Programación de scripts

*09/04/2019*

**Compilar en C:** cc fuente.c → resultado = a.out

cc -o salida fuente.c → resultado = salida

**Makefile:** se crea un fichero Makefile

obligatiorio linea *all: <nombre/s del ejecutable>*

y para cada uno, sin importar el orden una entrada *nombreEjecutable:*

*y* a continuación, sus dependencias (ficheros .c)

abajo **TABULACION** y línea de compilación (cc fuente.c -o salida)

se guarda y compila con *makeall: espejo*

*all: espejo*

*espejo: espejo.c*

*cc espejo.c -o micat*

Solo se vuelve a compilar si el script ha sufrido algún cambio

**Librerias a usar:** ncurses.h

cc -o salida fuente.c -lm librerias

-lLIBRERIA (JUNTO)

**Leer a variable:**

read <posicion tabla canales 0,1,2..>, &<variable>, <tamaño en bytes sizeof(variable)>

→ devuelve el numero de bytes o caracteres que realmente ha leído

**Escribir a variable:**

write <1 (salida estandar>, &<variable>, <tamaño en bytes sizeof(variable)>

→ devuelve el numero de bytes o caracteres que ha conseguido escribir

*12/04/19*

**Hilos:** comparten la zona de memoria entre ellos, variable global en C (declarada fuera del main)

*Hacer misma practica con procesos*

#include <pthread.h>

4 parámetros p\_thread\_create(pthread\_t +:

variable declarada de tipo pthread\_t

atributos del hilo: tipo pthread\_attr\_t → NULL

start\_routine: el nombre de la función donde se encuentra ese hilo

arg: parametros que recibe el hilo → NULL

pthread\_join(): se queda esperando el proceso principal hasta que acabe el hilo, para comunicarse, le pasa al principal como parametro l lo que devuelve el hilo

pthread\_detach(): ignora al hilo para seguir el proceso principal

Create, join y detach lo hace el principal. El hilo hac ella funcion exit

23/04/19

**Creación de procesos**

*fork()*: llamada al sistema que crea un nuevo proceso idéntico a sí mismo (ejecutándose a partir del fork, no desde el principio del código). No recibe parámetros. Hereda todo del original, copia el valor de las variables del proceso original, pero no las comparten. Devuelve al original el PID del proceso creado, y a la copia se le devuelve 0, o -1 si error

Para operar en uno u otro usaremos condicionales:

*int x = fork();*

*if (x==0)* → código de la copia

*else →* código del original

*execl (ruta y nombre del ejecutable, nombre del ejecutable, parámetros que le queremos pasar)*

Usaremos un 0 en los parámetros → *execl(“pr32”, ”pr32”, 0)*

Cuando hace la llamada, el padre muere. Usaremos un fork() para que la copia llame al execl

El execl a diferencia del fork, no hereda toda la tabla de canales, solo las 3 primeras posiciones.

Debajo del execl no hay código, solo un mensaje de error.

Si lo lanza una copia con fork, hereda como PID el de la copia.

Importante almacenar el PID al crear el fork

*(Mirar apuntes de manejo de archivos tema 7)*

*Close (posición) → cierra la posicion de la tabla de canales*

*Open(fichero, modo de abrirlo) →* modifica la salida estándar

Se cierra la salida estandar close(1) y el open modifica la salida estandar.

26/04/19

**Sincronización padre-hijo**

*Wait(int \* status):*Llamada al sistema que ejecuta el padre, para pararse hasta que termine 1 de sus hijos, por lo tanto necesita un wait por cada hijo que se cree. Devuelve un PID (del hijo que ha muerto).

Puede recibir un parámetro opcional para recibir en esa variable el estado que devuelve el exit (no lo usaremos).

*Exit(int status):* termina devolviendo el entero. 16 bits

exit 3 → 00000011 00000000 != 3

**Sincronización mediante señales**

1) Preparar el proceso para recibir señales, si no lo está, al recibir una, muere.

*Signal(int n señal, nombre de la funcion a ejecutar):* está preparado para recibir la señal **una sola vez**

*sigaction(int señal, struct sigaction \*accion, NULL).*

Registro con dos campos: .sa\_flags=0 y sa\_handler=rutina que atiende a la interrupción (driver)

*kill(int pid, int n señal):* enviar señal, si el proceso no está preparado, muere. para enviar señales entre procesos, necesitamos conocer sus PIDs

*alarm(int n segundos):* despertador, al cabo de n segundos recibe la señal 14 y se la autoenvía

el segundo alarm sustituye al primero. Devuelve un entero, el numero de segundos que faltaban para recibir el alarm.

*Alarm(10);*

.

. (2seg)

*Alarm(5);* → 8

*Alarm(0)* → borra la alarma anterior

El alarm no par al proceso, hace falta un pause

*Pause:* pausa un proceso hasta que reciba una señal cualquiera.

Si un proceso lo paramos para recibir señales, después de ejecutar su rutina, salta la instrucción que estaba haciendo el proceso antes de la interrupción.

30/04/17

**Comunicación entre procesos**

*creat:* crea un fichero.

*Open:* abre un fichero (para r, w… tiene opciones de append, abrir y crear). Devuelve la posición de la tabla de canales donde se asigna el fichero.

*read:* leer de un fichero

*write: escribir en un fichero*

**FIFOS**

*int mkfifo(const char \*camino, modo\_t modo):* equivalente a crear un fichero con creat, pero para fifos. Sigegundo parámetro son los permisos (igual que en chmod con un 0 delante). Una vez creado, se añade a la tabla de canales.

Una vez creado el fichero FIFO, cualquier proceso puede abrirlo para lectura o escritura, pero tiene que estar abierto con el open en escritura y lectura para operar con él.

Si se intenta leer de una FIFO que no tiene nada, se queda esperando a que haya algo.

Si no está abierto para escritura, devuelve un 0 y continúa la ejecución.

**Si se intenta escribir/leer en una FIFO que no está abierta por ningún otro proceso para lectura/escritura, el proceso que intenta realizar la escritura recibe la señal SIGPIPE.**

**Abriremos siempre con open para lectura y escritura.**

*Unlink():* es como hacer un rm de un fichero, lo borra. **El proceso que crea una FIFO, tiene que eliminarla.**

**TUBERÍA**

Cuando se crea un proceso con execl, hereda las 3 primera posiciones de la tabla de canales. Con un fork(), hereda la tabla completa. Teclado, pantalla, pantalla por defecto (bash).

*Pipe(int[] tubo)*: crea la tubería con puntero de escritura y puntero de lectura.

Tubo[0]: posición de la tabla de canales donde se ha colocado el puntero de lectura.

Tubo[1]: posición de la tabla de canales donde se ha colocado el puntero de escritura.

Usaremos Write(tubo[1], &variable, ) y read(tubo[0]).

Nos interesa que un proceso p1 se comunique con otro p2, para ello, p2 tiene que obtener el puntero de lectura de la tubería y p1 se queda con el puntero de escritura.

Como execl hereda las 3 primeras posiciones de la tabla de canales, tenemos que editarla (normalmente en la 2, la de los errores)

Secuencia:

Padre crea tubería

Padre hace fork en padre-copia

*Padre-copia:*

*close(2)*: *la copia* cierra la pos 2.

*dup(tubo[1]):* duplica lo pasado por parámetro en la primera posición libre de la tabla de canales.

*Execl()*: la copia hace execl y crea un hijo con Teclado, Pantalla y Puntero de escritura.

Padre cierra la posicion tomada por el hijo, y ya se pueden comunicar.

El hijo no conoce la variable tubo, tiene que usar la posicion.

03/05/19

**COLAS DE MENSAJES**

Tenemos un struct que define el tipo de lo que vamos a escribir/leer. Siempre long.

Así los procesos pueden identificar lo que van a leer y su tamaño.

struct Mensaje1{ Por ejemplo, 1 = int, 2 = char, 3 = float...

long tipo;

… (datos)

}

Para crear la cola necesitamos dos llamadas al sistema:

Se crea una clave con un fichero y un entero cualquiera

*key\_t ftok(char \*camino, int proy)*

*eg: key\_t Clave1 = ftok("./Makefile" , 33)*

Se crea la cola de mensajes y obtenemos un indicador para ella con *msgget*

i*nt id\_cola = msgget(Clave1, 0600 | IPC\_CREAT);*

Devuelve un entero que usaremos para escribir y leer de las colas.

Para leer y escribir no usamos read y write como en las pipes y fifos. Usaremos:

Devuelven un entero, -1 si hay fallo.

**Escribir**

*msgsnd( id\_cola, (struct msgbuf \*)&Mensaje1, sizeof(Mensaje1)* ***-*** *sizeof(long), IPC\_NOWAIT);*

Tamaño de la estructura **menos** el tipo.

El último parámetro puede ser:

IPC\_NOWAIT: que si al escribir en la cola de mensajes, está llena, no se queda esperando y continúa con la siguiente instrucción.

0: se queda esperando a que haya hueco en la cola de mensajes.

**Leer**

*msgrcv( id\_cola, (struct msgbuf \*)&Mensaje1, sizeof(Mensaje1) - sizeof(long), long tipo , 0)*

Penúltimo parámetro: el tipo a leer ( 0, 1, 2…)

Último parámetro: 0 ó IPC\_NOWAIT, importante.

IPC\_NOWAIT: no espera cuando no haya nada en la cola de mensajes de ese tipo.

0: se espera a que haya algo del tipo deseado. Usaremos esta.

**Borrar**

El ultimo en terminar debe borrar la cola **IMPORTANTE**

*msgctl ( id\_cola, IPC\_RMID, (struct msqid\_ds \*)NULL);*