación

- Programa en C/Java

```
int ticket= 0;
int fondo= 0;

int donar(int donacion) {
  fondo+= donacion; // Actualiza el fondo
  ticket++;         // Incrementa el número de ticket
  return ticket;    // Devuelve el número de ticket
}
```

- En assembler

```
load fondo
add donacion
store fondo

load ticket
add 1
store ticket

return reg
```

Un escenario posible

Dos procesos P_1 y P_2 ejecutan el mismo programa

P_1 y P_2 comparten variables `fondo` y `ticket`

P_1	P_2	r_1	r_2	`fondo`	`ticket`	ret_1	ret_2
donar(10)	donar(20)			100	5		
load fondo		100		100	5		
add 10		110		100	5		
	load fondo	110	100	100	5		
	add 20	110	120	100	5		
store fondo		110	120	110	5		
	store fondo	110	120	!! 120	0		
	load ticket	110	5	120	5		
	add 1	110	6	120	5		
load ticket		5	6	120	5		
add 1		6	6	120	5		
store ticket		6	6	120	6		
	store ticket	6	6	20	!! 6		
return reg		6	6	120	6	6	
	return reg	6	6	120	6		6

Analisis

- Si las ejecuciones hubiesen sido secuenciales, los resultados posibles eran que el fondo terminara con 130 y cada usuario recibiera los tickets 6 y 7 en algún orden.
- Sin embargo, terminamos con un resultado inválido. Toda ejecución debería dar un resultado equivalente a alguna ejecución secuencial de los mismos procesos.
- Lo que ocurrió se llama condición decarrera o race condition

Data Race

- Ocurre cuando más de un proceso accede y manipula un mismo recurso concurrentemente de modo que el resultado final depende del orden en el cual los accesos se llevaron a cabo.
- Ojo! Podemos tener race conditions o data races.

¿Tan terrible es?

Figure 1: Según esta noticia, esto le costo a **The Nasdaq Stock Market** 13 millones de dolares. Otros casos interesantes también con MySQL, Firefox, etc.

Es claro, tenemos que buscar alguna solución...

Sincronización de procesos

Motivación

¿Cómo logramos sincronizar los threads para tener algún orden particular de ejecución que nos sirva y evite problemas como los ya mencionados?

Aclaración:

- Para estudiar sincronización, nos da igual si tenemos un modelo **paralelo** o **multithread** (concurrente)
- El problema es el mismo: Conozco el orden de ejecución para un solo thread (o core) pero no si tengo muchos threads (o cores)

¿Cómo sincronizamos?

Como en "The mind", podríamos sincronizar con un clock. Pero eso, además de ser muy fuerte, puede fallar y traer demasiado **overhead**.

Veamos este ejemplo. ¿Cómo podemos hacer que Pepe siempre codee antes que Pato?

```
Pepe
a1. Desayunar
a2. Bañarse
a3. Codear
```

```
Pato
b1. Desayunar
b2. Codear
```

- Inicialmente, sabemos que a1 > a2 > a2 y por otro lado b1 > b2
- Agregamos a4. Avisar Bob y b2. Esperar aviso de Pepe, b3. Codear
- Es cierto que ahora b1 > a1 **NO!**. Ambos desayunan concurrentemente
- Es cierto que a1 > a2 > a3 y b1> b2 > a3?

Sincronización con mensajes

- ✓ Sincronización con mensajes . ¿Ventajas? ¿Desventajas?
- Sincronización con memoria compartida ?? To be continued...