CHAPTER VI: SIGNALS

B. Boufama

University of Windsor

Introduction

Signals are used for inter-process communication.

What is a signal?
a software interrupt that results from an asynchronous event, 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 the concerned process an event-specific signal.
A process can also send a signal to another process as long as it has permission. For instance, a parent process can send a 'kill' signal to its child.

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 $< sys/signal.h >$

For a particular signal, one may choose to

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

A default handler 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*, that 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*) **temporarily hanged or stopped**
- resumes process. If a process was stopped, the default action is to continue the process, otherwise, the signal is ignored

List of signals

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

positive integers and 0

Synopsis: **unsigned int alarm(unsigned int n)**
Asks the kernel to send a SIGALRM to the calling process after n seconds.

A previously scheduled alarm would be overwritten.
When n=*0*, any pending alarm will be canceled

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

  alarm(4);
  while(1){
    printf("Sleep for a second\n");
    sleep(1);
  }
  printf("Bye Bye, exiting\n");
  exit(0);
}
```

It wont actually take 1 second to print so we cannot assume 1 second for print operation. O/p will be Sleep for a second will be printed 4 times and then alarm will beep and exit the program.

This part wont be printed

Handling signals : signal() system call

System call *signal()* allows to specify what to do for a particular signal.

Part highlighted in yellow is the return type(void) of the method and parameter(int) that the method takes, this method's address is what signal method will return. which is the pointer to an old handler function

Synopsis :

void(*signal(int signo,void(*func)(int)))(int)

pointer to new handler function or default handler function

This system call has two arguments :

- *int signo* : the concerned signal number,
- *void (*func)(int)* : the value of this argument is either
  - constant SIG_IGN → ignore the signal *signo* or,
  - the constant SIG_DFL → use default handler or,
  - A user-defined function address → call this function when signal *signo* arrives.

About the prototype of *signal()*:
void (*signal(...))$_1$(*int*)$_2$;

- The pair $(*\ )_1$ is there because *signal()* returns a pointer to function.
- $(int)_2$ is there because *signal()* returns a function that takes one parameter of type *int*.

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

- If successful, *signal()* returns the address of the previous *handler()* associated with *signo* or,
- *SIG_ERR* (-1) otherwise.

Example 2: ignoring CTR-C and CTR-Z

```
int main(int argc, char *argv[]){
  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(int 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(int i=1; i<=10; i++){
    printf("I am sensitive to CTR-C/CTR-Z\n");
    sleep(1);
  }
}
```

pointer to SIGINT default handler will be returned by signal and stored in oldHandler1 and SIGINT is replaced with a new handler function which is SIG_ING

Due to first for loop for 10 seconds ctrl z and ctrl c will be ignored as the handlers have been changed, after which it will be restored from these two lines and next for loop will be sensitive to ctrl c and ctrl z

Ex. 3: replacing a default handler by our own

This alarm will trigger the myAlarmHandler function which will not exit like the first alarm example we saw, as alarm is called again inside the handler function making it to run forever.

```
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);                       // for first time

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

We are not giving &function_name becoz this is similar to arrays, if we just give the function name it will act as a pointer to the actual function.

new handler

Handling signals : sigaction() system call

The *signal()* function is the simplest interface to the signal features of Unix.
However, its semantics is not well defined among different implementations.
$\Rightarrow$ either *sigaction()* or an implementation of *signal()* using *sigaction()* should be used instead.

Synopsis :
int sigaction(int signo,const struct sigaction* act, const struct sigaction *oact)
where

- *signo* is the signal number, like in *signal()*
- if *act* is nonnull, we are modifying the action.
- if *oact* is nonnull, the system returns the previous action for the signal through *oact*.

The *the sigaction* structure is defined as

```
struct sigaction{
 void (*sa_handler)(int);
//handler address or SIG_IGN or SIG_DFL

 sigset_t sa_mask;   // additional signals to block
 int      sa_flag;   // signal options (page 351)
                     // alternate handler
 void (*sa_sigaction)(int, siginfo_t *, void *);
}
```

sa_mask specifies the set of to-be-blocked signals to add to the process signal mask, before the handler is called. The signal mask is reset after the the return of the handler.
Example:

```
sigemptyset(&act.sa_mask);
sigaddset(&act.sa_mask, SIGUSR1);
sigaddset(&act.sa_mask, SIGINT);
```

sa_flag specifies options for the signal handling.
Examples:

- *SA_INTERRUPT*: system calls interrupted by this signal are not automatically restarted.
  (option defined for Linux only, default for the others.)
- *SA_RESTART*: system calls interrupted by this signal are automatically restarted.
- *SA_SIGINFO*: provides more information to the handler, i.e., a pointer to *siginfo* structure and a pointer to an identifier for the process context.
  Note: when *SA_SIGINFO* is used, the alternate handler is invoked instead of the handler.

Here is a reliable version of *signal()*, using *sigaction()*.
Normally, this is already done for you by many platforms.

```
void (*signal(int signo, void(*func)(int)))(int){
  struct sigaction act, oact;

  act.sa_handler = func;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  if(signo == SIGALRM){
#ifdef SA_INTERRUPT
       act.sa_flags = act.sa_flags | SA_INTERRUPT;
#endif
  } else
       act.sa_flags = act.sa_flags | SA_RESTART;

  if (sigaction(signo, &act, &oact) < 0)
       return(SIG_ERR);

  return(oact.sa_handler);
}
```

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);
The *kill()* function 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 recieving 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 quiting
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 fellows :

- 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 sends 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.
  $\longrightarrow$ 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.
  $\longrightarrow$ 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
```

```
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("TOTO");
  if((pid2=fork()) == 0) child("TITI");

  sleep(1);
 signal(SIGUSR1, action);
  while(1){
    printf("\nReferee: TOTO 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, 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);
  }
}
```