Sistemas Operativos I

Programas Concurrentes

En esta clase introduciremos varios conceptos:

- Proceso vs Programa
- Vida de una Proceso
- Estructuras
- Programas Concurrentes
- Garantías del SO
- Programación en C: Fork/Exec/Wait/Exit

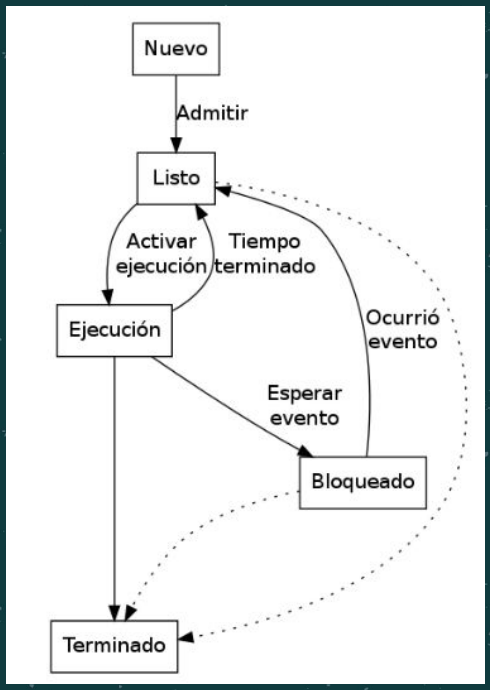
El Sistema Operativo brinda la ilusión de que varios procesos pueden ejecutarse simultáneamente, pero en realidad, es la cantidad de hardware subyacente quien dicta realmente cuántos procesos pueden estar ejecutándose simultáneamente.

Un proceso NO es sólo un programa en ejecución. Un proceso incluye también:

- Los Archivos abiertos
- las señales (signals) pendientes
- Datos internos del kernel
- El estado completo del procesador
- Un espacio de direcciones de memoria
- Uno o más hilos de Ejecución. Cada thread contiene:
    - Un único contador de programa
    - Un Stack
    - Un Conjunto de Registros
- Una sección de datos globales

La vida de un Proceso

- Nuevo
- Listo
- Ejecución
- Bloqueado
- Terminado
- Zombie

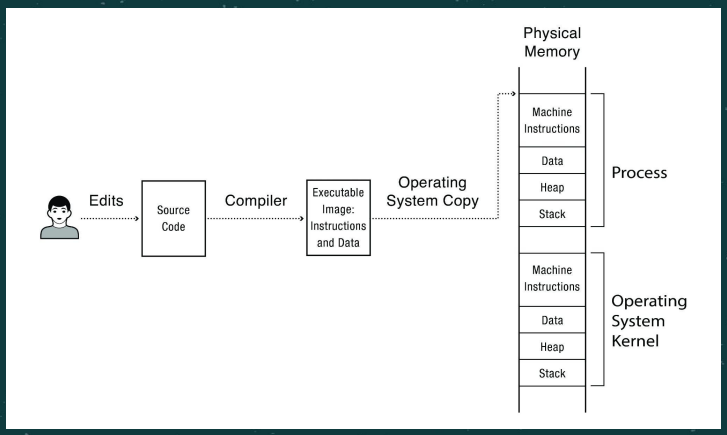


+ Nuevo: Se solicitó al SO la creación de un proceso cuyo recursos y estructuras están siendo creadas
+ Listo: Está listo para iniciar la ejecución pero el SO no le asignó ningún procesador
+ En Ejecución: El proceso está siendo ejecutado en éste momento.
+ Bloqueado: En espera de un evento para poder continuar su ejecución
+ Terminado: El proceso terminó de ejecutarse y sus estructuras están a la espera de ser eliminadas por el SO.
+ Zombie: El programa ha terminado su ejecución pero el SO todavía debe realizar algunas operaciones de limpieza para poder eliminarlo. Notificar al proceso padre, cerrar conexiones de red, liberar memoria, etc. Programas que terminaron (con `exit`) pero todavía se guarda información de ellos.

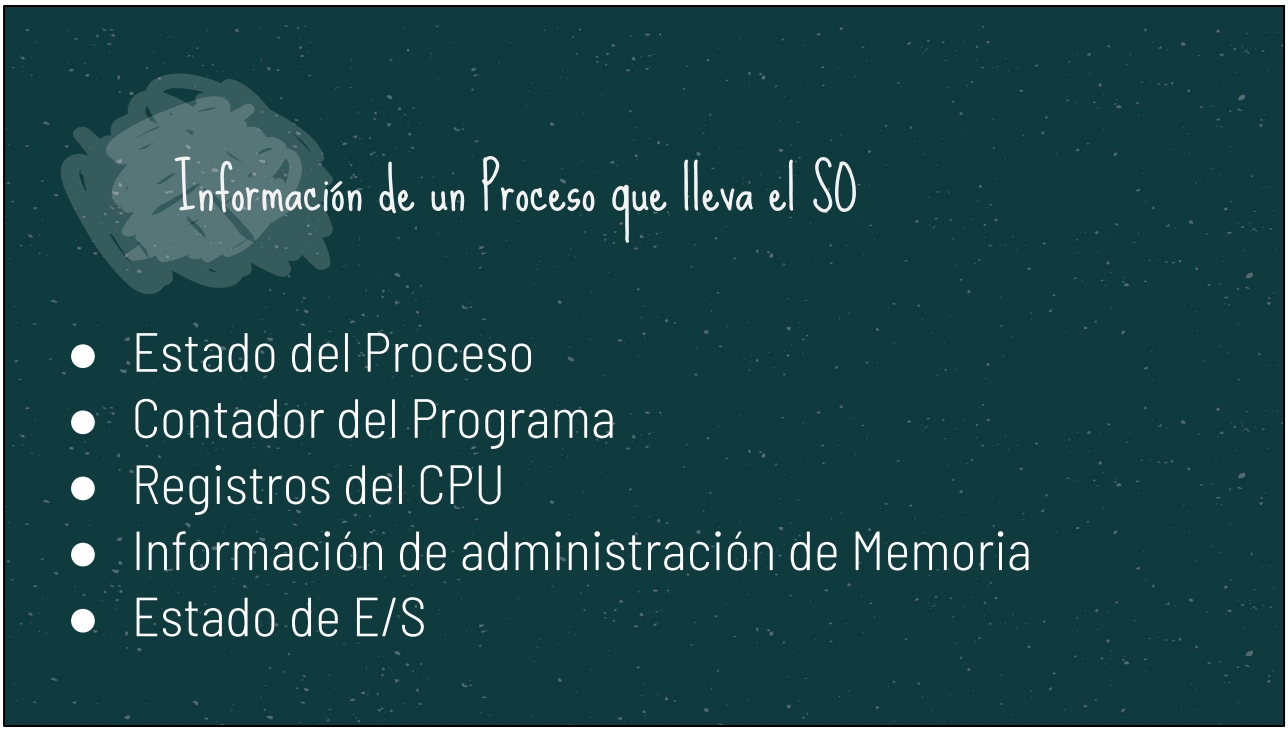
Áreas de Memoria que Ocupa un Proceso

Los procesos están dividido en 4 segmentos:

- **Text**
- **Data**
- **Heap**
- **Stack**

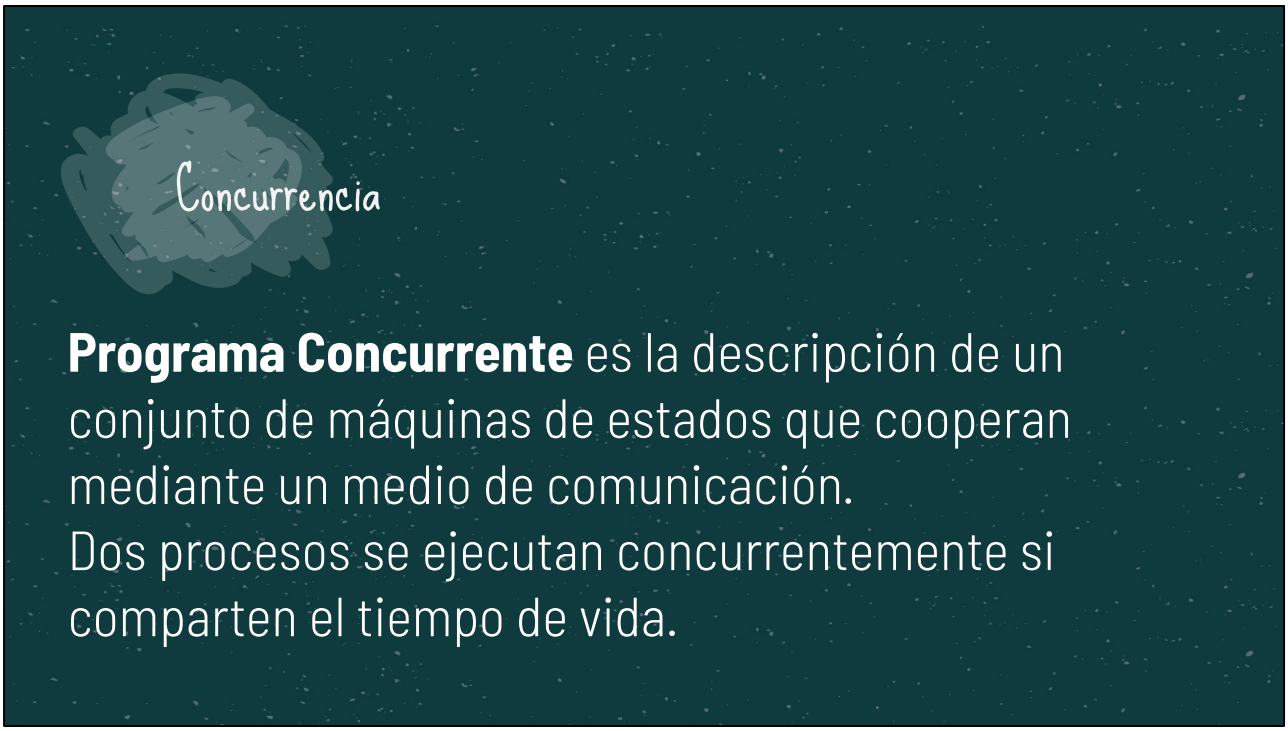


Text: Instrucciones del Programa.
Data: Variables Globales (extern o static en C). La zona de memoria estática es para datos que no cambian de tamaño, permite almacenar variables globales que son persistentes durante la ejecución de un programa.
Heap: permite almacenar variables adquiridas dinámicamente (vía funciones como malloc o calloc) durante la ejecución de un programa.
Stack: permite almacenar argumentos y variables locales durante la ejecución de las funciones en las que están definidas.

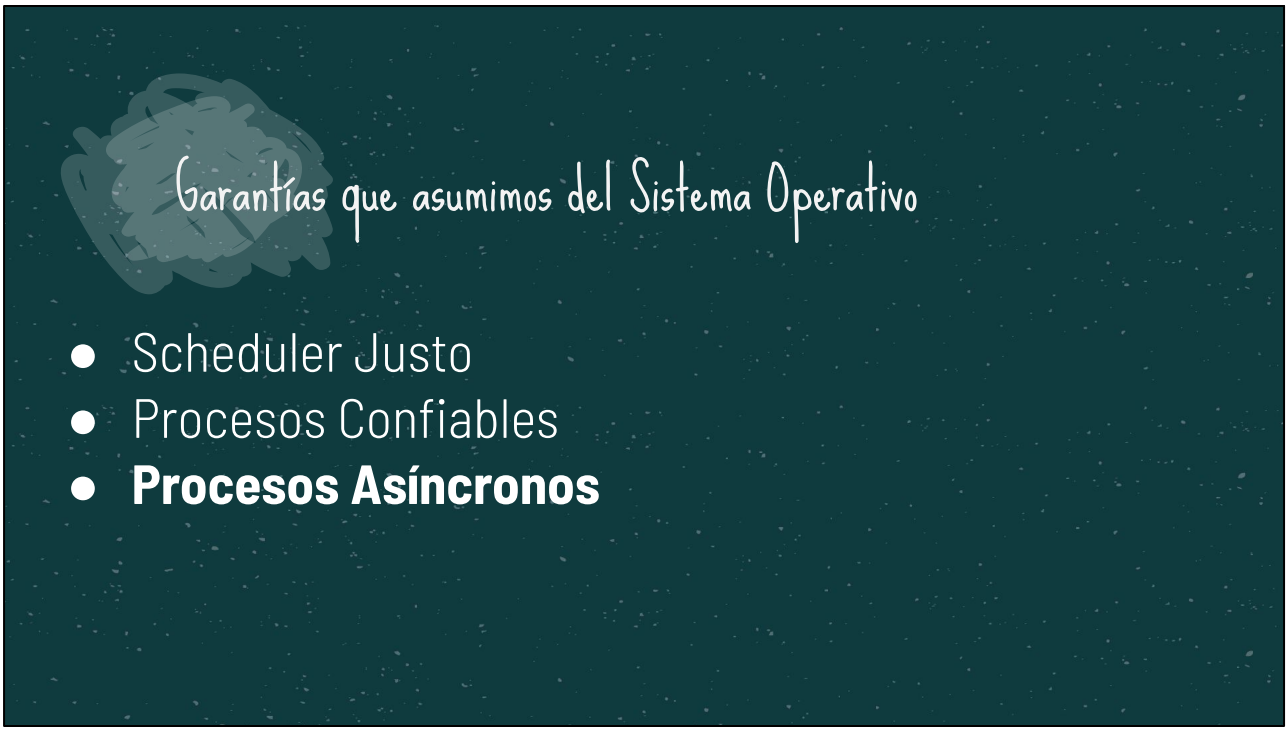


El sistema operativo lleva información del estado de ejecución de cada proceso:

+ El estado de ejecución, nuevo, listo, en ejecución, etc
+ La siguiente instrucción a ejecutar
+ Que registros del cpu está utilizando
+ información de la memoria que está utilizando
+ Lista de dispositivos de entrada y salida que el proceso a abierto



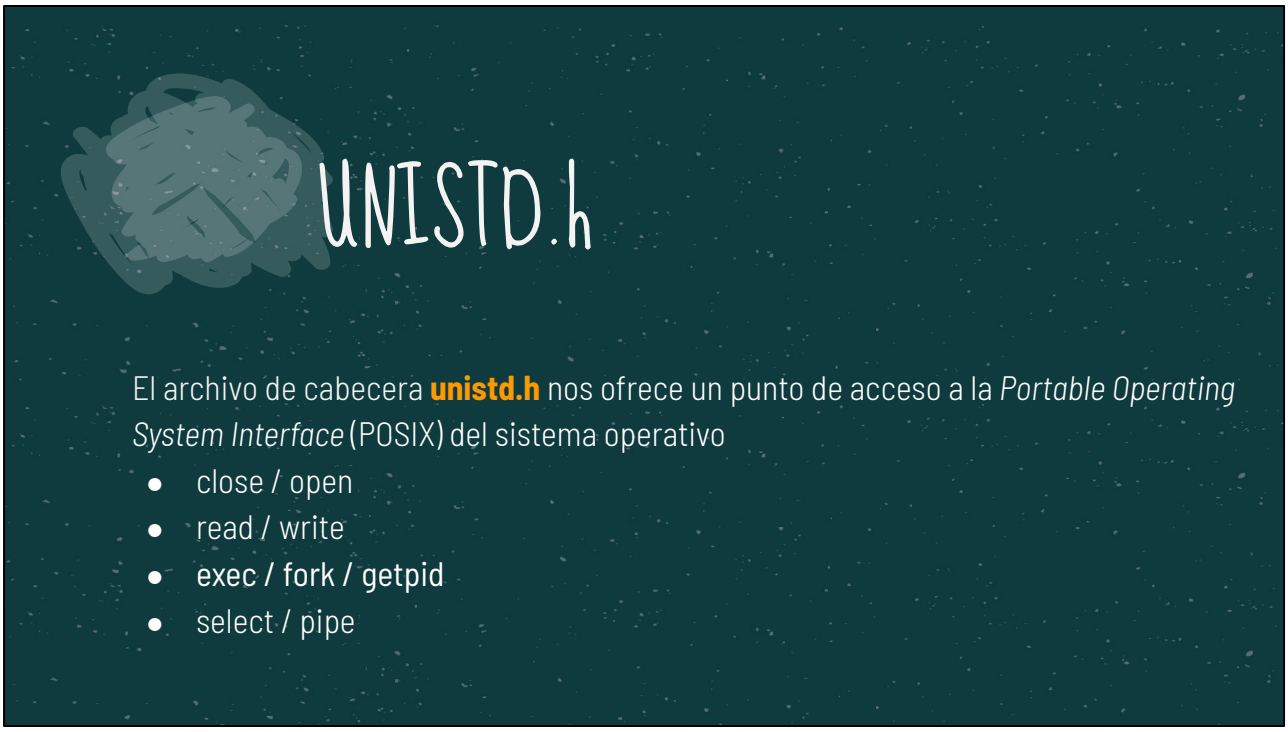
Un programa concurrente es la descripción de un algoritmo mediante la cooperación de un conjunto de máquinas de estados que se comunican de alguna manera.
Dos procesos se dicen concurrentes si comparten en (algún momento) un tiempo de vida. **No quiere decir que se ejecuten simultáneamente**, uno puede estar listo mientras otro puede estar en ejecución, etc. Para que se ejecuten en simultáneo es necesario contar con el hardware para hacerlo. La programación paralela se basa en la idea de la ejecución de procesos de forma simultánea, que entra bajo la definición de procesos concurrentes.



En el caso que tengamos menos procesadores (unidades de procesamiento) físicos que procesos tendremos que seleccionar que procesos se ejecutarán en cada momento. Eso lo hace el Sistema Operativo, y está totalmente oculto a los procesos (y nosotros).

+ El scheduler (o administrador de procesos) del Sistema Operativo es el encargado de seleccionar qué procesos, dentro del conjunto de procesos listos, comenzarán la ejecución. Nosotros asumimos que el scheduler es justo en el sentido que todo proceso listo eventualmente se ejecutará.
+ Los procesos harán lo que el programa les dice que hacer, y podemos confiar que su ejecución es fiel.
+ Además, no asumimos ninguna restricción de tiempo sobre la ejecución de las operaciones de cada uno de los procesos. En particular, no podemos asumir ningún orden de ejecución entre procesos concurrentes.

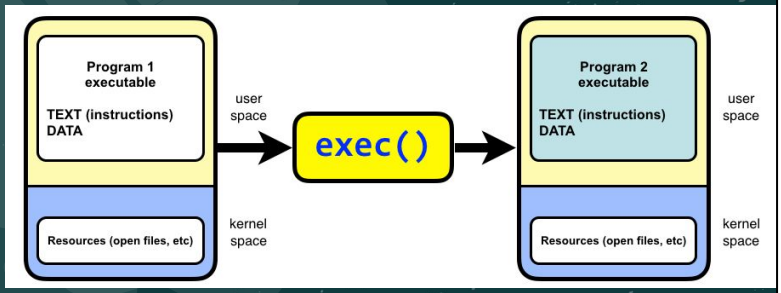
Vamos a pasar a aplicar los conceptos que aprendimos hoy.
Veremos cómo crear procesos mediante llamadas a sistema, en particular dos formas de hacerlo. Veremos formas primitivas de compartir información.



Comunicación con el Sistema Operativo por medio de la librería Unistd.
En particular, hoy utilizaremos **exec, fork and getpid.**

Exec

Reemplaza la imagen de programa del proceso actual por una nueva. Es decir, termina la ejecución del programa actual, y comienza la de otro **sin cambiar de Identificación de proceso**.



Notar que aquí no hay comunicación alguna, cuando un proceso invoca a una función de la familia de exec su programa es reemplazo, y otro se ejecuta en su lugar.

EJEMPLO!

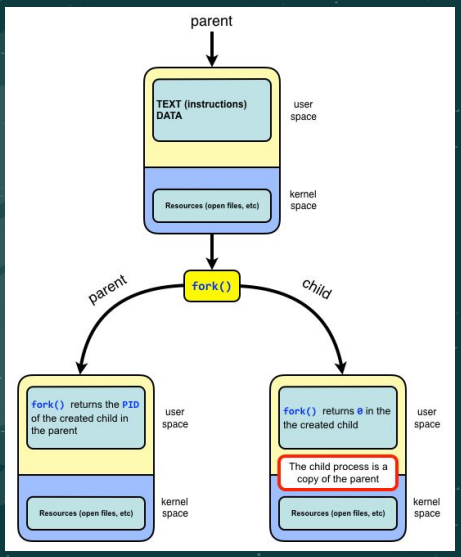
Proceso que invoca a otro mientras mostramos los PIDs!

Ver cómo funcinoa getpid()

Fork

Se crea una copia del proceso que invoca a **fork** pero en un nuevo espacio de memoria.
El proceso que invoca a **fork** se lo denomina **parent** mientras que el proceso creado por la invocación se denomina **child**.
Tenemos 3 resultados posibles a la llamada **fork**

- **-1** indicando un error
- **0** indicando que es el proceso child
- el PID (>0) del proceso child



Es decir, el proceso Parent sabe quien es porque el resultado de fork() es el PID del proceso child (y siempre es mayor a 0), mientras que el proceso Child no sabe quién es el Parent porque el resultado de invocar a fork() es 0!
Notar que la relación es asimétrica, por eso se llama Parent/Child, porque el proceso Parent sabe quien és el proceso Child mientras que es el proceso Child no siempre sabe quién es el proceso Parent. Y se utiliza la nomenclatura Parent/Child para establecer la cronología, Parent viene desde antes que Child, y es quien *lo crea.*

Por fin tenemos procesos compartiendo vida! Es decir, procesos concurrentes!!

Fork: los procesos NO comparten memoria.
Fork: los procesos si comparten la tabla de File Descriptors, esto es, ven los mismos archivos, pipes, E/S standard.
Fork: Se hace la copia de la memoria utilizando la técnica Copy-on-write. La memoria solo se copia bajo demanda. Esto es, la memoria se copia cuando el proceso Child quiere modificarla.

EJEMPLO!

Proceso que crea otro con fork, y mostramos que los PIDs varían!

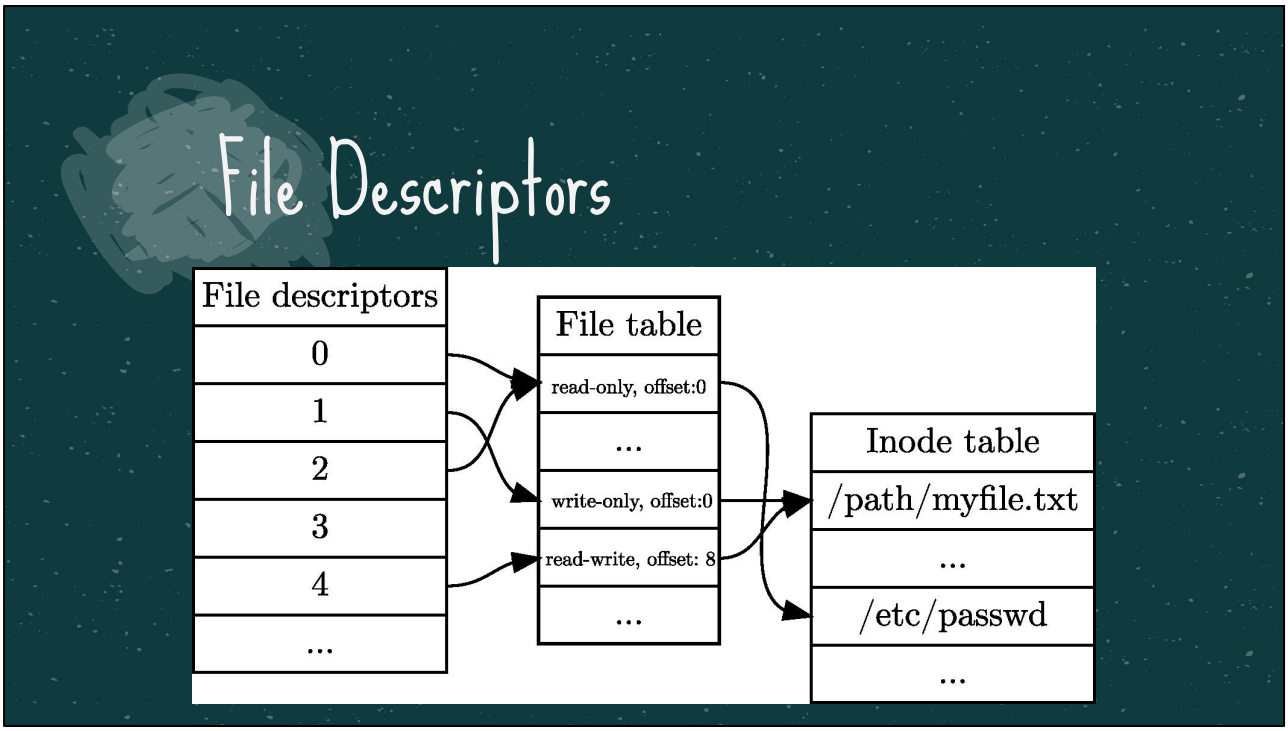
Notas sobre fork()

- El Child tendrá su propio ID de proceso único.
- Los procesos Parent/Child **NO comparten memoria**.
- Los procesos tienen una copia de la **tabla de File Descriptors**, esto es, ven los mismos archivos, pipes, E/S standard.
- La copia memoria se hace utilizando la técnica ***Copy-on-write***. La memoria solo se copia bajo demanda. Esto es, la memoria se copia cuando el proceso Child quiere modificarla.

EJEMPLO!

¿Cómo se comporta el siguiente programa?

```
int main()
{
  pid_t pid1 = fork();
  pid_t pid2 = fork();
  printf("pid1: %d pid2: %d!\n", pid1, pid2);
  return 0;
}
```



- Un **File Descriptor** es un unsigned integer utilizado por un proceso para identificar un archivo abierto.
- Para cada proceso en nuestro sistema operativo, hay un Process Control Block (PCB). PCB realiza un seguimiento del contexto del proceso. Entonces, uno de los campos dentro de esto es un array llamado File Descriptor Table.
- **File Descriptor Table:** es la colección de índices de enteros File Descriptors en los que los elementos son punteros a elementos de la **File Table**. Para cada proceso, se proporciona una File Descriptor Table única en el sistema operativo.
- La **File Table** es una estructura de datos residente en el kernel, que contiene detalles de todos los archivos abiertos. Un elemento en la **File Table** es una estructura en memoria para un archivo/recurso abierto, que se crea cuando se procesa una solicitud para abrir un archivo/recurso y estas entradas mantienen la posición del archivo/recurso. Los archivos/recursos podrían ser:

- File
- E/S de terminal
- Pipes
- Socket (para el canal de comunicación entre computadoras)
- Cualquier dispositivo (CD-ROM, USB, etc)

File Descriptors

- Cada proceso en ejecución comienza con tres archivos ya abiertos:

Nombre del File Descriptor	**Nombre corto**	**Número del File Descriptor**	**Descripción**
Entrada estándar	stdin	0	Entrada desde el teclado
Salida estándar	stdout	1	Salida a la consola
Salida de error estándar	stderr	2	Salida de error a la consola

creat/open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

- El valor de retorno de open() es un File Descriptor, que se puede utilizar por llamadas al sistema como read(2), write(2), lseek(2), fcntl(2), etc, para hacer referencia al archivo abierto. El File Descriptor devuelto será el descriptor de archivo con el número más bajo que no esté abierto actualmente para el proceso.

read/write/close

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
```

- **read()** intenta leer hasta contar bytes desde el descriptor de archivo fd en el búfer a partir de buf.
- En caso de éxito, se devuelve el número de bytes leídos (0 indica el final del archivo) y la posición del archivo avanza en este número. No es un error si este número es menor que el número de bytes solicitados; esto puede suceder, por ejemplo, porque hay menos bytes disponibles en este momento (tal vez porque estábamos cerca del final del archivo, o porque estamos leyendo de un pipe, o de un terminal), o porque read() fue interrumpido por una señal.

Ejercicio!

Abrir y leer de un archivo 1024 bytes.

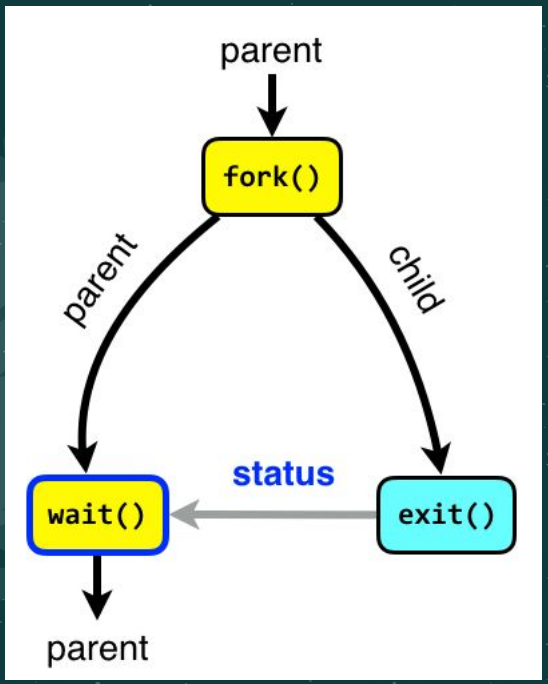
Ejercicio!

Escribir un programa en C que toma como entrada la dirección a un binario y un tiempo en segundos, de manera tal que ejecuta el comando cada los segundo especificados.

Función útil **sleep** debería ser útil

Wait

Cuando un proceso parent crea un proceso child éste puede esperar el resultado de la ejecución del proceso child.



EJEMPLO!

Proceso que crea otro con fork, el proceso parent espera a que finalice el proceso child!

Ejercicio!

Hacer una Shell, la misma debe poder leer comandos (con o sin argumentos) ingresados por teclado y ejecutarlo. Una vez finalizado el programa que se ejecutó, debe volver al prompt y permitir el ingreso de otro programa. Se debe también proporcionar el comportamiento de poder salir de la shell.

Sugerencia: utilizar fgets() para la lectura de la entrada estándar y strtok() para parsear el ejecutable y sus argumentos.