Sistemas Operativos I

Sincronización entre procesos
Introducción a Hilos.

Vamos introducir conceptos de sincronización de procesos, la problemática y algunas definiciones para ya adentrarnos a las soluciones la próxima clase.

Sincronización de Procesos

La sincronización se produce cuando uno o más procesos dependen de del comportamiento de otro proceso, y viene en dos sabores:

- Competencia: procesos compiten por un recurso
- Cooperación : procesos cooperan. Barrier + Productor/Consumidor

En general *sincronización* es el conjunto de reglas y mecanismos que permiten la especificación e implementación de propiedades secuenciales de cada proceso que garantizan la correcta ejecución de un programa concurrente.

Competencia, es lo que nos vamos a concentrar en la mayor parte, se da cuando varios procesos compiten por el mismo recurso.
Aunque también hay modelos donde los procesos cooperan, y esto se da cuando un conjunto de procesos dependen del avance de otros procesos. Por ejemplo, de la utilización de barreras que garantizan que los procesos han llegado a cierto punto de la ejecución, o el famoso problema del Productor/Consumidor.
El problema del productor/consumidor consiste en que haya dos procesos: uno dedicado a producir cierto token, y otro a consumirlo, ambos representados en un loop eterno.
Garantizando que:

+ Solo se consumen token generados por el productor
+ Cada token producido es conusmido exactamente una vez.

Una solución al problema del productor y consumidor es utilizando barreras, secuencializando cada juego, pero no es la mejor… Otra mejor puede consistir en el uso de un buffer que contenga productos a consumir. Cada vez que el productor produce agrega al buffer (hasta que se llena), y el consumidor consume hasta que se vacía.

Competencia

Varios procesos compiten para ejecutar una instrucción pero **solo uno puede hacerlo**.
Ejemplo: E/S a un archivo!

- **seek(x)** : mueve el cabezal a la posición **x**
- **read()** : retorna el valor que se encuentre en la posición que se encuentre el cabezal
- **write(v)** : escribe el valor **v** en la posición que se encuentre el cabezal

Programemos!

- unistd : write/read/lseek
- fctnl : open/close
- lseek+write ~> seek_write
- lseek+read ~> seek_read

Parent and Child compinting for reading and writing.

Problema de la Exclusión Mutua (Mutex)

Condición de Carrera : es una categoría de errores de programación que involucra a dos procesos que fallan al comunicarse su estado mutuo, llevando a resultados inconsistentes. Ocurre en general al no considerar la falta de atomicidad en las operaciones.

Problema de la Exclusión Mutua (Mutex)

Operación Atómica: Manipulación de datos que requiere la garantía de que se ejecutará **como** una sola unidad de ejecución, o fallará completamente, sin resultados o estados parciales observables por otro proceso o en el entorno.

Ejemplo: seek, write, read.

Notar que no es estrictamente necesario que el procesador o el SO ejecute la operación contiguamente o no se le retire el control de la unidad de tiempo. Lo que se expresa es que la operación se realiza o no, independientemente como fue exactamente ejecutada, y que no hay punto intermedio observable.

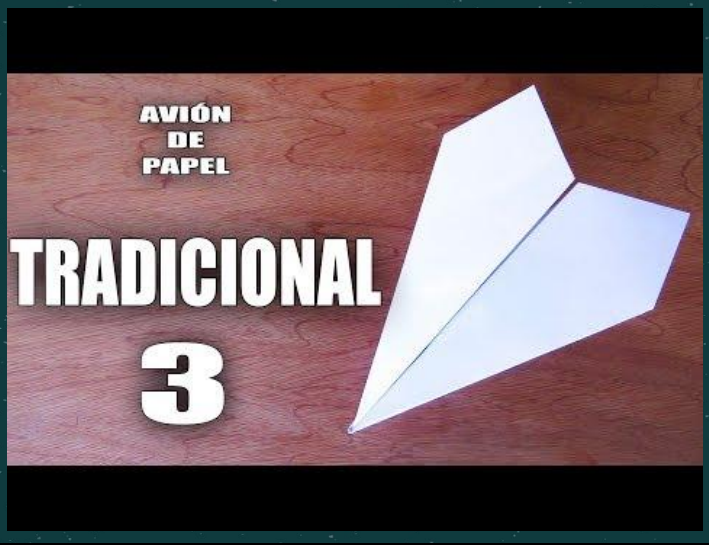
Problema de la Exclusión Mutua (Mutex)

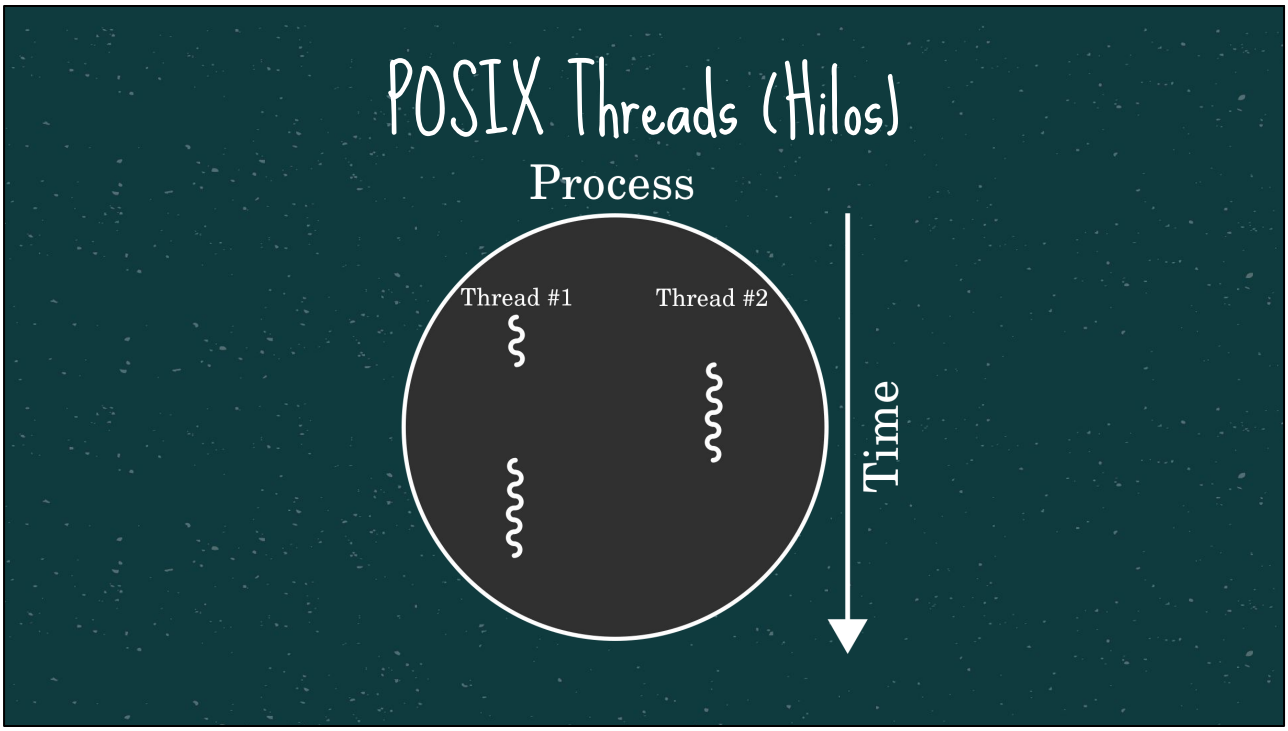
Sección crítica: sección que necesita ser protegida al ser usada por múltiples procesos.

Ejemplo: seek_write y seek_read son dos fragmentos de código que para garantizar un correcto funcionamiento se las tiene que proteger.

El problema de la exclusión mutua consiste en garantizar que las secciones críticas son accedidas por a lo sumo un sólo proceso.
Durante esta parte de la materia veremos diferentes mecanismos de sincronización para resolver el problema de la exclusión mutua, justamente para garantizar el correcto acceso a la secciones críticas de nuestros programas. Es decir, diseñar algoritmos que mediante sincronización garanticen que la sección crítica se accede de la manera adecuada.

Descanso: 10 mins





En general copiar todo el estado de un proceso es demasiado costoso para los objetivos de un programa concurrente, incluso puede volver prohibitivo el uso de la programación concurrente, desperdiciando muchos recursos en la generación de una copia (siguiendo el modelo fork).

Para evitar este problema se crea lo que se conoce como Hilos o procesos ligeros que comparten todo el estado de ejecución de un proceso pero
cada uno lleva solo la información necesaria para poder ejecutarse independientemente del otro (el PC, registros, y algo más, pero menos que todo el estado).
Por lo que los hilos comparten **mucha información** y se introducen **muchas regiones críticas.**

Los hilos pueden simularse enteramente a nivel de usuario (sin ayuda del SO) y suelen llamarse *hilos de usuario o hilos verdes*. Esto es muy útil si
en software embebido.
Mientras que los hilos que se crean con ayuda del sistema operativo se denominan *hilos de kernel* y se requiere una librería: pthreads

En la materia vamos a tener el placer de jugar con ambos. En C utilizando hilos de kernel y más adelante en Erlang utilizando procesos de Erlang.

POSIX Threads (pthreads.h)

Creación de Hilos

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg);

Compile and link with -pthread.
```

La creación de hilos se hace invocando la función `pthread_create` que toma:

- un elemento de tipo pthread_t
- una estructura de atributos utilizados para configurar diferentes tipos de threads, en general utilizaremos la configuración por defecto
- la función a invocar, notar que el procedimiento a invocar toma una dirección de memoria y devuelve otra. En C esto significa ‘toma algo y devuelve algo’.
- su argumento

Mostrar el funcionamiento

POSIX Threads (pthreads.h)

Esperar a que los hijos terminen

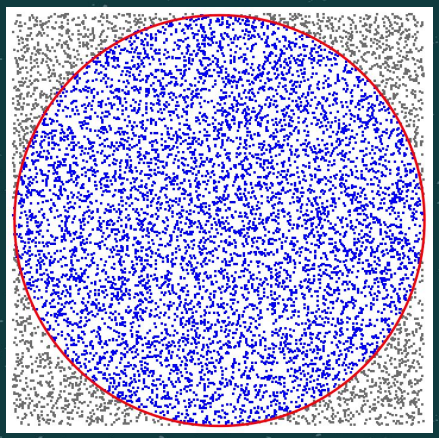
```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);

Compile and link with -pthread.
```

Como vimos en el ejemplo, como los hilos son parte de **un proceso** cuando el proceso termina, todos los hilos terminan con él.
Para eso utilizamos la función `pthread_join` donde indicamos el thread que vamos a esperar y el espacio de memoria destinado a su valor de retorno.

Revisitemos el ejemplo

POSIX Threads (Ejercicio)

Aproximación de π por método monte carlo



$A_s = (2 * r)^2 = 4 * r^2$

$A_c = \pi * r^2$

$A_c/A_s = (\pi * r^2) / (4 * r^2) = \pi / 4$

Puntos = sq_aleatorios(NPuntos)

$\pi = (4 * Puntos\_Circ) / NPuntos$

Para terminar la clase de hoy se deja como ejercicio la implementación de un programa concurrente que de una aproximación del número pi.
Para esto se propone utilizar el método de montecarlo de aproximación de Pi.

La idea consiste en sí el área del cuadrado es $A_s = (2*r)^2 = 4 * r^2$, mientras que la de la circunferencia es $A_c = pi * r^2$.
La razón de las áreas es entonces $A_c / A_s = (pi * r^2) / (4 * r^2) = pi / 4$, entonces, $pi = (4 * A_c) / A_s$.

Para calcular pi entonces utilizaremos la generación aleatoria de C. La razón de los puntos generados dentro y fuera de la circunferencia sigue la misma razón que las areas antes mencionadas.
Por ejemplo, si se generan $NPuntos$ de las cuales $CPuntos$ caen dentro de la circuferencia, $pi = (4 * CPuntos) / NPuntos$.

Se recomienda comenzar por una implementación secuencial, y luego una concurrente. Pensar cuántos procesos utilizar, y que se utilicen todos (htop viene bien)...