IPC: Interrupts and Signals

Subject:- Unix Operating System

System Lab Class :- TYIT

Name PRN

Aditi Sudhir Ghate 2020BTEIT00044

Assignment No - 2c

Title– Write a application or program that communicates between to process opened in two terminal using kill() and signal()

Objectives –

1. To learn about IPC through signal.
2. To know the process management of Unix/Linux OS
3. Use of system call to write effective application programs

Theory-

kill()

Syntax-

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

The **kill**() system call can be used to send any signal to any process group or process.
If pid is positive, then signal sig is sent to pid.
If pid equals 0, then sig is sent to every process in the process group of the current process.
If pid equals -1, then sig is sent to every process for which the calling process has permission to send signals, except for process 1 (init), but see below.
If pid is less than -1, then sig is sent to every process in the process group -pid.
If sig is 0, then no signal is sent, but error checking is still performed.
For a process to have permission to send a signal it must either be privileged (under Linux: have the **CAP_KILL** capability), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of SIGCONT it suffices when the sending and receiving processes belong to the same session.

signal()

Syntax-

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

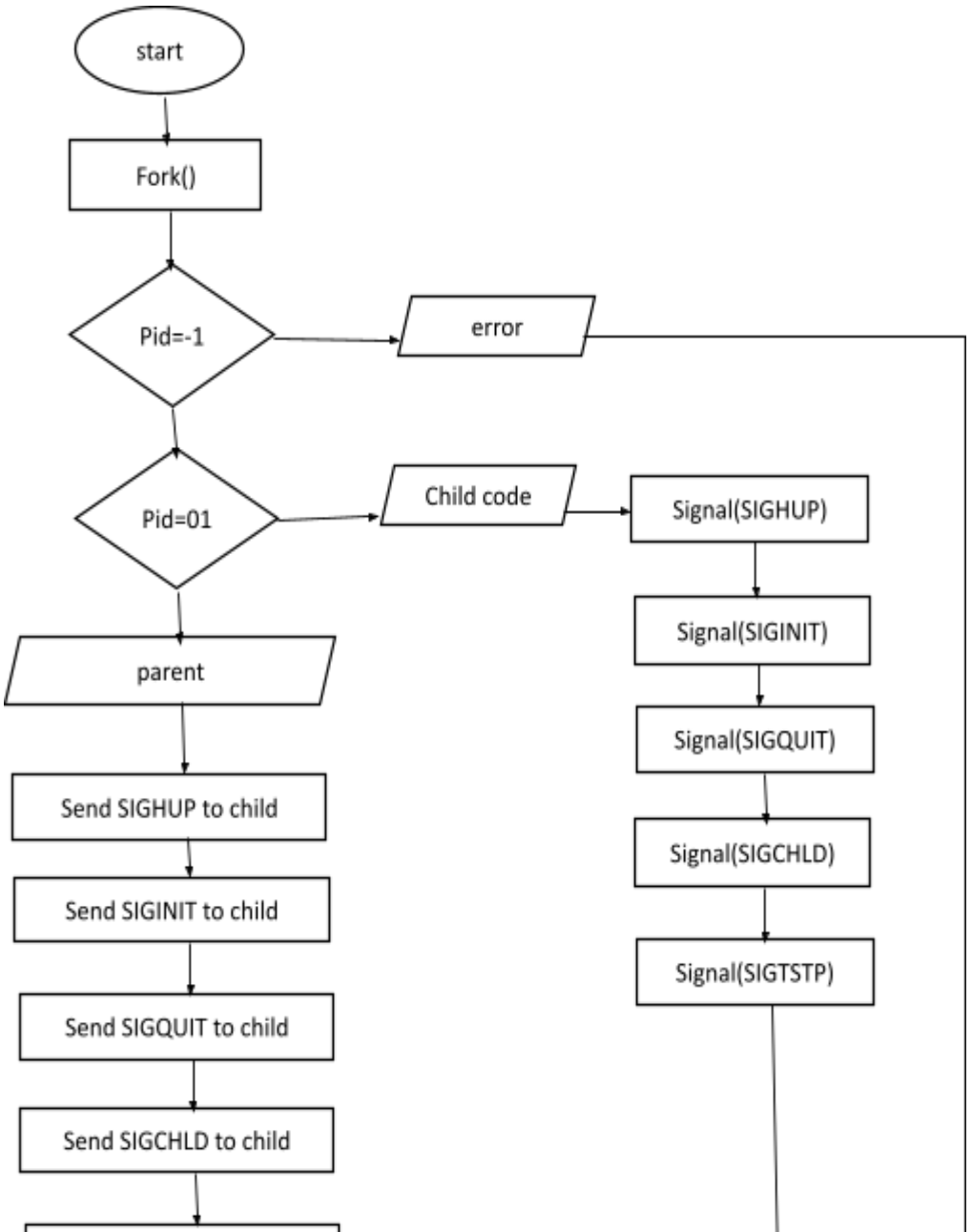
The signal() system call installs a new signal handler for the signal with number signum. The signal handler is set to sighandler which may be a user specified function, or either SIG_IGN or SIG_DFL.

Upon arrival of a signal with number signum the following happens. If the corresponding handler is set to SIG_IGN, then the signal is ignored. If the handler is set to SIG_DFL, then the default action associated with the signal (see signal(7)) occurs. Finally, if the handler is set to a function sighandler then first either the handler is reset to SIG_DFL or an implementation-dependent blocking of the signal is performed and next sighandler is called with argument signum.

Using a signal handler function for a signal is called "catching the signal". The signals SIGKILL and SIGSTOP cannot be caught or ignored.

The signal() function returns the previous value of the signal handler, or SIG_ERR on error. The original Unix signal() would reset the handler to SIG_DFL, and System V (and the Linux kernel and libc4,5) does the same. On the other hand, BSD does not reset the handler, but blocks new instances of this signal from occurring during a call of the handler. The glibc2 library follows the BSD behaviour.

Flowchart-



Program-

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include<stdlib.h>
#include<unistd.h>
void SIGINT_handler(int);
void SIGQUIT_handler(int);
int ShmID;
pid_t *ShmPTR;
void main(void)
{
int i;
pid_t pid = getpid();
key_t MyKey;
if (signal(SIGINT, SIGINT_handler) == SIG_ERR) {
printf("SIGINT install error\n");
exit(1);
}
if (signal(SIGQUIT, SIGQUIT_handler) == SIG_ERR) {
printf("SIGQUIT install error\n");
exit(2);
}
MyKey = ftok(".", 's');
ShmID = shmget(MyKey, sizeof(pid_t), IPC_CREAT | 0666);
ShmPTR = (pid_t *) shmat(ShmID, NULL, 0);
*ShmPTR = pid;
for (i = 0; ; i++) {
printf("From process %d: %d\n", pid, i);
sleep(1);
}
}
void SIGINT_handler(int sig)
{
signal(sig, SIG_IGN);
printf("From SIGINT: just got a %d (SIGINT ^C) signal\n",
sig); signal(sig, SIGINT_handler);
}
void SIGQUIT_handler(int sig)
{
signal(sig, SIG_IGN);
printf("From SIGQUIT: just got a %d (SIGQUIT ^\\) signal"
" and is about to quit\n",
sig); shmdt(ShmPTR);
shmctl(ShmID, IPC_RMID, NULL);
exit(3);
}
```

Output-

```
aditi@aditi-Lenovo-ideapad-330S-14IKB-U:~/ADnOR/Assignments/2C$ gcc 2C.c
aditi@aditi-Lenovo-ideapad-330S-14IKB-U:~/ADnOR/Assignments/2C$ ./a.out
From process 4284: 0
From process 4284: 1
From process 4284: 2
From process 4284: 3
From process 4284: 4
From process 4284: 5
From process 4284: 6
From process 4284: 7
From process 4284: 8
From process 4284: 9
From process 4284: 10
From process 4284: 11
From process 4284: 12
From process 4284: 13
From process 4284: 14
From process 4284: 15
From process 4284: 16
From process 4284: 17
From process 4284: 18
```

```
From process 4284: 75
From process 4284: 76
From process 4284: 77
From process 4284: 78
From process 4284: 79
From process 4284: 80
From process 4284: 81
From process 4284: 82
From process 4284: 83
From process 4284: 84
From process 4284: 85
From process 4284: 86
From process 4284: 87
From process 4284: 88
From process 4284: 89
From process 4284: 90
From process 4284: 91
From process 4284: 92
From process 4284: 93
From process 4284: 94
From process 4284: 95
```

Conclusion:

Processes opened in two terminals can also be handled using signal handlers and kill() function calls. Shared memory can be used as a mode of IPC.

References:

http://www.csl.mtu.edu/cs4411.ck/www/NOTES/signal/kill.html