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 del comportamiento de otro proceso, y puede darse de dos formas:

- **Competencia:** procesos compiten por un recurso (Ej: Acceso a una Impresora)
- **Cooperación :** procesos cooperan por un objetivo común. Barreras, Productor/Consumidor, etc.

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. Por ejemplo, el uso de un periférico (Ej: impresora)
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 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 consumido 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.

Los procesos en competencia luchan por recursos sin un fin común, mientras que los procesos en cooperación colaboran para alcanzar un objetivo compartido.
Ambos requieren mecanismos de sincronización (semáforos, mutex, etc.), pero en la cooperación también es clave la comunicación.

Competencia

Varios procesos compiten para ejecutar una instrucción pero solo uno puede hacerlo.

Veamos algunos ejemplos...

EJEMPLO de Competencia!

Dos procesos intentan escribir un string dado en la salida estándar (stdout). Escriben de a un caracter (Utilizar la función **putc()** para escribir y **setbuf()** sobre **stdout** para que no bufferee caracteres.).

Ejemplo minimal de que muestra un race condition cuando dos procesos intentar escribir de a 1 caracter en la salida estándar

EJEMPLO de Competencia!

Entrada/Salida a un archivo!

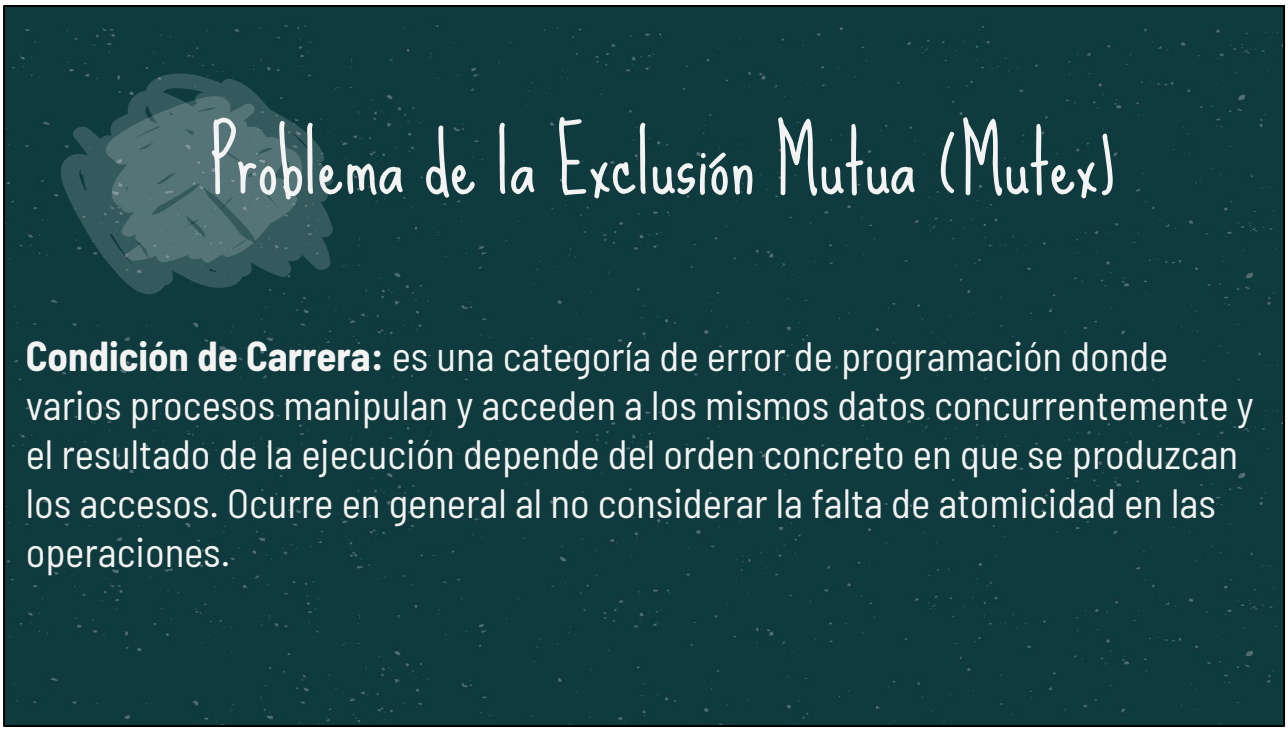
- **`lseek(x)`**: mueve el cabezal a la posición **`x`**
- **`read()`**: retorna el valor que se encuentre en la posición que se encuentre el cabezal
- **`write(v)`**: escribe el valor **`v`** en la posición que se encuentre el cabezal

Implementar:

- **`seek_write(x,v)`**: escribe el valor v en la posición x
- **`seek_read(x)`**: retorna el valor que se encuentre en la posición x

Librerías: `unistd(write/read/lseek)` y `fctnl(open/close)`

Parent y Child compiten por leer y escribir



Mutex: viene de Mutual Exclusion

Problema de la Exclusión Mutua (Mutex)

Operación Atómica: Manipulación de datos que requiere la garantía de que se ejecute 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.

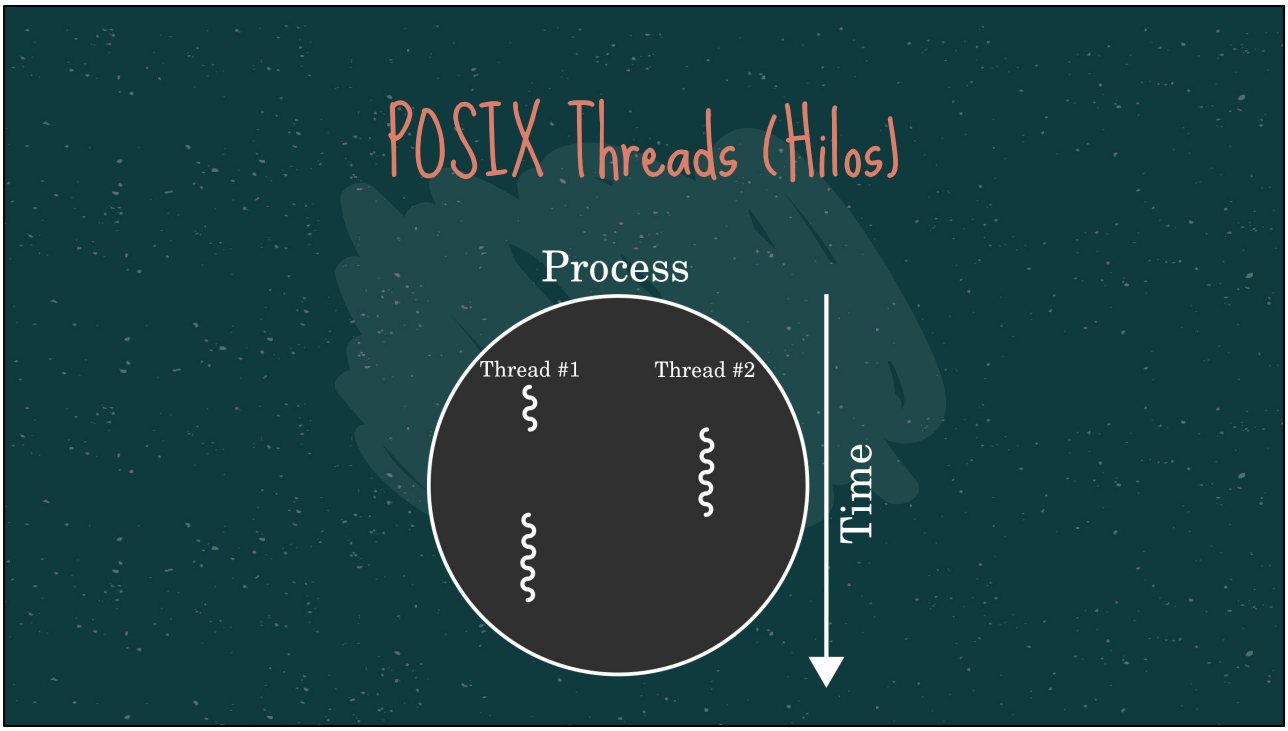
Problema de la Exclusión Mutua (Mutex)

El **problema de la exclusión mutua** consiste en garantizar que las secciones críticas son accedidas por a lo sumo un sólo proceso.

Sección crítica: segmento de código donde el proceso accede/modifica memoria compartida. Cuando un procesos está ejecutando una sección crítica, ningún otro proceso puede ejecutar su correspondiente sección crítica.

Ejemplo: seek_write y seek_read son dos segmentos 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.



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 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)

La función **pthread_create()** inicia un nuevo hilo en el proceso de llamada. El nuevo hilo comienza la ejecución invocando **start_routine()** con los argumentos en **arg**. La estrucutra **attr** permite configurar el thead ((tamaño del stack, prioridad del thread, etc).

```
#include <pthread.h>

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

Compilar con el flag -pthread.
```

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

- un elemento de tipo pthread_t: pthread_t es el tipo de datos que se utiliza para identificar de forma única un subproceso.
- una estructura de atributos utilizados para configurar diferentes threads (tamaño del stack, prioridad del thread, entre otros), en general utilizaremos la configuración por defecto (para esto pasar como argumento NULL).
- 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’.
- Los argumentos de la función
- Retorna 0 si tiene éxito, caso contrario retorna un número de error y el comportamiento es indefinido

Mostrar el funcionamiento

EJEMPLO!

Lanzar un thread que imprima en salida standard “Hello World!”

POSIX Threads (pthreads.h)

La función **pthread_join()** espera a que finalice el hilo especificado por el hilo. Si ese hilo ya terminó, entonces **pthread_join()** regresa inmediatamente.

```
#include <pthread.h>

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

Compilar con el flag -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

EJEMPLO!

El thread creador espera a que el nuevo thread termine.

POSIX Threads (pthreads.h)

Un thread puede terminar de la siguiente manera:

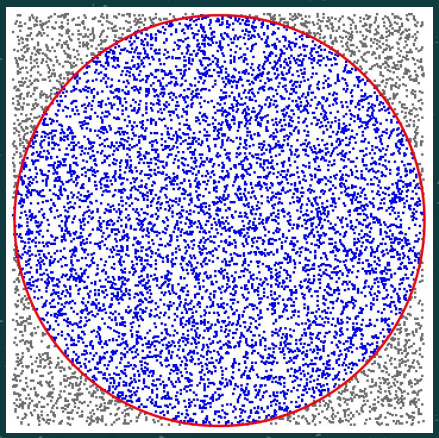
- Llama a **pthread_exit(3)**, especificando un valor de estado de salida que está disponible para otro hilo en el mismo proceso que llama a **pthread_join(3)**.
- Retorna **start_routine().** Esto es equivalente a llamar a **pthread_exit(3)** con el valor proporcionado si se llama a **return**.
- Se cancela (ver **pthread_cancel(3))**.
- Cualquiera de los threads en el proceso llama a **exit(3)**, o el subproceso principal realiza un retorno desde **main().** Esto provoca la terminación de todos los subprocesos en el proceso.

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 = π * r^2
A_c/A_s = (π * r^2) / (4 * r^2)
= π / 4
--------------------------------
Puntos = sq_aleatorios(NPuntos)
π = (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 áreas 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)...