**IPC: Interrupts and Signals**

**Subject - Unix Operating System**

**Name – Hemant Sharma**

**PRN – 22610001 Class – TYIT**

**Assignment No – 2b**

**Title-** Write a application or program that communicates between child and parent processes 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-**

1. **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.

1. **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

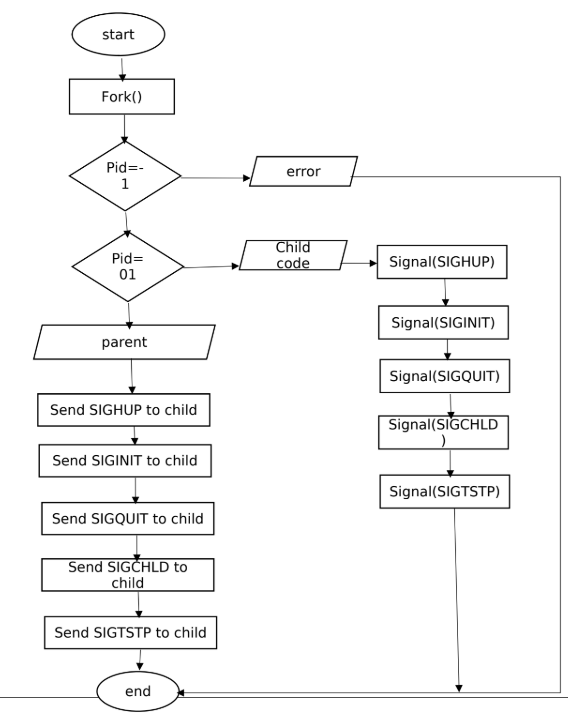
SIG\_DFL, and System V (and the Linux kernel and libc4,5) does the same. On

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 <signal.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

void sighup();

void sigint();

void sigquit();

void main()

{

int pid;

if ((pid = fork()) < 0) {

perror("fork");

exit(1);

}

if (pid == 0) {

signal(SIGHUP, sighup);

signal(SIGINT, sigint);

signal(SIGQUIT, sigquit);

for (;;) ;

}

else

{

printf("\nPARENT: sending SIGHUP\n\n");

kill(pid, SIGHUP);

sleep(3);

printf("\nPARENT: sending SIGINT\n\n");

kill(pid, SIGINT);

sleep(3);

printf("\nPARENT: sending SIGQUIT\n\n");

kill(pid, SIGQUIT);

sleep(3);

}

}

void sighup()

{

signal(SIGHUP, sighup);

printf("CHILD: I have received a SIGHUP\n");

}

void sigint()

{

signal(SIGINT, sigint);

printf("CHILD: I have received a SIGINT\n");

}

void sigquit()

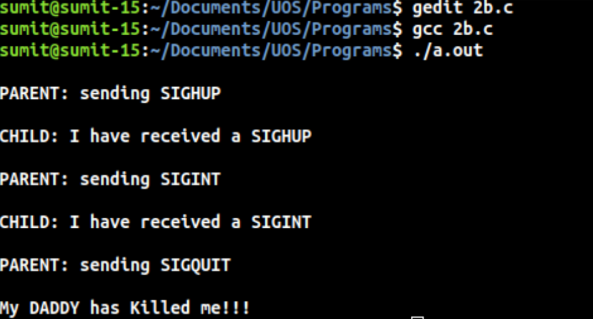
{

printf("My DADDY has Killed me!!!\n");

exit(0);

}

**Output-**

****

**Conclusion-**

alarm() signal can be used to raise alarm after particular time period. Signal() system call is evoked by alarm() which is further processed by signal handler