

# CHAPTER VI: SIGNALS

B. Boufama

UNIVERSITY OF WINDSOR  
(Chapter 10 from your textbook)

## Introduction

Signals are a form of inter-process communication.

What is a signals? a software interrupt.

Most nontrivial applications must deal with signals that result from asynchronous events, such as :

- a CTR-C or CTR-Z
- a death of a child
- a floating-point error
- an alarm clock ring

When such an event occurs, the kernel sends an event-specific signal to the appropriate process.

Note: a process can send a signal to another process as long as it has permission.

## Signal concepts

Every signal has a name that begins with **SIG**.  
For example, *SIGABRT*, *SIGINT*, *SIGALRM*,...

These names are all defined by positive integer constants in *< sys/signal.h >*

Note that the programmer may choose for a particular signal to

- trigger a user-supplied *signal handler* or,
- trigger the default kernel-supplied *handler* or,
- be ignored, this works for most signals except the two signals: *SIGKILL* and *SIGSTOP* that cannot be ignored.

The default handler usually performs one of the followings :

- terminates the process and generates a core file(*dump*). In this case, a memory image of the process is saved in the file *core*. The latter can be used by a debugger to find out the state of the process at the termination time.
- terminates the process without generating a core file(*quit*).
- ignores signal(*ignore*).
- suspends process(*suspend*)
- resumes process. If the process was stopped, the default action is to continue the process, otherwise, the signal is ignored

List of signals(page 316 on the book)
---------------------------------------

Name	Value	Action	Description
SIGHUP	1	quit	hang-up
SIGINT	2	quit	interrupt
SIGQUIT	3	dump	quit
SIGILL	4	dump	invalid instruction
SIGTRAP	5	dump	trace trap(used by debuggers)
SIGABRT	6	dump	abort
SIGEMT	7	dump	emulator trap instruction
SIGFPE	8	dump	arithmetic exception
SIGKILL	9	quit	kill (fatal)
SIGBUS	10	dump	bus error (bad format address)
SIGSEGV	11	dump	segmentation fault
SIGSYS	12	dump	bad argument to system call
SIGPIPE	13	quit	pipe/socket error
SIGALRM	14	quit	alarm clock
SIGTERM	15	quit	termination signal

Name	Value	Action	Description
SIGUSR1	16	quit	user signal 1
SIGUSR2	17	quit	user signal 2
SIGCHLD	18	ignore	child status changed
SIGPWR	19	ignore	power fail
SIGWINCH	20	ignore	window size change
SIGURG	21	ignore	urgent socket condition
SIGPOLL	22	exit	pollable event
SIGSTOP	23	suspend	stops a process
SIGSTP	24	suspend	interactive stop signal(CTR-Z)
SIGCONT	25	ignore	Continue executing if stopped
SIGTTIN	26	quit	Background process attempt read
SIGTTOU	27	quit	Background process attempt write
SIGVTALARM	28	quit	timer expired
SIGPROF	29	quit	profiling timer expired
SIGXCPU	30	dump	CPU time limit exceeded
SIGXFSZ	31	dump	file size limit exceeded

### Example 1: Requesting an alarm signal

Synopsis: **unsigned int alarm(unsigned int n)**

Asks the kernel to send a SIGALRM to the calling process after  $n$  seconds.

If a previous alarm was scheduled, it will be overwritten.

If  $n=0$ , any pending alarm will be canceled

```
#include <stdio.h>
```

```
int main(int argc, char *argv[]){
```

```
    alarm(4);
```

```
    while(1){
```

```
        printf("Going to sleep\n");
```

```
        sleep(1);
```

```
    }
```

```
    printf("Exiting\n"); // will never get executed
```

```
    exit(0);
```

```
}
```

## Handling signals : `signal()` system call

A process can use the *signal()* system call to specify what to do when a particular signal is received. This system call has two arguments :

- *int signo* : the signal number to be reprogrammed, a constant from the previous table.
- *void (\*func)(int)* : the value of this argument is either
  - the constant `SIG_IGN` → ignore the signal *signo* (`SIGKILL` and `SIGSTOP` cannot be ignored) or,
  - the constant `SIG_DFL` → use default handler or,
  - the address of a user-defined function → use this function when the signal *signo* arrives.



What does the *signal()* system call exactly return?

- If successful, *signal()* returns the previous *func()* value associated with *signo* (the previous handler) or,
- *SIG\_ERR* (-1) otherwise.

## Example 2: ignoring CTR-C and CTR-Z

```
int main(int argc, char *argv[]){
    int i;
    void (*oldHandler1)(); //to save default handlers
    void (*oldHandler2)(); //for CTR-C and CTR-Z

    oldHandler1=signal(SIGINT, SIG_IGN); //ignore CTR-C
    oldHandler2=signal(SIGTSTP, SIG_IGN); //ignore CTR-Z
    for(i=1; i<=10; i++){
        printf("I am not sensitive to CTR-C/CTR-Z\n");
        sleep(1);
    }
    signal(SIGINT, oldHandler1); // restore default
    signal(SIGTSTP, oldHandler2); // restore default
    for(i=1; i<=10; i++){
        printf("I am sensitive to CTR-C/CTR-Z\n");
        sleep(1);
    }
}
```

### Ex. 3: replacing a default handler by our own

```
void myAlarmHandler(){
    printf("I got an alarm, I took care of it\n");
    alarm(3);
}

int main(int argc, char *argv[]){

    signal(SIGALRM, myAlarmHandler); //install handler
    alarm(3);

    while(1){
        printf("I am working\n");
        sleep(1);
    }
}
```

## System call `pause()`

### **Synopsis :** `int pause(void);`

The *pause()* suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

*pause()* is typically used to wait for an alarm signal.

### **Example 4 :**

```
void AlarmHandler(){
    static int n = 0;
    if(n++ < 6 ){
        printf("Beeping%c%c%c\n", 7, 7, 7);
        alarm(10);
    }else
        exit(0);
}

int main(int argc, char *argv[]){
    signal(SIGALRM, AlarmHandler); //install a handler
    AlarmHandler();
    while(1) pause();
}
```

## System call `kill()` : sending a signal

**Synopsis :** `int kill(pid_t pid, int signo);`

*kill()* sends the signal *signo* to a process or a group of processes, defined by *pid*.

The signal is sent only when at least one of the following conditions is satisfied:

- The sending and receiving processes have the same owner.
- The sending process is owned by a super-user.

Example :

`kill(2344, SIGTERM);`

To terminate process 2344.

The *pid* parameter in *kill()* can take several values with different meanings:

- If *pid*  $> 1$ , the signal is sent to the process with id *pid*.
- If *pid* is 0, the signal is sent to all processes in the sender's process group.
- If *pid* is -1 then
  - if the sender is owned by a super-user, the signal is sent to all processes, including the sender.
  - otherwise, the signal is sent to all processes owned by the same owner as the sender.
- If *pid* is equal to  $-n$ , with  $n > 1$ , the signal is sent to all processes with a process group-id equal to  $n$ .

### **Example 5 :** Parent-Child synchronization I

```
void myAlarmHandler(){}; // to avoid quitting
int main(int argc, char *argv[]){
    pid_t pid;

    if((pid=fork()) > 0){
        printf("My child should wait until I am done\n");
        sleep(4);
        printf("Child, now you can do your job\n");
        kill(pid, SIGALRM); // let the child wake up
        printf("Parent Exiting\n");
    }else{
        printf("I have to wait for my parent\n");
        signal(SIGALRM, myAlarmHandler);
        pause();
        printf("OK, now I can do my job\n");
        sleep(2);
        printf("Child Exiting\n");
    }
    exit(0);
}
```

## Example 6 : Parent-Child synchronization II

```
void action(){
    sleep(2);
    printf("Switching\n");
}

int main(int argc, char *argv[]){
    pid_t pid;
    if((pid=fork())>0){
        signal(SIGUSR1, action);
        while(1){
            printf("Parent is running\n");
            kill(pid, SIGUSR1);
            pause();
        }
    }else{
        signal(SIGUSR1, action);
        while(1){
            pause();
            printf("Child is running\n");
            kill(getppid(), SIGUSR1);
        }
    }
}
```



```
}  
}
```

**Example 7:** a non-blocking wait()

```
void childDeath(){  
    pid_t pid;  
    int status;  
    pid = wait(&status);  
    printf("Child %d has terminated\n", pid);  
}  
  
int main(int argc, char *argv[]){  
    pid_t pid;  
    signal(SIGCHLD, childDeath);  
    while(1){  
        if((pid=fork())==0){  
            sleep(random()%20);  
            exit(0);  
        }  
        printf("Created a child, pid=%d\n", pid);  
        sleep(2);  
    }  
}
```

## Process groups and control terminal

Unix organizes processes as follows :

- Every process is a member of a *process group*. In particular, a child inherits its *process group* from its parent. When a process *execs*, its *process group* remains the same. However, a process may change its *process group* to a new value using *setpgid()*.
- Every process may have an associated *control terminal*. In particular, a child inherits its *control terminal* from its parent. When a process *execs*, its *control terminal* remains the same.
- Every terminal is associated with a *control process*, the piece of software that manages the terminal. For example, when CTR-C is typed, the terminal *control process* will send a *SIGINT* to all processes in the *process group* of its *control process*.

The shell uses these features as follows :

- When an interactive shell starts, it is the control process of its control terminal.
- When a shell executes a foreground process, the child shell changes its *group process* number to its *pid*, *execs* the command and, takes control of the terminal.  
—→ Signals generated from the terminal go to the command and not to the parent shell.  
When the command terminates, the parent shell takes back the control of the terminal.
- When a shell executes a background process, the child shell changes its *group process* number to its *pid* then *execs* the command. However, it does not take control of the terminal.  
—→ Signals generated from the terminal go to the original parent shell.

## System call : `setpgid()`

**Synopsis:** `pid_t setpgid(pid_t pid, pid_t pgid)`

*setpgid()* sets the process group ID of the process, whose ID is *pid*, to *pgid*.

- If *pgid* is equal to *pid*, the process becomes a process group leader.
- If *pid* is equal to 0, the calling process group ID is set to *pgid*.

*setpgid()* returns 0 if successful and -1 otherwise.

Note that *setpgid()* succeeds only when at least one of the following conditions is satisfied:

- The caller and the specified processes have the same owner.
- The caller process is owned by a super-user.

## System call: `getpgid()`

### Synopsis: `pid_t getpgid(pid_t pid)`

*getpgid()* returns the process group ID of the process with PID equal to *pid*.

If *pid* is 0, the calling process group ID is returned.

Ex.8: CTR-C goes to all processes in a process group

```
void CTR_handler(){
    printf("Process %d got a CTR-C, exit\n", getpid());
    exit(2);
}

int main(int argc, char *argv[]){
    int i;
    printf("First process, PID=%d, PPID=%d, PGID=%d\n",
        getpid(), getppid(), getpgid(0));
    signal(SIGINT, CTR_handler);
    for(i=1; i<=3; i++)
        fork();
    printf("PID=%d PGID=%d\n", getpid(), getpgid(0));
    pause();
}
```

## Outputs:

```
First process, PID=5257, PPID=393, PGID=5257
My PID=5260, my PGID=5257
My PID=5259, my PGID=5257
My PID=5262, my PGID=5257
My PID=5263, my PGID=5257
My PID=5258, my PGID=5257
My PID=5261, my PGID=5257
My PID=5264, my PGID=5257
My PID=5257, my PGID=5257
^CProcess5263 got a CTR-C
I am exiting
Process5262 got a CTR-C
I am exiting
Process5260 got a CTR-C
I am exiting
Process5259 got a CTR-C
I am exiting
Process5264 got a CTR-C
I am exiting
Process5261 got a CTR-C
```

## Chapter VI: Signals

```
I am exiting  
Process5258 got a CTR-C  
I am exiting  
Process5257 got a CTR-C  
I am exiting
```

### Example 9: Changing process group ID

```
void CTR_handler(){
    printf("Process%d got CTR-C, exiting\n", getpid());
    exit(0);
}

int main(int argc, char *argv[]){
    int i, pid;

    signal(SIGINT, CTR_handler);
    if((pid=fork()) == 0){
        setpgid(0, getpid()); //child is in its own group
        printf("PID=%d, PGID=%d\n",getpid(),getpgid(0));
    }else
        printf("PID=%d, PGID=%d\n", getpid(), getpgid(0));

    for(i=1; i<=10; i++){
        printf("Process %d is still alive\n", getpid());
        sleep(2);
    }
}
```



### Example 10: Changing process group ID

```
void TTIN_handler(){
    printf("Attempted to read from keyboard\n");
    exit(22);
}

int main(int argc, char *argv[]){
    int i, status, pid;

    if(!(pid=fork())){
        signal(SIGTTIN, TTIN_handler);
        setpgid(0, getpid());
        printf("Enter a value> \n");
        scanf("%d", &i);
    }else{
        wait(&status);
        if(WIFEXITED(status))
            printf("Exit status=%d\n",WEXITSTATUS(status));
        else
            printf("signaled by =%d\n",WTERMSIG(status));
    }
}
```

```
void action(){};    // Example 11: two-player game
void child(char *);
int main(int argc, char *argv[]){
    pid_t pid1, pid2;

    printf("This is a 2-players game with a referee\n");
    if((pid1=fork()) == 0) child("TOT0");
    if((pid2=fork()) == 0) child("TITI");

    sleep(1);
    signal(SIGUSR1, action);
    while(1){
        printf("\nReferee: TOT0 plays\n\n");
        kill(pid1, SIGUSR1);
        pause();
        printf("\n\nReferee: TITI plays\n\n");
        kill(pid2, SIGUSR1);
        pause();
    }
}
```

```
void child(char *s){
```

```
int points=0;
int dice;
long int ss=0;

while(1){
    signal(SIGUSR1, action); // block myself
    pause();
    printf("%s: playing my dice\n", s);
    dice =(int) time(&ss)%10 + 1;
    printf("%s: got %d points\n", s, dice);
    points+=dice;
    printf("%s: Total so far %d\n\n", s, points);
    sleep(3);
    if(points >= 50){
        printf("%s: game over I won\n", s);
        kill(0, SIGTERM);
    }
    kill(getppid(), SIGUSR1);
}
}
```