CL-2006 Operating Systems

LAB - 09

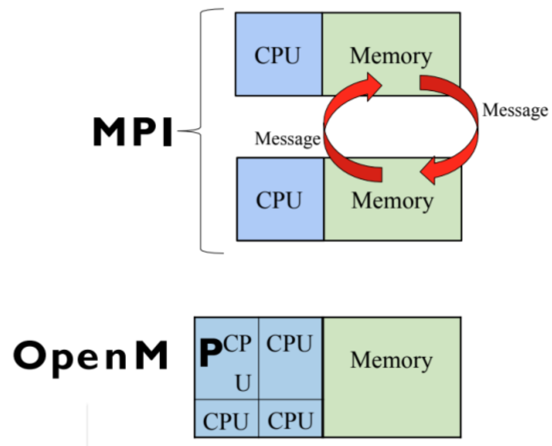
OMP parallel programming and Signal Handling

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
Spring 2023

Introduction to Parallel Programming with OpenMP

- MPI – Designed for distributed memory
  1. Multiple systems Message
  2. Send/receive messages.

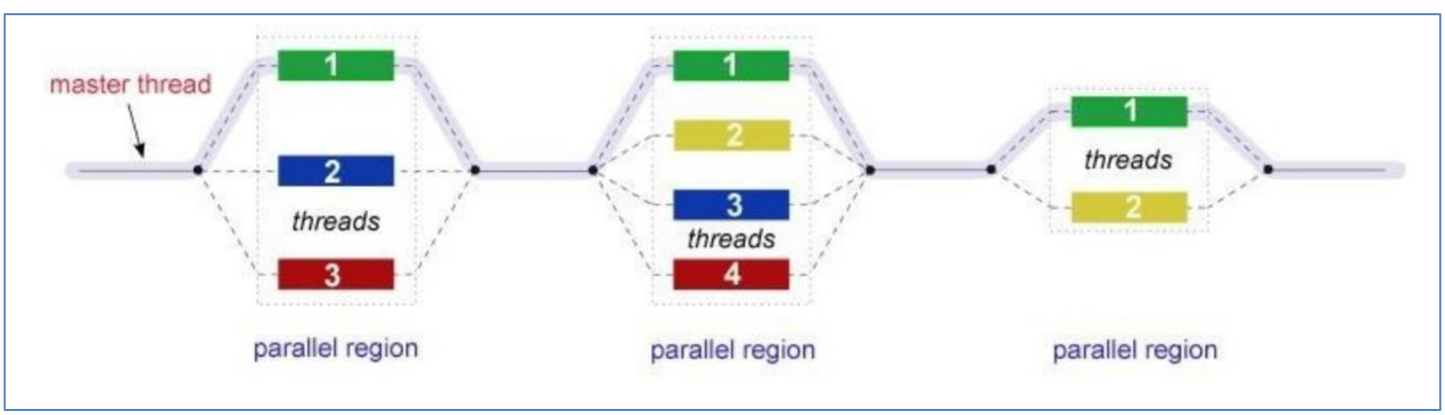
- OpenMP – Designed for shared memory
  1. Single system with multiple cores
  2. One thread/core sharing memory



- C, C++, and Fortran
- There are other options
- Interpreted languages with multithreading.
- Python, R, matlab (have OpenMP & MPI underneath)
- CUDA, OpenACC (GPUs)
- Pthreads, Intel Cilk Plus (multithreading)
- OpenCL, Chapel, Co-array Fortran, Unified Parallel C (UPC)

- Open Multi-Processing
  a. Completely independent from MPI
  b. Multi-threaded
- Standard since 1997
  a. Defined and endorsed by the major players
- Fortran, C, C++
- Requires compiler to support OpenMP
  a. Nearly all do.
- For shared memory machines
  a. Limited by available memory.
  b. Some compilers support GPUs.

Fork - Join Model: Three Components



Preprocessor directives

- Always start with #

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

- OpenMP directives tell the compiler to add machine code for parallel execution of the following

```
#pragma omp parallel
```

- “Run this next set of instructions in parallel”

OpenMP compiler directives Always start with # are used for various purposes:

a. Spawning a parallel region
b. Dividing blocks of code among threads
c. Distributing loop iterations between threads
d. Serializing sections of code
e. Synchronization of work among threads

Some OpenMP Subroutines

- Returns max possible (generally set by OMP_NUM_THREADS)

```
int omp_get_max_threads()
```

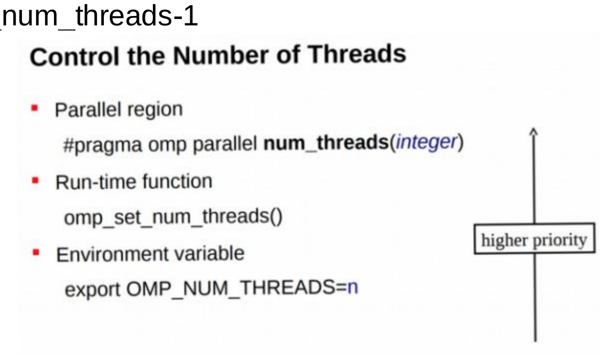
- Returns number of threads in current team

```
int omp_get_num_threads()
```

- Returns thread id of calling thread

```
int omp_get_thread_num()
```

- Between 0 and omp_get_num_threads-1



OpenMP Code Structure

```
#include<omp.h>
int main () {

int var1, var2, var3;
  Serial code
        .
        .
        .
Beginning of parallel region. Fork a team of threads.
   Specify variable scoping
   #pragma omp parallel private(var1, var2) shared(var3)
      {
      Parallel region executed by all threads           .
      Other OpenMP directives
      Run-time Library calls
      All threads join master thread and disband
      }
Resume Serial Code
     .
     .
     .
}
```

OpenMP Constructs

- Parallel region
    a. Thread creates team, and becomes master (id 0)
    b. All threads run code after
    c. Barrier at end of parallel section

```
#pragma omp parallel [clause ...]
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     num_threads (integer)


   structured_block                                (not a complete list)
```

First OpenMP program- Hello World

```
#include <omp.h> //<-- necessary header file for OpenMP API
#include <stdio.h>

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

   printf("OpenMP running with %d threads\n", omp_get_max_threads());

#pragma omp parallel
   {
      //Code here will be executed by all threads
      printf("Hello World from thread %d\n", omp_get_thread_num());
   }

   return 0;
}
```

Running OpenMP Hello World

```
[user@adroit4]$ gcc -o hello_world_omp hello_world_omp.c -fopenmp
```

Compiler flag to enable OpenMP

(-fopenmp for gcc)
(-qopenmp-stubs for icc serial)

Environment variable defining max threads

```
[user@adroit4]$ export OMP_NUM_THREADS=4
[user@adroit4]$ ./hello_world_omp
OpenMP running with 4 threads
Hello World from thread 1
Hello World from thread 0
Hello World from thread 2
Hello World from thread 3
```

- OMP_NUM_THREADS defines run time number of threads can be set in code as well using: omp_set_num_threads()

Shared and Private Variables

- Important note about compilers
- C (before C99) does not allow variables declared in for loop syntax
- Compiler will make loop variables private
- Still recommend explicit.

C

```
#pragma omp parallel private(i)
  for (i=0; i<N; i++) {
    b = a + i;
  }
```

C+ +

```
#pragma omp parallel
  for (int i=0; i<N; i++) {
    b = a + i;
  }
```

Automatically private

Example 1: Table Computation

Below code prints table using OpenMP for loop Parallelization

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

int main() {
        int num;
        int i;
        printf("Table [PROMPT] Enter Your Number: "); scanf("%d",&num);

        #pragma omp parallel num_threads(10)
        #pragma omp for
        for(i=0;i<10000;i++) {
                printf("Table [INFO] Thread ID: %d | %d X %d = %d \n", omp_get_thread_num(), i, num,
i*num );
        }

        return 0;
}
```

OMP parallel for Private Variable 1

```
#include <omp.h>
#include <stdio.h>
int main() {
  int i;
  const int N = 1000;
  int a = 50;
  int b = 0;

#pragma omp parallel for default(shared)
  for (i=0; i<N; i++) {
    b = a + i;
  }

  printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```

```
[user@adroit3]$ gcc -fopenmp omp_private_1.c -o omp_private_1
[user@adroit3]$ export OMP_NUM_THREADS=1
[user@adroit3]$ ./omp_private_1
a=50 b=1049 (expected a=50 b=1049)
[user@adroit3]$ export OMP_NUM_THREADS=4
[user@adroit3]$ ./omp_private_1
a=50 b=799 (expected a=50 b=1049)
```

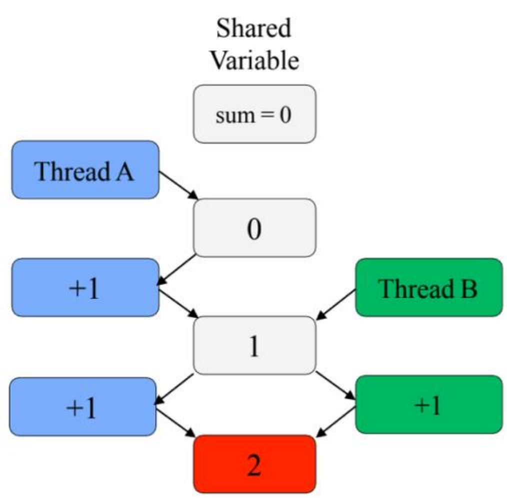
OMP parallel for Private Variable 2

```c
#include <omp.h>
#include <stdio.h>
int main() {
  int i;
  const int N = 1000;
  int a = 50;
  int b = 0;

#pragma omp parallel for default(none) private(i) private(a) private(b)
  for (i=0; i<N; i++) {
    b = a + i;
  }

  printf("a=%d b=%d (expected a=50 b=1049)\n", a, b);
}
```

```
[user@adroit3]$ gcc -fopenmp omp_private_2.c -o omp_private_2
[user@adroit3]$ export OMP_NUM_THREADS=4
[user@adroit3]$ ./omp_private_2
a=50 b=0 (expected a=50 b=1049)
```

Race Condition

- When multiple threads simultaneously read/write shared variable
- Multiple OMP solutions
  - Reduction
  - Atomic
  - Critical




```c
#pragma omp parallel for private(i) shared(sum)
  for (i=0; i<N; i++) {
    sum += i;
  }
```

Should be 3!

OMP Reduction:

```
#pragma omp reduction (operator:variable)
```

- Avoids race condition
- Reduction variable must be shared
- Makes variable private, then performs operator at end of loop
- Operator cannot be overloaded (c++)
- One of: +, *, -, / (and &, ^, |, &&, ||)
- OpenMP 3.1: added min and max for c/c++

```c
#include <omp.h>
#include <stdio.h>

int main()

  {  int i;
  const int N = 1000;
  int sum = 0;

#pragma omp parallel for private(i) reduction(+: sum)
  for (i=0; i<N; i++) {
     sum += i;
  }

  printf("reduction sum=%d (expected %d)\n", sum, ((N-1)*N)/2);
```

```
[user@adroit3]$ gcc -fopenmp omp_race.c -o omp_race.out
[user@adroit3]$ export OMP_NUM_THREADS=4
[user@adroit3]$ ./omp_race.out
reduction sum=499500 (expected 499500)
```

OMP Lab Exercise Questions:

1. Run the codes given in the manual.
2. Calculate the following series with and without open MP and calculate the time in each case . Also append the Time.

$$1 + 1/2 + 1/3 + 1/4 + .. + 1/n$$

3. Calculate Addition of two matrices with and without open MP and calculate the time.

Signal Handling

Introduction

Programs must sometimes deal with unexpected or unpredictable events, such as:

- a floating point error
- a power failure an alarm clock "ring"
- the death of a child process
- a termination request from a user (i.e., a Control-C)
- a suspend request from a user (i.e., a Control-Z)

These kinds of events are sometimes called interrupts, as they must interrupt the regular flow of a program in order to be processed. When UNIX recognizes that such an event has occurred, it sends the corresponding process a signal.

The kernel isn't the only one that can send a signal; any process can send any other process a signal, as long as it has permissions.

A programmer may arrange for a particular signal to be ignored or to be processed by a special piece of code called a signal handler. In the latter case, the process that receives the signal suspends its current flow of control, executes the signal handler, and then resumes the original flow of control when the signal handler finishes.

Signals inform processes of the occurrence of asynchronous events. Every type of signal has a handler which is a function. All signals have default handlers which may be replaced with user-defined handlers. The default signal handlers for each process usually terminate the process or ignore the signal, but this is not always the case.

Signals may be sent to a process from another process, from the kernel, or from devices such as terminals. The ^C, ^Z, ^S and ^Q terminal commands all generate signals which are sent to the foreground process when pressed.

The kernel handles the delivery of signals to a process. Signals are checked for whenever a process is being rescheduled, put to sleep, or re-executing in user mode after a system call.

Types of Signals

There are 64 different signals defined in "/usr/include/signal.h". A programmer may choose for a particular signal to trigger a user-supplied signal handler, trigger the default kernel-supplied handler, or be ignored. The default handler usually performs one of the following actions:

- terminates the process and generates a core file (dump) terminates
- the process without generating a core image file (quit) ignores and
- discards the signal (ignore)
- suspends the process (suspend)
- resumes the process.

SIGINT

Interrupt. Sent to every process associated with a control terminal when the interrupt key (Control-C) is hit. This action of the interrupt key may be suppressed, or the interrupt key may be changed using the stty command. Note that suppressing the interrupt key is completely different from ignoring the signal, although the effect (or lack of it) on the process is the same.

SIGTSTP
Interrupt. Sent to every process associated with a control terminal when the interrupt key (Control-Z) is hit. This action of the interrupt key may be suppressed, or the interrupt key may be changed using the stty command. Note that suppressing the interrupt key is completely different from ignoring the signal, although the effect (or lack of it) on the process is the same.

SIGQUIT
Quit. Similar to SIGINT but sent when the quit key (normally Control-\) is hit. Commonly sent in order to get a core dump.

SIGILL
Illegal instruction. Sent when the hardware detects an illegal instruction. Sometimes a process using floating point aborts with this signal when it is accidentally linked without the -f option on the cc command. Since C programs are in general unable to modify their instructions, this signal rarely indicates a genuine program bug.

SIGTRAP
Trace trap. Sent after every instruction when a process is run with tracing turned on with ‘ptrace’.

SIGIOT
I/O trap instruction. Sent when a hardware fault occurs, the exact nature of which is up to the implementer and is machine dependent. In practice, this signal is preempted by the standard subroutine abort, which a process calls to commit suicide in a way that will produce a core dump.

SIGEMT
Emulator trap instruction. Sent when an implementation-dependent hardware fault occurs. Extremely rare.

SIGFPE
Floating-point exception. Sent when the hardware detects a floating-point error, such as a floating point number with an illegal format. Almost always indicates a program bug.

SIGKILL
Kill. The one and only sure way to kill a process, since this signal is always fatal (can't be ignored or caught). To be used only in emergencies; SIGTERM is preferred.

SIGBUS
Bus error. Sent when an implementation-dependent hardware fault occurs. Usually means that the process referenced at an odd address data that should have been word aligned.

SIGALRM
Alarm clock. Sent when a process's alarm clock goes off. The alarm clock is set with the alarm system call.

SIGTERM
Software termination. The standard termination signal. It's the default signal sent by the kill command, and is also used during system shutdown to terminate all active processes. A program should be coded to either let this signal default or else to clean (e.g., remove temporary files) and call exit.

SIGUSR1

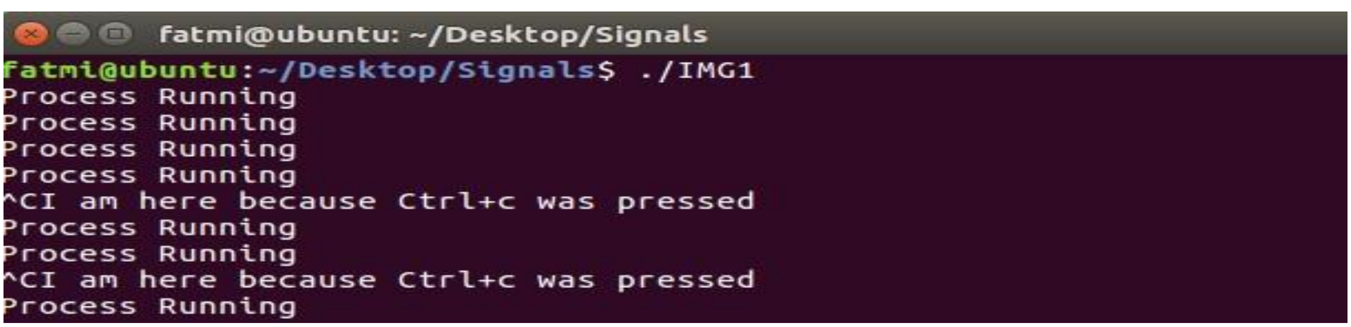
User defined signal 1. This signal may be used by application programs for interprocess communication. This is not recommended however, and consequently this signal is rarely used.

SIGUSR2

User-defined signal 2. Like SIGUSR1

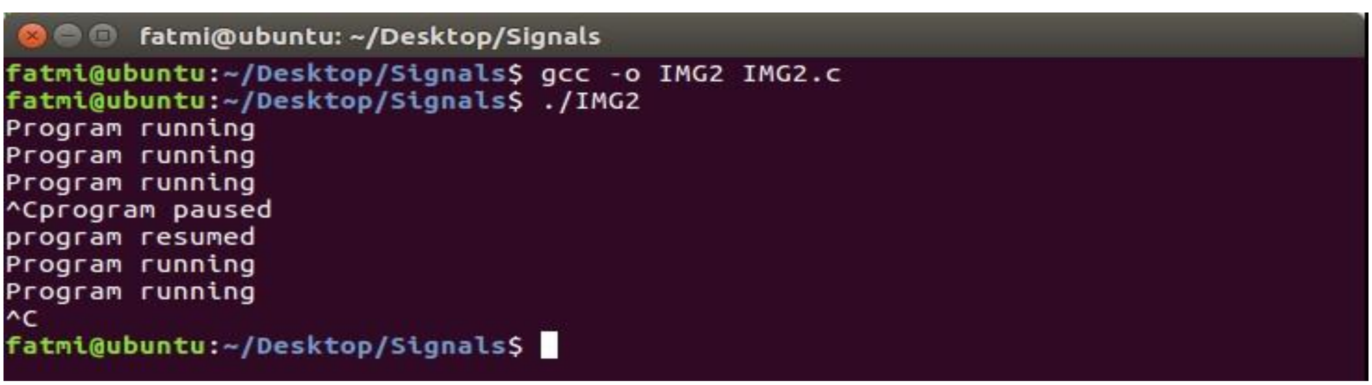
SIGINT Example #1

```
#include<stdio.h>
#include<unistd.h>
#include<wait.h>
#include<signal.h>
int state=1;
void signal_handler(int num){
            printf("I am here because Ctrl+c was pressed\n");
}
 int main()
 {
            signal(SIGINT, signal_handler);
            while(1){
            sleep(1);
            printf("Process Running \n");
                        }
            return 0;
  }
```



1. SIGINT Example #2

```
#include<stdio.h>
#include<unistd.h>
#include<wait.h>
#include<signal.h>
int state=0;

void signal_handler(int num){

if(state==0)
{
        printf("program paused\n");
        state=1;
}
if(state==1)
{       printf("program resumed\n");
        state=0;
}

 signal(SIGINT, SIG_DFL);
}

int main()
{
 signal(SIGINT, signal_handler);
 while(1){
        sleep(1);
        printf("Program running\n");
}
 return 0;
}
```



```
fatmi@ubuntu: ~/Desktop/Signals
fatmi@ubuntu:~/Desktop/Signals$ gcc -o IMG2 IMG2.c
fatmi@ubuntu:~/Desktop/Signals$ ./IMG2
Program running
Program running
Program running
^Cprogram paused
program resumed
Program running
Program running
^C
fatmi@ubuntu:~/Desktop/Signals$
```

Requesting An Alarm Signal: alarm ( )

One of the simplest ways to see a signal in action is to arrange for a process to receive an alarm clock signal, SIGALRM, by using alarm ( ). The default handler for this signal displays the message "Alarm Clock" and terminates the process. Here's how alarm ( ) works:

```
int alarm (int count)
```

alarm( ) instructs the kernel to send the SIGALRM signal to the calling process after count seconds. If an alarm had already been scheduled, it is overwritten. If count is 0, any pending alarm requests are cancelled. alarm( ) returns the number of seconds that remain until the alarm signal is sent.Here's a small program that uses alarm ( ) together with its output.

```
#include <stdio.h>
main ( )
{
//alarm (5) ; /* schedule an alarm signal in 5 seconds */
printf ("Looping forever ...\n") ;
while ( 1 ) ;
printf ("This line should never be executed.\n") ;
}
```

‘signal’ System Call

```
#include <signal.h>

void (*signal(sig, func))() /* Catch signal with func */

void (*func)();/* The function to catch the sig

/*Returns the previous handler or -1 on error */
```

The declarations here baffle practically everyone at first sight. All they mean is that the second argument to signal is a pointer to a function, and that a pointer to a function is returned.

The first argument, sig, is a signal number. The second argument, func, can be one of three things:

- **SIG_DFL**. This sets the default action for the signal.
- **SIG_IGN**. This sets the signal to be ignored; the process becomes immune to it. The signal **SIGKILL** can't be ignored. Generally, only **SIGHUP, SIGINT, and SIGQUIT** should ever be permanently ignored. The receipt of other signals should at least be logged, since they indicate that something exceptional has occurred.
- A pointer to a function. This arranges to catch the signal; every signal but SIGKILL may be caught. The function is called when the signal arrives.

The signals SIGKILL and SIGSTP may not be reprogrammed.

A child process inherits a parent’s action for a signal. Actions SIG_DFL and SIG_IGN are preserved across an exec but caught signals are reset to SIG_DFL. This is essential because the catching function will be overwritten by the new code. Of course, the new program can set its own signal handlers. Arriving signals are not queued. They are either ignored, they terminate the process, or they are caught. This is the main reason why signals are inappropriate for inter-process communication -- a message in the form of a signal might be lost if it arrives when that type of signal is temporarily ignored. Another problem is that arriving signals are rather rude. They interrupt whatever is currently going on, which is complicated to deal with properly, as we'll see shortly. The signal returns the previous action for the signal. This is used if it's necessary to restore it to the way it was.

Defaulting and ignoring signals is easy; the hard part is catching them. To catch a signal you supply a pointer to a function as the second argument to signal.

When the signal arrives two things happen, in this order:

- The signal is reset to its default action, which is usually termination. Exceptions are SIGILL and SIGTRAP, which are not reset because they are signaled too often.
- The designated function is called with a single integer argument equal to the number of the signal that it caught. When and if the function returns, processing resumes from the point where it was interrupted.
- If the signal arrives while the process is waiting for any event at all, and if the signal catching function returns, the interrupted system call returns with an error return of EINTR -- it is not restarted automatically. You must distinguish this return from a legitimate error. Nothing is wrong -- a signal just happened to arrive while the system call was in progress.

‘pause’ System Call

Int pause()

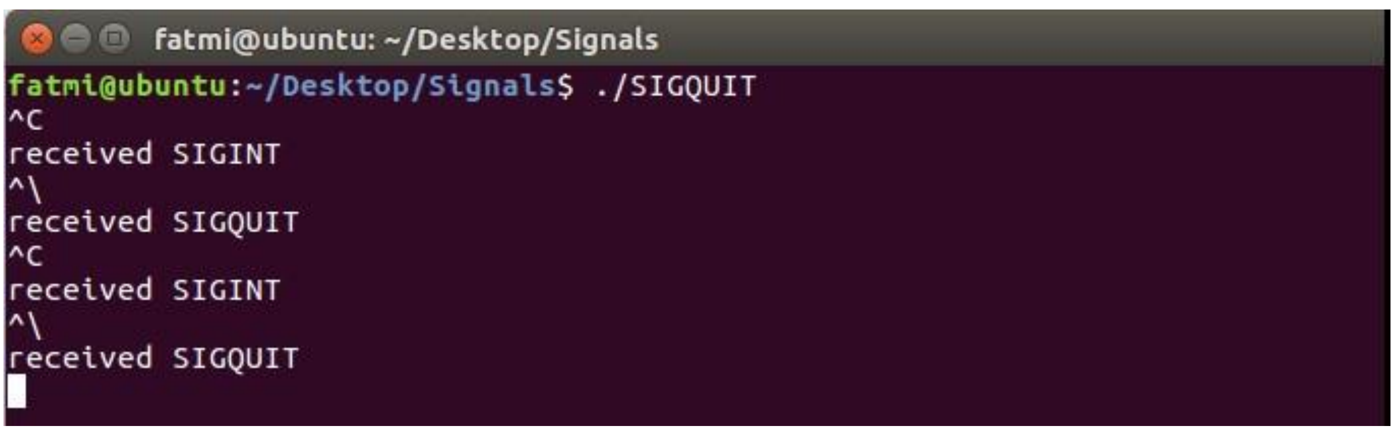
pause( ) suspends the calling process and returns when the calling process receives a signal. It is most often used to wait efficiently for an alarm signal. pause( ) doesn't return anything useful.

The following program catches and processes the SIGALRM signal efficiently by having user written signal handler, alarmHandler ( ), by using signal ( ).

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

int alarmFlag = 0 ;
void alarmHandler ( ) ;

  main ( ) {
  signal(SIGALRM, alarmHandler) ; /*Install signal Handler*/
  alarm (5) ;
  printf ("Looping ...\n") ;
  while (!alarmFlag) {
        pause ( ); /* wait for a signal */
        printf ("Loop ends due to alarm signal\n");
        }
   }

    void alarmHandler ( ) {
    printf ("An ALARM clock signal was received\n");
    alarmFlag = 1;
    }
```

2. SIGQUIT Example

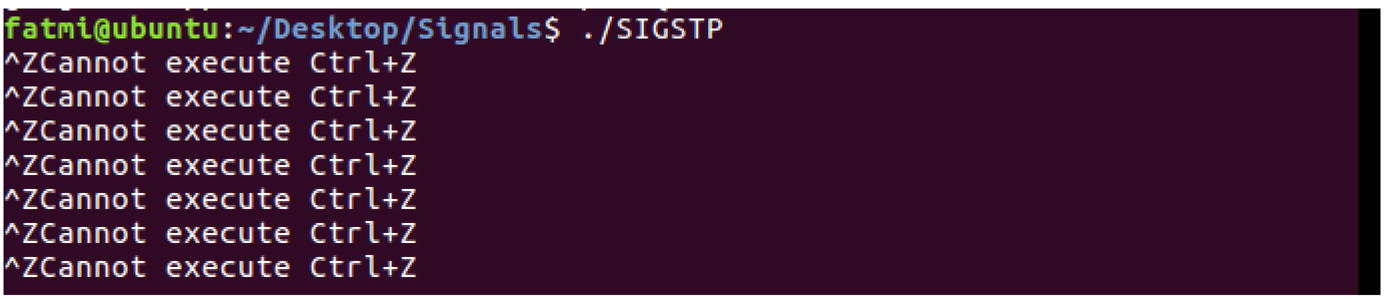
```
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
//signal handling function that will except ctrl-\ and ctrl-c
void sig_handler(int signo)
{
   //looks for ctrl-c which has a value of 2
   if (signo == SIGINT)
      printf("\nreceived SIGINT\n");
   //looks for ctrl-\ which has a value of 9
   else if (signo == SIGQUIT)
      printf("\nreceived SIGQUIT\n");
}

int main(void)
{
   //these if statement catch errors
   if (signal(SIGINT, sig_handler) == SIG_ERR)
      printf("\ncan't catch SIGINT\n");
   if (signal(SIGQUIT, sig_handler) == SIG_ERR)
      printf("\ncan't catch SIGQUIT\n");
   //Runs the program infinitely so we can continue to input signals
   while(1)
      sleep(1);
   return 0;
}
```



3. SIGTSTP Example

```
// C program that does not suspend when
// Ctrl+Z is pressed
#include <stdio.h>
#include <signal.h>

// Signal Handler for SIGTSTP
void sighandler(int sig_num)
{
    // Reset handler to catch SIGTSTP next time
  //  signal(SIGTSTP, sighandler);
```

```
  printf("Cannot execute Ctrl+Z\n");
}

int main()
{
   // Set the SIGTSTP (Ctrl-Z) signal handler
   // to sigHandler
   signal(SIGTSTP, sighandler);
   while(1)
   {
   }
   return 0;
```



Protecting Critical Code and Chaining Interrupt Handlers

The same techniques described previously may be used to protect critical pieces of code against Control-C attacks and other signals. In these cases, it's common to save the previous value of the handler so that it can be restored after the critical code has been executed. Here's the source code of theprogram that protects itself against SIGINT signals:

```
#include<stdio.h>
#include<signal.h>
main ( ) {
int (*oldHandler) ( ) ; /* holds old handler value */
printf ("I can be Control-C'ed \n") ;
sleep (5) ;
oldHandler = signal(SIGINT, SIG_IGN) ; /* Ignore Ctrl-C */
```

‘kill’ System Call

```
#include <signal.h>
int kill(pid, sig) /* Send the signal to the named process */
int pid;
int sig;
```

In the previous sections we mainly discussed signals generated by the kernel as a result of some exceptional event. It is also possible for one process to send a signal of any type to another process. pid is the process-ID of the process to receive the signal; sig is the signal number. The effective user-IDs of the sending and receiving processes must be the same, or else the effective user-ID of the sending process must be the super user.

If pid is equal to zero, the signal is sent to every process in the same process group as the sender. This feature is frequently used with the kill command (kill 0) to kill all background processes without referring to their process-IDs. Processes in other process groups (such as a DBMS you happened to have started) won't receive the signal.

If pid is equal to -1, the signal is sent to all processes whose real user-ID is equal to the effective user-ID of the sender. This is a handy way to kill all processes you own, regardless of process group.

In practice, kill is used 99% of the time for one of these purposes:

To terminate one or more processes, usually with SIGTERM, but sometimes with SIGQUIT so that a core dump will be obtained.

To test the error-handling code of a new program by simulating signals such as SIGFPE (floating-point exception).

Kill is almost never used simply to inform one or more processes of something (i.e., for inter-process communication), for the reasons outlined in the previous sections. Note also that the kill system call is most often executed via the kill command. It isn't usually built into application programs.

4. Using ‘sigaction’ to handle interrupts.

Another option that is frequently used to implement signal handling is sigaction, one working example isas follows

```
#include <stdio.h> //needs for perror
#include <signal.h> //signal.h
#include<wait.h>
#include<unistd.h>


  void handler(int signum){
   if(signum == SIGINT)
   {
   printf("CONTROL SIGNAL IS PRESSED!");
          }
    }
int main(){
 struct sigaction sa; //creating sa, which will be called in sigaction function with the control signal variable.
 sa.sa_handler = handler; //this is declaring which handler is used if control signal is passed to sa;
 while(1){
        printf("/");
        for(int i=0;i<=100000;i++){
        }
       if(sigaction(SIGINT, &sa, NULL) == -1)
       perror("S1GACTION");
        }
   return 0;
}
```