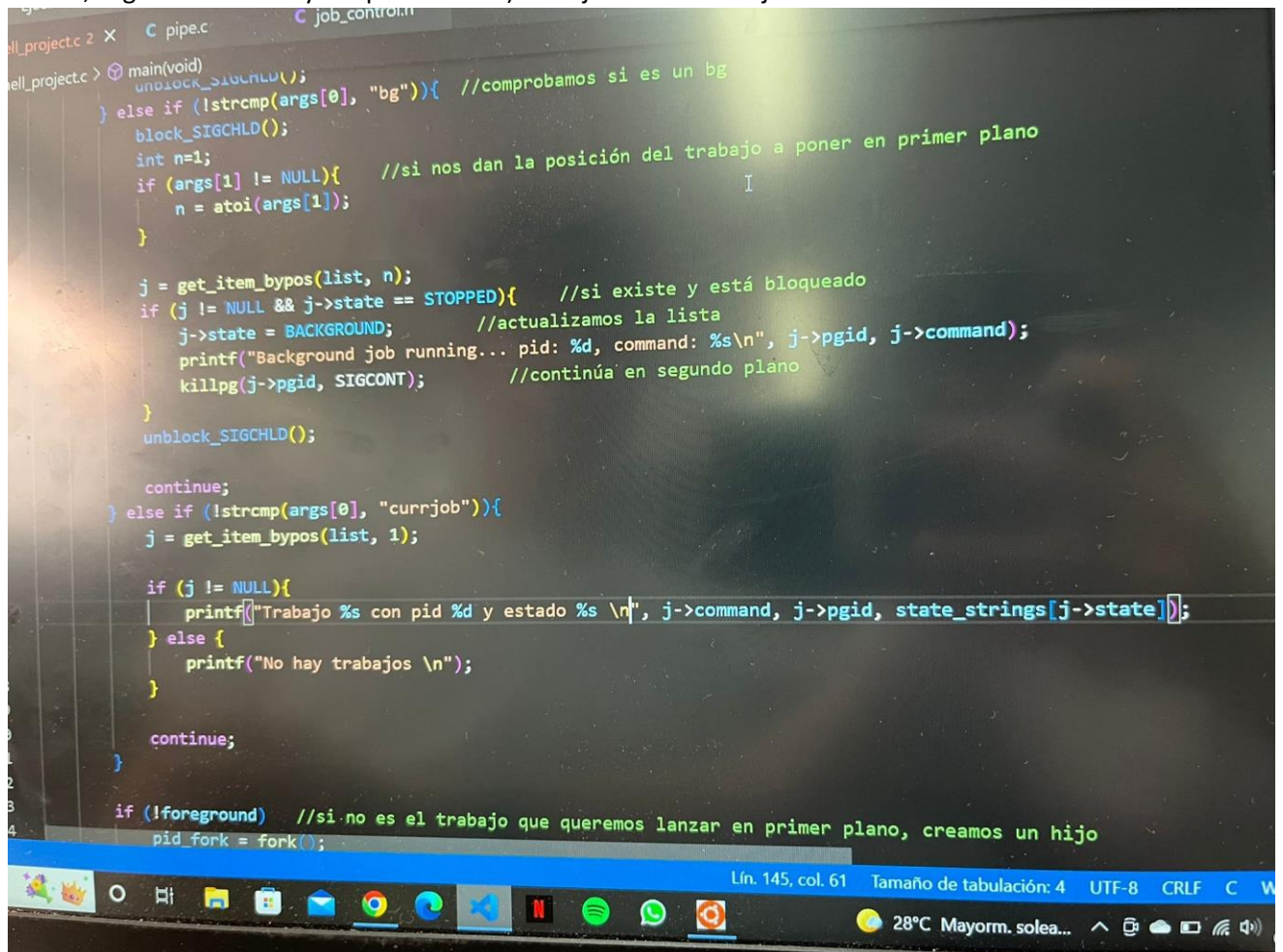


1- Crear un manejador para la señal SIGHUP, que al recibirla, creará si no existe un documento sighup.txt en la carpeta del Shell, y hará printf de "Señal SIGHUP recibida. \n".

Para probarlo, se puede abrir el Shell en una ventana del terminal, y en otra ventana nueva de terminal, buscar el pid del proceso ./Shell con el comando "ps -au", y escribir el comando "kill -SIGHUP <pid del proceso>".

```
71
72 int main(void)
73 {
74     char inputBuffer[MAX_LINE]; /* buffer to hold the comm
75     int background;             /* equals 1 if a command is
76     char *args[MAX_LINE/2];     /* command line (of 256) ha
77     // probably useful variables:
78     int pid_fork, pid_wait; /* pid for created and waited p
79     int status;             /* status returned by wait */
80     enum status status_res; /* status processed by analyze_
81     int info;               /* info processed by analyze_st
82     job *j;                 //variable trabajo
83     char *name;             //comando
84     int foreground=0;        //nos dice si queremos ponerlo e
85
86     ignore_terminal_signals(); //ignoramos las señales rela
87     signal(SIGCHLD, manejador); //si recibe la señal SIGCHLD
88     signal(SIGHUP, manejador1);
89     //signal(SIGHUP, manejador);
90     list = new_list("list"); //creamos la lista
91
92     while (1) /* Program terminates normally inside get
93     {
94
19 #define MAX_PRINTS 100
20
21 // -----
22 //                               MAIN
23 // -----
24
25 job *list; //lista de trabajos
26
27 void manejador1(int senal){
28
29
30     printf("Señal SIGHUP recibida. \n");
31
32
33 void manejador(int senal){
34     block_SIGCHLD(); // cuando llega una señ
35     job *j;          //bloqueo ya que se
36     int status, info, pid_wait=0; //variable con a qu
37     enum status status_res;
38
39
40
41     for (int i=1; i<=list_size(list); i++){
42         j = get_item_bypos(list, i);
43
44         if (j != NULL){
45             pid_wait = j->pid;
46             status_res = waitpid(pid_wait, &status, 0);
47             if (status_res == -1)
48                 perror("waitpid");
49             else
50                 printf("Process %d terminated with status %d\n", j->pid, status);
51             delete(j);
52         }
53     }
54 }
```

2- Añadir el comando interno "currjob", que mostrará por la salida estándar información sobre el primer (o último, según cómo se haya implementado) trabajo de la lista de jobs.



```
main(void)
{
    unblock_sigchld();
} else if (strcmp(args[0], "bg")){ //comprobamos si es un bg
    block_sigchld();
    int n=1;
    if (args[1] != NULL){ //si nos dan la posición del trabajo a poner en primer plano
        n = atoi(args[1]);
    }

    j = get_item_bypos(list, n);
    if (j != NULL && j->state == STOPPED){ //si existe y está bloqueado
        j->state = BACKGROUND; //actualizamos la lista
        printf("Background job running... pid: %d, command: %s\n", j->pgid, j->command);
        killpg(j->pgid, SIGCONT); //continúa en segundo plano
    }
    unblock_sigchld();

    continue;
} else if (strcmp(args[0], "currjob")){
    j = get_item_bypos(list, 1);

    if (j != NULL){
        printf("Trabajo %s con pid %d y estado %s \n", j->command, j->pgid, state_strings[j->state]);
    } else {
        printf("No hay trabajos \n");
    }

    continue;
}

if (!foreground) //si no es el trabajo que queremos lanzar en primer plano, creamos un hijo
    pid_fork = fork();
```

3- Añadir el comando interno "deljob", que eliminará el trabajo actual de la lista de tareas (siendo este el descrito en "currjob"). Eliminarla solamente significa eliminarlo de la estructura de datos, pero no hay que hacer nada más con la tarea (ni matarla, ni nada más), simplemente sacarla de la lista.


```

139         continue;
140     } else if (!strcmp(args[0], "currjob")){
141         j = get_item_bypos(list, 1);
142
143         if (j != NULL){
144             printf("Trabajo %s con pid %d y estado %s \n",
145                 j->command, j->pgid, j->state);
146         } else {
147             printf("No hay trabajos \n");
148         }
149
150         continue;
151     } else if (!strcmp(args[0], "deljob")){
152         j = get_item_bypos(list, 1);
153         if (j != NULL) {
154             delete_job(list, j);
155         }
156         continue;
157     }
158
159     if (!foreground) //si no es el trabajo que queremos
160         pid_fork = fork();
161
162     if (!pid_fork){ //código del hijo

```

1. Imprime una cadena cuando un proceso en segundo plano muera por recibir una señal

```

42     printf("Background pid: %d, command: %s, info: %d\n", j->pgid, j->command,
43           j->info);
44     //actualizamos los elementos de la lista
45     if (status_res == SUSPENDED){ //si se ha suspendido
46         j->state = STOPPED;
47     } else if (status_res == CONTINUED){ //si se ha reanudado
48         j->state = BACKGROUND;
49     } else { //si ha finalizado tanto por una señal como por si mismo
50         delete_job(list, j);
51         i--; //para que no se salte un trabajo en la siguiente iteración
52     }
53     if (j->state == BACKGROUND && status_res == SIGHUP){
54         printf("maria la shupa");
55     }
56 }
57
58
59
60
61
62     unblock_sigchld(); //desbloqueo ya que ya no se utiliza la lista
63 }
64
65 int main(void)
66 {
67

```

2. Comando interno que todas las tareas de segundo plano se maten

```
229 Shell_examen > C Shell_project.c
230
231 //no la añadimos a la
232 //Pero si solo se ha
233 if (status_res == SUSPENDED){
234     add_job(my_job_list, new_job(aux_pid, nombre, STOPPED));
235 }
236
237 printf("Foreground pid: %d, command: %s, %s, info: %d \n", aux_pid, nombre,
238
239 }
240
241 }else if (strcmp(args[0], "nuevo"){
242     block(SIGCHLD);
243     for (int i=list_size(my_job_list); i>=1; i--){
244         job * aux = get_item_bypos(my_job_list, i);
245         if (aux!=NULL){
246             if((*aux).state == BACKGROUND)
247                 killpg(SIGINT, (*aux).pgid);
248             i--;
249         }
250     }
251     unblock(SIGCHLD);
252
253 }
254
255 }
256
257 }
258
259 }
260
261 }
262
263 }else if (strcmp(args[0], "bg")==0){ //comando interno bg
264     //este comando pone a trabajar en segundo plano a una tarea que estaba suspendida
265     //esta primera parte igual que en comando interno fg
266     int n = 1;
267     int argumento;
268     if (args[1]!=NULL){
269         int
```


NUEVOS 2021

1. Poner cuando un proceso en segundo plano terminara: print he terminado

```
j != NULL){ //comprobamos si existe tal proceso
pid_wait = waitpid(j->pgid, &status, WUNTRACED | WNOHANG | WCONTINUED);

if (pid_wait == j->pgid){ //comprobamos si es el que buscamos
    status_res = analyze_status(status, &info);

    printf("Background pid: %d, command: %s, %s, info: %d\n", j->pid, j->command, j->args[0], status_res);

    //actualizamos los elementos de la lista
    if (status_res == SUSPENDED){ //si se ha suspendido
        j->state = STOPPED;
    } else if (status_res == CONTINUED){ //si se ha reanudado
        j->state = BACKGROUND;
    } else { //si ha finalizado tanto por una señal como por sí mismo
        if (status_res == EXITED && j->state == BACKGROUND)
            printf("He terminado");

        delete_job(list, j);
        i--; //para que no se salte un trabajo en la siguiente iteración
    }
}
```

2. Crear un contador para los procesos suspendidos o signaled y poner por pantalla el contador

```
}else{ //Padre

    if(!background){ //si se ha lanzado en primer plano
        waitpid(pid_fork, &status, WUNTRACED);
        set_terminal(getpid()); //le damos el terminal
        status_res=analyze_status(status,&info);

        if(status_res==SUSPENDED){ //en el caso en el que se haya suspendido
            block_SIGCHLD();
            if(pp){ //el trabajo que buscamos es el que se había puesto en primer plano
                job=new_job(pid_fork,co,STOPPED);
                printf("Foreground pid: %d, command: %s, %s, info: %d \n", pid_fork, co, status_strings[status_res], info);
            }else{ //el trabajo que buscamos es el suspendido
                job=new_job(pid_fork,args[0],STOPPED);
                printf("Foreground pid: %d, command: %s, %s, info: %d \n", pid_fork, args[0], status_strings[status_res], info);
            }
            add_job(list,job); //lo añadimos a la lista
            unblock_SIGCHLD();
        }else{
            if(pp){
                printf("Foreground pid: %d, command: %s, %s, info: %d \n", pid_fork, co, status_strings[status_res], info);
            }else{
                printf("Foreground pid: %d, command: %s, %s, info: %d \n", pid_fork, args[0], status_strings[status_res], info);
            }
        }
    }

    /*
    if(status_res==SUSPENDED || status_res == SIGNALED){
        contador++;
        printf("Contador: " + contador);
    }
    */
    pp=0;
}
```

```

job *list;
//int contador=0;

void manejador(int senal){           //se activa cndo llega una señal SIGCHLD
    block_SIGCHLD();                //se bloquean las señales para acceder a la lista
    job *job;
    int info, status, pid_wait=0;
    enum status status_res;

    for(int i=1; i<=list_size(list);i++){
        job=get_item_bypos(list,i);
        if(job!=NULL){
            pid_wait=waitpid(job->pgid,&status, WUNTRACED | WNOHANG | WCONTINUED); //hac
            if(pid_wait==job->pgid){           //coincide con el que buscamos
                status_res=analyze_status(status,&info);
                printf("Background pid: %d, command: %s, %s, info: %d \n", job->pgid, job->command, job->name, info);
                if(status_res==SUSPENDED){      //actualizamos la lista
                    job->state=STOPPED;
                }else if(status_res==CONTINUED){
                    job->state=BACKGROUND;
                }else{
                    delete_job(list,job);
                    i--;
                }
            }
            /*
            if(status_res==SUSPENDED || status_res == SIGNALED){
                contador++;
                printf("Contador: " + contador);
            }
            */
        }
    }
}

```

3.a) Comando interno killpg, todos los no-stopped los intenta matar, b) contador con la cantidad de procesos q ha intentado matar

```

if(!strcmp(args[0],"killpg")){
    block_SIGCHLD();
    int err=0;
    int cont1=0, cont2=0;
    for(int i=1;i<=list_size(list);i++){
        job=get_item_bypos(list,i);
        if(job!=NULL && job->state != STOPPED){
            err = killpg(job->pgid,SIGINT);
            cont1++;
            if(err==-1){
                printf("No se ha podido matar");
            }else{
                cont2++;
            }
        }
    }
    printf("Ha intentado matar %d procesos, sin embargo, ha matado realmente a %d procesos.\n", cont1, cont2);
    unblock_SIGCHLD();
    continue;
}

```

4. Crear un fichero y guardar en ese fichero el nombre del comando + pid , se ha matado correctamente con la funcion killpg

myatomo.

- ①. Mostrar la cadena "Ouch!" cada vez que el shell reciba un SIGCHLD [1 pto]

```
void manejador(int senal){           //se activa cndo llega una señal SIGCHLD
    block_SIGCHLD();                //se bloquean las señales para acceder a la lista
    job *job;
    int info, status, pid_wait=0;
    enum status status_res;
    //printf("auch");
    for(int i=1; i<=list_size(list);i++){
        job=get_item_bypos(list,i);
```

- ②. Mostrar la cadena "Quiero aprobar!" cada vez que se suspenda una tarea de 2º plano (background) [2 ptos]

```
void manejador(int senal){           //se activa cndo llega una señal SIGCHLD
    block_SIGCHLD();                //se bloquean las señales para acceder a la lista
    job *job;
    int info, status, pid_wait=0;
    enum status status_res;
    //printf("auch");
    for(int i=1; i<=list_size(list);i++){
        job=get_item_bypos(list,i);
        if(job!=NULL){
            pid_wait=waitpid(job->pgid,&status, WUNTRACED | WNOHANG | WCONTINUED); //hacemos un wait para evitar el bloqueo
            if(pid_wait==job->pgid){ //coincide con el que buscamos
                status_res=analyze_status(status,&info);
                printf("Background pid: %d, command: %s, %s, info: %d \n", job->pgid, job->command, status_strings[status_res], info);
                if(status_res==SUSPENDED){ //actualizamos la lista
                    job->state=STOPPED;
                    //printf("Quiero aprobar");
                }else if(status_res==CONTINUED){
                    job->state=BACKGROUND;
                }else{
                    delete_job(list,job);
                }
            }
        }
    }
}
```

3. Crear el comando interno "sig" que imprima el número de tareas que han terminado por recibir una señal (SIGNALED) (no las que han terminado normalmente, EXITED). El texto que debe aparecer tras ejecutar el comando "sig" debe ser similar al siguiente: 'El número de tareas que han terminado por recibir una señal es de 4' [2 ptos]

```
if(!strcmp(args[0],"sig)){
    printf("Hay signaled %d \n",contador);
    continue;
}
```

```

}else{ //Padre

    if(!background){ //si se ha lanzado en primer plano
        waitpid(pid_fork, &status, WUNTRACED);
        set_terminal(getpid()); //le damos el terminal
        status_res=analyze_status(status,&info);

        if(status_res==SUSPENDED){ //en el caso en el que se haya suspendido
            block_SIGCHLD();
            if(pp){ //el trabajo que buscamos es el que se había p
                job=new_job(pid_fork,co,STOPPED);
                printf("Foreground pid: %d, command: %s, %s, info: %d \n", pid_fork, co
            }else{ //el trabajo que buscamos es el suspendido
                job=new_job(pid_fork,args[0],STOPPED);
                printf("Foreground pid: %d, command: %s, %s, info: %d \n", pid_fork, a
            }
            add_job(list,job); //lo añadimos a la lista
            unblock_SIGCHLD();
        }else{
            if(pp){
                printf("Foreground pid: %d, command: %s, %s, info: %d \n", pid_fork, co
            }else{
                printf("Foreground pid: %d, command: %s, %s, info: %d \n", pid_fork, a
            }
        }
    }
    /*
    if(status_res==SUSPENDED || status_res == SIGNALED){
        contador++;
        printf("Contador: " + contador);
    }
    */
    if(status_res==SIGNALED){
        contador++;
    }
    pp=0;

```



```

void manejador(int senal){           //se activa cndo llega una señal SIGCHLD
    block_SIGCHLD();                //se bloquean las señales para acceder a la l
    job *job;
    int info, status, pid_wait=0;
    enum status status_res;
    //printf("auch");
    for(int i=1; i<=list_size(list);i++){
        job=get_item_bypos(list,i);
        if(job!=NULL){
            pid_wait=waitpid(job->pgid,&status, WUNTRACED | WNOHANG | WCONTINUED)
            if(pid_wait==job->pgid){           //coincide con el que buscamos
                status_res=analyze_status(status,&info);
                printf("Background pid: %d, command: %s, %s, info: %d \n", job->p
                if(status_res==SUSPENDED){      //actualizamos la lista
                    job->state=STOPPED;
                    //printf("Quiero aprobar");
                }else if(status_res==CONTINUED){
                    job->state=BACKGROUND;
                }else{
                    delete_job(list,job);
                    i--;
                    if(status_res==SIGNALLED){
                        contador++;
                    }
                }
            }
        }
    }
}
/*

```

Ampliación: Implementar un comando interno 'team' que recibe un número (n) y un comando (cmd)

p.ej.:>team 5 xclock -update 1

4. Lanzará n veces el comando cmd en background [1.5 ptos]
5. El manejador de SIGCHLD debe hacer 'wait' de los procesos de la tarea de tipo team para que no queden zombies (defunct) [1.5 ptos]

object.c > main(void)

```

    } else if (!strcmp(args[0], "team")){
        if (args[1] != NULL){
            int n = atoi(args[1]);

            if (args[2] != NULL){

                int tam=2;
                while (args[tam] != NULL){
                    tam++;
                }

                char **vargs=(char **) malloc(sizeof(char*)*tam);
                for(int i=2; i<tam; i++){
                    vargs[i-2] = strdup(args[i]);
                }

                for (int i=0; i<n; i++){
                    pid_fork = fork();

                    if (!pid_fork){
                        new_process_group(getpid()); //le creamos su grupo de procesos
                        restore_terminal_signals();

                        execvp(args[2], vargs); //ejecutamos el comando

                        printf("Error, command not found: %s\n", args[0]);
                        exit(-1);
                    } else {
                        block_SIGCHLD();
                        j = new_job(pid_fork, args[2], BACKGROUND);
                        add_job(list, j);
                        unblock_SIGCHLD();
                    }
                }
            }
        }
    }
}

```