ICS 431 Operating Systems
Lab 11: Inter-Process Communication - Pipes and Signals

****

Objective:

- To learn and practice how processes communicate among themselves
- To use Pipes and signals
- Practice the following system calls
  pipe
  dup / dup2
  alarm
  kill
  signal

Introduction:

Now that we know how to create processes, let us turn our attention to make the processes communicate among themselves. There are many mechanisms through which the processes communicate and in this lab we will discuss two such mechanisms: **Pipes** and **Signals**. A **pipe** *is used for one-way communication of a stream of bytes*. **Signals** *inform processes of the occurrence of asynchronous events*. In this lab we will learn how to create pipes and how processes communicate by reading or writing to the pipe and also how to have a two-way communication between processes. This lab also discusses how the default signals handlers can be replaced by user-defined handlers for particular signals and also how the processes can ignore the signals.

By learning about signals, you can "protect" your programs from *Control-C*, arrange for an alarm clock signal to terminate your program if it takes too long to perform a task, and learn how UNIX uses signals during everyday operations.

Pipes

Pipes are familiar to most Unix users as a shell facility. For instance, to print a sorted list of **who** is logged on, you can enter this command line:

```
who | sort | lpr
```

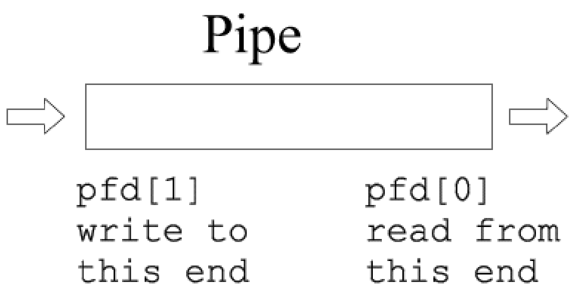
There are **three processes** here, connected with **two pipes**. **Data flows in one direction only**, from `who` to `sort` to `lpr`. It is also possible to set up **bidirectional pipelines** (from process A to B, and from B back to A) and **pipelines in a ring** (from A to B to C to A) using system calls. The shell, however, provides no notation for these more elaborate arrangements, so they are unknown to most Unix users.

We'll begin by showing some simple examples of processes connected by a **one-directional pipeline**.

pipe System Call

```
    int pfd[2];

    int pipe (pfd);   /* Returns 0 on success or -1 on error */
```



I/O with a pipe

These two file descriptors can be used for block I/O

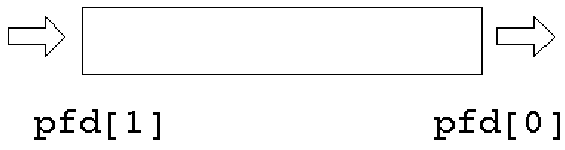
```
        write(pfd[1], buf, SIZE);

        read(pfd[0], buf, SIZE);
```

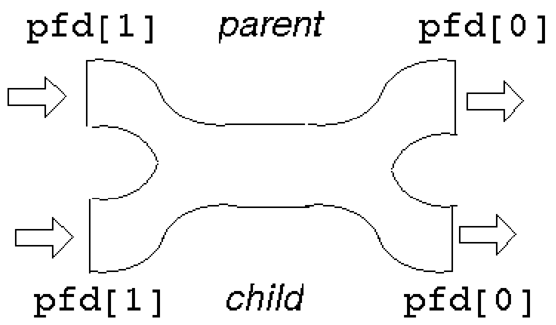
Fork and a pipe

A single process would not use a pipe. They are used when two processes wish to communicate in a one-way fashion. A process splits in two using **fork ( )**. A **pipe opened before the fork becomes shared between the two processes**.

Before fork



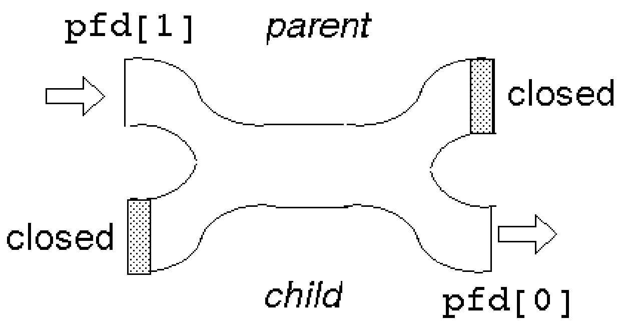
After fork



This gives ***two* read ends** and ***two* write ends**. **The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed.**

Either process can write into the pipe, and either can read from it. Which process will get what is not known.

For predictable behavior, **one of the processes must close its read end, and the other must close its write end**. Then it will become a simple pipeline again.



Suppose the parent wants to write down a pipeline to a child. **The parent closes its read end, and writes into the other end**. **The child closes its write end and reads from the other end.**

When the processes have ceased communication, the parent closes its write end. This means that the child gets eof on its next read, and it can close its read end.

Pipes use the buffer cache just as ordinary files do. Therefore, the benefits of writing and reading pipes in units of a block (usually 512 bytes) are just as great. A single **`write`** execution is atomic, so if 512 bytes are written with a single system call, the corresponding **read** will return with 512 bytes (if it requests that many). It will not return with less than the full block. However, if the writer is not writing complete blocks, but the reader is trying to read complete blocks, the reader may keep getting partial blocks anyhow. This won't happen if the writer is faster than the reader, since then the writer will be able to fill the pipe with a complete block before the reader gets around to reading anything. Still, it's best to buffer writes and reads on pipes, and this is what the Standard I/O Library does automatically.

```
#include <stdio.h>

#define SIZE 1024

main( )
{
  int pfd[2];
  int nread;
  int pid;
  char buf[SIZE];

  if (pipe(pfd) == -1)
  {
    perror("pipe failed");
    exit(1);
  }
  if ((pid = fork()) < 0)
  {
    perror("fork failed");
    exit(2);
  }

  if (pid == 0)
  {
    /* child */
    close(pfd[1]);
    while ((nread = read(pfd[0], buf, SIZE)) != 0)
      printf("child read %s\n", buf);
    close(pfd[0]);
  }
  else
  {
    /* parent */
    close(pfd[0]);
    strcpy(buf, "hello...");
    /* include null terminator in write */
    write(pfd[1], buf, strlen(buf)+1);
    close(pfd[1]);
  }
}
```

Given that **we have two processes, how can we connect them so that one can read from a pipe what the other writes? We can't.** Once the processes are created they can't be connected, because there's **no way for the process that makes the pipe to pass a file descriptor to the other process**. It can pass the file descriptor number, of course, but that number won't be valid in the other process. **But if we make a pipe in one process *before creating* the other process, it will inherit the pipe file descriptors, and they will be valid in both processes**. Thus, **two processes communicating over a pipe can be parent and child, or two children, or grandparent and grandchild, and so on, but they must be related, and the pipe must be passed on at birth**. In practice, this may be a severe limitation, because **if a process dies there's no way to recreate it and reconnect it to its pipes -- the survivors must be killed too, and then the whole family has to be recreated**.

In general, then, here is how to connect two processes with a pipe:

1. **Make the pipe.**
2. **Fork to create the reading child.**
3. **In the child close the writing end of the pipe, and do any other preparations that are needed.**
4. **In the child execute the child program.**
5. **In the parent close the reading end of the pipe.**
6. **If a second child is to write on the pipe, create it, make the necessary preparations, and execute its program. If the parent is to write, go ahead and write.**

Here’s a small program that uses a **pipe** to allow the parent to read a message from its child:

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

#define  READ    0
#define  WRITE   1

char*  phrase  =  "This is ICS431 lab time" ;

main ( )
{
 int    fd [2],  bytesread ;
 char    message [100] ;

 pipe ( fd ) ;
 if ( fork (  )  ==  0 )                              /*  child, writer */
 {
    close ( fd [READ] ) ;                              /*  close unused end */
    write  ( fd [WRITE], phrase, strlen (phrase) + 1) ;
    close ( fd [WRITE] ) ;                              /*  close used end  */
 }
 else                                                  /*  parent,  reader  */
 {
    close ( fd [WRITE] ) ;                              /* close unused end  */
    bytesread = read (fd [READ], message, 100) ;
    printf ("Read %d bytes : %s\n", bytesread, message) ;
    close ( fd [READ] ) ;                              /*  close used end  */
 }
}
```

****

Implementation of Redirection

When a process forks, the child inherits a copy of its parent's file descriptors. When a process execs, the standard input, output, and error channels remain unaffected. The UNIX shells uses these two pieces of information to implement redirection.

To perform redirection, the shell performs the following series of actions:

- **The parent shell forks and then waits for the child shell to terminate.**
- **The child shell opens the file "output", creating it or truncating as necessary.**
- **The child shell then duplicates the file descriptor of "output" to the standard output file descriptor, number 1, and then closes the original descriptor of "output". All standard output is therefore redirected to "output".**
- **The child shell then exec's the ls utility. Since the file descriptors are inherited during an exec ( ), all of standard output of ls goes to "output".**
- **When the child shell terminates, the parent resumes. The parent's file descriptors are unaffected by the child's actions, as each process maintains its own private descriptor table.**

```
#include <stdio.h>
#include <sys/file.h>
main (argc, argv)
int  argc ;
char  *argv[ ] ;
{
   int   fd ;              /* file descriptor or pointer */

   fd = open (argv[1], O_CREAT | O_TRUNC | O_RDWR, 0777) ;
                                /* open file named in argv[1] */
   dup2 (fd, 1) ;              /*  and assign it to fd file pointer */
   close (fd) ;               /* duplicate fd with 1 which is standard output (the monitor) */
   execvp (argv[2], &argv[2]) ;
                  /* the output is not printed on screen but is redirected to "output" file */
   printf ("End\n") ;                        /* should never execute */
}
```

****

dup / dup2 System Call

```
int dup   (int  oldfd )
int dup2 (int  oldfd, int newfd )
```

dup ( ) finds the smallest free file descriptor entry and points it to the same file as *oldfd*. **dup2 ( )** closes ***newfd*** if it's currently active and then points it to the same file as ***oldfd***. In both cases, the **original and copied file descriptors share the same file pointer and access mode**.

They both return the **index of the new file descriptor** if **successful**, and **–1** otherwise.

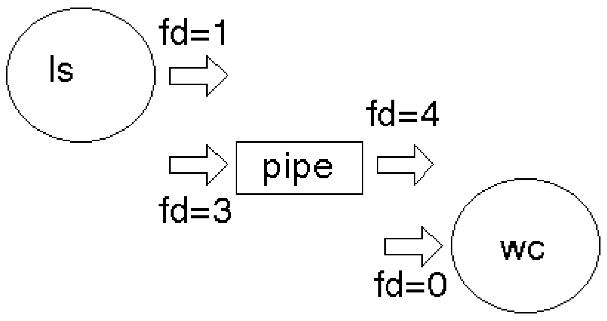
dup/dup2 duplicates an existing file descriptor, giving a new file descriptor that is open to the same file or pipe. The two share the same file pointer, just as an inherited file descriptor shares the file pointer with the corresponding file descriptor in the parent. **The call fails if the argument is bad (not open) or if 20 file descriptors are already open**.

A pipeline works because the two processes know the file descriptor of each end of the pipe. **Each process has a stdin (0), a stdout (1) and a stderr (2)**. The file descriptors will depend on which other files have been opened, but could be 3 and 4, say.

Suppose one of the processes replaces itself by an "**exec**". The new process will have files for descriptors 0, 1, 2, 3 and 4 open. How will it know which are the ones belonging to the pipe? It can't.

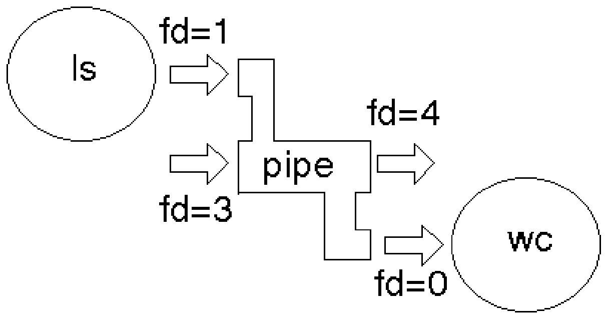
Example:

To implement "**ls | wc**" the **shell will have created a pipe and then forked**. The parent will **exec** to be replaced by "**ls**", and the child will **exec** to be replaced by "**wc**" The write end of the pipe may be descriptor 3 and the read end may be descriptor 4. "**ls**" normally writes to 1 and "**wc**" normally reads from 0. How do these get matched up?

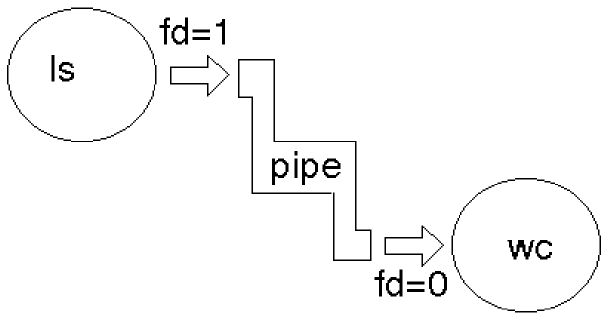


The **dup/dup2** function call takes an existing file descriptor, and another one that it "would like to be". Here, fd=3 would also like to be 1, and fd=4 would like to be 0. So we ***dup*** fd=3 as 1, and ***dup*** fd=4 as 0. Then the old fd=3 and fd=4 can be closed as they are no longer needed.

After dup



After close



The UNIX shells use unnamed pipes to build pipelines. They use a trick similar to the redirection mechanism to connect the standard output of one process to standard input of another. To illustrate this approach, here’s the program that executes two named programs, connecting the standard output of the first to the standard input of the second.

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

#define  READ    0
#define  WRITE   1

main (argc, argv)
int  argc ;
char*  argv[] ;
{
  int     pid, fd [2] ;

  if (pipe(fd) == -1)
  {
    perror("pipe failed");
    exit(1);
  }
  if ((pid = fork( )) < 0)
  {
    perror("fork failed");
    exit(2);
  }
  if ( pid  !=  0 )                          /*  parent, writer */
  {
      close ( fd [READ] ) ;               /*  close unused end */
      dup2 ( fd [WRITE], 1) ;          /* duplicate used end to standard out */
      close ( fd [WRITE] ) ;              /*  close used end  */
      execlp ( argv[1], argv[1], NULL) ;      /*  execute writer program  */
  }
  else                                        /*  child,  reader  */
  {
      close ( fd [WRITE] ) ;                /* close unused end */
      dup2 ( fd [READ], 0) ;              /* duplicate used end to standard input */
      close ( fd [READ] ) ;                /* close used end */
      execlp ( argv[2], argv[2], NULL) ;       /* execute reader program */
  }
}
```

Run the above program as

```
a.out  who  wc
```

Variations

Some common variations on this method of IPC are:

1. A pipeline may consist of three or more process (such as a C version of `ps | sed 1d | wc -l`). In this case there are lots of choices
   1. The parent can fork twice to give two children.
   2. The parent can fork once and the child can fork once, giving a parent, child and grandchild.

3. The parent can create two pipes before any forking. After a fork there will then be a total of 8 ends open (2 processes * two ends * 2 pipes). Most of these will have to be closed to ensure that there ends up only one read and only one write end.
4. As many ends as possible of a pipe may be closed before a fork. This minimises the number of closes that have to be done after forking.

2. A process may want to both write to and read from a child. In this case it creates two pipes. One of these is used by the parent for writing and by the child for reading. The other is used by the child for writing and the parent for reading.

****

Signals

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 kind 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 delivery of signals to a process is handled by the kernel. 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 31 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**

Some signals are widely used, while others are extremely obscure and used by only one or two programs. The following list gives a brief explanation of each signal. **The default action upon receipt of a signal is for the process to terminate**.

SIGHUP

Hangup. Sent when a terminal is hung up to every process for which it is the control terminal. Also sent to each process in a process group when the group leader terminates for any reason. This simulates hanging up on terminals that can't be physically hung up, such as a personal computer.

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 implementor 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.

SIGSEGV

Segmentation violation. Sent when an implementation-dependent hardware fault occurs. Usually means that the process referenced data outside its address space. Trying to use NULL pointers will usually give you a SIGSEGV.

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. Note that the standard shell (`sh`) makes each process in a pipeline the parent of the process to its left. Hence, the writer is not the reader's parent (it's the other way around), and would otherwise not be notified of the reader's death.

SIGALARM

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`.

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. Similar to SIGUSR1.

SIGPWR

Power-fail restart. Exact meaning is implementation-dependent. One possibility is for it to be sent when power is about to fail (voltage has passed, say, 200 volts and is falling). The process has a very brief time to execute. It should normally clean up and exit (as with SIGTERM). If the process wishes to survive the failure (which might only be a momentary voltage drop), it can clean up and then sleep for a few seconds. If it wakes up it can assume that the disaster was only a dream and resume processing. If it doesn't wake up, no further action is necessary.

Programs that need to clean up before terminating should arrange to catch signals **SIGHUP**, **SIGINT**, and **SIGTERM** . Until the program is solid, **SIGQUIT** should be left alone so there will be a way to terminate the program (with a core dump) from the keyboard. Arrangements for the other signals are made much less often; usually they are left to terminate the process. But a really polished program will want to catch everything it can, to clean up, possibly log the error, and print a nice error message. Psychologically, a message like ``Internal error 53: contact customer support'' is more acceptable than the message ``Bus error -- core dumped'' from the shell. For some signals, the default action of termination is accompanied by a core dump. These are SIGQUIT, SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, and SIGSYS.

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") ;
}
```

The output is:

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

---

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 parent's action for a signal is inherited by a child process. 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 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 interprocess 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. `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.

It's extremely difficult to take interrupted system calls into account when programming. You either have to program to restart every system call that can wait or else temporarily ignore signals when executing such a system call. Both approaches are awkward, and the second runs the additional risk of losing a signal during the interval when it's ignored. We therefore offer this rule: Never return from a signal-catching function. Either terminate processing entirely or terminate the current operation by executing a global jump (not described here).

If you make it a habit to always print out the value of **`errno`** when a system call fails (by calling **`perror`** for example) you won't be mystified for long since the EINTR error code will clarify what's going on.

Since the first thing that happens when a caught signal arrives is to change its action to the default (termination), another signal of the same type arriving immediately after the first can terminate the process before it has a chance to even begin the catching function. This is rare but possible, especially on a busy system.

This loophole can be tightened, but not eliminated, by setting the signal to be ignored immediately upon entering the catching function, before doing anything else. Since we're not using signals as messages, we don't care if an arriving signal is thereby missed. We're concerned only with processing the first one correctly and with not terminating prematurely.

---

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 ;
}
```

The output will be as:

```
Looping ...
An ALARM clock signal was received
Loop ends due to alarm signal
```

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 executed. Here's the source code of the program 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 */
  printf ("I am protected from Control-C now \n") ;
  sleep (5) ;
  signal (SIGINT, oldHandler) ; /* Restore old handler */
  printf ("I can be Control-C'ed again \n") ;
  sleep (5) ;
  printf ("Bye!!!!!!!\n") ;
}
```

Now run the program by pressing ***Control-C*** twice while the program sleeps.

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 superuser.**

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 interprocess 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.

Assignments:

Execute the C programs given in the following problems. ***Observe*** and **Interpret** the results. You will learn about ***child*** and ***parent*** processes, and much more about UNIX processes in general by performing the suggested experiments.

UNIX Calls used in the following problems: ***signal( ), alarm( ), pipe( ), sigkey( ),*** and ***exit( )***.

1) Create your own **pipe-redirect** command so that the output of given command goes to a given file.
Eg. ./a.out ls file1
The output of ls command is stored in file1.

2) Execute the following program and its suggested modifications. Observe and interpret the results.

```
#include <signal.h>
void my_routine ( ) ;
main ( )
{
   printf ("Process ID is: %d\n", getpid( ) ) ;
   signal (SIGINT, my_routine) ;
   for ( ; ; ) ;
}

void my_routine ( )
{
   printf ("Have a good day !!!!!!\n") ;
}
```

Modifications: **1)** Press the **CTRL-C** key. What happened? **2)** Omit the **signal ( )** statement in the *main* program, run the program, observe the result. **3)** In **main ( )**, replace the **signal ( )** statement by: **signal (SIGINT, SIG_IGN) ;** Observe the result. Relate the three parts and explain their results. If you get stuck, you can kill the processes by pressing **CTRL \**. **4)** The signal sent whrn **CTRL \** is pressed is **SIGQUIT** and the name of the signal service routine is **sigkey ( )**; Go back to the original code. Change the name **my_routine** to sigkey and observe what happens when you press **CTRL \**.

3) Observe and explain the behavior of the following program.

```
#include <signal.h>
void my_routine ( ) ;
int pid ;

main ( )
{
   pid = fork ( );
   signal (SIGINT, my_routine) ;
   for ( ; ; ) ;
}

void my_routine ( )
{
   printf ("My pid = %d\n", pid) ;
}
```

4) Find out what this program does and explain it.

```
#include <signal.h>
#include <stdio.h>
char msg[100] ;
main (argc, argv)
int argc ;
char *argv[ ] ;
{
   int time ;
   void my_secretary ( ) ;
   time = atoi (argv[2]) ;
   strcpy (msg, argv[1]) ;
   signal (SIGALRM, my_secretary) ;
   alarm (time) ;
   for ( ; ; ) ;
}

void my_secretary ( )
{
   printf ("%s\n", msg) ;
   exit (0) ;
}
```

5) Suspending and Resuming Processes. The **SIGSTOP** and **SIGCONT** signals **suspend** and **resume** a process, respectively. They are used by the UNIX shells to implement built-in commands like **stop**, **fg**, and **bg**. Observe and explain the behavior of the following program.

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

main ( )
{
  int  pid1, pid2 ;

  pid1 = fork ( ) ;
  if (pid1 == 0)     /*first child */
  {
    while (1)   /* infinite loop */
    {
      printf ("pid1 is alive\n") ;
      sleep (1) ;
    }
  }
  pid2 = fork ( ) ;
  if (pid2 == 0)     /*second child */
  {
    while (1)   /* infinite loop */
    {
      printf ("pid2 is alive\n") ;
      sleep (1) ;
    }
  }
  sleep (3) ;
  kill (pid1, SIGSTOP) ;
  sleep (3) ;
  kill (pid1, SIGCONT) ;
  sleep (3) ;
  kill (pid1, SIGINT) ;
  kill (pid2, SIGINT) ;
}
```

6) Discover the behavior by writing a **main program** and a **signal handling routine** to print a message in case of an ***illegal instruction*** (e.g. division by zero). Including a division by zero in main program. The signal sent for an illegal instruction is **SIGILL**.

CLICK HERE FOR LAB NOTES