Ampliación de Sistemas Operativos

PRÁCTICA Shell

A. Estructura y gestión de procesos

Un proceso se define mediante “un espacio de direcciones, sobre el que se ejecutan uno o más threads, y los recursos del sistema necesarios para la ejecución de los mismos”.

Simplificando, consideraremos de momento un proceso como un programa en ejecución caracterizado por su imagen de memoria y su contexto. Éste último se define en LINUX mediante una estructura denominada *task_struct,* que contienen toda la información necesaria para su gestión: identificador de usuario (UID), identificador de grupo (GID), tabla local de descriptores (LDT), etc. Aunque los detalles de implementación de esta estructura suelen estar ocultos al usuario, la declaración de `task_struct` puede ser consultada en el fichero `<linux/sched.h>`[1].

En UNIX un proceso se crea mediante la llamada `fork()` que duplica casi todo el proceso invocante y establece una relación padre-hijo entre el proceso creador y el creado. La distinción inicial entre padre e hijo se basa en el resultado devuelto por la llamada f`ork()`: en el padre, devuelve el PID del hijo; en el hijo, devuelve el valor 0.

```
SINOPSIS
#include <unistd.h>

pid_t fork(void);
```

Todos los procesos tienen un único proceso padre, pero pueden tener varios procesos hijos. El núcleo identifica cada proceso mediante su PID (*process ID*), que es un número asociado a cada proceso y que no cambia durante el tiempo de vida de este. El proceso *0* en los sistemas tipo Unix tradicionales es especial, se crea durante el arranque del sistema, es el único que no tiene proceso padre y se encarga de la gestión de la memoria virtual (swapper). El proceso *1* (denominado tradicionalmente como el proceso *init*) es el encargado de arrancar los demás procesos del sistema (la página de manual `init/systemd` describe los conceptos relevantes relacionados con el arranque del sistem). El árbol de procesos puede visualizarse mediante el comando `pstree`.

Un proceso que está ejecutando el código de un programa puede pasar a ejecutar el código de otro programa diferente, invocando para ello la llamada `execve()`. Existen varias funciones de la librería de C que sirven de interfaz con esta llamada al sistema y que se distinguen por el modo en que especifican sus argumentos (consultar `man 3 exec`): `execl()`, `execlp()`, `execle()`, `execv()`, `execvp()`.

```
SINOPSIS
#include <unistd.h>
int execve (const char *filename, const char *argv [],
            const char *envp[]);
```

El parámetro `filename` especifica el nombre del programa a ejecutar. Puede tratarse de un fichero ejecutable o de un guión shell (*shell-script*). En éste último caso, el fichero debe comenzar por una línea de la forma "`#! intérprete [arg]`"

1 `https://github.com/torvalds/linux/blob/master/include/linux/sched.h`

y la llamada `execve()` se correspondería con la ejecución del la línea de comando "`intérprete [arg] filename`".

Los parámetros `argv` y `envp` representan respectivamente la *lista de argumentos del programa* y la *lista de variables de entorno.* En ambos casos, se trata de arrays de cadenas de caracteres (`char *`) cuyo último elemento debe ser un puntero nulo para indicar el final del array. El propósito de estos parámetros es alterar el comportamiento inicial del programa.

En lenguaje C, ambas listas se pueden acceder mediante los parámetros de la función principal `main()`. Las variables de entorno pueden accederse también mediante la variable global `environ` (consultar `man environ`).

Funciones y comandos relacionados con la manipulación de los argumentos del programa y las variables de entorno:

- `getopt()`, `getopt_long()` – facilitan el análisis de los argumentos de la línea de ordenes.
- `getenv()`, `putenv()`, `setenv()`, `unsetenv()` – permiten consultar, modificar y añadir variables de entorno.
- `printenv`, `env` – comandos que permiten consultar el entorno actual y ejecutar un programa con un entorno modificado.

Cuando un proceso finaliza, invoca explícita o implícitamente a la llamada `_exit()` que entrega al proceso padre un código que expone el motivo (éxito o error) de la terminación. Normalmente se utiliza a través de la función `exit()` de la librería estándar.

```
SINOPSIS
#include <stdlib.h>

void exit (int status);
```

Mientras el proceso padre no pueda recoger esa información, el hijo ya acabado estará en estado *zombie*. Un proceso padre puede hacer una pausa en su ejecución hasta ser informado de la terminación de un proceso ejecutando las llamada `wait()` o `waitpid()`.

```
SINOPSIS
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options);
```

Para interpretar el significado de la variable `status` se puede hacer uso de las macros definidas en <sys/wait.h> (consultar `man 2 wait` para más información): *WIFEXITED()*, *WEXITSTATUS()*, etc.

En LINUX se puede consultar información de los procesos en ejecución a través del sistema de ficheros virtual `/proc` (consultar `man proc`) y/o mediante los siguientes comandos:

- `ps` – informa del estado de los procesos.
- `pgrep` – busca procesos basándose en el nombre del programa u otros criterios.
- `top` – muestra un listado de tareas con actualización continua.
- `pstree` – muestra el árbol de procesos.

Hay varias llamadas que permiten acceder a parte de la información del contexto de un

proceso:

- PIDs: `getpid()`, `getppid()`, `getpgid()`, `getpgrp()`
- UIDs: `getuid()`, `geteuid()`, `getgid()`, `getegid()`
- Directorio actual: `getcwd()`, `chdir()`
- Límites de recursos: `getrlimit()`, `getrusage()`, `setrlimit()`

En UNIX, normalmente, un proceso forma parte de un grupo de procesos, y varios grupos de procesos forman parte de una sesión. Cada sesión suele tener asociado un terminal de control (TC) que es manejado cada vez por uno de los grupos de procesos, del que se dice que se está ejecutando en *foreground*; los demás se ejecutan en *background*. Las siguientes llamadas al sistema y funciones de biblioteca permiten su manipulación:

- `setpgid()`, `setpgrp()` – permiten formar un grupo de procesos.
- `tcsetpgrp()` – permite modificar el grupo de procesos que maneja el TC.

El usuario real de un proceso es normalmente quien ordena ejecutarlo, pero el usuario efectivo puede modificarse:

- Mediante la ejecución de un fichero que tenga activado el bit *SET-UID* (`man 2 chmod` para más información). Al cargarlo el usuario efectivo del proceso pasa a ser el propietario del fichero.
- Mediante las llamadas `setuid()` y `setreuid()`.

B. Shell

Un *shell* es un intérprete de órdenes, parte de las cuales ejecuta como rutinas internas. Para las que no reconoce como internas, busca un fichero ejecutable que coincida con su nombre y lo ejecuta como un proceso hijo. Los directorios y el orden de búsqueda vienen determinados por la variable de entorno PATH. Para ejecutar un programa que no se encuentre en dichos directorios es necesario especificar la ruta completa o relativa al fichero ejecutable.

La sintaxis de las líneas de órdenes dadas a un *shell* tiene las siguientes características:

- Un *trabajo* o *job* (grupo de procesos) está constituido por una sola orden o varias ordenes conectadas mediante cauces. En este último caso, se emplea el carácter "|" para separarlas y la salida de una orden se convierte en la entrada de la siguiente.
- Se pueden secuenciar varios *jobs* en una misma línea separándolos con ';'.
- Los *jobs* pueden ejecutarse en *foreground* o en *background* (empleando el carácter '`&`' al final de la orden en este caso).
- Preprocesar algunos caracteres especiales (p.ej. '\' como escape, '*' '?' como comodines, '$' como referencia a variables, ' " '...)

C. Tiempos

Por lo general, las dos principales medidas de tiempo son:

- Fecha o *tiempo real*: tiempo transcurrido desde el inicio de los tiempos o *epoch*, que la mayor parte de los UNIX sitúan en el 1 de Enero de 1970 a las 00:00:00 GMT. Se obtiene mediante las llamadas `time()` y `gettimeofday()` y se puede dar formato mediante las funciones: `localtime()`, `gmtime()`, `asctime()`, `ctime()`, `mktime()`, `strftime()`, `strptime()`.

```
SINOPSIS
#include <time.h>

 time_t time(time_t *t);
```

```
SINOPSIS
#include <sys/time.h>
#include <unistd.h>

int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(const struct timeval *tv,const struct timezone
*tz);
```

El parámetro `timezone` no se utiliza (puntero nulo).

- Tiempo de CPU: tiempo consumido por la ejecución de un programa (usuario y/o sistema). Se obtiene mediante las llamadas `times()` y `getrusage()`, o mediante la función `clock()`.

```
SINOPSIS
#include <systime.h>

clock_t times(struct tms *buf);
```

```
SINOPSIS
#include <time.h>

clock_t clock(void);
```

```
SINOPSIS
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

int getrusage (int who, struct rusage *usage);
```

El siguiente esquema ilustra la relación entre funciones, tipos de dato y tiempos. Los arcos representan llamadas al sistema o funciones; las cajas representan tipos de datos; y los puntos gruesos representan la fecha y el tiempo de CPU.

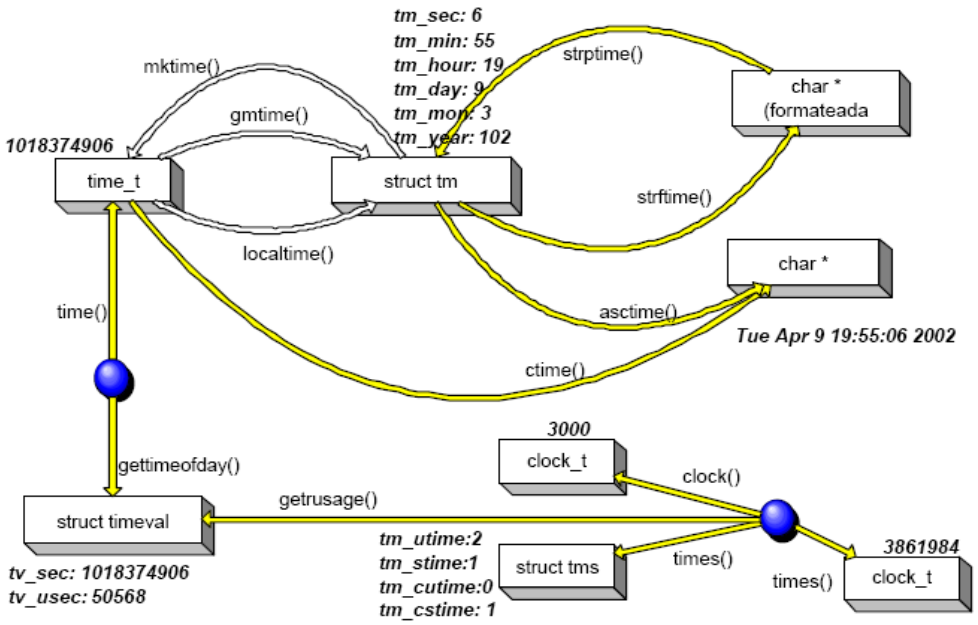

Fig. 1 Funciones y estructuras de datos de tiempo.

Otros comandos y funciones relacionados con la gestión de tiempos son:

- `date` – comando que muestra la fecha.
- `sleep(), usleep(), nanosleep()` – "duermen" el proceso durante un cierto tiempo especificado en segundos, microsegundos o nanosegundos.
- `sleep` – comando de usuario análogo a la función `sleep()`.
- `getitimer(), setitimer()` – obtiene/asigna el valor de un temporizador.

PRÁCTICA

Objetivo:

Programar un intérprete de órdenes sencillo que permita la ejecución de trabajos con un solo programa, sin cauces, ni redirección de la entrada/salida.

Descripción:

El programa debe soportar algunas de las características propias de los *shells que* detallaremos a continuación. Siempre que no se especifique lo contrario, el comportamiento será análogo al del *shell* por defecto (*bash*).

Sintaxis de línea de orden:

`orden argumentos [& | ;] orden argumentos [& | ;] ...`

Operación:

Las líneas de orden puede contener más de una orden, separadas mediante ";" o "&", en este último caso, la orden se ejecutará en *background*.

Como mínimo debe tener las siguientes *órdenes internas*:

- **`cd dir`** : cambia el directorio de trabajo a *dir*. Si se usa sin argumentos cambia al directorio raíz del usuario (variable de entorno `HOME`).
- **`pwd`** : muestra el directorio de trabajo actual.
- **`jobs`** : muestra la lista de trabajos (jobs), con el siguiente formato:

  `[jobID] estado línea_de_orden`

  dónde
  - `jobID` = struct `job.jobID`
  - `estado` = "Running" o "Stopped"
  - `línea_de_orden` = `struct job.texto`

- **`wait n`** : espera a que termine de ejecutarse el trabajo *n* . Si se usa sin ningún argumento, espera a que finalicen todos.
- **`kill n`** : elimina de ejecución el trabajo número *n.*
- **`stop n`** : detiene la ejecución del trabajo en *background* número *n* (*SIGSTOP*).
- **`fg n`** : pasa a *foreground* el trabajo número *n* (que debería estar detenido).
- **`bg n`** : pasa a *background* el trabajo número *n* (que debería estar detenido).
- **`times`** : (opcional) muestra el tiempo acumulado de usuario y de sistema.
- **`date`** : muestra la fecha actual.
- **`exit`** : finaliza la ejecución del programa de forma segura. Envía una señal *SIGKILL* a los trabajos en background y aguarda a que estos terminen antes salir.

Todo aquello que no sea una orden interna, se interpretará como un *comando externo* que es necesario ejecutar mediante `fork()` y `execvp()`.

Un trabajo en *foreground* puede ser detenido pulsando `CTRL+Z` desde el terminal de control, en cuyo caso el *shell* debe retomar el control. Para ello debe detectar mediante la llamada `waitpid()` que el trabajo por el que esperaba no ha acabado sino que ha sido detenido a causa de la señal *SIGTSTP*.

Para poder manipular el terminal de control sin que se pare nuestro *shell* es necesario ignorar la señal SIGTTOU.

Además, es necesario capturar la señal SIGINT (`CTRL+C` desde terminal) para evitar la finalización del shell como ocurre con bash.

La siguiente figura ilustra los tipos de datos que es necesario manejar, para llevar a cabo la implementación del *shell* .

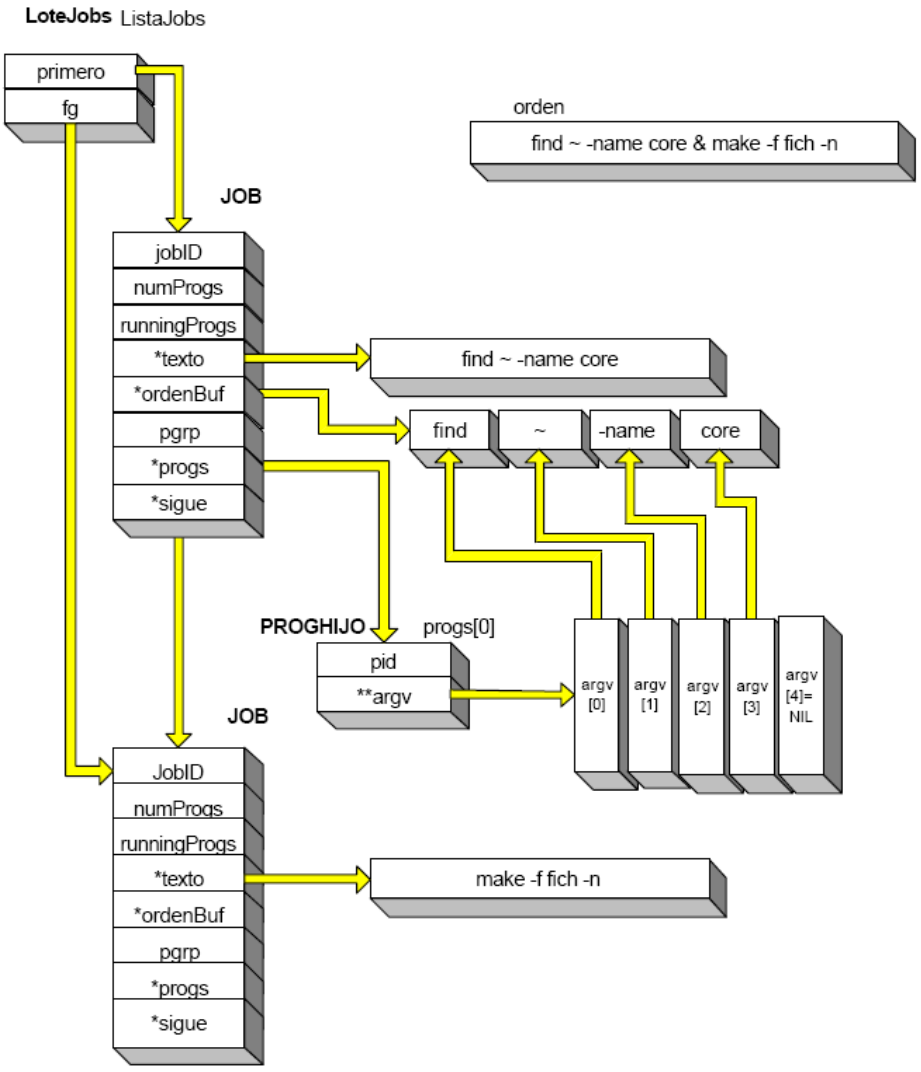

Fig. 2 Estructuras de datos manejadas por el shell.

El campo `jobID` de la estructura `job` se gestiona de una manera análoga al resto de los *shells*, cuando se inserta un nuevo trabajo:

- Si la lista esta vacia --> jobID = 1
- Si la lista no esta vacia --> jobID = jobID_último + 1

Cuando se ejecuta un trabajo en *background*, después de lanzarlo a ejecución es preciso informar del siguiente modo (PID representa el *pid* del proceso):

```
[jobID] PID
```

Para facilitar la implementación de la práctica se proponcionan los siguientes ficheros que podrán utilizarse como plantilla. Incluye un análisis de ordenes rudimentario básico.

`shell.h`	Definiciones de tipos de datos y prototipos de funciones
`shell.c`	Función principal (`main`)
`shell_orden.c`	Funciones de manejo de ordenes
`shell_jobs.c`	Funciones de manejo de jobs

La lectura de ordenes que se proporciona es muy básica y se puede mejorar haciendo uso de la función `readline()`. El fichero `shell-readline.txt` muestra el esquema general de shell usando `readline()` para la lectura de ordenes.