

Cracking Shells

Shell – A process that does command-line interpretation

ush.h

```
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <limits.h>
#include <errno.h>
#include <sys/stat.h>
#define STDMODE 0600
#ifndef MAX_CANON
#define MAX_CANON 256
#endif
int makeargv(char *s, char *delimiters, char ***argvp);
int parseandredirectin(char *inbuf);
int parseandredirectin(char *inbuf);
void executecmdline(char *cmd);
int executeredirect(char *s, int in, int out);
int signalsetup(struct sigaction *def, sigset_t *mask, void (*handler)(int));

#define TRUE 1
#define FALSE 0
#define BLANK_STRING " "
#define PROMPT_STRING ">>"
#define QUIT_STRING "q"
#define BACK_STRING "&"
#define PIPE_STRING "|"
#define NEWLINE_STRING "\n"
#define IN_REDIRECT_SYMBOL '<'
#define OUT_REDIRECT_SYMBOL '>'
#define NULL_SYMBOL '\0'
#define PIPE_SYMBOL '|'
#define BACK_SYMBOL '&'
#define NEWLINE_SYMBOL '\n'
#define FFLAG (O_WRONLY|O_CREAT|O_TRUNC)
#define FMODE (S_IRUSR|S_IWUSR)
```

```
#include "ush.h"
#define MAX_BUFFER 256
void main (void)
{
    char **chargv;
    char inbuf[MAX_BUFFER];
    for( ; ; ) {
        gets(inbuf);
        if (strcmp(inbuf, QUIT_STRING) == 0)
            return 0;
        if (fork() == 0) &&
            makeargv(inbuf, BLANK_STRING,
&chargv) > 0)
            execvp(chargv[0], argv);
        wait(NULL);
    }
}
```

ush1.c

Problems with ush1.c

- cd is not available
- No wildcards such as * and ?
- What if parent doesn't call wait?
 - Execute with wait removed and run an invalid command such as “xyz”
 - Since execvp does not catch error, child falls through and begins executing commands – ps shows two ush1 shells
 - The parent and child shells can execute concurrently.
Execute ls /user/s followed by ls
- Notice MAX_BUFFER is user defined, non-portable constant – uses gets rather than fgets

```
#include "ush.h"
void main (void)
{ pid_t childpid;
  char inbuf[MAX_CANON+1];
  int len;
  for( ; ; ) { if (fputs(PROMPT_STRING, stdout) == EOF)
                continue;
                if (fgets(inbuf, MAX_CANON, stdin) == NULL)
                    continue;
                len = strlen(inbuf);
                if (inbuf[len - 1] == NEWLINE_SYMBOL)
                    inbuf[len - 1] = 0;
                if (strcmp(inbuf, QUIT_STRING) == 0)
                    break;
                else { if ((childpid = fork()) == -1 )
                        perror("Fork failed")
                    elseif (childpid == 0)
                        executecmd(inbuf);
                        return 1; }
                else
                    wait(NULL); }
  return 0; }
```

ush2.c

executecmdsimple.c

```
#include "ush.h"
void executecmd(char *incmd)
{
    char **chargv;
    if (makeargv(incmd, BLANK_STRING, &chargv) <= 0) {
        fprintf(stderr, *Failed to parse command line\n");
        exit(1); }
    execvp(chargv[0], chargv)
    perror("Invalid command");
    exit(1);
}
```

Problems Solved by ush2.c

- Command prompt is displayed
- Proper termination of `execvp` on error
- System-defined constant `MAX_CANON` replaces `MAX_BUFFER`
- `fgets` replaces `gets`

Current Problems with ush2.c

- cd doesn't work – cd must change the user's environment
- Try ls -l and q with interspersed and leading extra blanks
 - ls -l works but q doesn't because
 - ls -l is handled by makeargv – q isn't
- Try commands found at stty -a such as ^c – What do they do to parent/child? Why?
- Try erase, ^h – It still works because characters entered are not in program yet but are stored in a temporary buffer
- Doesn't recognize redirect or pipe symbols

Adding in Redirection

- `compile < t.c`
- `cat file1 > file2`
- `cat < my.input > my.output`

executecmdredirect.c

```
#include "ush.h"
void executecmd (char *incmd)
{  char **chargv;
  if (parseandredirectout(incmd) == -1)
    perror("Failed to redirect Output);
  else if (parseandredirectin(incmd) == -1)
    perror("Failed to redirect input");
  else if (makeargv(incmd, BLANK_STRING, &chargv) <= 0)
    fprintf(stderr, "Fialed to parse command line\n");
  else {
    execvp(chargv[0], chargv)
    perror("Failed to execute command");
  }
  exit(1); }
```

parseandredirect.c – Top

```
#include "ush.h"
int parseandredirect(char *cmd) { /* redirect stdin if '<' */
    int error;
    int infd;
    char *infile;
    if ((infile = strchr(cmd, '<')) == NULL) return 0;
    *infile = 0 /* take everything after '<' out of cmd */
    infile = strtok(infile + 1, "\\t");
    if (infile == NULL) return 0;
    if ((infd = open(infile, O_RDONLY)) == -1) return -1;
    if (dup2(infd, STDIN_FILENO) == -1) {
        error = errno; /* make sure errno is correct */
        close(infd);
        errno = error;
        return -1; }
    return close(infd); }
```

parseandredirect.c - Bottom

```
int parseandredirectout(char *cmd) { /* redirect stdout if '>' */
    int error;
    int outfd;
    char *outfile;
    if ((outfile = strchr(cmd, '>')) == NULL) return 0;
    *outfile = 0 /* take everything after '>' out of cmd */
    outfile = strtok(outfile + 1, "\t");
    if (outfile == NULL) return 0;
    if ((outfd = open(outfile, O_WRONLY, 0)) == -1) return -1;
    if (dup2(outfd, STDOUT_FILENO) == -1) {
        error = errno;
        close(outfd);
        errno = error;
        return -1; }
    return close(infd); }
```

pipe and redirection

- `ush2.c`
- `executecmdpipe.c`
- `executeredirect.c`
- `parseandredirect.c`
- `makeargv.c`

executecmdpipe.c

```
void executecmd(char *cmds) {
    int child, count, fds[2], l;
    char **pipelist;
    count = makeargv(cmds, "|", &pipelist);
    if (count <= 0) { fprintf(stderr, "No Input Command\n"); exit(1); }
    for (i = 0; i < count - 1; i++) {
        if (pipe(fds) == -1) perror_exit("Failed to create pipes");
        else if (( child = fork () ) perror_exit("Fork failed");
        else if (child) {
            if (dup2 (fds[1], STDOUT_FILENO) == -1) perror_exit("Pipe failed");
            if (close (fd[0]) || close (fds[1])) perror_exit("Close failed");
            executeredirect(pipelist[i], i==0, 0); exit(1);
        }
        if (dup2 (fds[0], STDIN_FILENO) == -1) perror_exit("Last connect
failed");
        if (close (fds[0]) || close (fds[1])) perror_exit("Final close failed");
    }
    executeredirect(pipelist[i], i==0, 1); exit(1); }
```

executeredirect.c

```
void executeredirect (char *s, int in, int out) {
    char **chargv, *pin, *pout;
    if (in && ((pin = strchr(s, '>')) != NULL) && out && ((pout=strchr(s,'>'))
        != NULL) && (pin > pout) ) {
        if (parseandredirectin(s) == -1) { perror("Failed to redirect input); return; }
        in = 0; }
    if (out && (parseandredirectout(s) == -1)) perror("Output redirect failed");
    else if (in && (parseandredirectin(s) == -1)) perror("Input redirect failed");
    else if (makeargv(s, " \t", &chargv) <= 0)
        fprintf(stderr,"Failed to parse command line\n");
    else { excvp (chargv[0], argv); perror("Failed to execute command"); }
    exit(1); }
```

signalsetup.c

```
int signalsetup(struct sigaction *def, sigset_t *mask, void (*handler)(int)) {
    struct sigaction catch;
    catch.sa_handler = handler; /* Set up signal structures */
    def->sa_handler = SIG_DFL; catch.sa_flags = 0; def->sa_flags = 0;
    if ((sigemptyset(&(def->sa_mask)) == -1) ||
        (sigemptyset(&(catch.sa_mask)) == -1) ||
        (sigaddset(&(catch.sa_mask), SIGINT) == -1) ||
        (sigaddset(&(catch.sa_mask), SIGQUIT) == -1) ||
        (sigaction(SIGINT, &catch, NULL) == -1) ||
        (sigaction(SIGQUIT, &catch, NULL) == -1) ||
        (sigemptyset(mask) == -1) ||
        (sigaddset(mask, SIGINT) == -1) ||
        (sigaddset(mask, SIGQUIT) == -1))
        return -1;
    return 0; }
```

ush3.c – top

```
#include "ush.h"
int main (void) {
    sigset_t blockmask;
    pid_t childpid;
    struct sigaction defaction;
    char inbuf[MAX_CANON];
    int len;

    if (signalsetup(&defaction, &blockmask, SIG_IGN) == -1) {
        perror("Failed to set up shell signal handling");
        return 1;
    }
    if (sigprocmask(SIG_BLOCK, &blockmask, NULL) == -1) {
        perror("Failed to block signals");
        return 1;
    }
}
```

ush3.c – Bottom

```
for(;;) {
    if (fputs(PROMPT_STRING, stdout) == EOF) continue;
    if (fgets(inbuf, MAX_CANON, stdin) == NULL) continue;
    len = strlen(inbuf);
    if (inbuf[len - 1] == '\n') inbuf[len - 1] = 0;
    if (strcmp(inbuf, QUIT_STRING) == 0) break;
    if((childpid = fork()) == -1) perror("Failed to fork child to execute command");
    else if (childpid == 0) {
        if ((sigaction(SIGINT, &defaction, NULL) == -1) ||
            (sigaction(SIGQUIT, &defaction, NULL) == -1) ||
            (sigprocmask(SIG_UNBLOCK, &blockmask, NULL) == -1)) {
            perror("Failed to set signal handling for command "); return 1; }
        executecmd(inbuf); return 1; }
    wait(NULL); }
return 0; }
```

ush3.c Analysis

- Parent ignores signals
- Child does not
- Problem – On ^c child does not display prompt

ush4.c – Top

```
#include "ush.h"
static void jumphd(int signalnum) {
    if (!okaytojump) return;
    okaytojump = 0;
    siglongjmp(jumptoprompt, 1); }
int main (void) {
    sigset_t blockmask;
    pid_t childpid;
    struct sigaction defhandler;
    int len;
    char inbuf[MAX_CANON];
    if (signalsetup(&defhandler, &blockmask, jumphd) == -1) {
        perror("Failed to set up shell signal handling");
        return 1;
    }
}
```

ush4.c - Middle

```
for( ; ; ) {
    if ((sigsetjmp(jumptoprompt, 1)) && /* if return from signal, \n */
        (fputs("\n", stdout) == EOF) ) continue;
    wait(NULL);
    okaytojump = 1;
    if (fputs(PROMPT_STRING, stdout) == EOF) continue;
    if (fgets(inbuf, MAX_CANON, stdin) == NULL) continue;
    len = strlen(inbuf);
    if (inbuf[len - 1] == '\n')
        inbuf[len - 1] = 0;
    if (strcmp(inbuf, QUIT_STRING) == 0) break;
    if (sigprocmask(SIG_BLOCK, &blockmask, NULL) == -1)
        perror("Failed to block signals");
```

ush4.c - Bottom

```
if ((childpid = fork()) == -1) perror("Failed to fork");
else if (childpid == 0) {
    if ((sigaction(SIGINT, &defhandler, NULL) == -1) ||
        (sigaction(SIGQUIT, &defhandler, NULL) == -1) ||
        (sigprocmask(SIG_UNBLOCK, &blockmask, NULL) == -1)) {
        perror("Failed to set signal handling for command ");
        return 1; }
    executecmd(inbuf); return 1; }
if (sigprocmask(SIG_UNBLOCK, &blockmask, NULL) == -1)
    perror("Failed to unblock signals"); }
return 0; }
```

ush4.c Analysis

- Parent jumps to user prompt on SIGINT and SIGQUIT

kill Command

- `kill -INT -3245` sends SIGINT to all processes in process group 3245
- `kill -INT 3245` sends SIGINT to only process 3245

Process Group

In the command:

```
$ ls -l | sort -n +4 | more
```

the commands `ls`, `sort`, and `more` are all in the same process group.

`^c` will terminate all commands but the shell

Getting and Changing Process Group

SYNOPSIS

```
#include <unistd.h>  
pid_t getpgrp(void);  
int setpgid(pid_t pid, pid_t pgid);
```

POSIX

Get process group ID with `getpgrp` and change process group with `setpgid`

setpgid

- Sets the process group ID of process pid to the process group ID pgid.
- Child inherits process group ID of parent
- The parent can use setpgid to change the process group of a child so long as the child has not yet issued an exec
- A child can give itself a new process group by setting its process group to its own process ID

Background Process

- A background process does not receive $\wedge c$
- A background process brought back into the foreground, does receive $\wedge c$
- Background processes are more complicated to manage than foreground processes – for example, which process terminates them.

Getting and Changing Session ID

SYNOPSIS

```
#include <unistd.h>
```

```
pid_t getsid(pid_t pid);
```

POSIX: *XSI*

```
pid_t setsid(void);
```

POSIX

- A process can determine its session ID by calling `getsid`
- `Setsid` sets the the process group ID and session ID of the caller to its process ID.

Session/Group ID Example

```
$ ls -l | sort -n +4 | grep testfile > testfile.out &
```

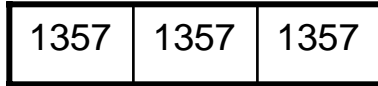
```
$ grep process | sort > process.out &
```

```
$ du . > du.out &
```

```
$ cat /etc/passwd | grep users | sort | head > users.out &
```

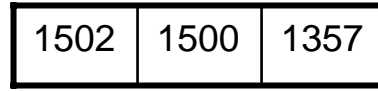
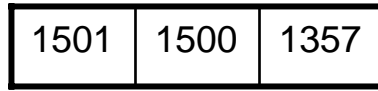
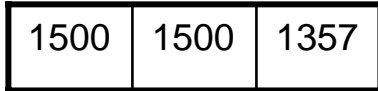
Session/Process Group Diagram

process
group
1357

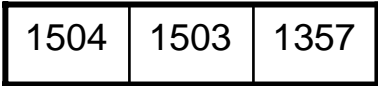
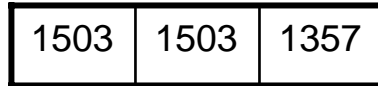


Shell

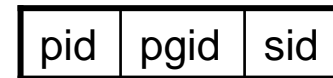
process
group
1500



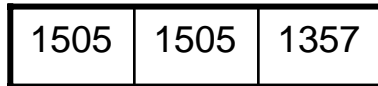
process
group
1503



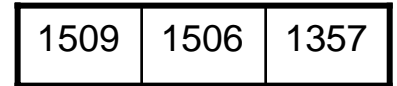
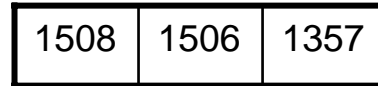
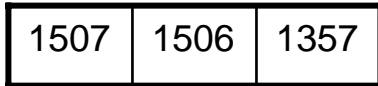
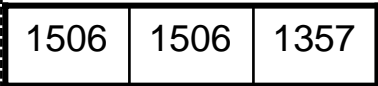
key



process
group
1505



process
group
1506



Background/Foreground Processes

- A session has a controlling terminal which is the controlling terminal of the shell
- At most one process group is in the foreground at a time
- All other processes are background processes
- Keyboard input and `^c` go only to foreground processes
- Screen output comes only from foreground processes

Job Control

A shell has job control if it allows users to move process groups from the background to the foreground and from the foreground to the background

tcgetpgrp/tcsetpgrp

SYNOPSIS

```
#include <unistd.h>
```

```
pid_t tcgetpgrp(int fildes);
```

```
int tcsetpgrp(int fildes, pid_t pgid);
```

POSIX

tcgetpgrp – returns the process group of the foreground process of a particular controlling terminal

tcsetpgrp – changes the process group associated with a controlling terminal fildes

ush5.c - Top

```
#include "ush.h"
static void jumphd(int signalnum) {
    if (!okaytojump) return;
    okaytojump = 0;
    siglongjmp(jumptoprompt, 1); }
int main (void) {
    char *backp;
    sigset_t blockmask;
    pid_t childpid;
    struct sigaction defhandler;
    int inbackground;
    char inbuf[MAX_CANON];
    int len;
    if (signalsetup(&defhandler, &blockmask, jumphd) == -1) {
        perror("Failed to set up shell signal handling");
        return 1; }
```

ush5.c - Middle

```
for( ; ; ) {
    if ((sigsetjmp(jumptoprompt, 1)) && /* if return from signal, \n */
        (fputs("\n", stdout) == EOF) )
        continue;
    okaytojump = 1;
    printf("%d", (int) getpid());
    if (fputs(PROMPT_STRING, stdout) == EOF) continue;
    if (fgets(inbuf, MAX_CANON, stdin) == NULL) continue;
    len = strlen(inbuf);
    if (inbuf[len - 1] == '\n') inbuf[len - 1] = 0;
    if (strcmp(inbuf, QUIT_STRING) == 0) break;
    if ((backp = strchr(inbuf, BACK_SYMBOL)) == NULL) inbackground =
0;
    else { inbackground = 1;
        *backp = 0; }
}
```

ush5.c - Bottom

```
if (sigprocmask(SIG_BLOCK, &blockmask, NULL) == -1)
    perror("Failed to block signals");
if ((childpid = fork()) == -1) perror("Failed to fork");
else if (childpid == 0) {
    if (inbackground && (setpgid(0, 0) == -1))
        return 1;
    if ((sigaction(SIGINT, &defhandler, NULL) == -1) ||
        (sigaction(SIGQUIT, &defhandler, NULL) == -1) ||
        (sigprocmask(SIG_UNBLOCK, &blockmask, NULL) == -1)) {
        perror("Failed to set signal handling for command ");
        return 1; }
    executecmd(inbuf);
    return 1; }
if (sigprocmask(SIG_UNBLOCK, &blockmask, NULL) == -1)
    perror("Failed to unblock signals");
if (!inbackground) /* only wait for child not in background */
    wait(NULL); }
return 0; }
```

ush5.c in Old Book – Top

```
#include "ush.h"
void main (void)
{
    char inbuf[MAX_CANON];
    pid_t child_pid;
    int inbackground;
    char *backp;

    for( ; ; ) {
        fputs(PROMPT_STRING, stdout);
        if (fgets(inbuf, MAX_CANON, stdin) == NULL)
            break;
        if (*(inbuf + strlen(inbuf) - 1) == NEWLINE_SYMBOL)
            *(inbuf + strlen(inbuf) - 1) = 0;
        if (strcmp(inbuf, QUIT_STRING) == 0)
            break;
    }
}
```

```
else {
    if ((backp = strchr(inbuf, BACK_SYMBOL)) ==
NULL)
        inbackground = FALSE;
    else {
        inbackground = TRUE;
        *(backp) = NULL_SYMBOL;
    }
    if ((child_pid = fork()) == 0) {
        if (inbackground)
            if (setpgid(getpid(), getpid()) == -1)
                exit(1);
        executecmdline(inbuf);
        exit(1);
    }
    else if (child_pid > 0 && !inbackground)
        waitpid(child_pid, NULL, 0);
    }
}
exit(0);
}
```

ush5.c in Old Book – Bottom

ush5.c Analysis

- Background processes are possible
- Terminated background processes do not get handled by parent or init since former parent (shell) never terminates
- Leaves zombies

ush6.c – Top

```
static void jumphd(int signalnum) {
    if (!okaytojump) return;
    okaytojump = 0;
    siglongjmp(jumptoprompt, 1); }
int main (void) {
    char *backp, inbuf[MAX_CANON];
    sigset_t blockmask;
    pid_t childpid;
    struct sigaction defhandler;
    int inbackground, len;
    if (signalsetup(&defhandler, &blockmask, jumphd) == -1) {
        perror("Failed to set up shell signal handling");
        return 1; }
```

ush6.c – Upper Middle

```
for( ; ; ) {
    if ((sigsetjmp(jumptoprompt, 1)) && /* if return from signal, \n */
        (fputs("\n", stdout) == EOF) ) continue;
    okaytojump = 1;
    printf("%d", (int) getpid());
    if (fputs(PROMPT_STRING, stdout) == EOF) continue;
    if (fgets(inbuf, MAX_CANON, stdin) == NULL) continue;
    len = strlen(inbuf);
    if (inbuf[len - 1] == '\n') inbuf[len - 1] = 0;
    if (strcmp(inbuf, QUIT_STRING) == 0) break;
    if ((backp = strchr(inbuf, BACK_SYMBOL)) == NULL) inbackground =
0;
    else { inbackground = 1;
          *backp = 0; }
}
```

ush6.c Lower Middle

```
if (sigprocmask(SIG_BLOCK, &blockmask, NULL) == -1)
    perror("Failed to block signals");
if ((childpid = fork()) == -1)
    perror("Failed to fork");
else if (childpid == 0) {
    if (inbackground) {          /* child creates another process */
        if ((childpid = fork()) == -1) {
            perror("Second fork failed");
            return 1; }
        if (childpid > 0) return 0;
        if (setpgid(0, 0) == -1) {
            perror("setpgod failed");
            return 1; }
    }
}
```

ush6.c – Bottom

```
if ((sigaction(SIGINT, &defhandler, NULL) < 0) ||
    (sigaction(SIGQUIT, &defhandler, NULL) < 0) ||
    (sigprocmask(SIG_UNBLOCK, &blockmask, NULL) == -1)) {
    perror("Failed to set signal handling for command ");
    return 1; }
executecmd(inbuf);
perror("Failed to execute command");
return 1;
}
if (sigprocmask(SIG_UNBLOCK, &blockmask, NULL) == -1)
    perror("Failed to unblock signals");
wait(NULL);
}
return 0;
```

ush6.c in Old Book - Top

```
#include "ush.h"
void main (void)
{  char inbuf[MAX_CANON];
   pid_t child_pid;
   char *backp;
   for( ; ; ) {
       fputs(PROMPT_STRING, stdout);
       if (fgets(inbuf, MAX_CANON, stdin) == NULL)
           break;
       if (*(inbuf + strlen(inbuf) - 1) == NEWLINE_SYMBOL)
           *(inbuf + strlen(inbuf) - 1) = 0;
       if (strcmp(inbuf, QUIT_STRING) == 0)
           break;
```

ush6.c Analysis

- If not in background, the child executes the command
- If in background,
 - Child forks a grandchild and then terminates
 - Grandchild executes the command
 - When grandchild terminates, the parent is terminated, so init handles grandchild (no zombies)

ush6.c in Old Book - Bottom

```
else if ((child_pid = fork()) == 0) {
    if ((backp = strchr(inbuf, BACK_SYMBOL)) != NULL) {
        *(backp) = NULL_SYMBOL; /* end command line
                                   before & */

        if (fork() != 0) exit(0);
        if (setpgid(getpid(), getpid()) == -1)
            exit(1);
    }
    executecmdline(inbuf);
    exit(1);
} else if (child_pid > 0)
    waitpid(child_pid, NULL, 0);
}
```

ush7.c in Old Book – Top

```
#include "ush.h"
void main (void)
{
    char inbuf[MAX_CANON];
    pid_t child_pid;
    pid_t wait_pid;
    char *backp;
    int inbackground;
    for( ; ; ) {
        fputs(PROMPT_STRING, stdout);
        if (fgets(inbuf, MAX_CANON, stdin) == NULL)
            break;
        if (*(inbuf + strlen(inbuf) - 1) == NEWLINE_SYMBOL)
            *(inbuf + strlen(inbuf) - 1) = 0;
        if (strcmp(inbuf, QUIT_STRING) == 0)
            break;
```

```

else {
    if ((backp = strchr(inbuf, BACK_SYMBOL)) ==
NULL)
        inbackground = FALSE;
    else { inbackground = TRUE;
        *(backp) = NULL_SYMBOL; }
    if ((child_pid = fork()) == 0) {
        if (inbackground)
            if (setpgid(getpid(), getpid()) == -1)
exit(1);
        executecmdline(inbuf);
        exit(1); }
    else if (child_pid > 0) {
        if (!inbackground)
            while((wait_pid = waitpid(-1, NULL, 0)) > 0)
                if (wait_pid == child_pid) break;
        while (waitpid(-1, NULL, WNOHANG) > 0); }
    }
}
exit(0);
}

```

ush7.c in Old Book – (bottom)

ush7.c Analysis

- Only one fork
- Child executes foreground processes and background processes
- `waitpid(childpid, NULL, 0)` handles foreground child
- `while waitpid(-1, NULL, WNOHANG)` handles background processes

Job Control – (1)

\$ a.out & ← a.out is an infinite loop

[1] 8694

\$ a.out &

[2] 8698

\$ a.out &

[3] 8708

\$ ps

PID	TTY	TIME	CMD
8694	pts/17	0:35	a.out
8698	pts/17	0:33	a.out
8708	pts/17	0:30	a.out
5889	pts/17	0:01	ksh

Job Control – (2)

\$ jobs

[1] + running a.out &

[2] - running a.out &

[3] running a.out &

\$ kill -KILL %2

[2] + killed

\$ jobs

[1] + running a.out &

[3] - running a.out &

\$ stop %3

\$ jobs

[3] + Stopped (SIGTSTP) a.out &

[1] - running a.out &

Job Control – (3)

```
$ bg %3
```

```
[3] a.out &
```

```
$ jobs
```

```
[3] + running a.out &
```

```
[1] - running a.out &
```

```
$ fg %1
```

```
a.out
```

```
← Type ^c here
```

```
$ jobs
```

```
[3] + running a.out &
```

```
$ kill -KILL %3
```

```
[3] + killed
```

```
$ jobs
```

```
← No jobs are left
```

Job Control – (4)

```
$ a.out
```

```
    ← a.out is running in the foreground – type ^z  
here
```

```
[1] + Stopped (SIGTSTP)a.out
```

```
$ bg
```

```
[1] a.out &
```

```
$ kill -KILL %1
```

```
[1] + Killed a.out &
```