OS Tutorial

BITS Pilani
Pilani Campus

Interprocess communication (IPC)

Why IPC

- To share resources
- Client/server paradigms
- Inherently distributed applications, etc.

IPC methods

- Signal
- Pipe
- FIFO
- Message passing
- Shared Memory
- Semaphores

Interprocess Communication

- Each process has a private address space. Normally, no process can write to another process's space.
- How to get important data from process A to process B?

Interprocess Communication…

- Message passing between different processes running on the same operating system is IPC
- Synchronization is required in case of IPC through shared memory or file system

Interprocess Communication…

- Processes executing concurrently in the OS may be either **independent processes** or **cooperating processes**.

Interprocess Communication…

- A process is **independent** if it cannot affect or be affected by the other processes executing in the system.
- Any process that does not share data with any other process is independent.

Interprocess Communication…

- A process is **cooperating** if it can affect or be affected by the other processes executing in the system.
- Clearly, any process that shares data with other processes is a **cooperating process**.

Interprocess Communication…

- Reasons for cooperating processes:
  - **Information sharing:** Since several users may be interested in the same piece of information, an environment to allow concurrent access to such information is necessary.
  - **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. It can be noted that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).

Interprocess Communication…

- Reasons for cooperating processes:
  - **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
  - **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Signal

- A signal is an ***asynchronous*** event which is delivered to a process.
- ***Asynchronous*** means that the event can occur at any time
  - may be unrelated to the execution of the process
  - e.g. user types ctrl-C

Signal

- 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 (Control-C)
  - a suspend request from a user (Control-Z)

Signal

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

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.

Signal

- 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.

Signal

- 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***.

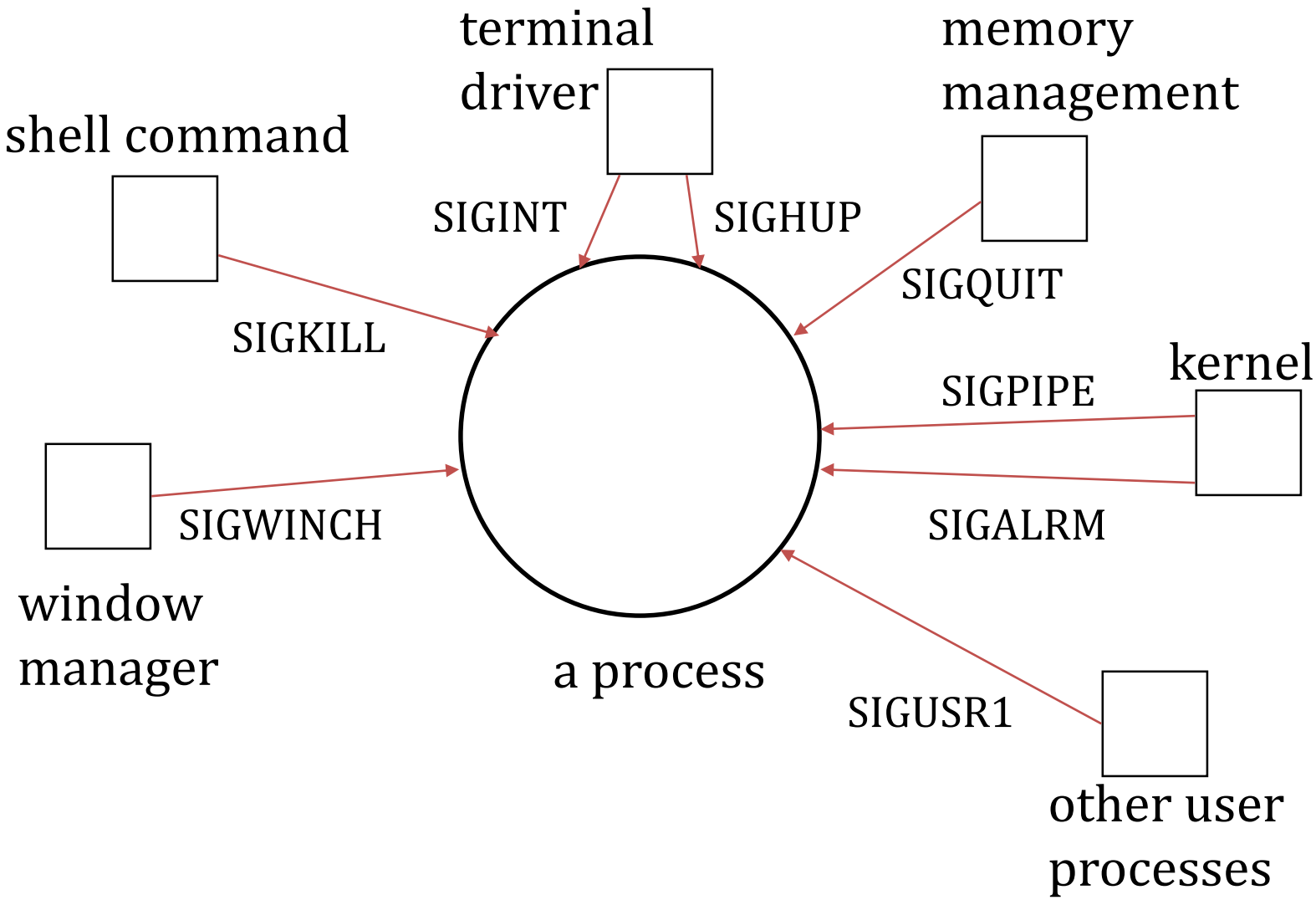
Signal

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

Signal

- 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.

Signal Sources



Types of Signal

- 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.

Signal

- 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

Signal

- There are a set of defined signals

  (1) SIGHUP (2) SIGINT (3) SIGQUIT (4) SIGILL (5) SIGTRAP (6) SIGIOT (7) SIGBUS (8) SIGFPE (9) SIGKILL (10) SIGUSR1 (11) SIGSEGV (12) SIGUSR2 (13) SIGPIPE (14) SIGALRM (15) SIGTERM (17) SIGCHLD (18) SIGCONT (19) SIGSTOP (20) SIGTSTP (21) SIGTTIN (22) SIGTTOU (23) SIGURG (24) SIGXCPU (25) SIGXFSZ (26) SIGVTALRM (27) SIGPROF (28) SIGWINCH (29) SIGIO (30) SIGPWR

- $ kill -l

Signal	Description	Default action
SIGABRT	Sent by `abort()`	Terminate with core dump
SIGALRM	Sent by `alarm()`	Terminate
SIGBUS	Hardware or alignment error	Terminate with core dump
SIGCHLD	Child has terminated	Ignored
SIGCONT	Process has continued after being stopped	Ignored
SIGFPE	Arithmetic exception	Terminate with core dump
SIGHUP	Process's controlling terminal was closed (most frequently, the user logged out)	Terminate
SIGILL	Process tried to execute an illegal instruction	Terminate with core dump
SIGINT	User generated the interrupt character (Ctrl-C)	Terminate
SIGIO	Asynchronous I/O event	Terminate[2]
SIGKILL	Uncatchable process termination	Terminate
SIGPIPE	Process wrote to a pipe but there are no readers	Terminate
SIGPROF	Profiling timer expired	Terminate
SIGPWR	Power failure	Terminate
SIGQUIT	User generated the quit character (Ctrl-\)	Terminate with core dump

Signal	Description	Default action
SIGSEGV	Memory access violation	Terminate with core dump
SIGSTKFLT	Coprocessor stack fault	Terminate[b]
SIGSTOP	Suspends execution of the process	Stop
SIGSYS	Process tried to execute an invalid system call	Terminate with core dump
SIGTERM	Catchable process termination	Terminate
SIGTRAP	Break point encountered	Terminate with core dump
SIGTSTP	User generated the suspend character (Ctrl-Z)	Stop
SIGTTIN	Background process read from controlling terminal	Stop
SIGTTOU	Background process wrote to controlling terminal	Stop
SIGURG	Urgent I/O pending	Ignored
SIGUSR1	Process-defined signal	Terminate
SIGUSR2	Process-defined signal	Terminate
SIGVTALRM	Generated by `settimer()` when called with the `ITIMER_VIRTUAL` flag	Terminate
SIGWINCH	Size of controlling terminal window changed	Ignored
SIGXCPU	Processor resource limits were exceeded	Terminate with core dump
SIGXFSZ	File resource limits were exceeded	Terminate with core dump

Signal

- **SIGHUP**

  Hangup. The SIGHUP signal is sent to a process when its controlling terminal is closed. Also sent to each process in a process group when the group leader terminates for any reason.

- **SIGINT**

  Interrupt. Sent to every process associated with a control terminal when the interrupt key **(Control-C)** is hit.

Signal

- **SIGTSTP**

  Interrupt. Sent to every process associated with a control terminal when the interrupt key (Control-Z) is hit.

- **SIGQUIT**

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

Signal

- **SIGILL**

  Illegal instruction. It is sent to a process when it attempts to execute an illegal, unknown, or privileged instruction.

- **SIGIOT**

  I/O trap instruction. Sent when a H/W fault occurs, the exact nature of which is up to the implementer and is machine-dependent.

Signal

- **SIGEMT**

  Emulator trap instruction. Sent when an implementation dependent H/W fault occurs. Extremely rare.

- **SIGFPE**

  Floating-point exception. Sent when the H/W detects a floating-point error, such as a floating point number with an illegal format (such as division by zero).

Signal

- **SIGKILL**

  Kill. The one and only sure way to kill a process, since this signal is always fatal (can't be ignored).

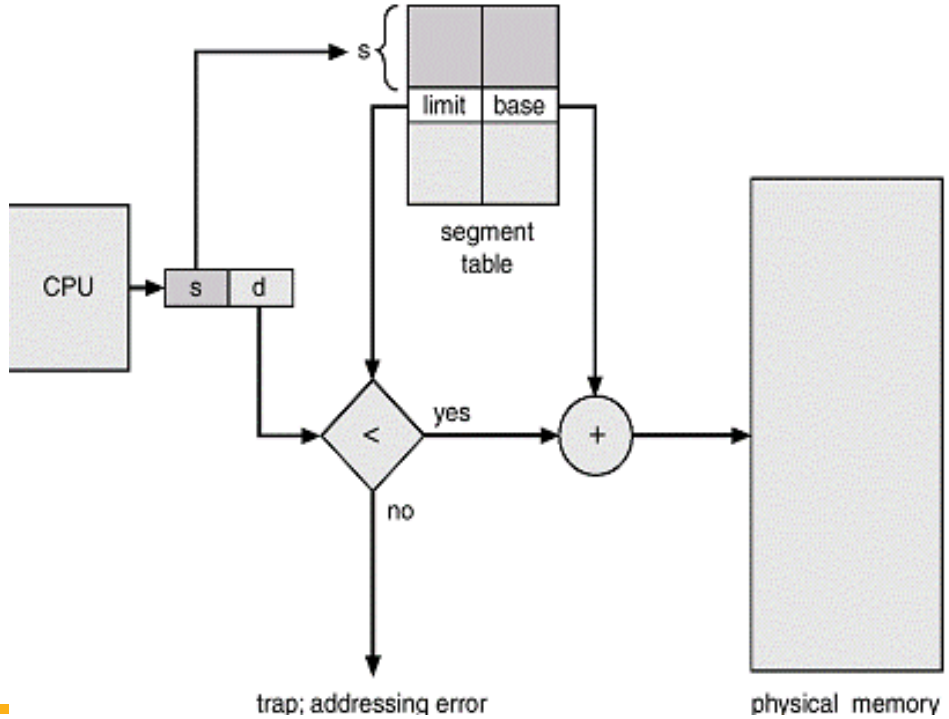
- **SIGBUS**

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

Signal

- **SIGSEGV**

  Segmentation violation. Sent when an implementation-dependent H/W fault occurs. Usually means that the process referenced data outside its address space.



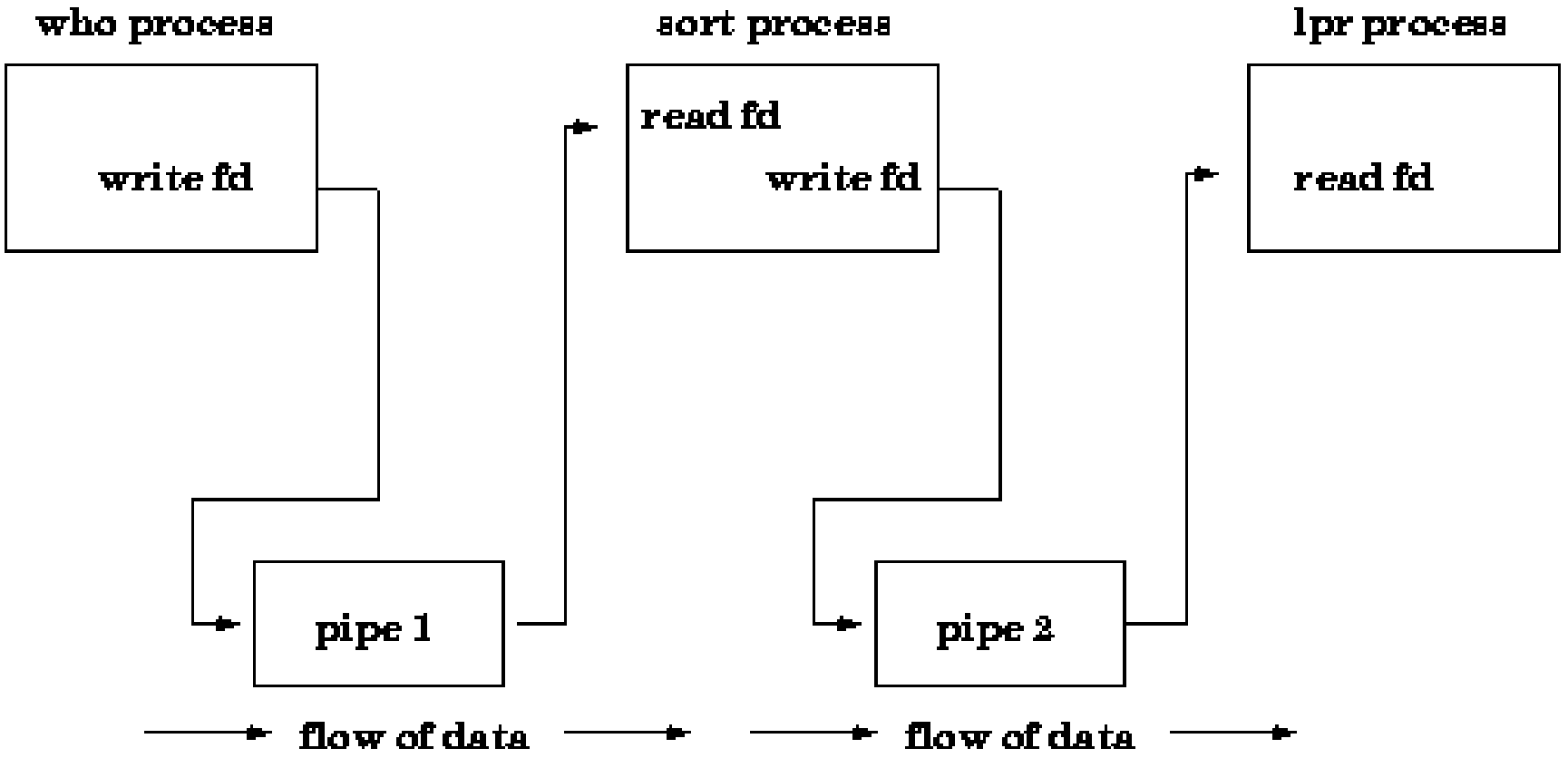
Signal

- **SIGPIPE**

  Write on a pipe not opened for reading. Sent to a process when it writes on a pipe that has no reader. Usually this means that the reader was another process that terminated abnormally. This signal acts to terminate all processes in a pipeline: When a process terminates abnormally, all processes to its right receive an end-of-file and all processes to its left receive this signal.

Signal

- SIGPIPE



Signal

- **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 up quickly (e.g., remove temporary files) and call exit.

Signal

- **SIGPWR**

  Power-fail restart. Exact meaning is implementation-dependent. The SIGPWR signal is sent to a process when the system experiences a **power failure**.

- **SIGXFSZ**

  The SIGXFSZ signal is sent to a process when it grows a **file** larger than the maximum allowed **size**.

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.
- Syntax: int alarm (int count)

Alarm Signal: alarm ()

- Syntax: 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.

```
#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") ;
}
```

The output is:

```
Looping forever...
Alarm clock
```

Signal()

- Programs can respond to signals three different ways. These are:
  - **Ignore the signal:** This means that the program will never be informed of the signal no matter how many times it occurs. ***The only exception to this is the SIGKILL signal which can neither be ignored nor caught.***

Signal()

- **A signal can be set to its default state:** It means that the process will pewrform default action when it receives that signal.
- SIGQUIT, SIGILL, SIGIOT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, or SIGSYS, the UNIX system will produce a core image (core dump), if possible, in the directory where the process was executing when it received the program-ending signal.

Signal()

- **Catch the signal:** When the signal occurs, the UNIX system will transfer control to a previously defined subroutine where it can respond to the signal as is appropriate for the program.

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

Signal()

- *function* is any of
  - 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.

Signal()

- SIG_DFL, meaning that you wish the UNIX system to take the default action when your program receives the signal.
- A pointer to a function. This arranges to catch the signal; every signal but, SIGKILL cannot not be caught. The function is called when the signal arrives.

Signal and value

Signal	Value	Signal	Value
SIGHUP	1	SIGKILL	9
SIGINT	2	SIGSEGV	11
SIGQUIT	3	SIGPIPE	13
SIGILL	4	SIGALRM	14
SIGABRT	6	SIGTERM	15

Signal()

- The function is given control when your program receives the signal, and the ***signal number*** is passed as an argument.
- Function can handle multiple functions
- signal() returns the previous behavior of the signal, which could be
- pointer to a signal handler,
- SIG_DFL, or SIG_IGN.

- On error, the function returnsSIG_ERR. It does not set errno.
- Reasons for failure:
  - Signal name is an illegal name or SIGKILL.
  - function points to an invalid memory address.

Catching a Signal

- To handle certain signals register a signal handling function to the kernel.
- prototype of a signal handling function :
- void <signal handler func name> (int sig)
- Eg: **void** my_handler (**int** signo);
- The signal handler function has void return type and accepts a signal number corresponding to the signal that needs to be handled.

Catching a Signal

- To get the signal handler function registered to the kernel, the signal handler function pointer is passed as second argument to the 'signal' function.
- The prototype of the signal function is :

```
void (*signal(int signo, void (*func )(int)))(int);
```

Catching a Signal

- **void (*signal(int signo, void (*func)(int)))(int);**
  - The function requires two arguments.
  - The first argument is an integer (signo) depicting the signal number or signal value.
  - The second argument is a pointer to the signal handler function that accepts an integer as argument and returns nothing (void).
  - While the 'signal' function itself returns function pointer whose return type is void.

pause()

- int pause()
- ***pause()*** suspends the calling process and returns when the calling process receives a signal which is either handled or terminates the process.
- What is signal is ignored?
- Doesn't return
- most often used to wait efficiently for an alarm signal.
- Returns -1 and sets errno to EINTR

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
/* handler for SIGINT */
static void sigint_handler (int signo)
{
printf ("Caught SIGINT!\n");
exit (EXIT_SUCCESS);
}
```

```c
int main (void)
{
/*
 * Register sigint_handler as our signal handler
 * for SIGINT.
 */
if (signal (SIGINT, sigint_handler) == SIG_ERR)
{
fprintf (stderr, "Cannot handle SIGINT!\n");
exit (EXIT_FAILURE);
}
for (;;)
pause ();
return 0;
}
```

```
stdio.h, signal.h, unistd.h
void signalHandler(int signum)
{
int   signum;
printf("\n Interrupt signal %d received \n", signum);
exit(signum);
}
int main()
{
signal(SIGINT, signalHandler);
while(1)
{
      printf("\n Going to sleep....\n");
      sleep(1);
}
return 0;
}
```

How to catch a Signal

```
wlab@wlab-VirtualBox:~/Documents/os_try$ ./a.out
 Going to sleep....
 Going to sleep....
 Going to sleep....
 Going to sleep....
 Going to sleep....
^C
 Interrupt signal 2 received
```

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

void signalHandler(int signum)
{
//int   signum;
printf("\n Interrupt signal %d received \n", signum);
exit(signum);
}
int main()
{
signal(SIGINT, SIG_IGN);
while(1)
{
        printf("\n Going to sleep....\n");
        sleep(1);
}
return 0;
}
```

```
wlab@wlab-VirtualBox:~/Documents/os_try$ ./a.out
 Going to sleep....

 Going to sleep....

 Going to sleep....
^C
 Going to sleep....
^C
 Going to sleep....

 Going to sleep....

 Going to sleep....
^Z
[3]+  Stopped                 ./a.out
```

```
#include <signal.h> #include <stdio.h>
int alarmFlag = 0 ;
void alarmHandler( ) ; /* signal handler */
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 ;
}
```

```
wlab@wlab-VirtualBox:~/Documents/os_try$
   ./a.out
Looping ...
An ALARM clock signal was received
Loop ends due to alarm signal
wlab@wlab-VirtualBox:~/Documents/os_try$
```

stdio.h, stdlib.h, unistd.h, signal.h, sys/signal.h

```
void alarm_handler(int signum)
{
printf("BITS Pilani\n");
}
int main()
{
signal(SIGALRM, alarm_handler);   //set up alarm handler

alarm(1);                  //schedule alarm for 1 second

pause();                  //do not proceed until signal is handled
}
```

- **Output:** BITS Pilani

- The signal is delivered via the alarm() call.
- The pause() call will "pause" the program until a signal is delivered and handled.

Recurring Alarms

- Alarm can be set continually, but only one alarm is allowed per process.
- Subsequent calls to alarm() will reset the previous alarm.
- Suppose, now, that we want to write a program that will continually alarm every 1 second, we would need to reset the alarm once the signal is delivered. The natural place to do that is in the signal handler:

```
void a_h(int sm)
{
printf("BITS Pilani\n");
alarm(1);        //set a new alarm for 1 second
}
int main()
{
signal(SIGALRM, a_h);      //set up alarm handler
alarm(1);                  //schedule the first alarm
while(1)
{
 pause();                  //pause in a loop
 }
 }
```

```
wlab@wlab-VirtualBox:~/Documents/os_try$
   ./a.out
1 second
1 second
1 second
1 second
1 second
1 second
1 second
^\Quit (core dumped)
wlab@wlab-VirtualBox:~/Documents/os_try$
```

Resetting Alarms

- Let's suppose we want to add a snooze function to our alarm.
- If the user enters ***Ctrl-c*** then we want to reset the alarm to 5 seconds before buzzing again, like snooze.
- We can easily add a signal handler to do that.

```
void sigint_handler(int signum)
{
printf("Snoozing!\n");
alarm(5); //schedule next alarm for 5 seconds
}
void alarm_handler(int signum)
{
printf("Buzz Buzz Buzz\n"); //set a new alarm for 1 second
alarm(1);
}
int main()
{
signal(SIGALRM, alarm_handler); //set up alarm handler
signal(SIGINT, sigint_handler); //set up signint handler
alarm(1); //schedule the first alarm
while(1)
{
pause(); //pause in a loop
} }
```

```
void sigquit_handler(int signum)
{
printf("Alarm Off\n");
alarm(0); //turn off all pending alarms
//reinstate default handler for SIGINT
// Ctrl-C will now terminate program
signal(SIGINT, SIG_DFL);
}

void sigint_handler(int signum)
{
printf("Snoozing!\n");
//schedule next alarm for 5 seconds
alarm(5);
}
```

```
void alarm_handler(int signum)
{
printf("Buzz Buzz Buzz\n");
//set a new alarm for 1 second
alarm(1);
}
int main()
{
signal(SIGALRM, alarm_handler);
signal(SIGINT, sigint_handler);
signal(SIGQUIT, sigquit_handler);
alarm(1);
while(1)
{
pause();
}
}
```