Procesos

Eduardo C. Garrido Merchán

Sistemas Operativos. Práctica 1. Semana 2.

Definición

- Un proceso es un programa en ejecución. Solo existe cuando se está ejecutando en el sistema operativo.
- Intuitivamente: Un proceso nace cuando se ejecuta un programa. Si un programa se ejecuta k veces, nacerán k procesos distintos. Residirán en el Sistema Operativo tantos procesos como ejecuciones activas de programas.
- El sistema operativo posee una estructura de datos que contiene información de todos los procesos en cada instante.
- Todo proceso (PID) dispone de un padre (PPID) y puede tener hijos, cada proceso tiene un propietario (UID).
- Ejemplo: Podemos consultar los procesos ejecutando el comando ps -ef.

Definición

- Un proceso es un programa en ejecución. Solo existe cuando se está ejecutando en el sistema operativo.
- Intuitivamente: Un proceso nace cuando se ejecuta un programa. Si un programa se ejecuta k veces, nacerán k procesos distintos. Residirán en el Sistema Operativo tantos procesos como ejecuciones activas de programas.
- El sistema operativo posee una estructura de datos que contiene información de todos los procesos en cada instante.
- Todo proceso (PID) dispone de un padre (PPID) y puede tener hijos, cada proceso tiene un propietario (UID).
- Ejemplo: Podemos consultar los procesos ejecutando el comando ps -ef.

Definición

- Un proceso es un programa en ejecución. Solo existe cuando se está ejecutando en el sistema operativo.
- Intuitivamente: Un proceso nace cuando se ejecuta un programa. Si un programa se ejecuta k veces, nacerán k procesos distintos. Residirán en el Sistema Operativo tantos procesos como ejecuciones activas de programas.
- El sistema operativo posee una estructura de datos que contiene información de todos los procesos en cada instante.
- Todo proceso (PID) dispone de un padre (PPID) y puede tener hijos, cada proceso tiene un propietario (UID).
- Ejemplo: Podemos consultar los procesos ejecutando el comando ps -ef.

Definición

- Un proceso es un programa en ejecución. Solo existe cuando se está ejecutando en el sistema operativo.
- Intuitivamente: Un proceso nace cuando se ejecuta un programa. Si un programa se ejecuta k veces, nacerán k procesos distintos. Residirán en el Sistema Operativo tantos procesos como ejecuciones activas de programas.
- El sistema operativo posee una estructura de datos que contiene información de todos los procesos en cada instante.
- Todo proceso (PID) dispone de un padre (PPID) y puede tener hijos, cada proceso tiene un propietario (UID).
- Ejemplo: Podemos consultar los procesos ejecutando el comando ps -ef.

Definición

- Un proceso es un programa en ejecución. Solo existe cuando se está ejecutando en el sistema operativo.
- Intuitivamente: Un proceso nace cuando se ejecuta un programa. Si un programa se ejecuta k veces, nacerán k procesos distintos. Residirán en el Sistema Operativo tantos procesos como ejecuciones activas de programas.
- El sistema operativo posee una estructura de datos que contiene información de todos los procesos en cada instante.
- Todo proceso (PID) dispone de un padre (PPID) y puede tener hijos, cada proceso tiene un propietario (UID).
- Ejemplo: Podemos consultar los procesos ejecutando el comando ps -ef.

Ejecución de procesos hijos

- Cada proceso padre puede lanzar los procesos hijo que desee. Los procesos hijos serán clones del padre, heredando los recursos, pero sin compartir memoria.
- Lanzamos un proceso hijo desde el padre mediante **fork()**. El flujo de ejecución se "clona" en ese instante, teniendo dos procesos en paralelo.
- En el código, podemos diferenciar al padre de los hijos mediante el PID. De esta forma, aunque sean idénticos en código el padre e hijos, podemos hacer que ejecuten código distinto, paralelizando tareas.
- Los procesos padre deben esperar a que los hijos terminen. Esto se puede conseguir mediante **wait()** y **waitpid()**. En el hijo, se finaliza con **exit()**. El hijo queda en estado zombie si el padre no obtiene su resultado.

Ejecución de procesos hijos

- Cada proceso padre puede lanzar los procesos hijo que desee. Los procesos hijos serán clones del padre, heredando los recursos, pero sin compartir memoria.
- Lanzamos un proceso hijo desde el padre mediante **fork()**. El flujo de ejecución se "clona" en ese instante, teniendo dos procesos en paralelo.
- En el código, podemos diferenciar al padre de los hijos mediante el PID. De esta forma, aunque sean idénticos en código el padre e hijos, podemos hacer que ejecuten código distinto, paralelizando tareas.
- Los procesos padre deben esperar a que los hijos terminen. Esto se puede conseguir mediante **wait()** y **waitpid()**. En el hijo, se finaliza con **exit()**. El hijo queda en estado zombie si el padre no obtiene su resultado.

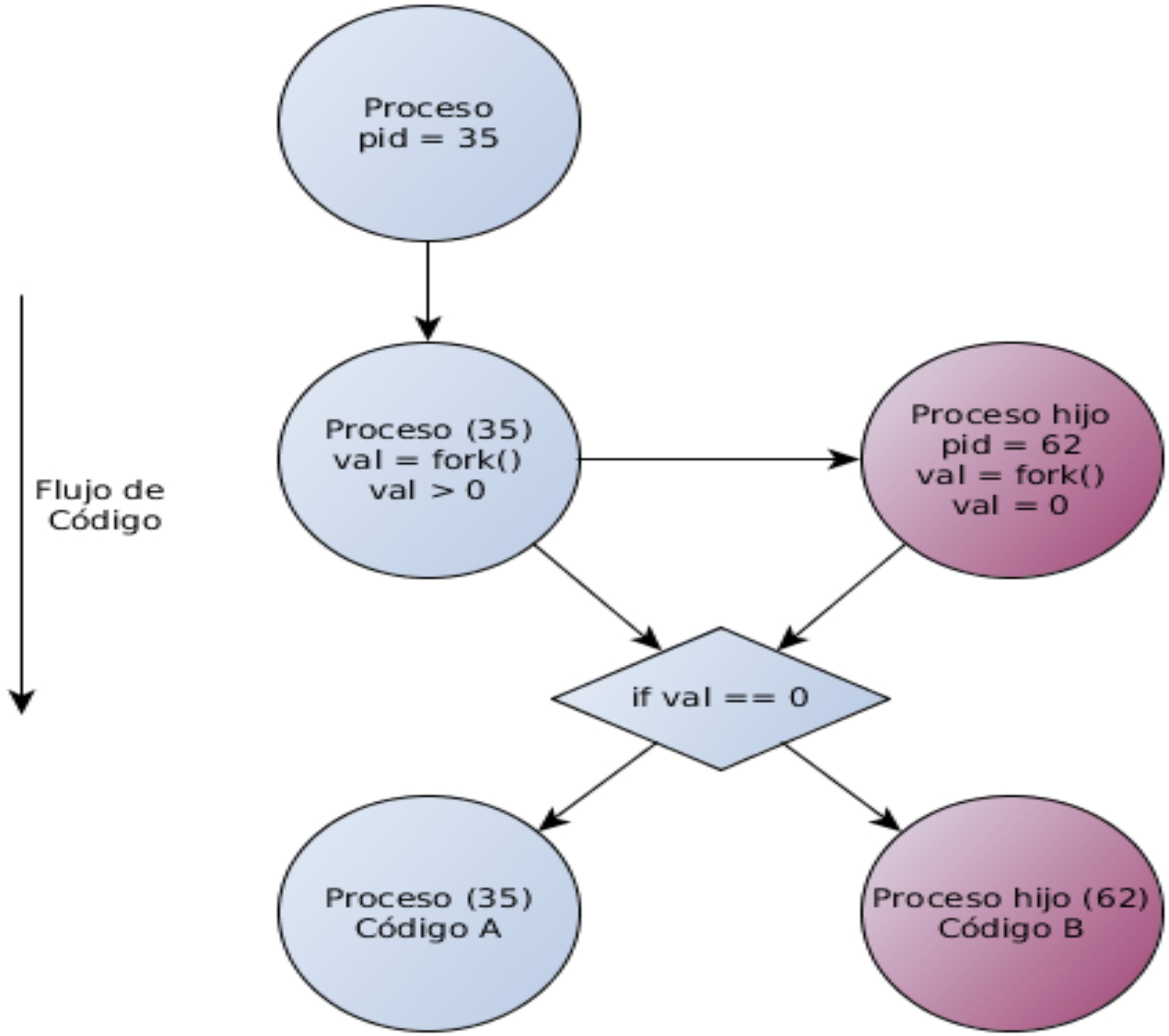
Ejecución de procesos hijos

- Cada proceso padre puede lanzar los procesos hijo que desee. Los procesos hijos serán clones del padre, heredando los recursos, pero sin compartir memoria.
- Lanzamos un proceso hijo desde el padre mediante **fork()**. El flujo de ejecución se "clona" en ese instante, teniendo dos procesos en paralelo.
- En el código, podemos diferenciar al padre de los hijos mediante el PID. De esta forma, aunque sean idénticos en código el padre e hijos, podemos hacer que ejecuten código distinto, paralelizando tareas.
- Los procesos padre deben esperar a que los hijos terminen. Esto se puede conseguir mediante **wait()** y **waitpid()**. En el hijo, se finaliza con **exit()**. El hijo queda en estado zombie si el padre no obtiene su resultado.

Ejecución de procesos hijos

- Cada proceso padre puede lanzar los procesos hijo que desee. Los procesos hijos serán clones del padre, heredando los recursos, pero sin compartir memoria.
- Lanzamos un proceso hijo desde el padre mediante **fork()**. El flujo de ejecución se "clona" en ese instante, teniendo dos procesos en paralelo.
- En el código, podemos diferenciar al padre de los hijos mediante el PID. De esta forma, aunque sean idénticos en código el padre e hijos, podemos hacer que ejecuten código distinto, paralelizando tareas.
- Los procesos padre deben esperar a que los hijos terminen. Esto se puede conseguir mediante **wait()** y **waitpid()**. En el hijo, se finaliza con **exit()**. El hijo queda en estado zombie si el padre no obtiene su resultado.

Redirección de código para procesos padre-hijo



Ejercicios

- Para consultar la jerarquía global de procesos: pstree 0 | nl | less.
- Idea intuitiva para los ejercicios: Clonar procesos diferenciándolos y referenciándolos mediante **getpid()** y **getppid()**.
- Realizar ejercicios 2,3,4,5 y 6 de los apuntes de prácticas.
- Ejercicio 3: http://pubs.opengroup.org/onlinepubs/9699919799/
- Ejercicio 4: Usar *srand(getpid());sleep(rand()%n);* entre procesos para dormirlos tiempos aleatorios.

Ejercicios

- Para consultar la jerarquía global de procesos: pstree 0 | nl | less.
- Idea intuitiva para los ejercicios: Clonar procesos diferenciándolos y referenciándolos mediante **getpid()** y **getppid()**.
- Realizar ejercicios 2,3,4,5 y 6 de los apuntes de prácticas.
- Ejercicio 3: http://pubs.opengroup.org/onlinepubs/9699919799/
- Ejercicio 4: Usar *srand(getpid());sleep(rand()%n);* entre procesos para dormirlos tiempos aleatorios.

Ejercicios

- Para consultar la jerarquía global de procesos: pstree 0 | nl | less.
- Idea intuitiva para los ejercicios: Clonar procesos diferenciándolos y referenciándolos mediante **getpid()** y **getppid()**.
- Realizar ejercicios 2,3,4,5 y 6 de los apuntes de prácticas.
- Ejercicio 3: http://pubs.opengroup.org/onlinepubs/9699919799/
- Ejercicio 4: Usar *srand(getpid());sleep(rand()%n);* entre procesos para dormirlos tiempos aleatorios.