COMP 8567 Advanced Systems Programming

Signals

Winter 2023

Outline

- Introduction
- Signal Concepts
- List of Signals
- **alarm()** system call
- Handling Signals: The **signal()** system call
- **pause()** system call
- **kill()** system call
- Process groups and control terminals
- **setpgid()** system call
- **getpgid()** system call
- Summary

Introduction..

- ctrlc.c

Introduction

Signals are used for inter-process communication.

Pipes, Sockets, Shared memory etc. can also be used for IPC

What is a signal?

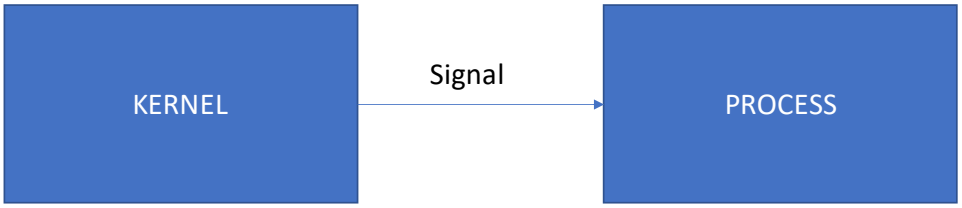
A **software interrupt** that results from an asynchronous event, such as :

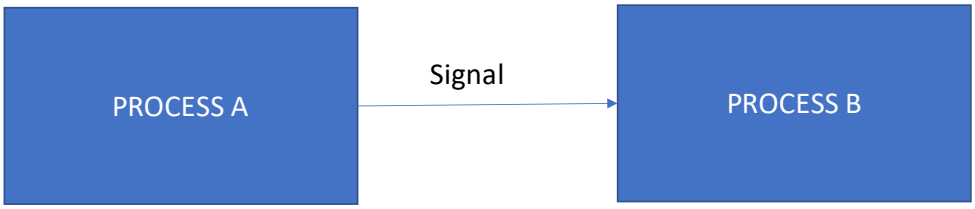
- a CTR-C or CTR-Z
- a death of a child process
- a floating-point error
- a system alarm clock going off

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 example, a parent process can send a 'kill' signal to its child process





A process can also 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 >**

For a particular signal, one may choose to:

- **Option 1:** Trigger the default **kernel-supplied handler** or,
- **Option 2:** Trigger a **user-supplied signal handler** or,
- **Option 3:** Ignore it, (ignoring works for all signals except for **SIGKILL** and **SIGSTOP**, that cannot be ignored)

Default Handler

A default (Kernel supplied) handler performs one of the following :

- **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** the signal **(ignore).**

- **Suspends** the process **(suspend)**

- **Resumes** a process. If a process was stopped, the default action is to continue the process, otherwise, the signal is ignored

List of signals and the corresponding integer values ($kill –L)

pranga@charlie:~/chapter6$ kill –L

1) SIGHUP	**2) SIGINT**	**3) SIGQUIT**	**4) SIGILL**	**5) SIGTRAP**
6) SIGABRT	**7) SIGBUS**	**8) SIGFPE**	**9) SIGKILL**	**10) SIGUSR1**
11) SIGSEGV	**12) SIGUSR2**	**13) SIGPIPE**	**14) SIGALRM**	**15) SIGTERM**
16) SIGSTKFLT	**17) SIGCHLD**	**18) SIGCONT**	**19) SIGSTOP**	**20) SIGTSTP**
21) SIGTTIN	**22) SIGTTOU**	**23) SIGURG**	**24) SIGXCPU**	**25) SIGXFSZ**
26) SIGVTALRM	**27) SIGPROF**	**28) SIGWINCH**	**29) SIGIO**	**30) SIGPWR**
31) SIGSYS	34) SIGRTMIN	35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3
38) SIGRTMIN+4	39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12	47) SIGRTMIN+13
48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14	51) SIGRTMAX-13	52) SIGRTMAX-12
53) SIGRTMAX-11	54) SIGRTMAX-10	55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7
58) SIGRTMAX-6	59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX			

Note: Signals 32 and 33 are not used anymore in Linux

#	Signal Name	Default Action	Comment	POSIX
1	SIGHUP	Terminate	Hang up controlling terminal or process	Yes
2	SIGINT	Terminate	Interrupt from keyboard, Control-C	Yes
3	SIGQUIT	Dump	Quit from keyboard, Control-\	Yes
4	SIGILL	Dump	Illegal instruction	Yes
5	SIGTRAP	Dump	Breakpoint for debugging	No
6	SIGABRT	Dump	Abnormal termination	Yes
6	SIGIOT	Dump	Equivalent to SIGABRT	No
7	SIGBUS	Dump	Bus error	No
8	SIGFPE	Dump	Floating-point exception	Yes
9	SIGKILL	Terminate	Forced-process termination	Yes
10	SIGUSR1	Terminate	Available to processes	Yes
11	SIGSEGV	Dump	Invalid memory reference	Yes
12	SIGUSR2	Terminate	Available to processes	Yes
13	SIGPIPE	Terminate	Write to pipe with no readers	Yes
14	SIGALRM	Terminate	Real-timer clock	Yes
15	SIGTERM	Terminate	Process termination	Yes
16	SIGSTKFLT	Terminate	Coprocessor stack error	No

17	SIGCHLD	Ignore	Child process stopped or terminated or got a signal if traced	Yes
18	SIGCONT	Continue	Resume execution, if stopped	Yes
19	SIGSTOP	Stop	Stop process execution, Ctrl-Z	Yes
20	SIGTSTP	Stop	Stop process issued from tty	Yes
21	SIGTTIN	Stop	Background process requires input	Yes
22	SIGTTOU	Stop	Background process requires output	Yes
23	SIGURG	Ignore	Urgent condition on socket	No
24	SIGXCPU	Dump	CPU time limit exceeded	No
25	SIGXFSZ	Dump	File size limit exceeded	No
26	SIGVTALRM	Terminate	Virtual timer clock	No
27	SIGPROF	Terminate	Profile timer clock	No
28	SIGWINCH	Ignore	Window resizing	No
29	SIGIO	Terminate	I/O now possible	No
29	SIGPOLL	Terminate	Equivalent to SIGIO	No
30	SIGPWR	Terminate	Power supply failure	No
31	SIGSYS	Dump	Bad system call	No
31	SIGUNUSED	Dump	Equivalent to SIGSYS	No

alarm() system call: 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.
- A previously scheduled alarm would be overwritten.
- When *n=0,* all pending alarm/s will be canceled

```
alarm() returns the number of seconds remaining until
any previously scheduled alarm was due to be delivered,
or zero if there was no previously scheduled alarm.
```

Ex1.c Requesting an alarm signal

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>
int main(int argc, char *argv[]){
//After 4 seconds, SIGALRM is sent by the kernel to the process
//The default action is to quit
alarm(4);
while(1)
{
printf("Sleep for a second\n");
sleep(1);
}
printf("Exiting on Alarm\n");
exit(0);
}
```

Handling Signals: signal() system call

System call signal() allows to specify **the action** for a particular signal.

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signo, sighandler_t handler);
```

This system call has **two** arguments :

- **int signo** : the concerned signal number
- **void (*func)(int)** : the value of this argument is either
  - **SIG_IGN (system defined handler)** : ignore the signal *signo* or,
  - **SIG_DFL (system define handler)** : use default(system specified) handler or
  - **Address of a user-defined function/handler** : This function is invoked when signal *signo* arrives.
    - In this case, the function/handler is called with argument *signo*
- Note: SIG_IGN and SIG_DEL are **NOT** signals, but are **system defined handlers**

signal() ..

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signo, sighandler_t handler);
```

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

ex2.c Ignoring CTR-C

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>


int main(int argc, char *argv[]){
void (*oldHandler1)(int);  //to save default handler for CTR-C
oldHandler1=signal(SIGINT,SIG_IGN);  //ignore CTR-C
for(int i=1; i<=10; i++){
printf("I am not sensitive to CTR-C\n");
sleep(1);
}

signal(SIGINT, oldHandler1);                // restore default SIG_DFL
for(int i=1; i<=10; i++){
printf("I am sensitive to CTR-C\n");
sleep(1);
}
}
```

ex3.c Replacing a default handler with a user-defined handler

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>
//Replaces the default handler with a user-defined handler
void myAlarmHandler(int signo){
printf("Received the alarm signal %d, returning to main\n“,signo);
//Additional processing can be implemented if required
}
int main(int argc, char *argv[]){
//install handler
signal(SIGALRM, myAlarmHandler);
//System alarm goes off 5 seconds from this point
alarm(5);
while(1){
printf("I am working\n");
sleep(1);
}}
```

System Call: pause()

- Synopsis : **int pause(void);**
- The pause() system call **suspends** the **calling process** until it receives a signal (any signal)
  - i.e any signal that is **not currently set to be ignored** by the calling process.
- pause() is typically used to wait for an alarm signal
- **pause**() returns only when a signal was caught and the signal- catching function returned successfully after execution.
  - In this case, **pause**() returns -1

//ex4.c pause()

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>
void AlarmHandler(){
printf("\nIn the alarm handler\n");
}
int main(int argc, char *argv[])
{
signal(SIGALRM, AlarmHandler); //install the handler
alarm(5);
printf("\nThe system is pausing\n");
int i=pause();
printf("\nThe process has resumed after pause\n");
printf("\nThe return value of pause() is %d\n",i);
}
```

System Call: kill()

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

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

- On success (at least one signal was sent), zero is returned.
- On error, -1 is returned

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

- The sending and receiving processes have the **same owner (user ex:pranga)**.
- The sending process is owned by a **super-user.**

Example :

kill(2344, SIGTERM);

To terminate process 2344.

kill()..

The pid parameter in **int kill(pid_t pid, int signo)** 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 process 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 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

ex5b.c //show ckp.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>

int main(int argc, char *argv[]){
pid_t pid;
int p=getpid();

if((pid=fork()) > 0){ //Parent Process
int i=0;
for(;;){
printf("Parent process id is %d\n", getpid());
if(i==5){
printf("\n The child process will now be killed\n"); //killed after 10 seconds
kill(pid,SIGINT);
}
sleep(2);
i=i+1;
}
}
else{ //Child Process
int k=0;
for(;;)
{
printf("Child process id is %d\n", getpid());
sleep(2);

}
}}
```

ex6.c //Kill with SIGINT

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>

void  action(){
sleep(2);
printf("Switching\n");
}
int main(int argc, char *argv[]){
pid_t pid;
if((pid=fork())>0){  //Parent
signal(SIGINT, action);
while(1){
printf("Parent  is running\n");
kill(pid, SIGINT);
pause();}}
else{ //Child
signal(SIGUSR1,  action);
while(1){
pause();
printf("Child  is running\n");
kill(getppid(), SIGINT);}}}
```

Can we use any other signal to implement the outcome of program(ex6.c) ?

ex6a.c
ex6b.c

Note:

- **Kill(pid_t pid, int signo)** can be used to send **any signal** to a process or a group of processes as indicated by the value of pid

Process Groups and Controls

Unix organizes processes as follows :

Every process is a member of a process group.

- 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 has an associated control terminal.

- 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 (All the processes which has the same groupid of that of the terminal 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 (fork and exec):**
  - 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 (ex ctrl+c)
  - When the command/executable terminates, the parent shell takes back the control of the terminal.
  - i.e when a foreground process is executed from a shell, the control of the terminal is transferred to the foreground process from the shell
- 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.

  (ctrl + c will have no effect on the background process, but will have an effect on the foreground process)

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.
- Returns -1 if unsuccessful

ex8.c CTR-C goes to all the processes in the process group

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>

//CTR-C goes to all processes in the process group
void CTR_handler(){
printf("Process %d received 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));
//getpgid(0) 0 indicates that the calling process' pgid will be returned
signal(SIGINT, CTR_handler); //Register the handler
for(i=1; i<=4; i++)
fork();
printf("PID=%d PGID= %d\n", getpid(), getpgid(0));
pause();
}
```

ex9.c Changing Process Group ID-1

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>
//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); //register the handler
if((pid=fork()) == 0)
{
setpgid(0, getpid());//child is in its own group
printf("Child PID= %d, PGID= %d\n",getpid(),getpgid(0));
}
else
printf("Parent PID= %d, PGID=%d\n", getpid(), getpgid(0));
for(i=1; i<=5; i++){
printf("Process %d is still alive\n", getpid());
sleep(2);
}}
```

ex10.c Changing Process Group ID-2

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>
void TTIN_handler(){
printf("Attempted to read from keyboard\n");
exit(24);
}
int main(int argc, char *argv[])
{
int i, status, pid;
if(!(pid=fork()))//child{
signal(SIGTTIN, TTIN_handler);
setpgid(0, getpid());
printf("Enter a value> \n");
scanf("%d", &i);
}
else //parent{
wait(&status);
if(WIFEXITED(status))
printf("Exit status= %d\n",WEXITSTATUS(status));
else
printf("signaled by = %d\n",WTERMSIG(status));
}}
```

Summary

- Introduction
- Signal Concepts
- List of Signals
- Handling Signals: The **signal()** system call
- **pause()** system call
- **kill()** system call
- Process groups and control terminals
- **setpgid()** system call
- **getpgid()** system call

THANK YOU

Appendix

- Additional examples

```
#include <stdio.h>   //ex3b.c
#include <stdlib.h>
#include <sys/signal.h>

void myAlarmHandler(int signum){  //void (* handlername) (void or int)
printf("I got a signal %d, I took care of it\n",signum);

}

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);
}
}
```