Programación Sobre Redes

T5: Procesos

Nicolás Mastropasqua

April 28, 2020

Instituto Industrial Luis A. Huergo

Contenidos

1. Repaso

2. Procesos

3. Creación de Procesos

4. Terminación de un proceso

Repaso

Previamente...

- Procesos vs Programas
- Estados de un proceso
- Context Switch
- Representación de procesos: PCBs

Procesos

¿Qué puede hacer un proceso?

- Terminar $\rightarrow$ exit()
- Lanzar un proceso hijo $\rightarrow$ fork()
- Ejecutar en la CPU
- Realizar E/S

Creación de Procesos

Creación de procesos

- Un proceso puede crear varios procesos hijos.
- De esta forma, se obtiene una estructura en forma de árbol

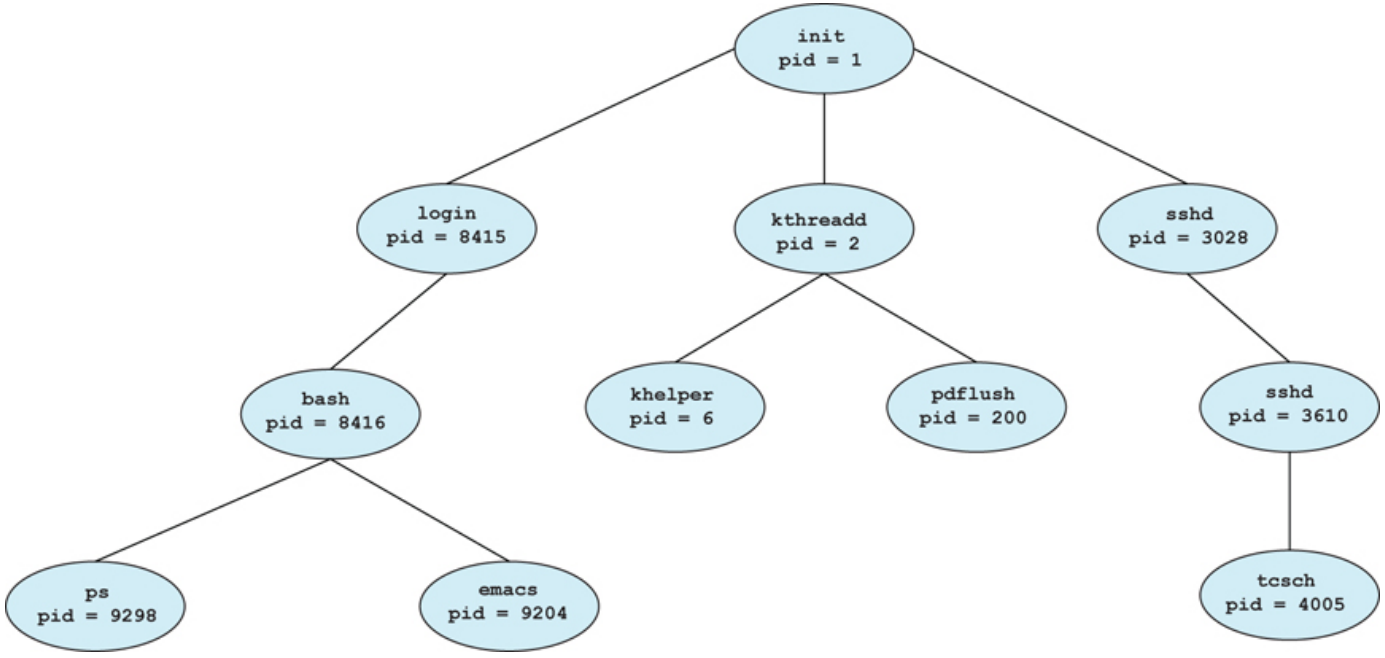


Figure 1: Ejemplo de un árbol de procesos en UNIX

Sugerencia: Jugar con **pstree** en linux y **ps** en linux. Pueden ver el siguiente link para el anterior

Relación padre-hijo

Relación padre-hijo

- El proceso padre puede ejecutar concurrentemente con el hijo o bien esperar a que alguno de sus hijos termine
- El hijo es creado como un clon del padre (corren el mismo programa)

¿Qué pasa cuando ejecuto ”ls” en consola?

- El shell hace un fork() y se ”clona”
- El hijo hace un exec() para correr el programa ls

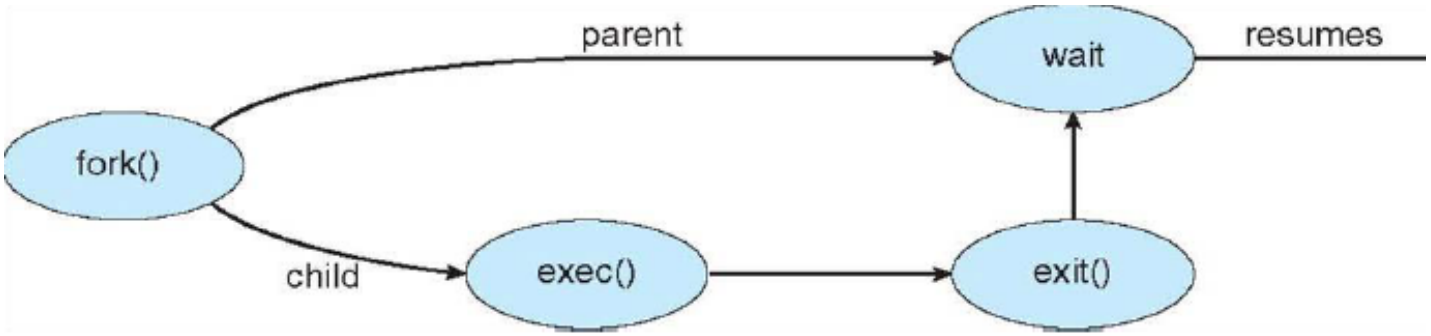


Figure 2: En este ejemplo, se ve que el padre no ejecuta concurrentemente con el hijo, sino que lo espera

Esquema general para crear un hijo

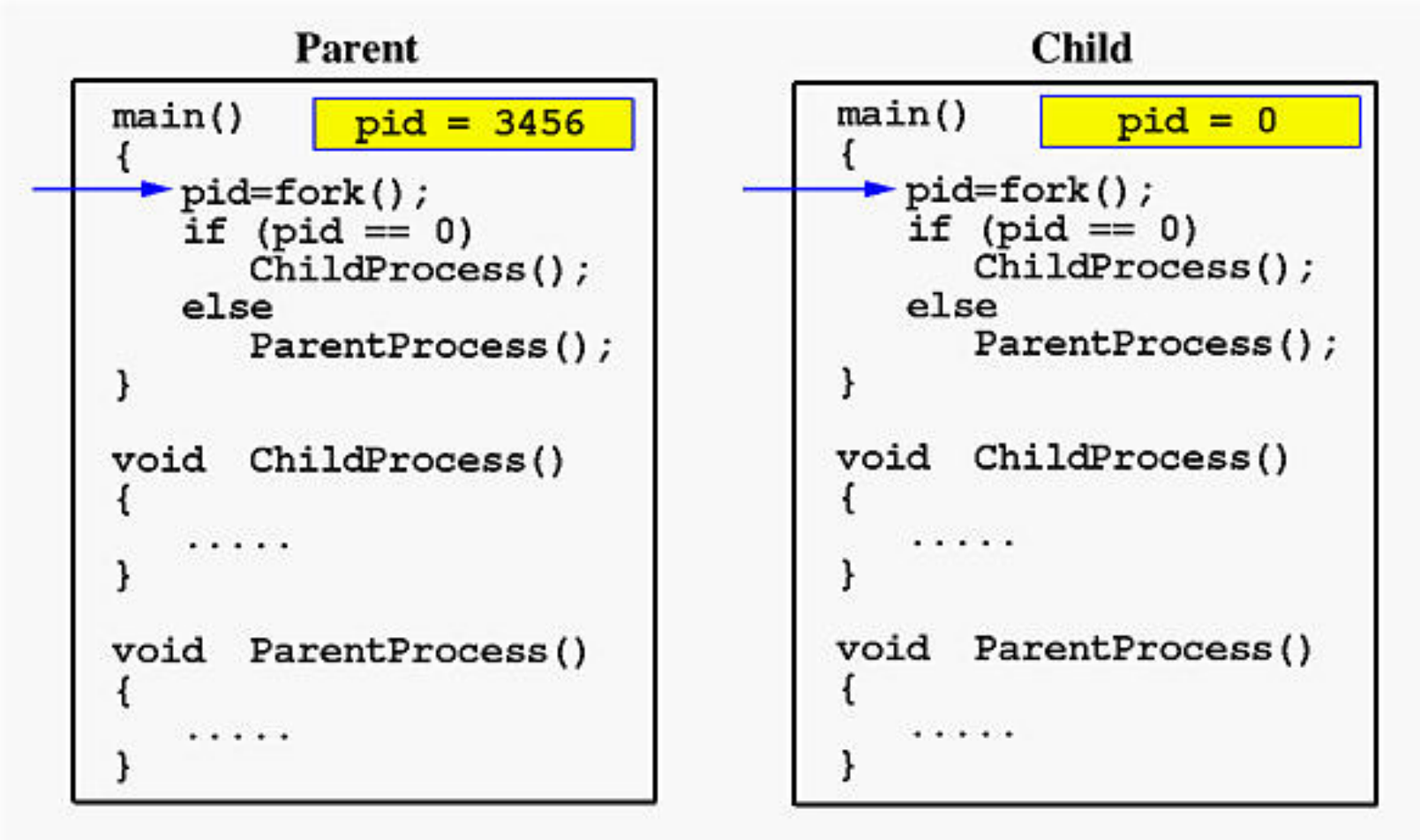


Figure 3: Esquema general para crear un proceso hijo y controlar su ejecución

¿Cómo hago el ejemplo en C?

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) {/* error occurred */
       fprintf(stderr, "Fork Failed");
       exit(-1);
    }
    else if (pid == 0) {/* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else {/* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
       exit(0);
    }
}
```

Figure 4: Ejemplo en C, válido para un sistema UNIX

Analizando el ejemplo

- fork crea un proceso que es una copia del espacio de memoria del original.
- El cpu puede continuar en poder del proceso padre o bien, puede ser adquirido por su hijo.
- Tener en cuenta que la función fork devuelve un entero. Para el caso del padre, dicho entero será el PID del hijo. En cambio, el hijo recibe el valor cero. ¡La variable pid de cada proceso es distinta!
- Eventualmente, si no hubo errores, el hijo entrará en la rama del else if y sobreescribirá su memoria al cargar /bin/ls (utilizando execlp)
- De esta forma, cada proceso sigue distintos caminos. El padre ejecutará la rama del else y al hacer wait(), será desalojado hasta que el hijo termine.
- Cuando el hijo termina, el padre ejecuta exit para finalizar su ejecución

Para pensar...

¿Cuántos procesos se crean?

```
int main()
{
    fork();
    fork();
}
```

¿ Y si agrego un fork más?

Fork bomb

```
while(1) fork();
```

Figure 5: ¿Hacemos una demo? ;)

Ejercicio

Determinar la salida del siguiente programa (Unix)

```
int value = 5;
int main(){
    int pid;
    pid = fork();
    if (pid == 0){
            value += 15;
            exit();
    }
    elseif(pid > 0){
            wait(NULL);
            printf("value = %d", value);
            exit();
    }
    return 0;
}
```

Terminación de un proceso

Terminación de proceso

- El proceso puede ejecutar exit() para que el sistema operativo libere los recursos asociados al mismo. Además, retorna un valor al procesos padre. ¿Qué recursos son liberados?
- También podría pasar que otro proceso, el padre por ejemplo, decida **matar** a algún hijo

Figure 6: La situación se pone un poco extraña...

Proceso huérfano

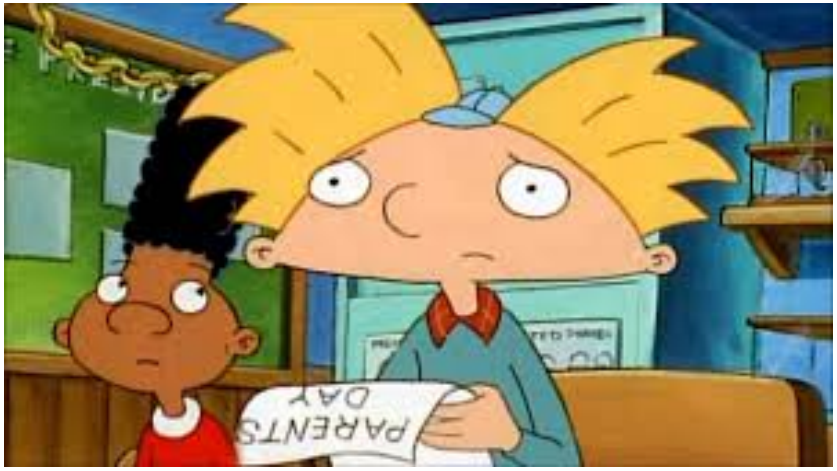
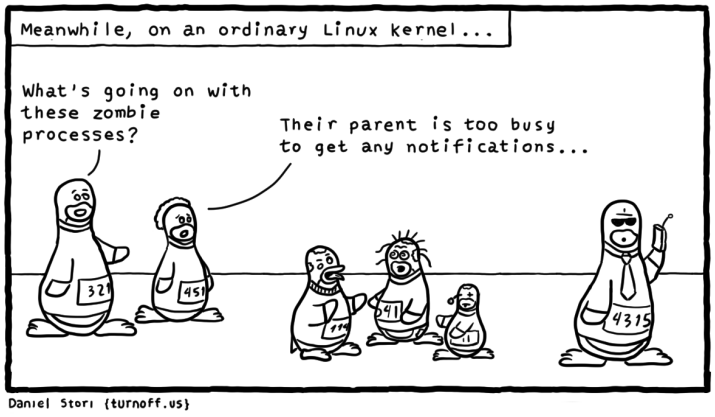


Figure 7: Aunque ¿es el caso?

- Típicamente, en UNIX, si el padre estaba haciendo wait() y uno de sus hijos termina, entonces wait devuelve el pid del proceso que finalizó.
- Si el padre muere, entonces los hijos son adoptados por el proceso init (UNIX)

Proceso Zombie



- Cuando un hijo hace **exit**, los recursos asociados son liberados y pasa a estado **terminado**. Se vuelve un zombie porque aún mantiene un PID y está en la tabla global de procesos.
- Si el padre lo estaba esperando, entonces recibe la notifiación de exit, y **recolecta su alma** eliminando correctamente la entrada de su PID
- Si no, el proceso permanecerá como zombie produciendo leak de recursos

¿Estamos perdidos?

¿Cómo podríamos cuidar nuestro sistema operativo de procesos que tiendan a spawnear zombies?

Figure 8: En UNIX, se adoptan distintas medidas para controla la proliferación de zombies