

CS 332/532 Systems Programming

Lecture 25
Signals / 2

Professor : Mahmut Unan – UAB CS

Exam 2

- Friday
 - Review
- Monday
 - Exam 2

Agenda

- Signals
- Signal Handling

signal()

- signal - ANSI C signal handling

-

```
typedef void (*sighandler_t) (int);
```

- `sighandler_t signal(int signum, sighandler_t handler);`

- **signal()** sets the disposition of the signal *signum* to *handler*, which is either **SIG_IGN**, **SIG_DFL**, or the address of a programmer-defined function (a "signal handler").

Linux signal handling

- Perform the default action. Most signals have a default action associated with them, if the process does not change the default behavior then the default action specific to that particular signal will occur.
 - The predefined default signal is specified as `SIG_DFL`.
- Ignore the signal. If a signal is ignored, then the default action is performed.
 - The predefined ignore signal handler is specified as `SIG_IGN`.
- Catch and Handle the signal. It is also possible to override the default action and invoke a specific user-defined task when a signal is received.
 - This is usually performed by invoking a signal handler using `signal()` or `sigaction()` system calls.
- However, the signals `SIGKILL` and `SIGSTOP` cannot be caught, blocked, or ignored.

Default Action Of Signals

- Each signal has a default action, one of the following:
 - **Term:** The process will terminate.
 - Core:** The process will terminate and produce a core dump file.
 - Ign:** The process will ignore the signal.
 - Stop:** The process will stop.
 - Cont:** The process will continue from being stopped.
- Default action may be changed using handler function. Some signal's default action cannot be changed. **SIGKILL** and **SIGABRT** signal's default action cannot be changed or ignored
- https://linuxhint.com/signal_handlers_c_programming_language/

Basic Signal handler examples

```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
void sig_handler(int signum){
    //Return type of the handler function should be void
    printf("\nInside handler function\n");
}

int main(){
    signal(SIGINT,sig_handler); // Register signal handler
    for(int i=1;;i++){          //Infinite loop
        printf("%d : Inside main function\n",i);
        sleep(1); // Delay for 1 second
    }
    return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example1.c -o Example1
tump@tump:~/Desktop/c_prog/signal$ ./Example1
1 : Inside main function
2 : Inside main function
^C
Inside handler function
3 : Inside main function
4 : Inside main function
^\\Quit (core dumped)
tump@tump:~/Desktop/c_prog/signal$
```

```
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
int main(){
    signal(SIGINT,SIG_IGN); // Register signal handler for ignoring the signal

    for(int i=1;;i++){        //Infinite loop
        printf("%d : Inside main function\n",i);
        sleep(1); // Delay for 1 second
    }
    return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example2.c -o Example2
tump@tump:~/Desktop/c_prog/signal$ ./Example2
1 : Inside main function
2 : Inside main function
^C3 : Inside main function
4 : Inside main function
^C5 : Inside main function
^\\Quit (core dumped)
tump@tump:~/Desktop/c_prog/signal$
```



```

#include<stdio.h>
#include<unistd.h>
#include<signal.h>

void sig_handler(int signum){
    printf("\nInside handler function\n");
    signal(SIGINT,SIG_DFL);    // Re Register signal handler for default action
}

int main(){
    signal(SIGINT,sig_handler); // Register signal handler
    for(int i=1;;i++){        //Infinite loop
        printf("%d : Inside main function\n",i);
        sleep(1);    // Delay for 1 second
    }
    return 0;
}

```

```

tump@tump:~/Desktop/c_prog/signal$ gcc Example3.c -o Example3
tump@tump:~/Desktop/c_prog/signal$ ./Example3
1 : Inside main function
2 : Inside main function
3 : Inside main function
^C
Inside handler function
4 : Inside main function
^C
tump@tump:~/Desktop/c_prog/signal$

```

```

#include<stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include<signal.h>
void sig_handler_parent(int signum){
    printf("Parent : Received a response signal from child \n");
}

void sig_handler_child(int signum){
    printf("Child : Received a signal from parent \n");
    sleep(1);
    kill(getppid(),SIGUSR1);
}

int main(){
    pid_t pid;
    if((pid=fork())<0){
        printf("Fork Failed\n");
        exit(1);
    }
    /* Child Process */
    else if(pid==0){
        signal(SIGUSR1,sig_handler_child); // Register signal handler
        printf("Child: waiting for signal\n");
        pause();
    }
    /* Parent Process */
    else{
        signal(SIGUSR1,sig_handler_parent); // Register signal handler
        sleep(1);
        printf("Parent: sending signal to Child\n");
        kill(pid,SIGUSR1);
        printf("Parent: waiting for response\n");
        pause();
    }
    return 0;
}

```

```

#include<stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include<signal.h>
void sig_handler_parent(int signum){
    printf("Parent : Received a response signal from child \n");
}

void sig_handler_child(int signum){
    printf("Child : Received a signal from parent \n");
    sleep(1);
    kill(getppid(),SIGUSR1);
}

```

```

int main(){

```

```

    pid_t pid;
    if (fork() < 0) {
        perror("fork failed");
        exit(1);
    }
    /* Parent process */
    printf("Parent: sending signal to Child\n");
    kill(pid, SIGUSR1);
    printf("Parent: waiting for response\n");
    /* Child process */
    printf("Child : Received a signal from parent\n");
    sleep(1);
    printf("Child : Sending response signal to parent\n");
    kill(getppid(), SIGUSR1);
    /* Parent process */
    printf("Parent : Received a response signal from child\n");
    /* Child process */
    printf("Child : Received a signal from parent\n");
    sleep(1);
    printf("Child : Sending response signal to parent\n");
    kill(getppid(), SIGUSR1);
    /* Parent process */
    printf("Parent: sending signal to Child\n");
    kill(pid, SIGUSR1);
    printf("Parent: waiting for response\n");
    pause();
}
return 0;
}

```

```

    signal(SIGUSR1,sig_handler_parent); // Register signal handler
    sleep(1);
    printf("Parent: sending signal to Child\n");
    kill(pid,SIGUSR1);
    printf("Parent: waiting for response\n");
    pause();
}
return 0;
}

```

Exercise Sigint

- Till now we used the kill command and the keyboard to generate the signals.
- Now we will generate the signal in a program using the C API for *kill* or *raise* system call.
- In this example, we replace the SIGINT signal handler with our own and in the signal handler asks the user if the process should be terminated and then based on the response continue with either terminating the process or let it continue to run.
- We use read function instead of scanf to emphasize that scanf is not a reentrant function

sigint

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <signal.h>
5
6  static void sig_int(int signo) {
7      ssize_t n;
8      char buf[2];
9
10     signal(signo, SIG_IGN); /* ignore signal first */
11     printf("Do you really want to do this: [Y/N]? ");
12     fflush(stdout);
13     n = read(STDIN_FILENO, buf, 2);
14     if ( buf[0] == 'Y' ) {
15         raise(SIGTERM); // or kill(0, SIGTERM); // or exit (-1);
16     } else {
17         printf("Ignoring signal, be careful next time!\n");
18         fflush(stdout);
19     }
20     signal(signo, sig_int); /* reinstall the signal handler */
21 }
```



```

23 int main(void) {
24     if (signal(SIGINT, sig_int) == SIG_ERR) {
25         printf("Unable to catch SIGINT\n");
26         exit(-1);
27     }
28     for ( ; ; )
29         pause();
30
31     return 0;
32 }
33

```

```

(base) mahmutunan@MacBook-Pro lecture23 % gcc -Wall sigint.c -o sigint
(base) mahmutunan@MacBook-Pro lecture23 % ./sigint
^CDo you really want to do this: [Y/N]? N
Ignoring signal, be careful next time!
^CDo you really want to do this: [Y/N]?
Ignoring signal, be careful next time!
^CDo you really want to do this: [Y/N]? Y
zsh: terminated ./sigint
(base) mahmutunan@MacBook-Pro lecture23 %

```

recall -forkexecvp.c

```
int main(int argc, char **argv) {
    pid_t pid;
    int status;

    if (argc < 2) {
        printf("Usage: %s <command> [args]\n", argv[0]);
        exit(-1);
    }

    pid = fork();
    if (pid == 0) { /* this is child process */
        execvp(argv[1], &argv[1]);
        printf("If you see this statement then execl failed ;-(\n");
        perror("execvp");
        exit(-1);
    } else if (pid > 0) { /* this is the parent process */
        printf("Wait for the child process to terminate\n");
        wait(&status); /* wait for the child process to terminate */
        if (WIFEXITED(status)) { /* child process terminated normally */
            printf("Child process exited with status = %d\n", WEXITSTATUS(status));
        } else { /* child process did not terminate normally */
            printf("Child process did not terminate normally!\n");
            /* look at the man page for wait (man 2 wait) to determine
               how the child process was terminated */
        }
    } else { /* we have an error */
        perror("fork"); /* use perror to print the system error message */
        exit(EXIT_FAILURE);
    }

    printf("[%ld]: Exiting program ..... \n", (long)getpid());

    return 0;
}
```

- Let us now consider how signals impact child processes created using fork/exec. We will be running a c program that consist of fork and execvp system calls (hw1.c)
- This code illustrates the use of dynamic memory allocation to create contiguous 2D-matrices and use traditional array indexing. It also illustrate the use of gettime to measure wall clock time.


```
$ ./a.out /home/UAB/puri/CS332/shared/hw1 1000
```

```
Wait for the child process to terminate
```

```
^C
```

```
$ ps -u
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
puri	19202	0.3	0.1	125440	3912	pts/0	Ss	10:05	0:00	-bash
puri	19320	0.0	0.0	161588	1868	pts/0	R+	10:06	0:00	ps -u

```
$
```

```
$ ./a.out /home/UAB/puri/CS332/shared/hw1 1000
```

```
Wait for the child process to terminate
```

```
^Z
```

```
[1]+  Stopped                  ./a.out /home/UAB/puri/CS332/shared/hw1 1000
```

```
$
```

```
$ jobs
```

```
[1]+  Stopped                  ./a.out /home/UAB/puri/CS332/shared/hw1 1000
```

```
$ ps -u
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
puri	19202	0.1	0.1	125440	3912	pts/0	Ss	10:05	0:00	-bash
puri	19351	0.0	0.0	4220	352	pts/0	T	10:08	0:00	./a.out /home/U
puri	19352	29.0	0.6	27800	23844	pts/0	T	10:08	0:01	/home/UAB/puri/
puri	19353	0.0	0.0	161588	1864	pts/0	R+	10:08	0:00	ps -u

```
$ kill -CONT 19352
```

```
$
```

```
$ ps -u
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
puri	19202	0.0	0.1	125440	3912	pts/0	Ss	10:05	0:00	-bash
puri	19351	0.0	0.0	4220	352	pts/0	T	10:08	0:00	./a.out /home/U
puri	19352	17.9	0.6	27800	23844	pts/0	R	10:08	0:04	/home/UAB/puri/
puri	19355	0.0	0.0	161588	1872	pts/0	R+	10:08	0:00	ps -u

```
$ Time taken for size 1000 = 32.274239 seconds
```

```
$ jobs
```

```
[1]+  Stopped                  ./a.out /home/UAB/puri/CS332/shared/hw1 1000
```

```
$ kill -CONT 19351
```

```
$ Child process exited with status = 0
```

```
[19351]: Exiting program .....
```

```
[1]+  Done                    ./a.out /home/UAB/puri/CS332/shared/hw1 1000
```

```
$ jobs
```

```
$
```