Shell

Alan Nicolás Martellotti
X3865204P

-Glosario
-Versión Usuario
-Versión Extendida

Glosario de comandos internos:

- Comando>jobs
- Comando>cd Documentos
- Comando>cd ..
- Comando> fg
- Comando>fg x
- Comando>bg
- Comando>bg x
- Comando>historial
- Comando>historial x

VERSION USUARIO.

Procedimiento main:

Antes del main tenemos que destacar los procesos internos, el main se encarga de leer y almacenar en args el comando dado por teclado, si args esta vacio se volverá al principio del bucle, en otro caso puede ser que se quiera ejecutar otro comando, supongamos que quiere ejecutar uno interno, entraría a comprobar si args[0] es alguno de estos y entraría en el if, ejecutaría uno de estos y volvería al bucle, en el caso que sea historial, esta la opción que poner historial x, siendo x un numero de la lista de historial, que nos permitirá ejecutar el comando en esa posición del historial.

Usaremos los procesos block_SIGCHLD() y unblock_SIGCHLD() para bloquear/desbloquear la señal SIGCHLD y evitar que interfiera en el programa y la ejecución.

Si entra un proceso interno, como el jobs, se activa el subprograma y se ejecuta, si es un comando externo entonces pasa a crear un fork() que le dará al padre el PID del hijo y a este un 0, el proceso entrara con un PID diferente de 0, irá a la parte del padre y este comprobara si esta en segundo plano o no, en el caso que si, añadirá este proceso a la lista jobs y desbloqueará la señal SIGCHLD() que activara el manejador, sino se encargara de asignarle el terminal y hacer un wait al proceso hijo para recogerlo al terminar, también se encargara de mostrar información acerca del proceso, luego este proceso ira a la parte del hijo(if pid_fork==0) y pasaremos a darle un grupo de procesos y verificar si esta o no en segundo plano, y ejecutarlo. En el caso que no pueda ejecutarse se mostrara un error de que el comando no se encuentra.

Procedimiento Manejador (int signal):

Este procedimiento se encarga de tratar las señales de los procesos en segundo plano terminados y/o suspendidos, en este proceso recorreremos la lista hasta encontrar el proceso del cual se ha mandado la señal, una vez lo encontremos procederemos a eliminarlo de la lista si no esta suspendido, en este caso diremos que esta suspendido y le cambiaremos el estado a parado, con lo cual en la siguiente búsqueda será eliminado.

Este proceso no podrá ser accedido desde el Shell (es solo interno para manejar los procesos)

Procedimiento jobs ():

Este procedimiento solo se encarga de mostrar por pantalla la lista de procesos que estén en segundo plano y/o suspendidos, si la lista esta vacía se muestra un mensaje por pantalla diciendo que la lista esta vacía.

Se puede acceder desde la Shell de esta manera:

- **Comando>jobs**

Procedimiento cd(char * args[])

Esto nos permite movernos entre los directorios del sistema, se puede acceder a este proceso desde el Shell de muchas maneras, las cuales enumerare:

- **Comando>cd Documentos**
  - si el directorio existe accederemos a el con la función chdir y se mostrara por pantalla que se a accedido al directorio en cuestión.
- **Comando>cd ..**
  - tiene que haber un espacio entre cd y los dos puntos, esto nos devolverá al directorio anterior y nos lo notificara
- En otros casos nos devolverá los oportunos errores que podamos haber cometido introduciendo el proceso, ya sea que hemos escrito un directorio que no existe o si no lo hemos escrito.

Procedimiento fg(char * args[])

Este se encarga de pasar un proceso de la lista de procesos(jobs) a primer plano y luego eliminarlo de la lista. Este procedimiento se encarga también de hacerle un waitpid() al proceso en cuestión para que cuando termine, este mande un señal de finalización y se recoja y no quede zombie, de esta manera se acaba con el proceso y también con todo el grupo de procesos que se le había asignado antes.

El comando fg tiene dos funcionalidades en la Shell:

- **Comando> fg**
  - Se encargara de eliminar el primer proceso de la lista jobs
- **Comando>fg x**
  - X seria el número que indicaría la posición en la lista jobs, se pasa a ejecutar el fg del proceso que se encuentre en el numero x de la lista (ejemplo; [1]sleep [2] man man ,-> fg 2 haría fg de man man).
- En otro caso se manda el error que seria el caso que la lista de procesos este vacía o el número no se corresponda con un número de la lista.

La separare en dos partes, con capturas de pantalla, ya que el main es bastante largo.

Procedimiento bg(char * [])

El comando bg se utiliza para mover un trabajo a segundo plano. Este buscara de la lista jobs, pone su estado en Background y mata a todo el grupo de procesos que se le había asignado a tal proceso, tiene dos posibles usos:

- **Comando>bg**
  - Hace background del primer proceso en la lista jobs
- **Comando>bg x**
  - Siendo x el número de un proceso de la lista jobs, se encarga de ejecutar el bg del proceso que se encuentre en la posición x.
- En otro caso se envía un mensaje por pantalla mostrando el error que puede ser que no haya procesos en la lista o que el número introducido no es valido.

Historial

El historial es la implementación que hemos hecho, pasare a comentar los subprogramas que usamos antes de explicar el historial en si.

- Typedef struct H
  - Definimos una estructura que contenga el comando y una lista(lista H), también.
- Nueva_lista(char * c)
  - Usaremos esta función para crear una nueva lista.
- Int tam (H * l)
  - Esta nos servirá para ver el tamaño de la lista actual.
- H * new_command(const char * c)
  - Nos servirá para crear un nuevo comando en la lista.
- Void almacenar (H * list, H * item)
  - Este procedimiento lo usaremos en el main para poder almacenar todo lo que reciba en la lista dada.

- H* get_command_bypos(H * l , int pos)
    - Esta función se usara para devolver el comando que se encuentra en la listaH en la posición que le digamos.
- Void Historial (char * args[])
    - El Historial tiene dos formas de funcionamiento
        - **Comando>historial**
            - Imprime la listaH donde se encuentra todos los comandos que se han introducido.
        - **Comando>historial x**
            - Si recibe algún valor de 2º parámetro entonces pasaremos a poner el valor recibido en la posición x en la variable comando. En el main luego seguirá desde historial y podrá ejecutar la acción que se le diga, (ejemplo, [1]ls [2] jobs Comando>historial 2 -ejecutaría jobs).

Versión Extendida:

En esta parte se explica más en profundidad las clases main e historial ya que son las más largas e importantes, si el resumen de arriba ha dejado clara la función de los procedimientos esta parte solo concretara algunas partes de código y entrara más en detalle, pero no es necesaria si antes se ha entendido.

Procedimiento main:

```
199 int main(void)
200 {
201     char inputBuffer[MAX_LINE];
202     int background;
203     char *args[MAX_LINE/2];
204     int pid_fork, pid_wait;
205     int status;
206     enum status status_res;
207     int info;
208     ignore_terminal_signals();
209     lista = new_list("procesos");
210     listaH = nueva_lista("history");
211 
212     while (1)
213     {
214         signal(SIGCHLD,manejador);
215         printf("\nCOMANDO-> ");
216         fflush(stdout);
217         get_command(inputBuffer, MAX_LINE, args, &background);
218 
219         if(args[0]==NULL) continue;
220 
221         almacenar(listaH,new_command(args[0]));
222 
223         if(strcmp(args[0],"historial")==0){
224             historial(args);
225             if(args[1]!=NULL){
226                 if((args[1]!=NULL)&&(strcmp(comando,"historial")==0)){
227                     printf(" -> ya estÃ¡ ejecutandose(justo arriba).\n");
228                     continue;
229                 }else{
230                     args[0]=comando;
231                     args[1]=NULL;
232                 }
233             }
234         }
```

En el main declaramos las variables que vayamos a utilizar, las que usaremos mas adelante las explicare luego, solo cabe destacar de esta parte la creación de una lista de procesos llamada “procesos”, y la creación de otra lista (listaH) que una lista que usaremos para la función historial que luego explicare.

Comenzamos iniciando el bucle (infinito) para aceptar los comandos, activamos la función signal (línea 214) que cada vez que reciba la señal SIGCHILD (señal que manda si un proceso en segundo plano termina) vaya al proceso manejador, que explicare mas adelante.

Se recibe el comando escrito por el usuario y comprobamos que si no se ha introducido nada siga imprimiendo por pantalla COMANDO, a continuación almacenamos en la listaH(la de historial) el comando que se ha introducido y evaluamos con la función strcpm si lo que recibimos en args 0 es un comando interno, en nuestro caso los que hemos implementado, como cd, jobs, bg o fg, si recibimos un 0 significa que es el comando dado y a continuación procedemos a ejecutarlo, si vemos que es la función historial pasamos a ejecutarla y si el historial viene acompañado de un numero procedemos a ejecutar el comando que se encuentre en la posición de ese numero en la lista del historial, también se encargara de guardar este comando en la variable "comando"(vista en el main mas adelante), en el caso que el número indique que se ejecute el historial de nuevo, saldrá el mensaje que dirá que ya esta ejecutando arriba, en otro caso se ejecutaran los demás comandos, ya sean internos o externos. Podemos ver que si en historial no se ejecuta el historial, sino otro comando (por ejemplo cd o ls) pondremos el argumento 1 a nulo para así eliminar el numero del historial (por ejemplo si ponemos historial 2 y esta el ls, quedaría en args[0]=ls y args[1]==null, para ejecutar el ls).

Podemos ver que si recibimos historial hace un continue, esto es para evitar que el comando historial se ejecute de nuevo (solo comprobación, ya que a args[0] le asignamos comando).

```
}
if(strcmp(args[0],"cd")==0){
        cd(args);
}else if(strcmp(args[0],"jobs")==0){
        jobs();
}else if(strcmp(args[0],"bg")==0){
        bg(args);
}else if(strcmp(args[0],"fg")==0){
        fg(args);
}else{
    block_SIGCHLD();
    pid_fork=fork();
    if(pid_fork==0){
        new_process_group(getpid());
        if(background!=1){
            set_terminal(getpid());
        }
        restore_terminal_signals();
        execvp(args[0],args);
        printf("Error, Command not found: %s\n",args[0]);
        exit(1);
```

En el caso que no se ejecute ningún comando interno pasamos a la ejecución de los demás comandos, bloqueamos la señal para evitar funcionamientos indeseados con block_SIGCHILD(), y creamos un proceso hijo igual al padre y le asignamos el PID del hijo en pid_fork,(el padre tendrá el PID del hijo y el hijo tendrá 0).

A continuación verificaremos si el proceso es un proceso hijo y si lo es le asignaremos un grupo de procesos y verificaremos si esta en primer plano (background!=1), si lo esta, le asignamos el terminal, luego restauramos las señales del terminal y ejecutamos el comando dado en args[0], si el comando no existe se manda error y vuelve al principio del bucle.

```
258                 }else{
259                     if(background==1){
260                         add_job(lista,new_job(pid_fork,args[0],background));
261                         printf("Background job running... pid: %d, command: %s\n",pid_fork,args[0]);
262                         unblock_SIGCHLD();
263                     }else{
264                         set_terminal(pid_fork);
265                         pid_wait=waitpid(pid_fork,&status,WUNTRACED);
266                         status_res=analyze_status(status,&info);
267                         set_terminal(getpid());
268                         if(info != 1){
269                             printf("Foreground pid: %d, command: %s %s info: %d\n"
270                             ,pid_fork,args[0],status_strings[status_res],info);
271                         }
272                         if(status_strings[status_res] == "Suspended"){
273                             add_job(lista,new_job(pid_fork,args[0],STOPPED));
274                             printf("Proceso %d suspendido.\n",getpid());
275                             unblock_SIGCHLD();
276                         }
277                     }
278 
279                 }
280             }
281 
282         }
283 }
```

En el caso de que el PID sea diferente de 0, es decir que sea el padre, verificaremos si esta en segundo plano(línea 259), si lo esta añadiremos el proceso a la lista de procesos de 2º y/o suspendidos/parados y mostramos un mensaje con información de este proceso y pasamos a desbloquear la señal SIGCHLD() para activar el manejador.

En el caso que este en primer plano nos encargaremos de asignarle el terminal al proceso hijo y hacer que el padre espere a que este termine para poder recogerlo, analizamos el estado del proceso y luego le asignamos el terminal al proceso que lo pide, que en este caso es el padre, verificamos que la información este correcta y mostramos por pantalla la información del proceso que se esta ejecutando.

Si el estado es suspendido añadimos en la lista de procesos (jobs) este proceso y activamos otra vez la señal SIGCHLD para ir al manejador.

Historial

El historial es la implementación que hemos hecho, pasare a comentar los subprogramas que usamos antes de explicar el historial en si, adjuntare imágenes para la mejor comprensión.

```
typedef struct H_{
    char * command;
    struct H_* next;
}H;
H* listaH;
char* comando;
```

Definimos una estructura que contenga el comando y una lista la cual llamaremos listaH, también usaremos comando para guardar lo que el usuario introduce por teclado, se mostrara mas adelante el uso.

```
H * nueva_lista(char * c){
    H* aux = (H *) malloc(sizeof(H));
    aux->command = c;
    return aux;
}
```

Usaremos esta función para crear una nueva lista, esta estructura sigue el tipo que se sigue en job_control.c

```
int tam (H * l){
    H* aux=l;
    int cont=0;
    while(aux!=NULL){
        cont++;
        aux=aux->next;
    }
    return cont;
}
```

Esta nos servirá para ver el tamaño de la lista actual.

```
H * new_command(const char * c){
    H * aux;
    aux = (H *) malloc(sizeof(H));
    aux->command = strdup(c);
    aux->next = NULL;
    return aux;
}
```

Nos servirá para crear un nuevo comando en la lista, no lo almacenara directamente, sino que le asignara al comando de la estructura lo que se le de por argumentos.

```
void almacenar(H * list, H * item){
    H * aux = list->next;
    list->next = item;
    item->next = aux;
}
```

Este procedimiento lo usaremos en el main para poder almacenar todo lo que reciba en la lista dada.

```
H * get_command_bypos(H * l, int pos){
    H * aux = l;
    if(pos<1||pos>tam(l)) return NULL;
    pos++;
    while(aux->next != NULL && pos){
        aux= aux->next;
        pos--;
    }
    return aux;
}
```

Esta función se usara para devolver el comando que se encuentra en la listaH en la posición que le digamos.

```
void historial(char * args[]){
    int cont=1;
    H * aux;
    if(args[1] != NULL){
        if(tam(listaH)<atoi(args[1]) || 0>atoi(args[1])){
            printf("No hay suficientes comandos en el historial.\n");
        }else{
            cont = atoi(args[1]);
            aux = get_command_bypos(listaH,cont);
            //printf(aux->command);
            comando=aux->command;
        }
    }else{
        aux=listaH;
        printf("%s\n",aux->command);
        while(aux->next!=NULL){
            aux=aux->next;
            printf("%d  ",cont);
            printf(aux->command);
            printf("\n");
            cont++;
        }
    }
}
```

El Historial puede recibir 1 argumentos o 2, si recibe un solo argumento salta al segundo else y se encarga de mostrar por pantalla el nombre de la lista y la lista de procesos con su correspondiente posición en la lista.

En el caso que reciba 2 argumentos comprobara que ese numero esta en la lista, sino mostrara error, si ese numero esta se encarga de acceder a la posición esa y coger el comando almacenado con la función get_item_bypos y almacenarlo en la variable global comando.

Esto sirve porque en main si se ejecuta historial, se ejecuta este y luego se cambia los args para que si recibe la posición donde se encuentra un comando interno pueda ejecutarlo luego en el main(pueda seguir el código). Historial solo guarda un comando de los usados, por ejemplo si se usa ls –a solo guardaría ls y ejecutaria este.