Sistemas Operativos I

Programas Concurrentes

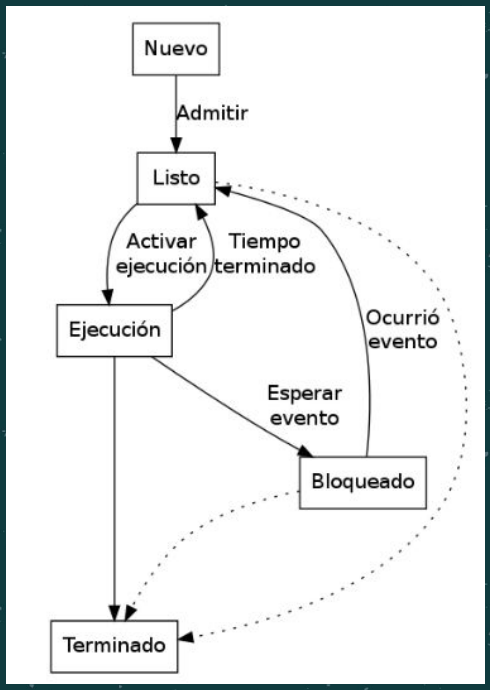
En esta clase introduciremos varios conceptos:

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

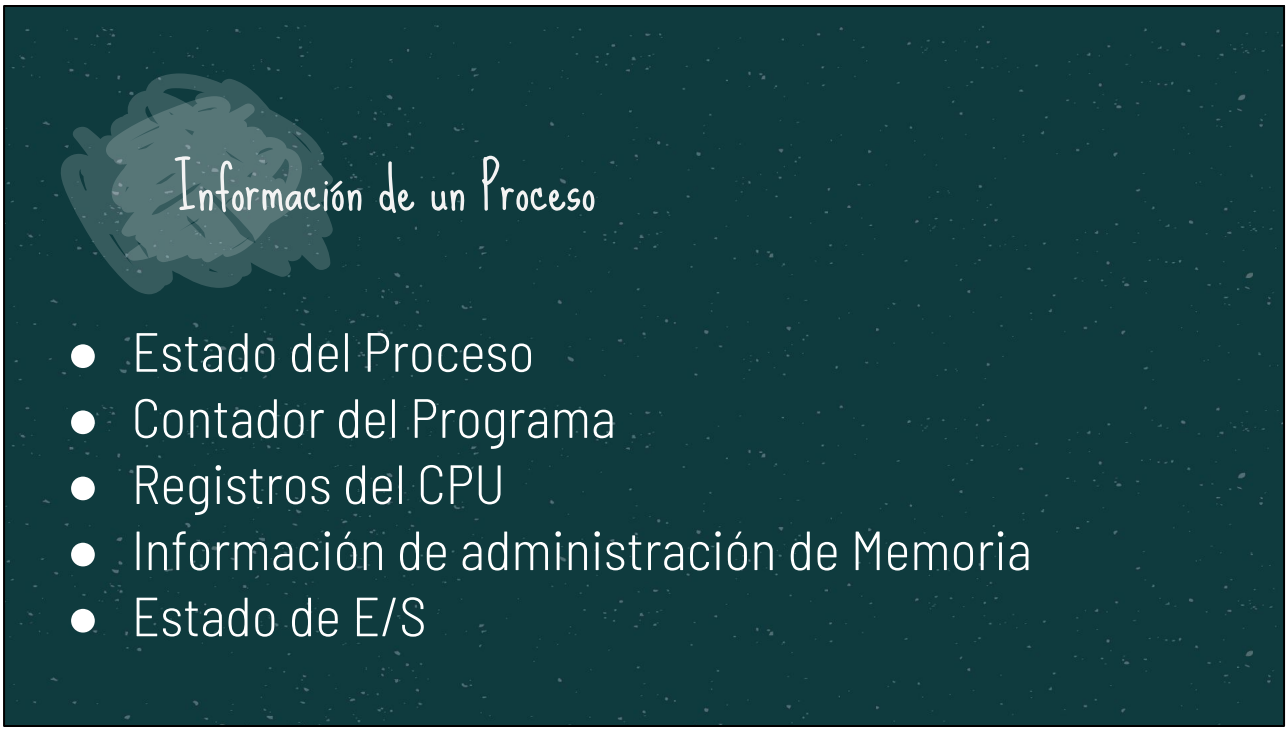
El Sistema Operativo brinda la ilusión de que varios procesos pueden ejecutarse simultáneamente, pero en realidad, es la cantidad de hardware subyacente quien dicta realmente cuántos procesos pueden estar ejecutándose simultáneamente.

La vida de un Proceso

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

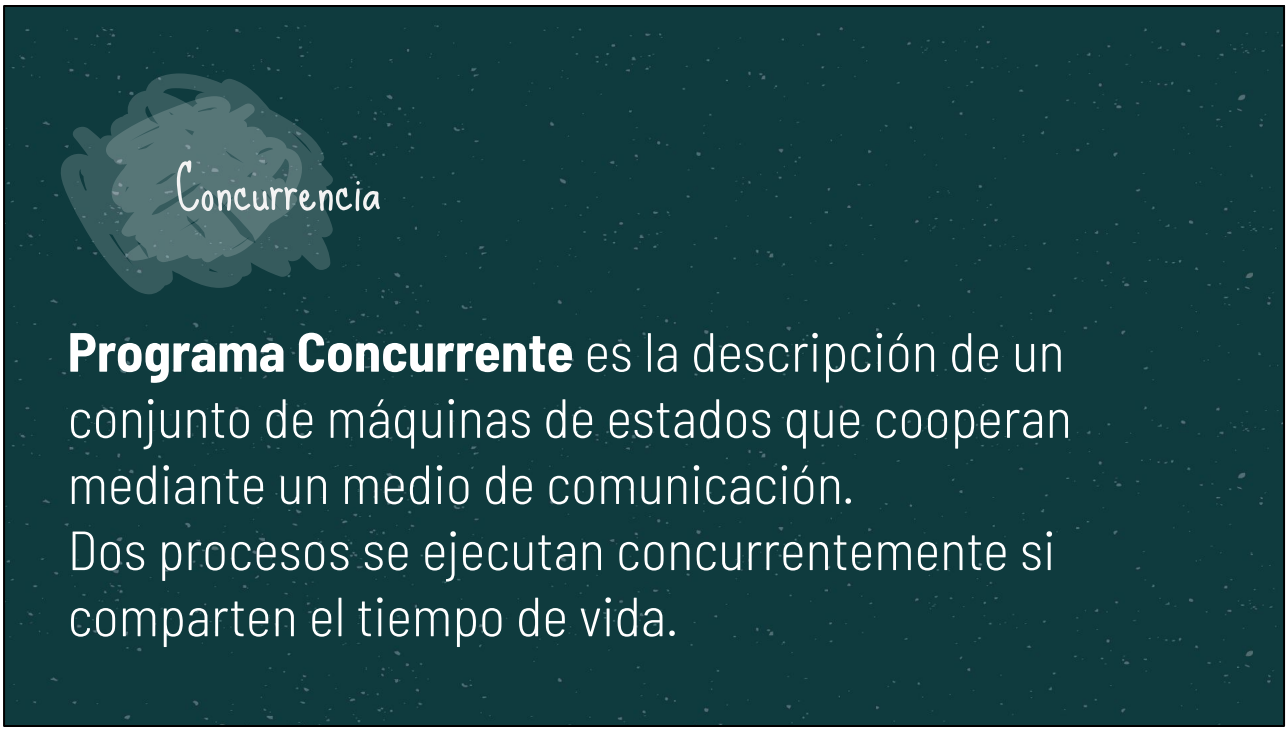


+ Nuevo: Se solicitó al SO la creación de un proceso cuyo recursos y estructuras están siendo creadas
+ Listo: Está listo para iniciar la ejecución pero el SO no le asignó ningún procesador
+ En Ejecución: El proceso está siendo ejecutado en éste momento.
+ Bloqueado: En espera de un evento para poder 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.

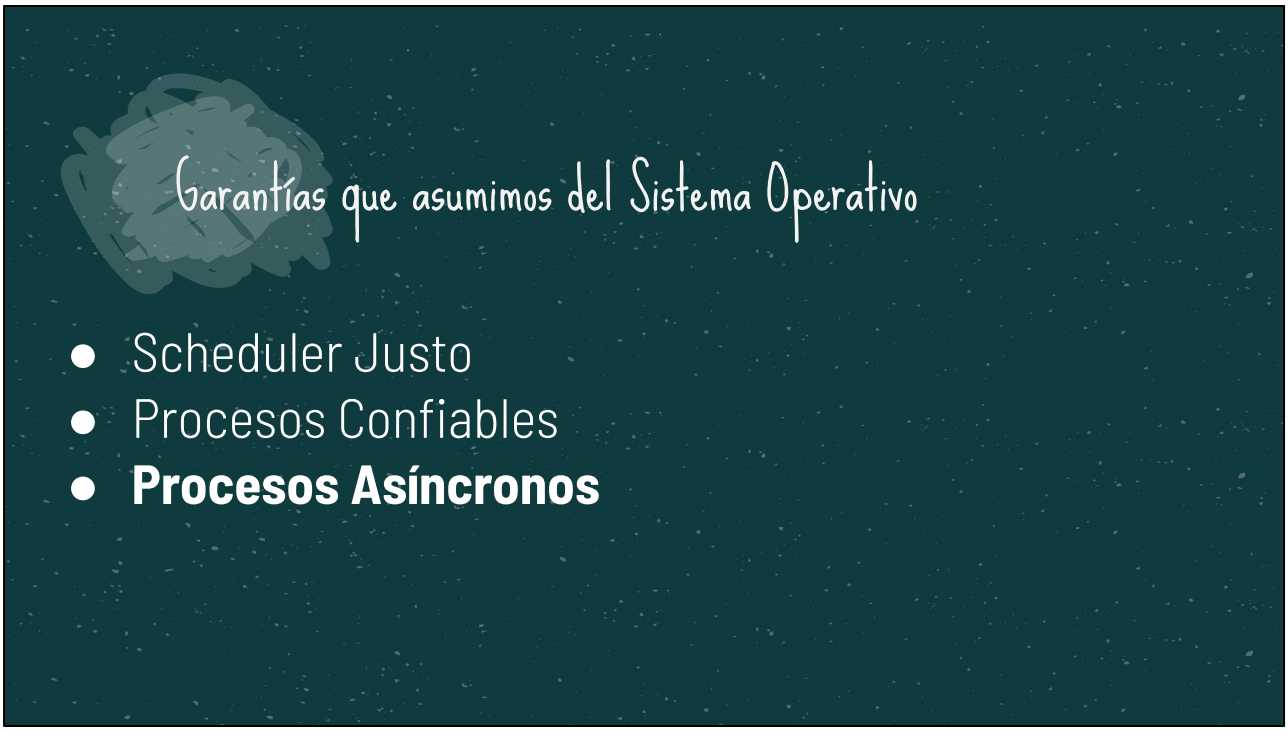


El sistema operativo lleva información del estado de ejecución de cada proceso:

+ 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 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.

Descanso: 5 mins

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.

UNISTD.h

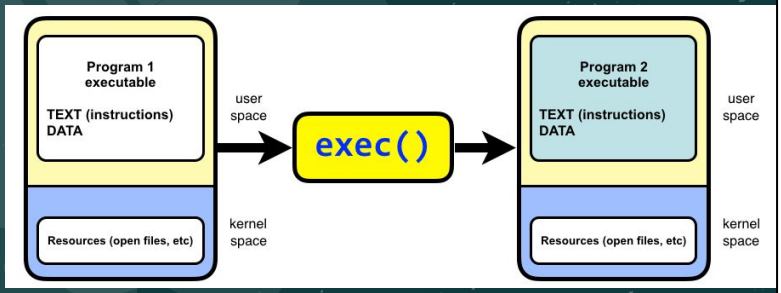
El archivo de cabecera **unistd.h** nos ofrece un punto de acceso a la *Portable Operating System Interface* (POSIX) del sistema operativo

- close / open
- read / write
- **exec / fork / getpid**
- select / pipe

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.

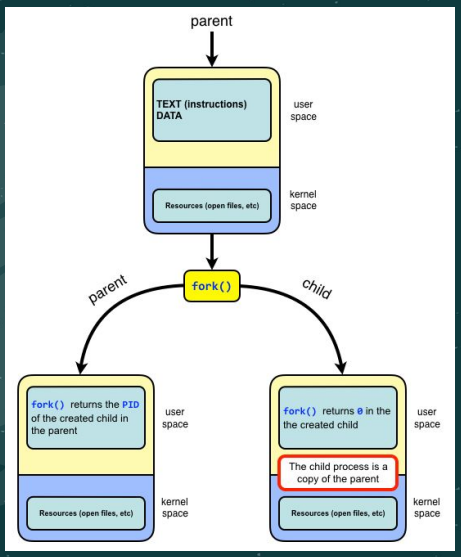
EJEMPLO!

Proceso que invoca a otro mientras mostramos los PIDs!

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 sabe quien 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!!

EJEMPLO!

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

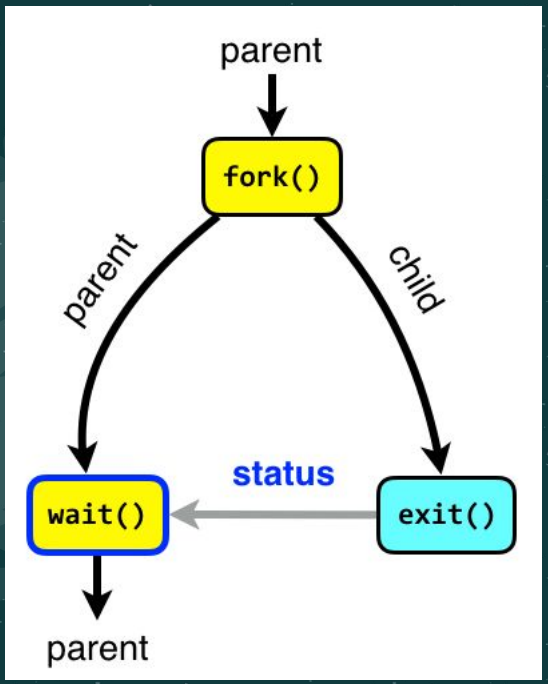
Ejercicio!

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!