System Programming

10. Signal (1)

Seung-Ho Lim

Dept. of Computer & Electronic Systems Eng.

Signal

- **A user process can handle an asynchronous urgent event like an interrupt handling by using signals.**
  - ***User's view***

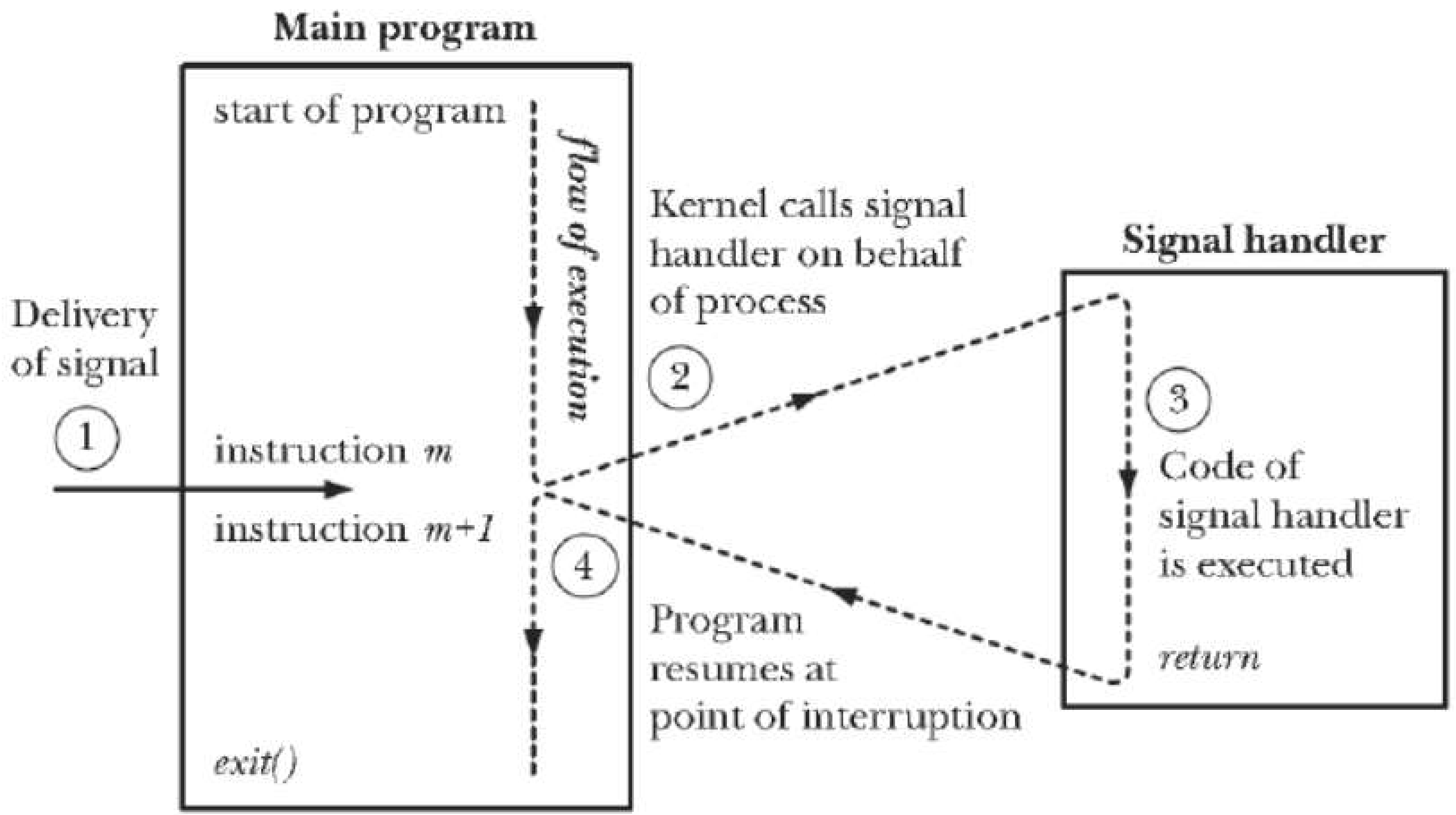
    user mode running → signal occurs → user program is interrupted → ***signal handler*** (user mode running, like an interrupt handling) → exit or return to the interrupted user program
  - ***Kernel's view***

    an event occurs(ex: segment fault) → mark that the signal (SIGSEGMENT) happened to the relevant process's signal table in PCB → just before return to user mode of the process, control goes to the ***signal handler*** (modify the process's stack) → exit or return to user mode program: **so there can be some delay (different from interrupt handling (no delay)**)
- **Signal delivery**
  - **kernel → user, user → user**

Signal Handling (1)

- When a signal is delivered to a process
  - mark (set) it to the pending signal table of the PCB.
- When will the delivered signal be handled(processed)?
  - When the process becomes running (by context switch) in the kernel, the process will eventually go to user mode.
    - Just before return to the user mode, the signal will be processes by the signal handler.
- A signal that is not handled yet
  - called “a pending signal”.
- A signal handling can be temporarily blocked by a user.
  - called “signal blocking”

Signal Handling (2)


Main program
start of program
flow of execution
Kernel calls signal handler on behalf of process
2
Signal handler
Delivery of signal
1
3
instruction m
Code of signal handler is executed
instruction m+1
4
Program resumes at point of interruption
return
exit()

Three Ways of Signal Handling

- Use a default signal handler **(SIG_DFL)**
  - in general, a signal happens when there is an error.
  - the handler is provided by the kernel
  - so in the default handler, do exit or core-dump & exit.
- Ignore a signal **(SIG_IGN)**
  - a signal is ignored.
  - however, SIGKILL & SIGSTOP cannot be ignored.
- **Use a user defined signal handler**
  - A process can register it's own signal handler.
    - e.g. ^C (kill a process).
    - but by a user defined handler, ^C can do a shutting in a game program.

Signals (1)

* SP server: defined in a header file "/usr/include/asm-generic/signal.h"

Signal name	***Reason of a signal***	***value***	***Default handler***
SIGABRT	Program abort (abort())	6	Core-dump & exit
SIGALRM	Timer alarm	14	Exit
SIGBUS	Bus error	7	Core-dump & exit
SIGCHLD	Death of a child process	17	Ignore
SIGCONT	Continue a stopped process	18	Restart/Ignore
SIGEMT	Emulation Trap	-	Core-dump & exit
SIGFPE	Arithmetic exception	8	Core-dump & exit
SIGHUP	TTY disconnected	1	Exit
SIGILL	Not an instruction (an illegal jump to a data section)	4	Core-dump & exit

Signals (2)

Signal name	*Reason of a signal*	*value*	*Default handler*
SIGINT	^C	2	Exit
SIGIO	Asynchronous I/O done	29	Exit
SIGIOT	H/W fault	6	Core-dump & exit
SIGKILL	immediate kill request (kill (9)) :	9	Exit
SIGPIPE	Write attempt to a pipe with no reader (broken pipe)	13	Exit
SIGPOLL	A pollable event occurred in I/O (poll())	29	Exit
SIGPROF	Profiling timer expired	27	Exit
SIGPWR	Power failure	30	Ignore
SIGQUIT	^Z, quit	3	Core-dump & exit

Signals (3)

Signal name	*Reason of a signal*	*value*	*Default handler*
SIGSEGV	illegal memory access (pointer, access kernel's or other process's area, write on read-only area)	11	Core-dump & exit
SIGSTOP	Stop (ex: debugger)	19	Stop
SIGSYS	Illegal system call	31	Core-dump & exit
SIGTERM	kill (15) : safe process termination	15	Exit
SIGTRAP	Trace/Breakpoint Trap	5	Core-dump & exit
SIGTSTP	^Z	20	Stop
SIGTTIN	A background attempts reading	21	Stop
SIGTTOU	A background attempts writing	22	Stop

Signals (4)

Signal name	***Reason of a signal***	***value***	***Default handler***
SIGURG	An urgent socket event	23	Ignore
SIGUSR1	User defined signal 1	10	Exit
SIGUSR2	User defined signal 2	12	Exit
SIGVTALRM	Virtual timer alarm	26	Exit
SIGWINCH	Size change in a `tty` window	28	Ignore
SIGXCPU	CPU time-limit expire	24	Core-dump & exit
SIGXFSZ	File size-limit violation	25	Core-dump & exit

* Use a signal name, not the magic number
- the numbers may different from system to system.

signal(2) function

```
#include <signal.h>

typedef void (*sighandler_t)(int);  // void function pointer
sighandler_t signal(int signum, sighandler_t handler);
```

- set a user-defined signal handler
- parameters
  - *signum* : signal number
  - *handler* : user defined signal handler (function) for the *signum signal*
- return:
  - old signal handler's address (if OK)
  - SIG_ERR on an error
- the signal handler is used only once
  - after the first signal reception, reset to SIG_DFL
  - some OS versions maintain the user defined handler

Usage of Signal Handler (1)

sighandler.c

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

static void sigcatcher(int);
void (*was)(int);

int main(void)
{
	if( was = signal( SIGINT, sigcatcher ) == SIG_ERR) {
		perror("SIGINT");
		exit(1);
	}
	while(1) pause();
}
```

Usage of Signal Handler (2)

```
static void sigcatcher( int signo )
{
   switch( signo ) {
        case SIGINT :
                printf("PID %d caught signal SIGINT.\n", getpid());
                signal(SIGINT, was); // dependent on linux, bsd versions
                break;
        default :
                fprintf(stderr, "something wrong\n");
                exit(1);
   }
}
```

- *Run & Result*

```
$ ./a.out
^CPID 22986 caught signal SIGINT.
$
```

`kill()` : sending a signal

```
#include <sys/types.h>
#include <signal.h>

int kill ( pid_t pid, int sig);
```

- parameters:
  - *pid* : process id
  - sig : signal number
- return:
  - 0 if OK
  - -1 on an error

pid	*target process*
> 0	to the process with pid
0	to all processes in my group
-1	to every process except for the init(pid =1) process (broadcasting)

kill() usage

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

int main()
{
   int pid;
   if ((pid = fork()) == 0) {    // child
        while(1);
   } else {      // parent
        kill (pid, SIGKILL);     // kill the child, signal number = 9
        printf("send a signal to the child\n");
        wait();
        printf("death of child\n");
   }
}
```

raise(): send a signal to itself

```
#include <signal.h>
int raise (int sig);
```

- parameter
  - sig : signal number
- return:
  - 0 if OK
  - -1 on an error
- usage

```
#include <siganl.h>
int main()
{   printf("Self Process signal : \n");
    raise(SIGUSR1);
}
```

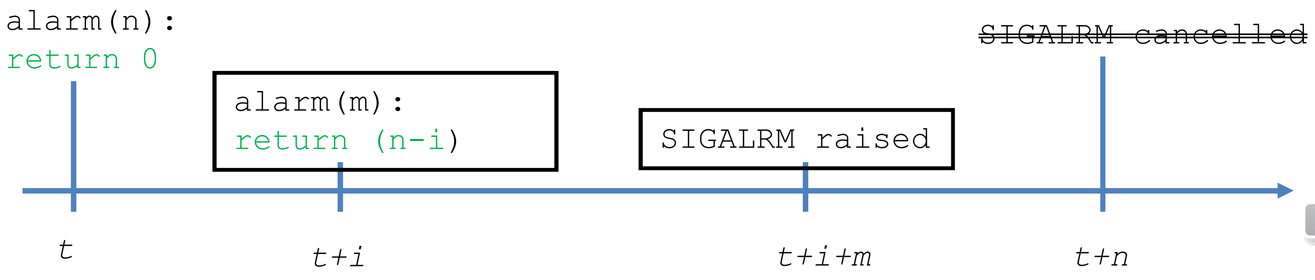
Run & Result

```
$ ./a.out
Self Process signal;
User defined Signal 1
$
```

`alarm()`: set an alarm clock

```
#include <unistd.h>
unsigned alarm(unsigned sec);
```

- parameter
  - *sec* : after sec, send SIGALRM to itself
- return:
  - if there is a previous alarm( ) call
    - time left to the alarm time of the previous alarm() call
  - at the first alarm( ) call : 0
- Linux allows only one alarm per process at a time
  - a new alarm replace a previous alarm
- alarm(0) cancels a previous alarm.



alarm() usage

alarm.c

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
static void sig_catcher(int);
int alarmed = 0;

int main()
{
   int pid;
   signal(SIGALRM, sig_catcher);
   alarm(3);
     do something;
   while(alarmed == 0);
   printf("after alarm in main\n");
}
```

```
void sig_catcher(int signo)
{
   alarmed = 1;
   alarm(0);
}
```

- sleep() call also use the SIGALRM
  - if alarm and sleep are used together, we may get a non-deterministic result

abort()

```
#include <stdlib.h>
void abort (void);
```

- cause abnormal process termination
  - generate a SIGABRT → core dump and exit
- usage

```
#include <stdlib.h>

int main()
{
   abort();
}
```

Run & Result

```
$ ls
a.out prog.c
$ ./a.out
Abort (core dumped)
```

pause()

```
#include <unistd.h>
int pause(void);
```

- wait until a (any) signal is delivered.
- return
  - when a default handler is used
    - the process exit (not return from the pause call)
  - when a user defined signal handler is used
    - return after the handler is invoked

Usage of `pause()`

pause.c

```
....
void sig_catch (int sig_no) {  ..... }
int main()
{
        int pid;
        int status;
        signal(SIGUSR1, sig_catch);
        if ((pid = fork()) == 0) {          // child
                // this pause may block forever: why?
                pause();
                printf("Child wake up\n");
                exit(0);
        } else {    // parent
                sleep(1);
                kill (pid, SIGUSR1); // send SIGUSR1 to the child process
                wait(&status);
        }
}
```

HW5

- Semaphore의 wait/post 동작방식 이해
  - Semaphore의 prod-con.c 코드 작성 및 테스트
  - Sleep의 위치에 따라서 producer와 consume가 각각 연속적으로 실행되는 상황을 테스트하시오.
- Signal 및 Pause 함수
  - Pause.c 코드 작성 및 테스트
  - Parent의 sleep()에 따라서 pause 실행 전후에 signal이 전달되는 상황을 테스트하시오.
- Due date
  - 5/28