UNIX Programming

Module 5 – FAQ's in VTU

1. **What is a signal? Mention the different sources of signals. Discuss any 4 POSIX defined signals.**

➢ Signal

A signal is a notification to a process that an event has occurred. Signals are sometimes called "software interrupts".

➢ Features of Signal

- Signal usually occurs asynchronously.
- The process does not know ahead of time exactly when a signal will occur.
- Signal can be sent by one process to another process (or to itself) or by the kernel to a process.

Sources of signals:

- The **terminal-generated signals** occur when users press certain terminal keys. Pressing the DELETE key or Control-C on the terminal normally causes the interrupt signal (SIGINT) to be generated.
- **Hardware exceptions** generate signals. For example, divide by 0 and invalid memory reference. These conditions are usually detected by the hardware, and the kernel is notified. The kernel then generates the appropriate signal for the process that was running at the time the condition occurred. For example, SIGSEGV is generated for a process that executes an invalid memory reference.
- The kill(2) function allows a process to send any signal to another process or process group, with limitations:
- The kill(1) command allows us to send signals to other processes. This program is just an interface to the kill function. This command is often used to terminate a runaway background process.
- Software conditions can generate signals when a process should be notified of various events. For example:
  - SIGURG: generated when out-of-band data arrives over a network connection),

- SIGPIPE: generated when a process writes to a pipe that has no reader)
- SIGALRM: generated when an alarm clock set by the process expires).

POSIX defined signals.

- SIGABRT: generated by calling the abort function. The process terminates abnormally.
- SIGALRM:
  - This signal is generated when a timer set with the alarm function expires.
  - This signal is also generated when an interval timer set by the setitimer(2) function expires.
- SIGBUS: indicates an implementation-defined hardware fault. Implementations usually generate this signal on certain types of memory faults.
- SIGCHLD: Whenever a process terminates or stops, the SIGCHLD signal is sent to the parent. By default, this signal is ignored, so the parent must catch this signal if it wants to be notified whenever a child's status changes. The normal action in the signal-catching function is to call one of the wait functions to fetch the child's process ID and termination status

2. Write a program to setup signal handler for SIGINT and SIGALRM

SIGINT

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

void     INThandler(int);

void  main(void)
{
    signal(SIGINT, INThandler);
    while (1)
        pause();
}

void  INThandler(int sig)
{
    char  c;

    signal(sig, SIG_IGN);
    printf("OUCH, did you hit Ctrl-C?\n"
           "Do you really want to quit? [y/n] ");
    c = getchar();
```

```
if (c == 'y' || c == 'Y')
    exit(0);
else
    signal(SIGINT, INThandler);
```

SIGALRM

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

void sig_handler(int signum){

 printf("Inside handler function\n");
}

int main(){

 signal(SIGALRM,sig_handler); // Register signal handler

 alarm(2);  // Scheduled alarm after 2 seconds

 for(int i=1;;i++){

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

3. **What is a Daemon? Discuss the characteristics and basic coding rules**

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down.

The characteristics of daemons are:

- Daemons run in background.
- Daemons have super-user privilege.
- Daemons don’t have controlling terminal.
- Daemons are session and group leaders.

Coding rules

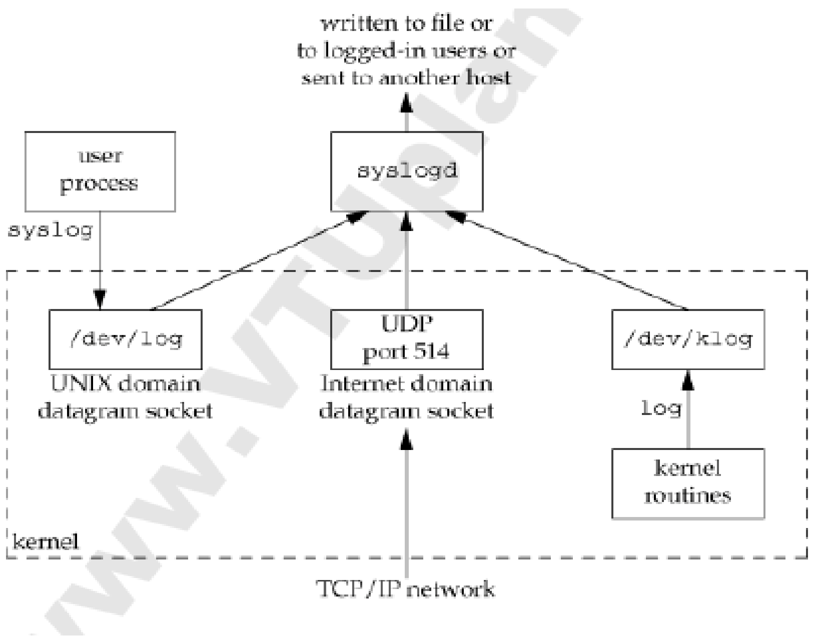
- Call umask to set the file mode creation mask to 0.
- Call fork and have the parent exit.
- Call setsid to create a new session.
- Change the current working directory to the root directory.
- Unneeded file descriptors should be closed
- Some daemons open file descriptors 0, 1, and 2 to /dev/null so that any library routines that try to read from standard input or write to standard output or standard error will have no effect.

Example Program:

```
#include <unistd,h>
#include <sys/types.h>
#include <fcntl.h>
 int daemon_initialise( )
{
pid_t pid;
if (( pid = for() ) < 0)
 return –1;
else if ( pid != 0)
 exit(0); /* parent exits */ /* child continues */
setsid( );
chdir(“/”);
umask(0);
return 0;
}
```

4. Explain error handling for Daemon process with a neat block diagram

One problem a daemon has is how to handle error messages. It can't simply write to standard error, since it shouldn't have a controlling terminal. A central daemon error-logging facility is required.



- **There are three ways to generate log messages:**

- ❖ Kernel routines can call the log function. These messages can be read by any user process that opens and reads the /dev/klog device.
- ❖ Most user processes (daemons) call the syslog(3) function to generate log messages. This causes the message to be sent to the UNIX domain datagram socket /dev/log.
- ❖ A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the syslog function never generates these UDP datagrams: they require explicit network programming by the process generating the log message**.**

Normally, the syslogd daemon reads all three forms of log messages. On start-up, this daemon reads a configuration file, usually /etc/syslog.conf, which determines where different classes of messages are to be sent. For example, urgent messages can be sent to the system administrator (if logged in) and printed on the console, whereas warnings may be logged to a file**.**

5. Explain sigsetjmp and sigsetlongjmp

setjmp and longjmp functions can be used for nonlocal branching. The longjmp function is often called from a signal handler to return to the main loop of a program, instead of returning from the handler.

However, there is a problem in calling longjmp. When a signal is caught, the signal-catching function is entered, with the current signal automatically being added to the signal mask of the process. This prevents subsequent occurrences of that signal from interrupting the signal handler. If we longjmp out of the signal handler, what happens to the signal mask for the process depends on the platform.

POSIX.1 does not specify the effect of setjmp and longjmp on signal masks

Instead, two new functions, sigsetjmp and siglongjmp, are defined by POSIX.1. These two functions should always be used when branching from a signal handler.

- The function prototypes of the APIs are:

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int savemask);
int siglongjmp(sigjmp_buf env, int val);
```

- The sigsetjmp and siglongjmp are created to support signal mask processing.
- Specifically, it is implementation-dependent on whether a process signal mask is saved and restored when it invokes the setjmp and longjmp APIs respectively.
- The only difference between these functions and the setjmp and longjmp functions is that sigsetjmp has an additional argument. If *savemask* is nonzero, then sigsetjmp also saves the current signal mask of the process in *env*. When siglongjmp is called, if the *env* argument was saved by a call to sigsetjmp with a nonzero *savemask*, then siglongjmp restores the saved signal mask

6. Write the prototype of ALARM and PAUSE and explain how they operate.

The **alarm function** sets a timer that will expire at a specified time in the future. When the timer expires, the SIGALRM signal is generated. If we ignore or don't catch this signal, its default action is to terminate the process.

```
#include <unistd.h>


unsigned int alarm(unsigned int seconds);


/* Returns: 0 or number of seconds until previously set alarm */
```

- The *seconds* value is the number of clock seconds in the future when the signal should be generated. When that time occurs, the signal is generated by the kernel, although additional time could elapse before the process gets control to handle the signal, because of processor scheduling delays.
- There is only one of these alarm clocks per process. If alarm is called with a previously registered alarm clock not yet expired, then:
  - The number of seconds left for previous alarm clock is returned as the value of this function.
  - The previous alarm clock is replaced by the new value.
- If a previously registered alarm clock for the process has not yet expired and if the seconds value is 0, the previous alarm clock is canceled. The number of seconds left for that previous alarm clock is still returned as the value of the function.

- The **pause function** suspends the calling process until a signal is caught.

```
#include <unistd.h>

int pause(void);

/* Returns: −1 with errno set to EINTR */
```

- The only time pause returns is if a signal handler is executed and that handler returns. In that case, pause returns −1 with errno set to EINTR.

7. Explain the sigaction () function and sigprocmask() by giving the prototype and discuss their features

- ➢ The sigaction API is a replacement for the signal API in the latest UNIX and POSIX systems.
- ➢ The sigaction API is called by a process to set up a signal handling method for each signal it wants to deal with.
- ➢ sigaction API returns the previous signal handling method for a given signal.
- ➢ The sigaction API prototype is:

```
#include <signal.h>
int sigaction(int signal_num, struct sigaction *action, struct sigaction *old_action);
```

The struct sigaction data type is defined in the <signal.h> header as:

```
struct sigaction
{
void (*sa_handler)(int);
sigset_t sa_mask;
int sa_flag;
;
```

- ➢ The sa_handler field can be set to SIG_IGN, SIG_DFL, or a user defined signal handler function.
- ➢ The sa_mask field specifies additional signals that process wishes to block when it is handling signal_num signal.
- ➢ The signal_num argument designates which signal handling action is defined in the *action* argument.

- The previous signal handling method for signal_num will be returned via the old_action argument if it is not a NULL pointer.
- If action argument is a NULL pointer, the calling process's existing signal handling method for signal_num will be unchanged.
- The following program illustrates the uses of sigaction:

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
void callme(int sig_num) { cout<<"catch signal:"<<sig_num<<endl; }
int main(int argc, char* argv[])
 {
 sigset_t sigmask;
struct sigaction action,old_action;
sigemptyset(&sigmask);
if(sigaddset(&sigmask,SIGTERM)==-1 ||
sigprocmask(SIG_SETMASK,&sigmask,0)==-1)
 perror("set signal mask");
sigemptyset(&action.sa_mask); sigaddset(&action.sa_mask,SIGSEGV);
action.sa_handler=callme;
action.sa_flags=0; if(sigaction(SIGINT,&action,&old_action)==-1)
perror("sigaction"); pause(); cout<<argv[0]<<"exists\n";
return 0;
 }
```

sigprocmask()

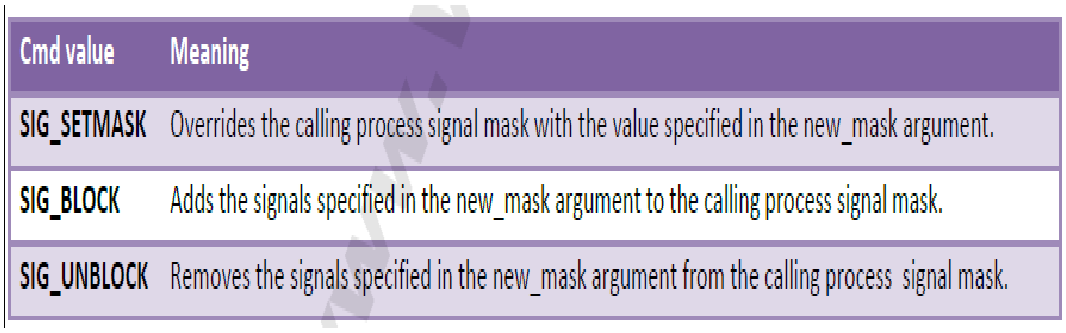
- **A process may query or set its signal mask via the sigprocmask API:**

  #include <signal.h>

  int sigprocmask(int cmd, const sigset_t *new_mask, sigset_t *old_mask);

  Returns: 0 if OK, 1 on error
- The new_mask argument defines a set of signals to be set or reset in a calling process signal mask, and the cmd argument specifies how the new_mask value is to be used by the API.

- If the actual argument to new_mask argument is a NULL pointer, the cmd argument will be ignored, and the current process signal mask will not be altered.
- If the actual argument to old_mask is a NULL pointer, no previous signal mask will be returned.
- The sigset_t contains a collection of bit flags.



Cmd value	Meaning
SIG_SETMASK	Overrides the calling process signal mask with the value specified in the new_mask argument.
SIG_BLOCK	Adds the signals specified in the new_mask argument to the calling process signal mask.
SIG_UNBLOCK	Removes the signals specified in the new_mask argument from the calling process signal mask.

8. Briefly explain the kill() API and alarm() API

Kill()

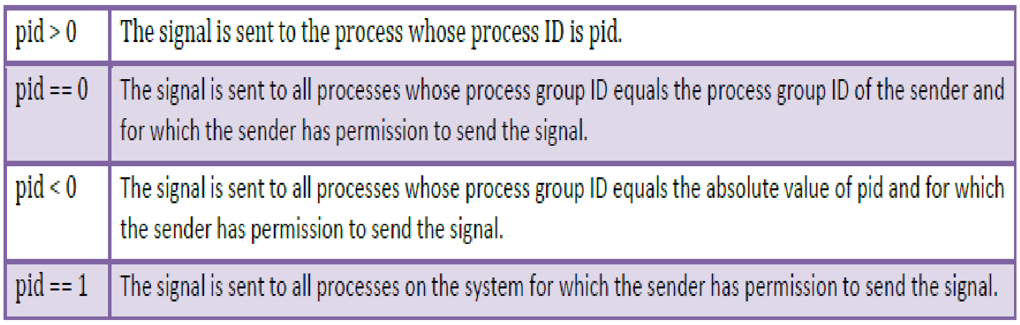
A process can send a signal to a related process via the kill API. This is a simple means of inter-process communication or control. The function prototype of the API is:

#include<signal.h>

int kill(pid_t pid, int signal_num);

Returns: 0 on success, -1 on failure.

- The signal_num argument is the integer value of a signal to be sent to one or more processes designated by pid. The possible values of pid and its use by the kill API are:



pid > 0	The signal is sent to the process whose process ID is pid.
pid == 0	The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal.
pid < 0	The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal.
pid == 1	The signal is sent to all processes on the system for which the sender has permission to send the signal.

The UNIX kill command invocation syntax is: **Kill [-<signal_num>] <pid>...... Where signal_num can be an integer number or the symbolic name of a signal. <pid> is process ID.**

Alarm:

- The alarm API can be called by a process to request the kernel to send the SIGALRM signal after a certain number of real clock seconds.
- The function prototype of the API is:

  #include<signal.h>

  Unsigned int alarm(unsigned int time_interval);

  Returns: 0 or number of seconds until previously set alarm

- The alarm API can be used to implement the sleep API:

```
 #include<signal.h>
 #include<stdio.h>
#include<unistd.h>
 void wakeup( ) { ; }
unsigned int sleep (unsigned int timer )
{
 struct sigaction action;
 action.sa_handler=wakeup;
action.sa_flags=0;
sigemptyset(&action.sa_mask); if(sigaction(SIGALARM,&action,0)==-1)
{ perror("sigaction");
return -1;
}
 (void) alarm (timer);
 (void) pause( );
 return 0;
}
```

9. **What are the three ways a process can react to pending signals:**

Signal dispositions

We can tell the kernel to do one of three things when a signal occurs. This is called the **disposition of the signal**, or the **action associated with a signal**.

1. **Ignore the signal**. Most signals can be ignored, but two signals can never be ignored: SIGKILL and SIGSTOP.
   - The reason these two signals can't be ignored is to provide the kernel and the superuser with a surefire way of either killing or stopping any process.

- If we ignore some of the signals that are generated by a hardware exception (such as illegal memory reference or divide by 0), the behavior of the process is undefined.

2. **Catch the signal**. To do this, we tell the kernel to call a function of ours whenever the signal occurs. In our function, we can do whatever we want to handle the condition. For example:
    - If we're writing a command interpreter, when the user generates the interrupt signal at the keyboard, we probably want to return to the main loop of the program, terminating whatever command we were executing for the user.
    - If the SIGCHLD signal is caught, it means that a child process has terminated, so the signal-catching function can call waitpid to fetch the child's process ID and termination status.
    - If the process has created temporary files, we may want to write a signal-catching function for the SIGTERM signal (the termination signal that is the default signal sent by the kill command) to clean up the temporary files.
    - Note that the two signals SIGKILL and SIGSTOP can't be caught.
3. **Let the default action apply**. Every signal has a default action. The default action for most signals is to terminate the process.

    The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

10. Write a C program that checks whether SIGINT signal is present in a process signal mask and adds it to the mask if it not there. It should clear SIGSEGV signal from the process signal mask
11. **What is signal mask of a Process**
    - Each process in UNIX or POSIX.1 system has signal mask that defines which signals are blocked when generated to a process.
    - A blocked signal depends on the recipient process to unblock it and handle it accordingly.
    - If a signal is specified to be ignored and blocked, it is implementation dependent on whether the signal will be discarded or left pending when it is sent to the process.
    - A process initially inherits the parent's signal mask when it is created, but an pending signals for the parent process are not passed on.

- A process may query or set its signal mask via the sigprocmask API

12. Explain the Client – server model

- In general, a server is a process that waits for a client to contact it, requesting some type of service.
- In two-way communication between a client and a server, the client sends a request to the server, and the server sends a reply back to the client.
- A common use for a daemon process is as a server process.
- We can call the syslogd process a server that has messages sent to it by user processes (clients) using a UNIX domain datagram socket.
- The service being provided by the syslogd server is the logging of an error message.

13. Write a C/C++ program to show the use of alarm API

```
} #include<signal.h>
 #include<stdio.h>
#include<unistd.h>
 void wakeup( ) { ; }
unsigned int sleep (unsigned int timer )
{
 struct sigaction action;
 action.sa_handler=wakeup;
action.sa_flags=0;
sigemptyset(&action.sa_mask); if(sigaction(SIGALARM,&action,0)==-1)
{ perror("sigaction");
return -1;
}
 (void) alarm (timer);
 (void) pause( );
 return 0;
```

14. Explain kernel support for signals

- When a signal is generated for a process, the kernel will set the corresponding signal flag in the process table slot of the recipient process.
- If the recipient process is asleep, the kernel will awaken the process by scheduling it.
- When the recipient process runs, the kernel will check the process U-area that contains an array of signal handling specifications.
- If array entry contains a zero value, the process will accept the default action of the signal.
- If array entry contains a 1 value, the process will ignore the signal and kernel will discard it.
- If array entry contains any other value, it is used as the function pointer for a user-defined signal handler routine.
- Finally, if the array entry contains any other value, it is used as the function pointer for a used defined signal hander routine.
- The kernel will setup the process to execute the function immediately, and the process will return to its current point of execution (or to some other place if signal handler does a long jump), if the signal hander does not terminate the process.
- If there are different signals pending on a process, the order in which they are sent to a recipient process in undefined
- If multiple instances of a signal are pending on a process, it is implementation – dependent on whether a single instance or multiple instances of the signal will be delivered to the process.
- In UNIX System V.3, each signal flag in a process table slot records only whether a signal is pending, but not how many of them are present