Tema 4: Padre e hijo

Enrique Soriano

Laboratorio de Sistemas,
Grupo de Sistemas y Comunicaciones,
URJC

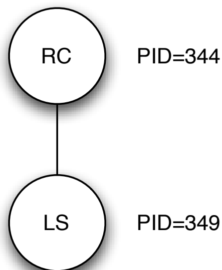
11 de marzo de 2010

(cc) 2010 Grupo de Sistemas y Comunicaciones.

Algunos derechos reservados. Este trabajo se entrega bajo la licencia Creative Commons Reconocimiento - NoComercial - SinObraDerivada (by-nc-nd). Para obtener la licencia completa, véase http://creativecommons.org/licenses/by-sa/2.1/es. También puede solicitarse a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Hasta ahora...

- ... sólo hemos creado procesos con el Shell.
  P. ej.
  `term% ls`



Llamadas al sistema

- Una para crear un proceso (fork) y otra para ejecutar un programa (exec).
- Podemos crear un nuevo flujo de control, para ejecutar el mismo programa o para ejecutar otro.
- Podemos configurar el nuevo proceso antes de que ejecute otro programa.
- Se podría hacer todo en una función, pero tendría demasiados parámetros.

Fork

- `int fork(void);`
- Se crea un proceso con nuevo PID.
- En el padre, retorna el PID del hijo.
- En el hijo, retorna 0.

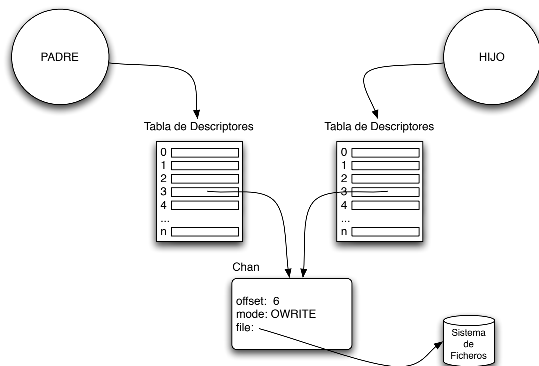
ver fork0.c

- El hijo es un clon exacto del padre.
- Padre e hijo ejecutan concurrentemente.
- El hijo es totalmente independiente del padre: abstracción de proceso.
- Se *"copia"* la memoria: TEXT, DATA, STACK. ¿Qué valor tienen las variables?

ver fork1.c , fork2.c, dos.c

Fork: compartiendo recursos

- La tabla de FDs se copia, pero los Chans abiertos se comparten.
- Los ficheros abiertos después del fork no estarán el la tabla del otro.



Condiciones de carrera

- No se sabe el orden en el que se van a ejecutar las instrucciones de los dos procesos.
- El resultado final depende de la carrera entre los dos procesos: condición de carrera.
- Programación concurrente: evitar condiciones de carrera.

Exec

- `int exec(char *name, char *argv[]);`
- `name` es la ruta del fichero ejecutable que queremos cargar.
- `argv[]` es un array de strings con los argumentos para el programa.
  Tiene que terminar en nil.
- ¡No se puede usar sintaxis del shell!
  P. ej, "$home" como argumento, "*" etc...

ver exec.c

Execl

- `int execl(char *name, ...);`
- Ideal para cuando se saben los argumentos de antemano.
- El último argumento tiene que ser nil.

ver execl.c

Fork + exec

- Las dos funciones juntas nos permiten crear procesos para cualquier programa.
- Desde `init`, todos los procesos se crean así.

ver forkexec.c

Wait

- `Waitmsg *wait(void);`
- Retorna cada vez que un hijo muere.
- Devuelve una estructura Waitmsg reservada con malloc.
  - pid: id del proceso.
  - time: ms usuario, sistema y real del proceso y descendientes.
  - msg: estatus del proceso.
- Si no hay hijos por los que esperar, devuelve nil.
- Si sólo nos interesa el pid, hay otra función: `int waitpid(void);`.

ver wait.c

En el shell

- `comando &` : el comando se ejecuta de forma asíncrona, el shell no le espera.
- `$pid` : el pid del shell.
- `$apid` : el pid del último procesos ejecutado en background.

Programas interpretados

- Son programas que son interpretados por otros programas.
  P. ej. scripts de shell.
- La primera línea del fichero tiene que indicar el programa intérprete.
  P. ej.:
  `#!/bin/rc`

Programas interpretados

- Son ejecutables, se pueden ejecutar con `exec`.
- Cuando Plan 9 ejecuta un fichero de este tipo, ejecuta el programa indicado en esta línea, pasándole el programa interpretado.

ver cd.rc, add.hoc