Processes and Signals	**LABORATORY**	**TWO**
OBJECTIVES		
1. Processes in Unix / Linux 2. Unix / Linux Signals 3. Signal Handling 4. More Signal Handling (Optional) 5. Signal Generation (Optional)		

Processes in Unix / Linux

In Unix / Linux, one or more processes will be created when executing programs. When you execute a program, e.g. by typing **`a.out`**, the command line interpreter, called the *shell*, will interpret your input line (**`a.out`**) and will create a process to host that program for execution. In general, when a program is executed, a process is created for it.

The set of processes existing in the system could be viewed by means of the **`ps`** command. For example, you could type the command **`ps`** to see a list of processes. If you want to see more information about the processes, you could type the command **`ps -elf`**. Each process is identified by a *process id* or simply *pid*. Most processes in the system would be in "**S**" state (***suspended***) under the the "**S**" or "**STAT**" column. The process executing **`ps`** is itself in "**R**" state (***running***). There could be several processes in this "**R**" state, when there are multiple CPUs with the machine, like **apollo2**. You may monitor the list of processes currently executing in the machine, by the command **`top`**. You should be seeing many more processes on **apollo2**, inclusive of other users' processes.

The invocation of a program is called a *job* or a *task* in Unix / Linux. A job normally creates one *process*, but could also create multiple processes to complete the mission. For example, the commonly used *pipe* or "**`|`**" operator in the shell generates multiple processes. For example, type **`ls -l | more`** and you can use **`ps`** to check that there are really two processes created, one for **`ls -l`** and the other for **`more`**. The **`ls`** and **`more`** commands are executed together for a job. In this case, you have to use another shell (e.g., use another *ssh client*) to check because you could not type the **`ps`** command when the **`ls`** and **`more`** commands are still being executed. We could try a *sequence of commands* for a job like **`ps; pwd; cal 2023`**. Processes belonging to the same job form a *process group*.

You could use multiple *ssh clients* when you are using the computers in the laboratory so that each program you want to execute can be executed on different windows (**`ssh`** or **`putty`** sessions). However, if you are connecting from home, you may have only one terminal or only one connection to Linux. Then is it possible for you to execute multiple jobs together? Luckily, the answer is ***yes***, and it works with C-shell (**`csh`**) and Bourne-again shell (**`bash`**). The default shell on **apollo** running a variant of Linux called CentOS is **`bash`** (version 4.2 for **apollo**).

At any moment, there is only *one* process group derived from one job that is connected to the keyboard for input, but there could be multiple jobs ***not*** *connected to the keyboard* running at the same time. We say that these multiple jobs, except for the one currently connected to the keyboard, are being run in *background*. The one connected to the keyboard is running in *foreground*. All these multiple *background processes* could still produce outputs to the screen, and all the outputs from all the jobs will be *intermixed* together. To see the list of jobs, execute the command **`jobs`**. To put a job to background, use the "**`&`**" operator when you run the job in the command line. Alternatively, to put an already running job to the background, you will first type "**`<Ctrl>-Z`**" to *stop* the job and then type **`bg`** to put it to the background. To put a specific background job back to the foreground, use **`fg`** command with the job number as seen with the command **`jobs`**.

Try running the long printing program **lab2A.c** three times with different inputs simultaneously and you could see an intermixed output. Type **`ps -lf`** to see your processes (perhaps using another window).

```
cc lab2A.c -o lab2A
./lab2A first 50 &
./lab2A second 60 &
./lab2A third 40 &
```

```
// lab 2A
#include <stdio.h>

int main(int argc, char* argv[])
{
  int n, i;
  n = atoi(argv[2]);
  for (i = 0; i < n; i++) {
      sleep(2);
      printf("program name: %s, argument: %s, output: %d\n",argv[0],argv[1],i);
  }
}
```

Repeat the above execution of the three processes on three different windows (terminals). Now we do not need to use "**`&`**" operator. All processes can receive inputs from the keyboard and produce outputs to the appropriate windows. Ability to switch jobs between background and foreground via **`bg`** and **`fg`** will be useful when you do not have the luxury of multiple windows but have to do multiple tasks, for example, when you are connecting from remote PC at home via one connection.

Do you find that the lines are intermixed together to make it very difficult to type in the new command when a single window is used? To relieve the problem, you could create a script to contain all the commands. For example, you could create a (script) file called **`allLab2`** to contain the following lines:

```
./lab2A first 50 &
./lab2A second 60 &
./lab2A third 40 &
```

You should normally type **`chmod 700 allLab2`** to make the script file **`allLab2`** *executable*. You can then execute the jobs by typing **`allLab2`**. The effect is just like typing the three lines on the keyboard by yourself, like a *batch file*.

If you find that a particular process may have fallen into an infinite loop, you could attempt to terminate that process by *killing* it. The default way to terminate a running process is to hit **`<Ctrl>-C`**. It is most natural. However, you cannot terminate background processes via **`<Ctrl>-C`**. You will need to use the **`kill`** command instead, by supplying the process ID, which can be seen when you type the command **`ps -elf`** for the list of processes. For example, to terminate process 12345, type **`kill 12345`**. You may also terminate a background process number 2 (**`%2`**), by typing **`kill %2`**. Sometimes, this default action may not be successful. If you still cannot kill the process, use the *stronger* option **`kill -9 12345`** that specifically sends a strong *kill signal* (signal number 9) to terminate the process.

Here is a list of common commands used with Unix/Linux processes.

Command	Usage	Description
`ps`	**`ps -elf`**	Show the processes in the computer
`bg`	**`bg`**	Move a stopped foreground job to background
`fg`	**`fg %2`**	Move background job number 2 to foreground
`jobs`	**`jobs`**	Show the list of background jobs
`time`	**`time a.out`**	Show the total execution time of a program and its share of CPU time
`uptime`	**`uptime`**	Show the current workload of the computer, e.g., number of users
`top`	**`top`**	Show CPU and resource utilization of computer with a list of active processes
`kill`	**`kill %2`** **`kill 12345`** **`kill -2 12345`** **`kill -9 12345`**	Terminate a background job Terminate a process Interrupt a process as like using **`<Ctrl>-C`** Terminate a process more forcefully

Unix / Linux Signals

A *signal* in Unix/Linux is a "message" sent to a process. It is an *asynchronous notification* sent to the target process. A signal can be considered like an *interrupt*, but a signal is more often generated by external events, for example, a keyboard interrupt. A most common form of "interrupt" is for you to terminate a running program (i.e. a process), by pressing **`<Ctrl>-C`**. When **`<Ctrl>-C`** is pressed, a signal **`INT`**, defined as a system constant **`SIGINT`**, is sent to the process connected to the keyboard, i.e. the foreground process. Actually **`SIGINT`** has a value of **2**, i.e. signal number **2** and is called the *terminal interrupt* signal. Under normal situation, the process will be terminated. The effect of pressing **`<Ctrl>-C`** on the keyboard is the same as executing the command **`kill -2 12345`**, assuming that the process id of the foreground process is 12345.

What does **`kill -9 12345`** mean? There are a number of signals that could be sent to a process. The most common one is **`SIGINT`**, which is generated by pressing **`<Ctrl>-C`**. The default signal without beings specified in a **`kill`** command is signal number **15**, i.e. **`SIGTERM`**, or the *termination signal*. Thus, executing **`kill 12345`** is the same as executing **`kill -15 12345`**. You can see that **`kill -9 12345`** means to send signal number **9**, or **`SIGKILL`**, to process **`12345`**. This is the *strongest signal* that could be sent, and all processes cannot ignore or handle this strongest signal. Other common signals include **`SIGQUIT`** (signal number **3**, *quit terminal*), **`SIGFPE`** (signal number **8**, *floating point error*), **`SIGPIPE`** (signal number **13**, writing to a *pipe without a reader*; pipes to be covered in **Interprocess communication and programming**), **`SIGSEGV`** (signal number **11**, *invalid memory reference*; to be covered in **Memory management**), **`SIGILL`** (signal number **4**, *illegal instruction*; to be covered in **COMP 2421**). The full list of signals available on the particular Unix/Linux system can be found inside **`/usr/include/bits/signum.h`**. A number of signals are hardware-dependent.

Signal **`SIGSEGV`** is the cause of the most common error when you run a C program: **`Segmentation fault (core dumped)`**. For example, try the following simple buggy C program **lab2B.c**.

```
// lab 2B
#include <stdio.h>

int main() {
  int n;
  printf("Please enter a number\n");
  scanf("%d",n); // forget to use address in scanf, i.e. &n
  printf("The number is %d\n",n);
  printf("Program terminates\n");
}
```

When a signal is *sent* to a process, the operating system *interrupts* the execution of the target process to *deliver* the signal. If there is a *handler* (like an interrupt handler) for the signal, called *signal handler*, it would be executed in a similar way as an interrupt handler. The process execution will be *suspended* and control is passed to the signal handler. When the signal handler returns, the process execution will *resume*. This action of having a signal handler to handle a given signal is called "*catching* the signal". For those who are familiar with Java, this is similar to the exception handling mechanism using exceptional handlers, by treating a signal like an exception. It is also somewhat similar to an *event handler* or *event listener* in event-driven programming, e.g. to handle a mouse button event.

If there is no handler for a particular signal, i.e., the signal is not caught by a signal handler, the process will normally be *terminated*. Conceptually, you could treat *signals* like *interrupts*. The difference between interrupts and signals is that interrupts are often generated by hardware for the operating system kernel to respond to, e.g. completion of I/O, via interrupt handlers inside the operating system, and signals are often generated by software or the operating system for a process to respond to, e.g. a command at the shell asking to send a signal to a process via **`kill`**. Some hardware interrupts are passed by the operating system kernel as signals to the process, e.g. **`SIGFPE`**, **`SIGSEGV`** and **`SIGILL`**. The interrupt handlers inside the operating system for these hardware interrupts will process them first, and then generate the corresponding signals to the process causing them for handling inside user program.

Signal Handling

A *signal handler* is like a procedure or function. Furthermore, since a signal handler is *asynchronous* with the execution of the process, there is a possibility of *race conditions* when accessing to shared variables. Race condition will be cover later in the lecture on **Synchronization**. For simplicity, just imagine that two users are modifying a *shared linked list* together and the pointers could be pointing to wrong nodes when the two users are updating them in bad timing or ordering. It is often safer to use the **`write`**() system call (details to be covered in future labs) instead of the **`printf`**() library routine for output inside a signal handler, in order to reduce the problem of potential race conditions.

After defining the code for the signal handler, you need to *install* the signal handler to the program before it takes effect. You will use the system call **`signal`**() to install it. Without this line, the signal will not be caught inside the process. The following program **lab2B1.c** shows an example to install the **`SEGV`** signal handler in order to be able to catch the error in **lab2B.c** above.

```
// lab 2B1
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

void sighandler(int signo) {
  char s[]="SEGV signal is caught, please debug your program\n";
  write(1,s,strlen(s)); // print to screen
  exit(1);
}

int main() {
  int n;
  // you could also try removing the following line
  signal(SIGSEGV, sighandler); // install signal handler
  printf("Please enter a number\n");
  scanf("%d",n); // forget to use address in scanf, i.e. &n
  printf("The number is %d\n",n);
  printf("Program terminates\n");
}
```

Try the following program **lab2C.c** on catching the **`INT`** signal for the **`<Ctrl>-C`** keyboard event.

```
// lab 2C
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

void sighandler(int signo) {
  char s[]="INT signal is caught, program terminated by signal handler\n";
  write(1,s,strlen(s)); // print to screen
  exit(1);
}

int dosomething(int n) { // use up some time
  int i;
  for (i = 0; i < 1000000000; i++) {}
  printf("%d\n",n);
}

int main() {
  int i;
  signal(SIGINT, sighandler); // install signal handler
  printf("Program starts\n");
  for (i = 1; i <= 10; i++) dosomething(i);
  printf("Program terminates\n");
}
```

You could choose to *reset* the signal handler back to its default setting in the middle of program execution. Try the following program **lab2C1.c** and hit **<Ctrl>-C**. Here we choose to *revert back* to the default behavior after the first **INT** signal generated by **<Ctrl>-C** is caught, by setting the *default* option (**SIG_DFL**) for the signal when the first time the signal handler is executed. Now you can use **<Ctrl>-C** to terminate the program after the first **<Ctrl>-C**, since *termination* is the *default* action for the subsequent **INT** signal. You will *not* see any message generated by **<Ctrl>-C** this second time, since the signal handler **sighandler()** that prints the message has been reset to the default one in Linux. The process will be terminated as the default action.

```
// lab 2C1
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

void sighandler(int signo) {
  char s[]="INT signal is caught, but future <ctrl>-C will work as usual\n";
  write(1,s,strlen(s)); // print to screen
  signal(SIGINT, SIG_DFL); // reset to default, i.e. terminate process
}

int main() {
  signal(SIGINT, sighandler); // install signal handler
  printf("Program starts\n");
  while (1) { } // an infinite loop
  printf("Program terminates\n");
}
```

More Signal Handling (Optional)

Warning: before proceeding with this optional part, you *must* be familiarized with ways to terminate a process via the **`kill`** command. Otherwise, you may not be able to take back the terminal or keyboard to continue with your work if you do not have multiple windows with multiple connections. If you are not too sure about your ability and competence on using Linux, you should simply ***stop here***.

Try the following program **lab2D.c**. It looks very similar to **lab2C.c**, but now you are unable to terminate this program via **`<Ctrl>-C`**. Compare this program with **lab2C.c** and try to explain why there is such a difference in behavior.

```
// lab 2D
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

void sighandler(int signo) {
  char s[]="INT signal is caught, sorry that your <ctrl>-C is not working\n";
  write(1,s,strlen(s)); // print to screen
}

int dosomething(int n) { // use up some time
  int i;
  for (i = 0; i < 1000000000; i++) {}
  printf("%d\n",n);
}

int main() {
  int i;
  signal(SIGINT, sighandler); // install signal handler
  printf("Program starts\n");
  for (i = 1; i <= 10; i++) dosomething(i);
  printf("Program terminates\n");
}
```

Besides catching and processing a signal, you could choose to *ignore* the signal instead of installing the signal handler. Try the program **lab2D1.c**. Here we choose to ignore the second time when this **`INT`** signal is caught, by setting the *ignore* option (**`SIG_IGN`**) when the first time the signal handler is executed. Now you will no longer see any message inside the signal handler, since **`<Ctrl>-C`** is simply ignored, and the corresponding **`INT`** signal is *not* sent to the process. The process remains running inside the infinite loop unless being killed explicitly, via the **`kill`** command, or other proper means. Note that ignoring a signal is similar in effect to disabling the interrupt at the kernel.

```
// lab 2D1
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

void sighandler(int signo) {
  char s[]="INT signal is caught , but future <ctrl>-C will be ignored\n";
  write(1,s,strlen(s)); // print to screen
  signal(SIGINT, SIG_IGN); // ignore future SIGINT signal
}

int main() {
  signal(SIGINT, sighandler); // install signal handler
  printf("Program starts\n");
  while (1) { } // an infinite loop
  printf("Program terminates\n");
}
```

Can you figure out why the use of **<Ctrl>-C** can terminate the following program **lab2D2.c**?

```
// lab 2D2
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

int n = 1;

void sighandler(int signo) {
  char s[]="INT signal is caught, your <ctrl>-C may not work\n";
  write(1,s,strlen(s)); // print to screen
  n = 2; // possibility of race conditions in accessing n
}

int main() {
  signal(SIGINT, sighandler); // install signal handler
  printf("Program starts\n");
  while (n == 1) { } // an infinite loop
  printf("Program terminates\n");
}
```

You can install *multiple* signal handlers in a program, to handle different signals. The following program **lab2E.c** is able to handle several signals, including **<Ctrl>-C**, segmentation fault, arithmetic error, as well as termination signal (i.e. a **kill** command without specifying the signal to be sent to the process). Try to hit **<Ctrl>-C**, then try **kill**, **kill -15**, **kill -2**, **kill -8**, **kill -11**, **kill -13**, **kill -1**. You may need to execute the process in the background, or put it to background, in case you have just one terminal, so that you could execute those **kill** commands on the shell. What do you see? Run the program again and then **kill -2**, **kill -9**. What do you see? Note the existence of a message to be printed inside the signal handler for **KILL**.

```
// lab 2E
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

int n = 1;

void sighandler2(int signo) {
  char s[]="INT signal is caught\n";
  write(1,s,strlen(s)); // print to screen
}

void sighandler8(int signo) {
  char s[]="FPE signal is caught\n";
  write(1,s,strlen(s)); // print to screen
}

void sighandler11(int signo) {
  char s[]="SEGV signal is caught\n";
  write(1,s,strlen(s)); // print to screen
}

void sighandler13(int signo) {
  char s[]="PIPE signal is caught\n";
  write(1,s,strlen(s)); // print to screen
}

void sighandler15(int signo) {
  char s[]="TERM signal is caught\n";
  write(1,s,strlen(s)); // print to screen
}
```

```
void sighandler9(int signo) {
  char s[]="KILL signal is caught\n";
  write(1,s,strlen(s)); // print to screen
}

int main() {
  signal(SIGINT, sighandler2);
  signal(SIGFPE, sighandler8);
  signal(SIGSEGV, sighandler11);
  signal(SIGPIPE, sighandler13);
  signal(SIGTERM, sighandler15);
  signal(SIGKILL, sighandler9);
  printf("Program starts\n");
  while (n == 1) { } // an infinite loop
  printf("Program terminates\n");
}
```

For program **lab2E.c** above, even though you have installed the signal handler for **`SIGKILL`**, it has *no effect* on the program, so that you *do not see* the message inside the signal handler for **`SIGKILL`**. Even if we send the **`KILL`** signal by executing **`kill -9`** to that process, there is also no effect. This is because the **`KILL`** signal *cannot be caught nor ignored*, as designed by Unix / Linux systems.

Question: what is the consequence if the **`KILL`** signal can be caught or ignored, just like **`SIGINT`** or **`SIGTERM`**?

Normally, signal handlers are installed so as to *change the default behavior* of the corresponding signals. If you do not want to change the default behavior, you do not need to install the signal handlers. This has the same effect as executing **`signal(SIGnnn,SIG_DFL)`** in the program, where **`SIG_DFL`** indicates the default. You could even change the signal handler for a signal *dynamically*, as in **lab2E1.c** below. In this program, three different signal handlers for **`SIGINT`** have been installed over time. Try hitting **`<Ctrl>-C`** multiple times and see the different outputs from different signal handlers.

```
// lab 2E1
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

void sighandler3(int signo) {
  char s[]="INT signal is caught the third time, program will terminate\n";
  write(1,s,strlen(s)); // print to screen
  exit(1);
}

void sighandler2(int signo) {
  char s[]="INT signal is caught again\n";
  write(1,s,strlen(s)); // print to screen
  signal(SIGINT, sighandler3); // install a third signal handler
}

void sighandler(int signo) {
  char s[]="INT signal is caught\n";
  write(1,s,strlen(s)); // print to screen
  signal(SIGINT, sighandler2); // install another signal handler
}

int main() {
  signal(SIGINT, sighandler); // install first signal handler
  printf("Program starts\n");
  while (1) { } // an infinite loop
  printf("Program terminates\n");
}
```

Signal Generation (Optional)

We have mentioned about sending signals from the shell or terminal to a process, using the command **`kill`**. However, we can also *send* signals from *inside a program* (running as a process) using the **`kill`**() system call, instead from the terminal to another process using the **`kill`** command. As a result, one can write a program to *terminate* selected processes belonging to the same user, by iterating through the processes found and sending them the **`KILL`** signals. This is a common procedure executed by the system administrator when shutting down a system in order to terminate other processes. In fact, the system administrator has the right to terminate *all processes* (except **`init`** process with pid **`1`**), by setting pid to be **`-1`**.

Since signals can be sent to another process for it to catch, one might even design the programs in a way to achieve *communication* by sending signals, as a handicapped version of TCP/IP in networking. For example, process *P* can send a signal, e.g. **`SIGUSR1`** or **`SIGUSR2`** (user signal 1 or user signal 2) to process *Q* to inform *Q* that *P* has done something important and *Q* should now proceed with execution instead of waiting inside an infinite loop, upon catching the signal. A better mechanism to communicate is via the **`pipe`** mechanism that we will cover in **Lecture 5** and subsequent lab. The **`pipe`** can be considered as a local version of TCP/IP, for communication within the *same* machine (**localhost**) instead of across different machines.

Here is a very simple implementation of the **`kill`** command by means of the **`kill`**() system call. You could compile **lab2F.c** and use the resultant executable like a simplified **`kill`** command.

```
// lab 2F
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int main(int argc, char *argv[]) {
  int pid, signo;

  if (argc < 2 || argc > 3) { // give usage hint
     printf("Usage: %s [-signo] pid\n", argv[0]);
     exit(1);
  }
  if (argv[1][0] == '-') { // kill -signo pid
     signo = atoi(&argv[1][1]);
     pid = atoi(argv[2]);
  } else { // kill pid
     signo = SIGTERM;
     pid = atoi(argv[1]);
  }
  printf("Sending signal %d to process %d\n", signo, pid);
  kill(pid,signo);
}
```

Besides sending a signal from inside a process to another process with the **`kill`**() system call, a process can also send a signal to *itself* when it is executing, by executing the **`raise`**() system call. Again, this arrangement resembles the Java mechanism in raising and catching an exception. Try to run the following program **lab2G.c** and try to hit **`<Ctrl>-C`**. What do you see? When do you think that the program would terminate?

```
// lab 2G
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

int n = 1;

void sighandler2(int signo) {
```

```
  char s[]="INT signal is caught\n";
  write(1,s,strlen(s)); // print to screen
  n++;
  raise(SIGSEGV);
}

void sighandler11(int signo) {
  char s[]="SEGV signal is caught\n";
  write(1,s,strlen(s)); // print to screen
  puts("Is it really segmentation fault?");
}

int main() {
  signal(SIGINT, sighandler2);
  signal(SIGSEGV, sighandler11);
  printf("Program starts\n");
  while (n < 5) { } // an infinite loop
  printf("Program terminates\n");
}
```

Try the following awkward program **lab2H.c**. To make the program continue executing, hit **<Ctrl>-C**. Observe the outputs and try to follow the execution sequence of the program. It is a very bad way to control the ordering of program statement execution in such a strange way that is so hard to understand. Yet jumping around within a program in some strange way could be considered an *obfuscation technique* adopted in computer security to produce **Dalvik** (the virtual machine for **Android**) **.dex** bytecode that could possibly avoid other people from reverse-engineering the original code from the bytecode.

```
// lab 2H
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

int n = 1;

void sighandler2(int signo) {
  char s[]="INT signal is caught\n";
  write(1,s,strlen(s)); // print to screen
  n = n+2;
  if (n % 2 == 1) raise(SIGSEGV);
  else raise(SIGFPE);
}

void sighandler8(int signo) {
  char s[]="FPE signal is caught\n";
  write(1,s,strlen(s)); // print to screen
  n--;
}

void sighandler11(int signo) {
  char s[]="SEGV signal is caught\n";
  write(1,s,strlen(s)); // print to screen
  n--;
}

int dosomething(int n) { // use up some time to reduce amount of screen output
  int i;
  for (i = 0; i < 1000000000; i++) {}
  printf("%d\n",n);
}

int main() {
  signal(SIGINT, sighandler2);
  signal(SIGFPE, sighandler8);
  signal(SIGSEGV, sighandler11);
  printf("Program starts\n");
```

```
   while (n < 10) {
      while (n == 1) dosomething(n); // an infinite loop
      printf("Past first loop\n");
      while (n == 2) dosomething(n); // another infinite loop
      printf("Past second loop\n");
      while (n == 3) dosomething(n); // third infinite loop
      printf("Past third loop\n");
      while (n == 4) dosomething(n); // fourth infinite loop
      printf("Past fourth loop\n");
      while (n == 5) dosomething(n); // fifth infinite loop
      printf("Past fifth loop\n");
   }
   printf("Past outer loop\n");
   printf("Program terminates\n");
}
```