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 (race condition)
- 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 código que 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 del Jardín Ornamental

Un gran jardín ornamental se abre al público para que todos puedan apreciar sus fantásticas rosas, arbustos y plantas acuáticas. Por supuesto, se cobra una módica suma de dinero a la entrada para lo cual se colocan dos molinetes, uno en cada una de sus dos entradas. Se desea conocer cuánta gente ha ingresado al jardín así que se instala una computadora conectada a ambos molinetes: estos envían una señal cuando una persona ingresa al jardín.

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.

Programemos

Se simulará un experimento en el que 20 visitantes ingresan por cada molinete. Al final de la simulación deberá haber 40 visitantes contados.

Ejemplo minimal de que muestra un race condition en el problema del jardín ornamental.

Puede darse la siguiente secuencia de eventos durante la ejecución de estas instrucciones:

1. cuenta = 0
2. torniquete1: LEER (resultado: rax de p1 = 0, cuenta = 0)
3. torniquete1: INC (resultado: rax de p1 = 1, cuenta = 0)
4. El sistema operativo decide cambiar de tarea, suspende torniquete1 y continúa con torniquete2.
5. torniquete2: LEER (resultado: rax de p2 = 0, cuenta = 0)
6. torniquete2: INC (resultado: rax de p2 = 1, cuenta = 0)
7. torniquete2: GUARDAR (resultado: rax de p2 = 1, cuenta = 1)
8. El sistema operativo decide cambiar de tarea, suspende torniquete2 y continua con torniquete1.
9. torniquete1: GUARDAR (resultado: rax de p1 = 1, cuenta = 1)

Ambos procesos ejecutaron sus instrucciones para incrementar en 1 el contador. Sin embargo, ¡en este caso cuenta tiene el valor 1! A este problema también se lo conoce como *problema de las actualizaciones múltiples*.

Una instrucción aparentemente simple, como cuenta = cuenta + 1 habitualmente

implica varias operaciones de más bajo nivel:

LEER Leer cuenta desde la memoria (p. ej. mov $cuenta,%rax).

INC Incrementar el registro (p. ej. add $1,%rax).

GUARDAR Guardar el resultado nuevamente en memoria (p. ej. mov%rax, $cuenta).

En el problema hay claramente un *recurso compartido* que es la cuenta, por tanto, el código que modifica la cuenta constituye una *sección crítica* y la operación cuenta = cuenta + 1 debe ser una operación atómica. La secuencia de eventos que se mostró es una *condición de carrera*.

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

Programemos un primer intento de solución

Intento con una bandera: mediante una variable de bandera se indica si hay un proceso en la región crítica

Acá el problema es que la sección crítica ahora es la bandera.

Programemos un segundo intento de solución

La idea es siempre tratar de dejar que el otro proceso ejecute antes que el actual.

Veamos otro 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 {
  ...
  turno = i;
  while( turno == i );
 …
}
unlock{} // me quedo con el turno hasta que termine
```

Acá se puede pensar la variante de pasarle el turno al otro en el unlock.

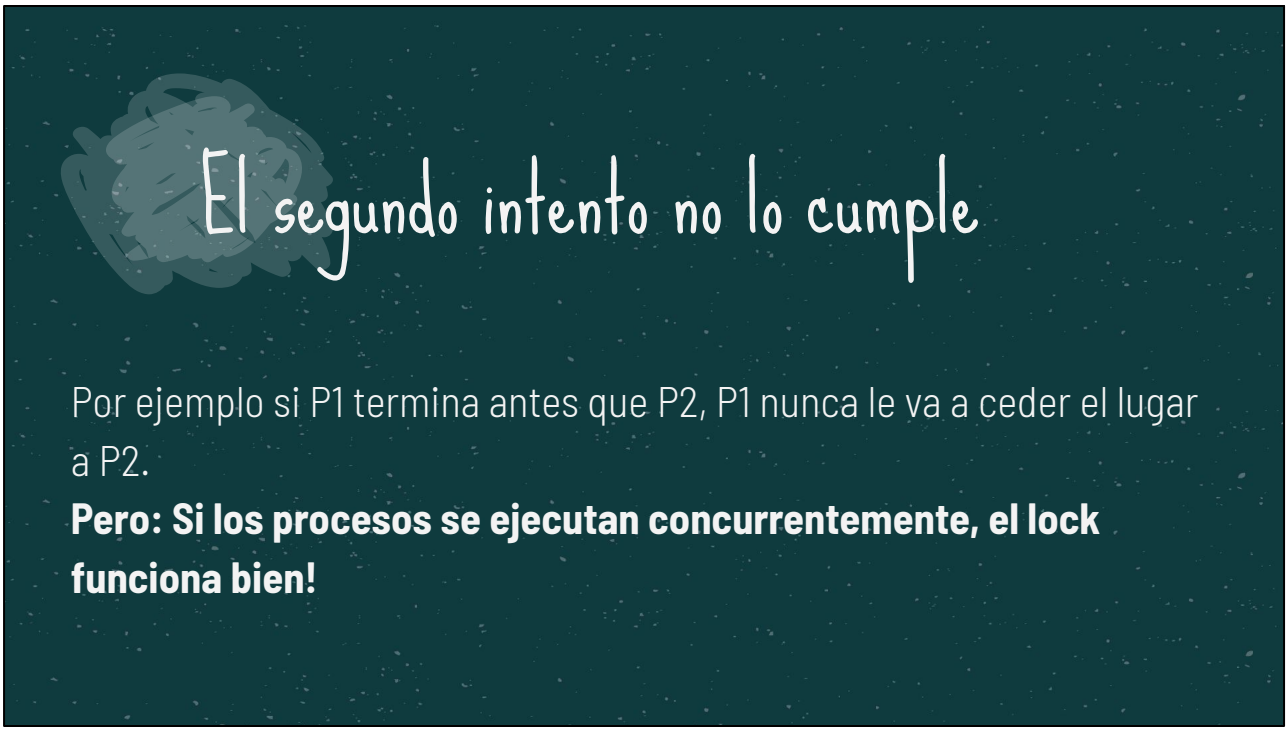
```
lock_i {
  ...
  turno = i;
  while( turno == i );
 …
}
unlock{
  turno = i+1; // le cedo el turno al otro thread
}
```

Esto resuelve el problema pero secuencializa el programa haciendo que entre una persona por molinete alternadamente si o si.

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.



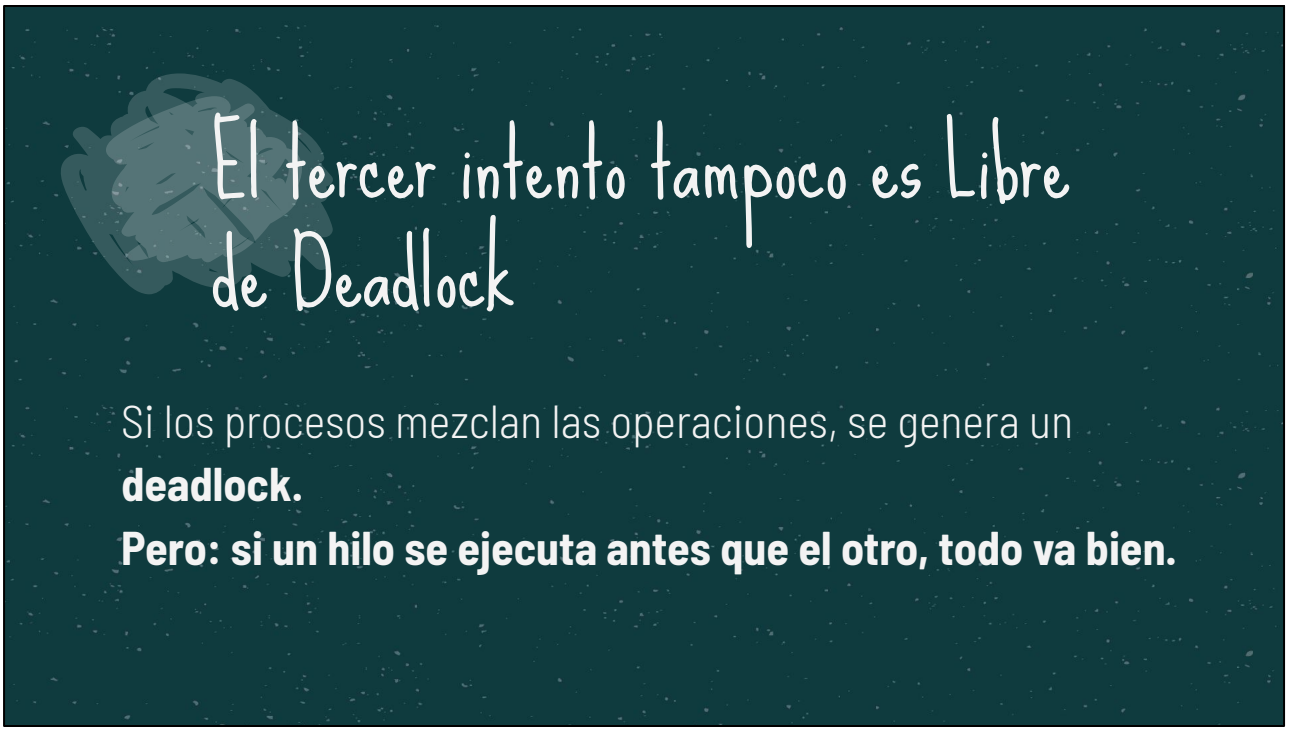
El segundo intento no lo cumple porque necesita que los dos procesos se ejecuten concurrentemente. Si P1 se completa antes que P2, P2 nunca podrá terminar su ejecución.
Pero tiene una propiedad interesante: Si los procesos se ejecutan concurrentemente, el lock funciona bien!.

Programemos un tercer intento de solución

Indicar la intención de entrar a la región crítica.

Para paliar los efectos de la solución anterior se puede intentar indicar si el otro proceso también está queriendo entrar en la región crítica.

Veamos un tercer 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 tercer 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 :(!

Vamos a requerir usar mfence!!!

—————————————————
Thread 1
—————————————————

```
// lock()
flag[0] = 1;
turn = 2;
while (flag[1] && (turn == 2))
      ;

/* región crítica */

/* unlock */
flag[0]=0;
```

—————————————————
Thread 2
—————————————————

```
// lock()
flag[1] = 1;
turn = 1;
while (flag[1] && (turn == 1))
      ;

/* región crítica */

/* unlock */
flag[1]=0;
```

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.

Algoritmo de la Panadería Simplificado

```
/* Variables globales */
Número: vector [1..N] de enteros = {todos a 0};

/* Código del hilo i */
lock(i)
{
  /* Calcula el número de turno */
  Número[ i ] = 1 + max(Número[ 1],..., Número[ N ]);

  /* Compara con todos los hilos */
  for j in 1..N
  {
    /* Si el hilo j tiene más prioridad, espera a que ponga su número a cero */
    /* j tiene más prioridad si su número de turno es más bajo que el de i */
    while ((Número[ j ] != 0) && (Número[ j ] < Número[ i ])) { /* busy waiting */ }
  }
}

/* Sección crítica */

unlock(i){
  Número[ i ] = 0;
}
```

Este algoritmo de la panadería simplificado tiene un problema. Puede ser que dos procesos simultáneamente calculen un número de turno (ejecutando al mismo tiempo la función max), obteniendo ambos el mismo número de turno.

Algoritmo de la Panadería

```
/* Variables globales */
Número: vector [1..N] de enteros = {todos a 0};
Eligiendo: vector [1..N] de booleanos = {todos a false};

/* Código del hilo i */
lock(i)
{
 /* Calcula el número de turno */
 Eligiendo[i] = true;
 Número[i] = 1 + max(Número[1],..., Número[N]);
 Eligiendo[i] = false;

 /* Compara con todos los hilos */
 for j in 1..N
 {
  /* Si el hilo j está calculando su número, espera a que termine */
  while ( Eligiendo[j] ) { /* busy waiting */ }

  /* Si el hilo j tiene más prioridad, espera a que ponga su número a cero */
  /* j tiene más prioridad si su número de turno es más bajo que el de i, */
  /* o bien si es el mismo número y además j es menor que i */
  while ((Número[j] != 0) &&
    ((Número[j] < Número[i]) || ((Número[j] == Número[i]) && (j < i)))) { /* busy waiting */ }
 }
}

/* Sección crítica */

unlock(i)
{
 Número[i] = 0;
}
```

En primer lugar, el proceso establece que su variable de Eligiendo sea true, lo que indica su intención de ingresar a la sección crítica. Luego, se le asigna el número de ticket más alto teniendo en cuenta al de los otros procesos. Luego, la variable Eligiendo se establece en false, lo que indica que ahora tiene un nuevo número de ticket. ¡En realidad es una pequeña sección crítica en sí misma! El propósito de las primeras tres líneas es que si un proceso está modificando su valor de ticket, en ese momento no se debe permitir que otro proceso chequee su valor de ticket anterior, que ahora está obsoleto. Es por eso que dentro del ciclo for antes de verificar el valor del ticket, primero nos aseguramos de que todos los demás procesos tengan la variable Eligiendo como false.

Después de eso, procedemos a verificar los valores de los tickets de los procesos donde el proceso con el menor número de ticket entra en la sección crítica. En caso de que haya dos procesos con el mismo número de ticket se desempata a favor del que tenga menor ProcessID. La sección de salida simplemente restablece el valor del boleto a cero.

El busy waiting while (Eligiendo[j]) { /* busy waiting */ } asegura que no se le está asignando un nuevo número de ticket a otro proceso. Ejemplo, supongamos que no exista Eligiendo. El thread 0 busca su número y le da 42, pero no lo llega a escribir en Número[i] = 1 + max(Número[1],..., Número[N]); y se interrumpe. El thread 1 también elige 42 (porque nadie escribió) y ve que el 0 no tiene número, así que entra a la RC. Ahora el thread 0 se despierta, escribe su 42, y ve que el thread 1 también tiene el 42 pero como 0<1, también entra a su RC.

La incorporación del array Eligiendo con el busy waiting while (Eligiendo[j]) { /* busy waiting */ } y la condición de ((Número[j] == Número[i]) && (j < i)) podemos lidiar con la falta de atomicidad de Número[i] = 1 + max(Número[1],..., Número[N]);.

Consistencia Secuencial

Asumimos

1. que las operaciones de cada procesador se realizan en el orden especificado, y
2. que los *stores* (escrituras a memoria) son inmediatamente visibles al otro procesador.

Esto no es cierto en procesadores modernos (Ej: x86). Estos implementan **ejecución fuera de orden** y **store buffering**.

Ejecución fuera de orden

Con ejecución fuera de orden, puede ser que algunas instrucciones se hagan antes de lo estipulado en el código si la CPU se encuentra bloqueada para seguir el pipeline de instrucciones secuencialmente. Ejemplo:

```
divl %ecx # eax := eax/ecx
movl arr( %eax), dest
movl flag, %ebx
```

Considerar en el algoritmo de Peterson qué pasa si la lectura de los flag en la condición del while se ejecuta como primer paso.

Supongamos que la división puede tardar alrededor de 20 ciclos. El primer movl depende del resultado de la división, con lo cual no podemos ejecutarlo sin esperar al resultado de la división. Sin embargo, el segundo movl no depende de ninguna instrucción previa, y la CPU puede detectar esto y ejecutarlo mientras se espera el resultado de la división. Estos reordenamientos pueden ocurrir con cientos de instrucciones de por medio.

No todos los reordenamientos son permitidos, las reglas dependen de cada arquitectura. En x86/x64, se garantiza que los únicos reordenamientos que pueden ocurrir son mover loads (lecturas de memoria) antes de stores (escrituras a memoria), obviamente mientras no sean a la misma dirección.

Store buffering

Los sistemas multiprocesador no garantizan que sus cachés son consistentes. Cuando un procesador escribe en una dirección de memoria, esto no es inmediatamente visible a los otros procesadores. Nuevamente, esto implica que ambos procesadores ejecutando el algoritmo de Peterson pueden avanzar en sincronía y entrar a sus RC. (En x86/x64, esto se debe en particular al store-buffering.)

Si bien la CPU reordena y no mantiene cachés coherentes, hace la burocracia necesaria para fingir que las operaciones se realizan secuencialmente. Dentro de cada procesador, no podemos observar estos efectos. Al escribir primitivas de sincronización, esta ilusión se rompe, y debemos hacer algunos cambios explícitos.

Fence (barrera de memoria)

Un fence (instrucción mfence) causa que la CPU:

1. No reordene instrucciones a través del mismo
2. Garantice que todas las escrituras previas al fence son visibles a todos los procesadores antes de continuar

Peterson con mfence quedaría:

```
f1 = true
turn = 2
asm("mfence");
while (f2 && turn == 2)
/* esperar */;
CRIT;
f1 = false

——------------------------------------------------------------------------------------

f2 = true
turn = 1
asm("mfence");
while (f1 && turn == 1)
/* esperar */;
CRIT;
f2 = false
```

Optimizaciones del Compilador

Otro problema independiente son las optimizaciones del compilador. En general, el compilador asume que las variables no cambian espóntaneamente. Es decir, si tenemos el fragmento:

```
v = 1;
/* ... mucho código que no modifica a 'v' */
x = v;
```

El compilador está autorizado a optimizar la última asignación a x=1. Nuevamente, esto nos puede traer problemas (explicar cómo, y verificarlo usando -O3).

Estas optimizaciones ocurren en tiempo de compilación, antes de correr el programa. Puede inspeccionarse el código assembly generado para determinar si fueron hechas o no. Por otro lado, los problemas de la sección anterior ocurren en tiempo de ejecución.

Para evitar esto, se usa en general la palabra clave volatile. Usada en declaraciones de variables le dice al compilador que la variable puede cambiar espontáneamente, sin acción del programa. Es decir, si v fuera declarada como volatile, la optimización anterior ya no está permitida.

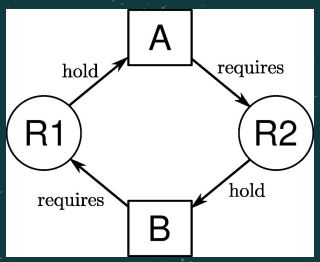
Instrucciones de Sincronización (x86)

Usualmente, no se usa el algoritmo de Peterson para sincronizar procesos. Tampoco el de Lamport, ni ningún otro que funcione solamente por memoria compartida. En cambio, se usan instrucciones especiales provistas por la arquitectura que permiten sincronizar de manera más eficiente.

El ejemplo clásico es usando una instrucción compare-and-swap (CAS) atómica, la cual es provista por varias arquitecturas. **CAS(l,a,b)** lee la dirección de memoria l, y si contiene un valor igual a **a**, lo sobreescribe con **b**, pero no lo modifica en otro caso. A la vez, retorna un booleano que indica si la escritura fue realizada o no. De esta forma, podemos implementar mutexes usando una variable en memoria que valga 0 cuando el mutex está libre y 1 cuando está tomado. Con esto, **lock() puede implementarse como while (!CAS(l, 0, 1));**

Deadlock

El término *deadlock* se utiliza cuando un programa concurrente entra en un estado donde **ningún proceso puede progresar.**



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.**

Ejemplo de Deadlock

Cree un programa que tenga deadlock.

Livelock

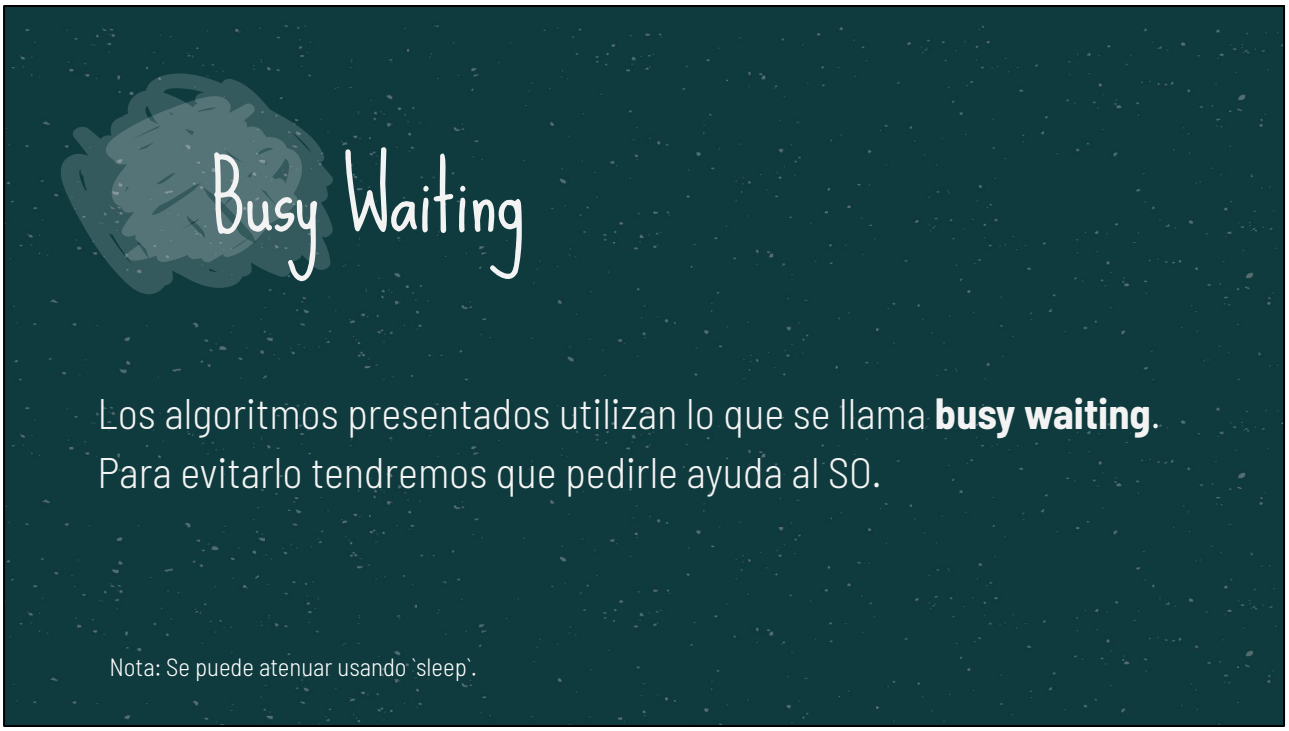
El término *livelock* se utiliza cuando dos o más procesos (activamente) realizan pasos que previenen el progreso de los otros procesos.

Un ejemplo conocido es el del código que intenta detectar y manejar situaciones de deadlock. Si dos threads detectan un deadlock e intentan "hacerse a un lado" el uno del otro, sin cuidado terminarán atrapados en un bucle haciendose siempre a un lado y nunca lograrán avanzar. Por "hacerse a un lado" nos referimos a que liberarían el lock e intentarían dejar que el otro lo adquiriera.

```
// thread 1
getLocks12(lock1, lock2)
{
  lock1.lock();
  while (lock2.locked())
  {
    // attempt to step aside for the other thread
    lock1.unlock();
    wait();
    lock1.lock();
  }
  lock2.lock();
}

// thread 2
getLocks21(lock1, lock2)
{
  lock2.lock();
  while (lock1.locked())
  {
    // attempt to step aside for the other thread
    lock2.unlock();
    wait();
    lock2.lock();
  }
  lock1.lock();
}
```

Por ejemplo, si dos personas van caminando y se van a chocar, ambas deciden moverse a un mismo costado al mismo tiempo, y como se van a chocar ambas deciden moverse nuevamente 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);
```

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

```
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

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).

El pthread_spin_lock es un bloqueo de "busy waiting". Su principal ventaja es que mantiene el thread activo y no provoca un cambio de contexto, por lo que si se sabe que solo se esperará durante muy poco tiempo (porque su sección crítica es muy rápida), entonces puede brindar un mejor rendimiento que un pthread_mutex_lock. Por el contrario, un pthread_mutex_lock provocará una menor demanda en el sistema si la sección crítica lleva mucho tiempo y es deseable un cambio de contexto.

Para poder ver estas funciones en el man de linux es necesario instalar el manual POSIX Programmer's Manual:
sudo apt-get install manpages-dev
sudo apt-get install manpages-posix-dev