Sistemas Operativos I

Mutex: Locks

Mini-repaso de la última clase

- Definimos la idea de competencia y cooperación de procesos (aunque nos concentramos en la competencia)[Pero dejé un ejercicio de cooperación (?)]
- Condición de Carrera
- Operación Atómica
- Sección Crítica

La clase anterior hablamos de dos clasificaciones de problemas, procesos/hilos que cooperan para obtener una solución o bien compiten por los recursos.
Vimos que la competencia introduce una nueva categoría de errores que llamamos condiciones de carrera introducida por comportamiento no determinista de la ejecución de las operaciones.
Vimos que además podíamos considerar cierta operaciones atómicas, como operaciones que podían verse como fragmentos de código que o bien se ejecutan completamente o fallan, pero no hay estados intermedios observables.
Y finalmente definimos la sección crítica como el fragmento de códigoque debe ser protegido cuando se lo ejecuta concurrentemente.
Hoy nos concentramos en continuar un poco más la teoría dentro del mundo Mutex y presentaremos el primer mecanismo de sincronización llamado Lock.

Problema de los Jardines Ornamentales

Como no encontré ninguna imagen relevante, la hacemos usando la pizarra de Google Meet!!
Y lo programamos en c. Asumimos una cantidad de visitantes arbitraria y usamos **dos hilos** para simular los molinetes de las entradas utilizando memoria compartida entre los hilos.

Jardín Ornamental

ZOO

COVID 19

Propiedad Safety

Propiedad Mutex: A lo sumo un proceso accede a la Sección Crítica.

Cuando definimos el problema Mutex lo definimos en base a una propiedad que tiene que ser garantizada: la región crítica es accedida por lo sumo un proceso.
Las propiedades de 'Safety' son las que garantizan que **nada malo puede pasar.**
Esto lo lograremos insertando mecanismos de sincronización antes y después de la sección crítica.
entry_protocol; cs_code(in); exit_protocol.

entry_protocol == tomar_mutex
exit_protocol == soltar_mutex

Lo asumimos para luego romper el espejo… Y generar un momento ‘que carajo’

Problema de los Jardines Ornamentales (I)

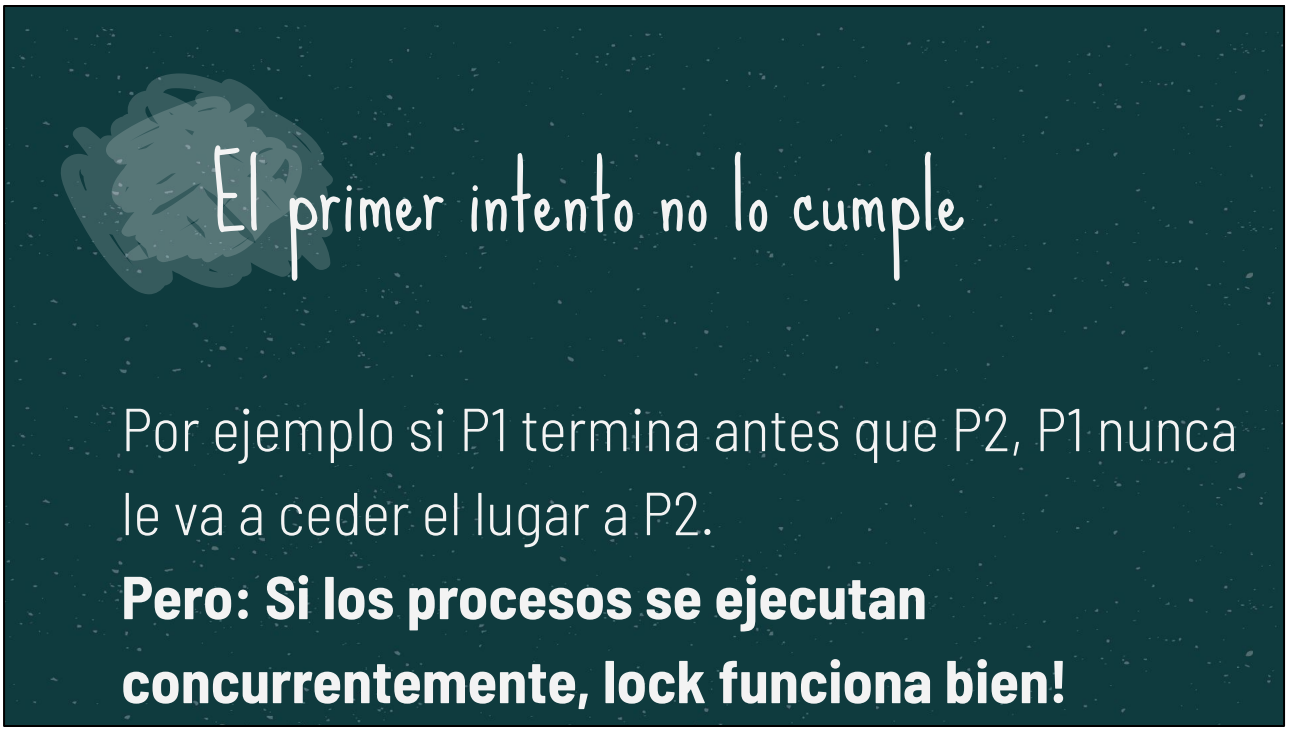
Veamos un intento de solución.
La idea es siempre tratar de dejar que el otro proceso ejecute antes que el actual.
Para esto usaremos una variable int para indicar a quién le toca.

```
lock_i {
  ...
  victima = i;
  while( victima == i );
 …
}
unlock{}
```

Propiedad Liveness

Ausencia de *Deadlock* : Si hay procesos intentado tomar/soltar a un lock, algún proceso va a tomar/soltar el lock.

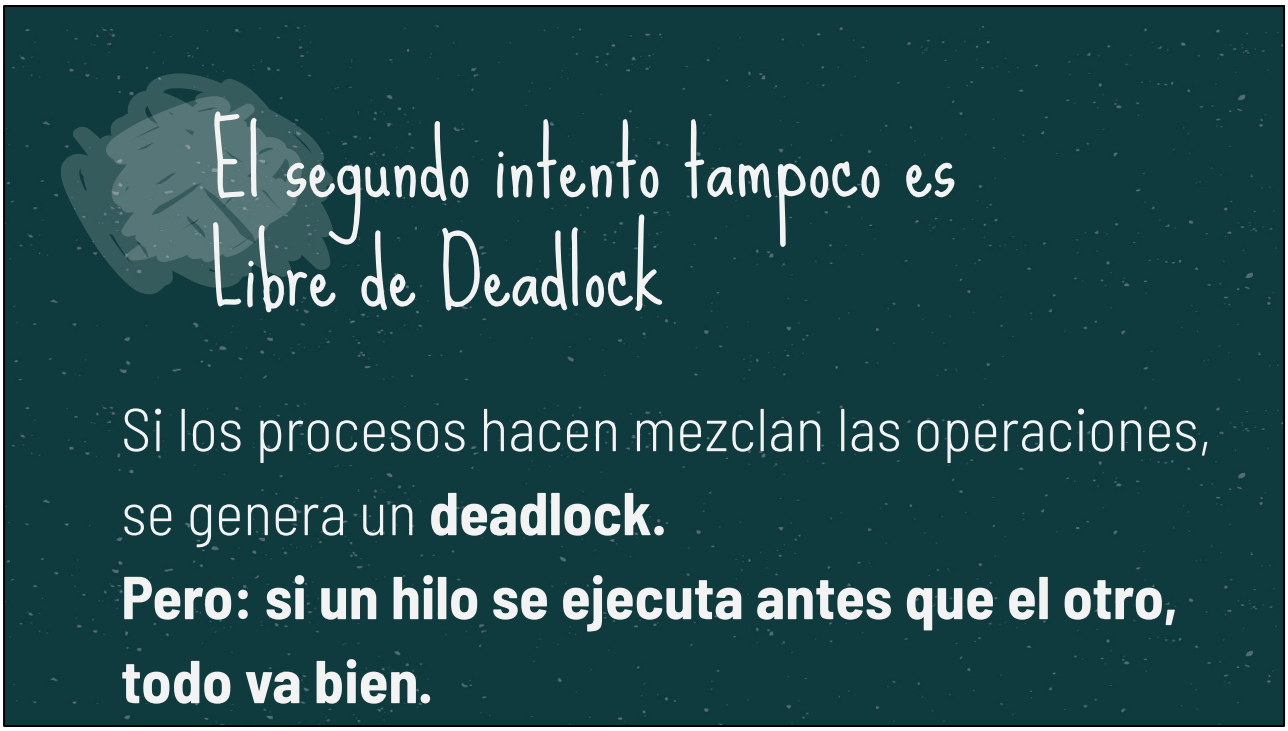
Si bien el programa que vimos cumple con la propiedad de ‘Safety’ no es suficiente para garantizar la correcta ejecución del programa concurrente. Necesitamos una propiedad que predique sobre la ejecución del programa.



El primer intento no lo cumple porque necesita que los dos procesos se ejecuten concurrentemente. Si P1 se completa antes que P2, P2 nunca podrá terminar su ejecución.
Pero tiene una propiedad interesante: Si los procesos se ejecutan concurrentemente, lock funciona bien!.

Problema de los Jardines Ornamentales (II)

Veamos un segundo intento de solución… La idea es que vamos a anunciar cuando un proceso tiene la intención de ejecutarse, si el otro proceso no tiene la intención de ejecutarse entonces se ejecuta el proceso actual.
Esto soluciona el problema anterior! Si P1 termina antes que P2, P1 no va a tener intención de ejecutarse entonces P2 se ejecutará tranquilo.



El segundo intento tampoco cumple con la propiedad. Si las operaciones de escritura/lectura de los procesos se mezclan llegamos a un caso donde ambos se estarán esperando mutuamente.
P1 -> intents; P2 -> intents; P1 while because P2 has raised their flag; P2 while because P1 has raised their flag
Pero tiene una propiedad interesante: si un hilo se ejecuta antes que el otro todo va bien.

Algoritmo de Peterson

Intento 1 + 2 ~ Algoritmo de Peterson! [Al parecer ése agradable señor es Peterson]

Acá se nos rompe el tema que la lectura/escritura de variables en C no es atómica :(!

Propiedad Liveness

Ausencia de *Inanición* : Siempre que un proceso quiera tomar un lock, eventualmente lo hará.

Asombrosamente, el algoritmo de Peterson cumple con otra propiedad muy interesante.
Ausencia de Inanición implica Ausencia de Deadlock. Pero no funciona al revés…
Podemos pensar en 3 procesos, p1, p2 y p3… Si siempre toman el lock P1 y P2, P3 nunca podrá tomar el lock. Esa posible ejecución (infinita claramente) cumple con Ausencia de Deadlock (p1 y p2) siempre acceden a la región crítica, pero P3 no lo hace nunca (violando la propiedad de ausencia de Inanición).

Problema con Peterson

Solo sirve para dos procesos!

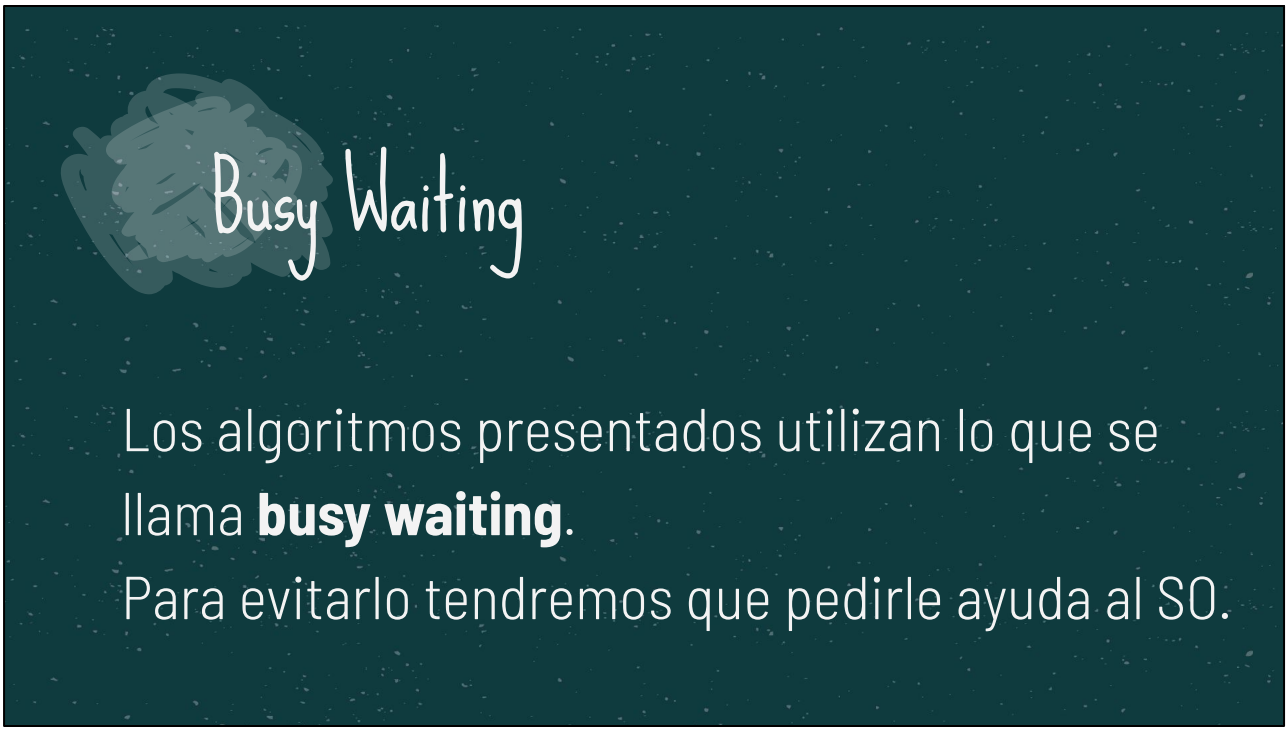
Se puede modificar para aceptar **N** procesos, introduciendo **N** niveles de espera.

Algoritmo de la Panadería

Pero hay un algoritmo que es mucho mejor, inventado por Leslie Lamport (una de las grandes figuras de la época).
La idea es la de simular el sistema de tickets de una panadería.
La analogía va en la líneas de: al aparecer un cliente, toma un número y espera hasta que sea llamado. Hay una entidad central (el panadero) que va atendiendo y llamando los números.

Se da en general cuando los procesos compiten por varios recursos. Por ejemplo, asumamos que hay dos locks en vez de uno, programados utilizando el algoritmo de Perterson (que ya sabemos que es correcto.
Y pensemos que tenemos dos procesos A y B que comparten L0 y L1. De manera que necesitan ambos locks para acceder continuar ejecutándose. Si A toma L0 y B toma L1, y luego A necesita L1 y B necesita L0, entonces ninguno de los dos podrá progresar generando un **deadlock.**

Por ejemplo, si dos personas van caminando y se van a chocar, ambas deciden moverse a un costado al mismo tiempo, y como se van a chocar ambas deciden moverse al mismo tiempo, y como ….



Notar que busy waiting a los ojos del SO es un desperdicio del CPU! Se puede atenuar usando `sleep`.

PThreads Locks

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Hoy en día la mayoría de los procesadores implementan directivas que bloquean a los procesos de forma atómica, sin la necesidad de tener bucles (busy waiting)