UNIT-V SIGNALS

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

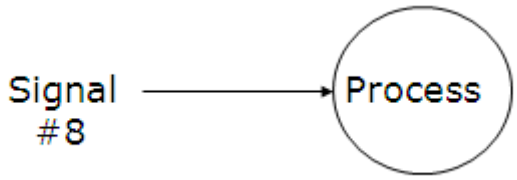
- a floating point error
- a power failure
- an alarm clock "ring"
- the death of a child process
- a termination request from a user (i.e., Control-C)
- a suspend request from a user (i.e., Control-Z)

- 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

There is a unique, numbered signal for each possible event.

For example,

if a process causes a floating point error,the kernel sends the offending process signal number 8:



any process can send any other process a signal, as long as it has permission to do so.

- 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*.
- the process that receives the signal

1) suspends its current flow of control,
2) executes the signal handler,
3) and then resumes the original flow of control when the signal handler finishes

Signals are defined in "/usr/include/sys/signal.h".

The default handler usually performs one of the following actions:

- terminate the process and generate a core file (*dump*)
- terminate the process without generating a core image file (*quit*)
- ignore and discard the signal (*ignore*)
- suspend the process (*suspend*)
- resume the process

A List of Signals

- Here's a list of the System V predefined signals, along with their respective macro definitions, numerical values, and default actions, as well as a brief description of each:

Macro	#	Default	Description
SIGHUP	1	quit	hang up
SIGINT	2	quit	interrupt
SIGQUIT	3	dump	quit
SIGILL	4	dump	illegal instruction
SIGTRAP	5	dump	trace trap(used by debuggers)
SIGABRT	6	dump	abort
SIGEMT	7	dump	emulator trap instruction
SIGFPE	8	dump	arithmetic execution
SIGKILL	9	quit	kill(cannot be caught, blocked, or ignored)
SIGBUS	10	dump	bus error(bad format address)
SIGSEGV	11	dump	segmentation violation(out-of-range address)
SIGSYS	12	dump	bad argument to system call
SIGPIPE	13	quit	write on a pipe or other socket with no one to read it.
SIGALRM	14	quit	alarm clock
SIGTERM	15	quit	software termination signal(default signal sent by kill)
SIGUSR1	16	quit	user signal 1
SIGUSR2	17	quit	user signal 2
SIGCHLD	18	ignore	child status changed
SIGPWR	19	ignore	power fail or restart
SIGWINCH	20	ignore	window size change
SIGURG	21	ignore	urgent socket condition
SIGPOLL	22	ignore	pollable event
SIGSTOP	23	quit	stopped(signal)
SIGSTP	24	quit	stopped(user)
SIGCONT	25	ignore	continued
SIGTTIN	26	quit	stopped(tty input)
SIGTTOU	27	quit	stopped(tty output)
SIGVTALRM	28	quit	virtual timer expired
SIGPROF	29	quit	profiling timer expired
SIGXCPU	30	dump	CPU time limit exceeded
SIGXFSZ	31	dump	file size limit exceeded

Terminal Signals

- The easiest way to send a signal to a foreground process is by pressing Control-C or Control-Z from the keyboard.
- When the terminal driver(the piece of software that supports the terminal) recognizes that Control-C was pressed. it sends a SIGINT signal to all of the processes in the current foreground job.
- Similarly, Control-Z causes it to send a SIGTSTP signal to all of the processes in the current foreground job.
- By default,SIGINT terminates a process and SIGTSTP suspends a process.

Requesting an Alarm Signal : alarm()

- SIGALRM, by using "alarm()".

The default handler for this signal displays the message "Alarm clock" and terminates the process.

Here's how "alarm()" works:

System Call : unsigned int **alarm**(unsigned 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, that alarm 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.

a small program that uses "alarm()", together with its output:

```
$ cat  alarm.c        --->   list the program.
#include <stdio.h>
main()
{
   alarm(3);  /* Schedule an alarm signal in three seconds */
   printf("Looping forever… \n");
   while(1)
      printf("This line should never be executed \n");
}
$ alarm                 ---> run the program.
Looping forever…
Alarm clock             ---> occurs three seconds later.
$ -
```

Handling Signals : signal()

System Call: void(*signal(int *sigCode*, void (**func*)(int))) (int)

"signal()" allows a process to specify the action that it will take when a particular signal is received.

The parameter *sigCode* specifies the number of the signal that is to be reprogrammed, and *func* may be one of several values:

- SIG_IGN, which indicates that the specified signal should be ignored and discarded.
- SIG_DFL, which indicates that the kernel's default handler should be used.
- an address of a user-defined function, which indicates that the function should be executed when the specified signal arrives.

The valid signal numbers are stored in "/usr/include/signal.h".

The signals SIGKILL and SIGSTP may not be reprogrammed.

A child process inherits the signal settings from its parent during a "fork()".

"signal()" returns the previous *func* value associated with *sigCode* if successful; otherwise, it returns a value of -1.

- The "signal()" system call may be used to override the default action.
- a couple of changes to the previous program so that it caught and processed the SIGALRM signal efficiently:
  - installed my own signal handler, "alarmHandler()", by using "signal()".

System Call: int pause(void)

"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.

$ cat handler.c ---> list the program.

```
#include <stdio.h>
#include <signal.h>
int alarmFlag = 0;      /* Global alarm flag */
void alarmHandler();   /* Forward declaration of alarm handler */
/*****************************************************/
main()
{
  signal( SIGALRM, alarmHandler );  /* Install signal handler */
  alarm(3);  /* Schedule an alarm signal in three seconds */
  printf("Looping…\n");
  while( !alarmFlag )  /* Loop until flag set */
   {
      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;
}
```

```
$ handler          ---> run the program.
Looping…
An alarm clock signal was received  ---> occurs three seconds later.
Loop ends due to alarm signal
```

Protecting Critical Code and Chaining Interrupt Handlers

- The same techniques that I just described may be used to protect critical pieces of code against Control-C attack and other such signals.
- it can be restored after the critical code has executed.

Here's the source code of a program that protects itself against SIGINT signals:

```
$ cat  critical.c          --->  list the program.
#include <stdio.h>
#include <signal.h>
main()
{
  void (*oldHandler) ()                    /* To hold old handler value */
  printf("I can be Control-C'ed \n");
  sleep(3);
  oldHandler = signal(SIGINT, SIG_IGN);       /* Ignore Control-C */
  printf("I'm protected from Control-C now\n");
    sleep(3);
    signal(SIGINT, oldHandler);   /* Restore old handler */
    printf("I can be Control-C'ed again \n");
    sleep(3);
    printf("Bye! \n");
}
$ critical                 ---> run the program.
I can be Control-C'ed
^C                         ---> Control-C works here.
$ critical                 ---> run the program again.
I can be Control-C'ed
I'm protected from Control-C now
^C
I can be Control-C'ed again
Bye!
```

- **Sending Signals: kill()**

- A process may send a signal to another process by using the "kill()" system call.
- "kill()" is a misnomer, since many of the signals that it can send to do not terminate a process.
- It's called "kill()" because of historical reasons;

the main use of signals when UNIX was first designed was to terminate processes.

System Call: int kill(pid_t *pid*, int *sigCode*)

"kill()" sends the signal with value *sigCode* to the process with PID *pid*.

"kill()" succeeds and the signal is sent as long as at least one of the following conditions is satisfied:

- The sending process and the receiving process have the same owner.
- The sending process is owned by a super-user.

There are a few variations on the way that "kill()" works:

- If *pid* is zero, the signal is sent to all of the processes in the sender's process group.
- If *pid* is -1 and the sender is owned by a super-user, the signal is sent to all processes, including the sender.

If *pid* is -1 and the sender is not owned by a super-user, the signal is sent to all of the processes owned by the same owner as that of the sender, excluding the sending process.

- If the *pid* is negative, but not -1, the signal is sent to all of the processes in the process group.

Death of Children

- When a parent's child terminates, the child process sends its parent a SIGCHLD signal.
- A parent process often installs a handler to deal with this signal; this handler typically executes a "wait()" system call to accept the child's termination code and let the child dezombify.
- the parent can choose to ignore SIGCHLD signals, in which case the child dezombifies automatically.

- The program works by performing the following steps:

1. The parent process installs a SIGCHLD handler that is executed when its child process terminates.
2. The parent process forks a child process to execute the command.
3. The parent process sleeps for the specified number of seconds. when it wakes up, it sends its child process a SIGINT signal to kill it.
4. If the child terminates before its parent finishes sleeping, the parent's SIGCHLD handler is executed, causing the parent to terminate immediately.

- Here's the source code for and sample output from the program.

```
$ cat limit.c                    ---> list the program.
#include <stdio.h>
#include <signal.h>
int delay;
void childHandler();
/*************************************************/
main( argc, argv )
int argc;
char* argv[];
{
```

```
    int pid;
    signal( SIGCHLD, childHandler );  /* Install death-of-child handler */
    pid = fork();  /* Duplicate */
    if ( pid == 0 )  /* Child */
      {
         execvp( argv[2], &argv[2] );   /* Execute command */
      perror("limit");   /* Should never execute */
    }
else   /* Parent */
 {
  sscanf( argv[1], "%d", &delay );  /* Read delay from command-line */
  sleep(delay);   /* Sleep for the specified number of seconds */
  printf("Child %d exceeded limit and is being killed \n", pid );
  kill( pid, SIGINT );   /* Kill the child */
 }
}
/*****************************************************/
void childHandler()  /* Executed if the child dies before the parent */
{
   int  childPid, childStatus;
   childPid = wait(&childStatus);   /* Accept child's termination code */
   printf("Child %d terminated within %d seconds \n", childPid, delay);
   exit(/* EXIT SUCCESS */  0);
}
$ limit  5  ls   ---> run the program; command finishes OK.
 a.out         alarm          critical         handler         limit
 alarm.c       critical.c       handler.c      limit.c
 Child 4030  terminated within 5 seconds.
 $ limit  4  sleep  100  ---> run it again;  command takes too long
 Child 4032 exceeded limit and is being killed.
 $ -
```

pause - wait for signal

#include <unistd.h>

int pause(void);

DESCRIPTION

The **pause**() library function causes the invoking process (or thread) to sleep until a signal is received that either terminates it or causes it to call a signal-catching function.

RETURN VALUE

The **pause**() function only returns when a signal was caught and the signal-catching function returned. In this case **pause**() returns -1, and *errno* is set to **EINTR**.

NAME

kill - send signal to a process
SYNOPSIS
#include <sys/types.h>
#include <signal.h>
int kill(pid_t *pid***, int** *sig***);**
DESCRIPTION
The **kill**() system call can be used to send any signal to any process group or process.
If *pid* is positive, then signal *sig* is sent to *pid*.
If *pid* equals 0, then *sig* is sent to every process in the process group of the current process.
If *pid* equals -1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (init), but see below.
If *pid* is less than -1, then *sig* is sent to every process in the process group *-pid*.
If *sig* is 0, then no signal is sent, but error checking is still performed.
For a process to have permission to send a signal it must either be privileged or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of SIGCONT it suffices when the sending and receiving processes belong to the same session.
RETURN VALUE
On success (at least one signal was sent), zero is returned. On error, -1 is returned, and *errno* is set appropriately.
Abort():
The C library function **void abort(void)** abort the program execution and comes out directly from the place of the call.
Declaration
Following is the declaration for abort() function.
void abort(void)
Return Value
This function does not return any value.
Example
The following example shows the usage of abort() function.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  FILE *fp;
  printf("Going to open nofile.txt\n");
  fp = fopen( "nofile.txt","r" );
  if(fp == NULL)
  {
    printf("Going to abort the program\n");
    abort();
```

```
   }
   printf("Going to close nofile.txt\n");
   fclose(fp);
   return(0);
}
```

Let us compile and run the above program that will produce the following result when it tries to open **nofile.txt** file, which does not exist:

```
Going to open nofile.txt
Going to abort the program
Aborted (core dumped)
```

Raise():

The C library function int raise(int sig) causes signal sig to be generated. The sig argument is compatible with the SIG macros.

Declaration

Following is the declaration for signal() function.

```
int raise(int sig)
```

Parameters

sig − This is the signal number to send. Following are few important standard signal constants −macro signal

SIGABRT (Signal Abort) Abnormal termination, such as is initiated by the abort function.

SIGFPE (Signal Floating-Point Exception) Erroneous arithmetic operation, such as zero divide or an operation resulting in overflow (not necessarily with a floating-point operation).

SIGILL (Signal Illegal Instruction) Invalid function image, such as an illegal instruction. This is generally due to a corruption in the code or to an attempt to execute data.

SIGINT (Signal Interrupt) Interactive attention signal. Generally generated by the application user.

SIGSEGV (Signal Segmentation Violation) Invalid access to storage − When a program tries to read or write outside the memory it is allocated for it.

SIGTERM (Signal Terminate) Termination request sent to program.

Return Value

This function returns zero if successful, and non-zero otherwise.

Example

The following example shows the usage of signal() function.

```
#include <signal.h>
#include <stdio.h>
void signal_catchfunc(int);
int main()
{
   int ret;
```

```
   ret = signal(SIGINT, signal_catchfunc);

   if( ret == SIG_ERR)
   {
      printf("Error: unable to set signal handler.\n");
      exit(0);
   }
   printf("Going to raise a signal\n");
   ret = raise(SIGINT);
   if( ret !=0 )
   {
      printf("Error: unable to raise SIGINT signal.\n");
      exit(0);
   }
   printf("Exiting...\n");
   return(0);
}
void signal_catchfunc(int signal)
{
   printf("!! signal caught !!\n");
}
```

Let us compile and run the above program to will produce the following result −

Output:

```
Going to raise a signal
!! signal caught !!
Exiting...
```

Interprocess Communication:

Interprocess Communication(IPC) is the generic term describing how two processes may exchange information with each other.

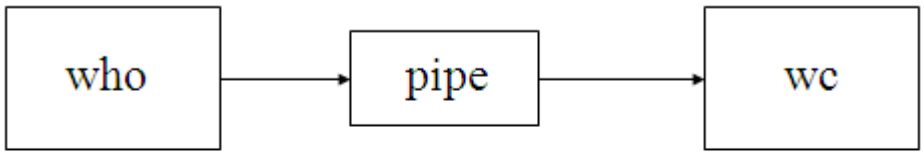
- In general, the two processes may be running on the same machine or on different machines, although some IPC mechanisms may only support local usage
  (e.g., signals and pipes)
- This communication may be an exchange of data for which two or more processes are cooperatively processing the data or synchronization information to help two independent, but related, processes schedule work so that they do not destructively overlap.

Pipes are an interprocess communication mechanism that allows two or more processes to send information to each other.

- commonly used from within shells to connect the standard output of one utility to the standard input of another.
- For example, here's a simple shell command that determines how many users there are on the system:

$ who | wc -l

- The who utility generates one line of output per user. This output is then "piped" into the wc utility, which, when invoked with the "-l" option, outputs the total number of lines in its input.



Bytes from "who" flow through the pipe to "wc"

- It's important to realize that both the writer process and the reader process of a pipeline execute concurrently;

- a pipe automatically buffers the output of the writer and suspends the writer if the pipe gets too full.
- Similarly, if a pipe empties, the reader is suspended until some more output becomes available.
- All versions of UNIX support unnamed pipes, which are the kind of pipes that shells use.
- System V also supports a more powerful kind of pipe called a *named pipe*.

Unnamed Pipes: "pipe()"

- An unnamed pipe is a unidirectional communications link that automatically buffers its input (the maximum size of the input varies with different versions of UNIX, but is approximately 5K)and may be created using the "pipe()" system call.
- Each end of a pipe has an associated file descriptor.

The "write" end of the pipe may be written to using "write()", and the "read" end may be read from using "read()".

- When a process has finished with a pipe's file descriptor.

it should close it using "close()".

System Call : int **pipe**(int fd[2])

"pipe()" creates an unnamed pipe and returns two file descriptors:

The descriptor associated with the "read" end of the pipe is stored in fd[0], and the descriptor associated with the "write" end of the pipe is stored in fd[1].

- If a process reads from a pipe whose "write" end has been closed, the "read()" call returns a value of zero, indicating the end of input.

- If a process reads from an empty pipe whose "write" end is still open, it sleeps until some input becomes available.

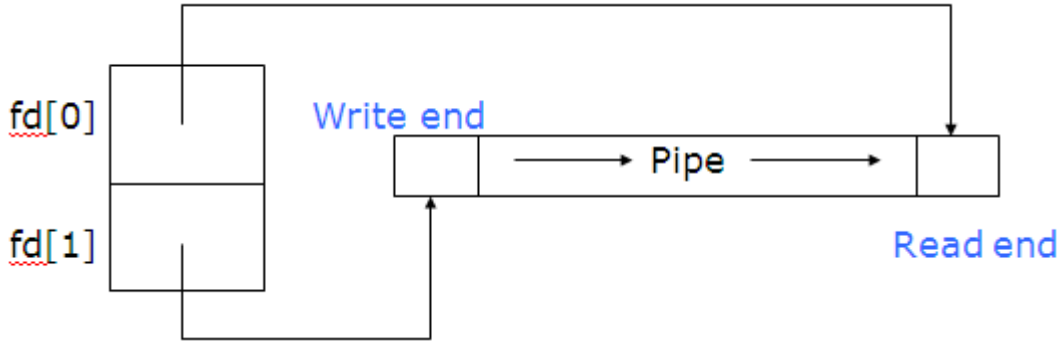
If a process tries to read more bytes from a pipe than are present, all of the current contents are returned and "read()" returns the number of bytes actually read.

- If a process writes to a pipe whose "read" end has been closed, the write fails and the writer is sent a SIGPIPE signal. the default action of this signal is to terminate the receiver.
- If a process writes fewer bytes to a pipe than the pipe can hold, the "write()" is guaranteed to be atomic; that is, the writer process will complete its system call without being preempted by another process.

If the kernel cannot allocate enough space for a new pipe, "pipe()" returns a value of -1; otherwise, it returns a value of 0.

- Assume that the following code was executed:

```
int fd[2];
pipe(fd);
```



\

Unnamed pipes are usually used for communication between a parent process and its child, with one process writing and the other process reading.

The typical sequence of events for such a communication is as follows:

1. The parent process creates an unnamed pipe using "pipe()".
2. The parent process forks.
3. The writer closes its "read" end of the pipe, and the designated reader closes its "write" end of the pipe.
4. The processes communicate by using "write()" and "read()" calls.
5. Each process closes its active pipe descriptor when it's finished with it.

- Bidirectional communiation is only possible by using two pipes.

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

```
$ cat talk.c            ---> list the program.
#include <stdio.h>
#define  READ   0     /* The index of the "read" end of the pipe */
#define  WRITE  1     /* The index of the "write" end of the pipe */
char*  phrase ="Stuff this in your pipe and smoke it";
```

```
main()
{
  int fd[2], bytesRead;
  char message[100];   /* Parent process' message buffer */
  pipe(fd);  /* Create  an unnamed pipe */
 if ( fork() == 0 )  /* Child, write */
  {
     close(fd[READ]); /* Close unused end */
     write(fd[WRITE], phrase, strlen(phrase)+1); /* Send */
     close(fd[WRITE]); /* Close used end */
  }
 else      /* Parent, reader */
  {
    close(fd[WRITE]);  /* Close unused end */
    bytesRead = read( fd[READ], message, 100 ); /* Receive */
    printf("Read %d bytes: %s \n", bytesRead, message );
    close(fd[READ]);  /* Close used end */
  }
}
$ talk        ---> run the program.
Read 37 bytes: Stuff this in your pipe and smoke it
$ _
```

The child included the phrase's NULL terminator as part of the message so that the parent could easily display it.

- When a writer process sends more than one variable-length message into a pipe, it must use a protocol to indicate to the reader the location for the end of the message.

Methods for such indication include :

- sending the length of a message(in bytes) before sending the message itself
- ending a message with a special character such as a new line or a NULL

- UNIX shells use unnamed pipes to build pipelines.

connecting the standard output of the first to the standard input of the second.

```
$ cat  connect.c   ---> list the program.
#include <stdio.h>
#define  READ    0
#define  WRITE   1
main( argc, argv )
int argc;
char* argv[];
{
   int fd[2];
   pipe(fd);   /* Create an unnamed pipe */
   if ( fork()!=0 )  /* Parent, writer */
```

```
    {
      close( fd[READ] );  /* Close unused end */
      dup2( fd[WRITE], 1);  /* Duplicate used end to stdout */
      close( fd[WRITE] );     /* Close original used end */
      execlp( argv[1], argvp[1], NULL );  /* Execute writer program */
      perror( "connect" );    /* Should never execute */
    }
  else    /* Child, reader */
    {
      close( fd[WRITE] );    /* Close unused end */
      dup2( fd[READ], 0 );  /* Duplicate used end to stdin */
      close( fd[READ] );      /* Close original used end */
      execlp( argv[2], argv[2], NULL );  /* Execute reader program */
      perror( "connect" );    /* Should never execute */
    }
}
$ who          ---> execute "who" by itself.
gglass          ttyp0      Feb  15  18:15 (xyplex_3)
$ connect  who  wc      ---> pipe "who" through "wc".
         1        6        57        …1 line,  6 words, 57 chars.
$ _
```

- **Named Pipes**

- Named pipes, often referred to as FIFOs(first in, first out), are less restricted than unnamed pipes and offer the following advantages:
  - They have a name that exists in the file system.
  - They may be used by unrelated processes.
  - They exist until explicitly deleted.
- Unfortunately, they are only supported by System V.
  named pipes have a larger buffer capacity, typically about 40K.
- Named pipes exist as special files in the file system and may be created in one of two ways:
  - by using the UNIX mknod utility
  - by using the "mknod()" system call
- To create a named pipe using mknod, use the "p" option.
  The mode of the named pipe may be set using chmod, allowing others to access the pipe that you create.
  Here's an example of this procedure:

```
$ mknod myPipe  p          ---> create pipe.
$ chmod  ug+rw  myPipe   ---> update permissions.
$ ls -lg myPipe              ---> examine attributes.
prw-rw----   1   glass     cs          0  Feb  27  12:38  myPipe
```

- To create a named pipe using "mknod()", specify "S_IFIFO" as the file mode.

The mode of the pipe can then be changed using "chmod()".

- C code that creates a name pipe with read and write permissions for the owner and group:

```
mknod("myPipe", SIFIFO, 0);   /* Create a named pipe */
chmod("myPipe", 0660);        /* Modify its permission flags */
```

- Once a named pipe is opened using "open()", "write()" adds data at the start of the FIFO queue, and "read()" removes data from the end of the FIFO queue.

When a process has finished using a named pipe, it should close it using "close()", and

- when a named pipe is no longer needed, it should be removed from the file system using "unlink()".
- Like an unnamed pipe, a named pipe is intended only for use as a unidirectional link.
- Writer processes should open a named pipe for writing only, and reader processes should open a pipe for reading only.

Although a process can open a named pipe for both reading and writing, this usage doesn't have much practical application.

- an example program that uses named pipes, here are a couple of special rules concerning their use:

- If a process tries to open a named pipe for reading only and no process currently has it open for writing, the reader will wait until a process opens it for writing, unless O_NONBLOCK or O_NDELAY is set, in which case "open()" succeeds immediately.

- If a process tries to open a named pipe for writing only and no process currently has it open for reading, the writer will wait until a process opens it for reading, unless O_NONBLOCK or O_NDELAY is set, in which case "open()" fails immediately.
- Named pipes will not work across a network.

The next examples uses two programs, "reader" and "writer", to demonstrate the use of named pipes,

- A single reader process that creates a named pipe called "aPipe" is executed.

It then reads and displays NULL-terminated lines from the pipe until the pipe is closed by all of the writing processes.

- One or more writer processes are executed, each of which opens the named pipe called "aPipe" and sends three messages to it.

If the pipe does not exist when a writer tries to open it, the writer retries every second until it succeeds. When all of a writer's messages are sent, the writer closes the pipe and exits.

- Sample Output

```
$ reader & writer & writer &    ---> start 1 reader, 2 writers.
```

```
[1] 4698                    ---> reader process.
[2] 4699                    ---> first writer process.
[3] 4700                    ---> second writer process.
Hello from PID 4699
Hello from PID 4700
Hello from PID 4699
Hello from PID 4700
Hello from PID 4699
Hello from PID 4700
[2] Done  writer      ---> first writer exists.
[3] Done  writer      ---> second writer exists.
[4] Done  reader      ---> reader exists.
```

- Reader Program

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>          /* For SIFIFO */
#include <fcntl.h>
/*********************************************/
main()
{
  int fd;
  char str[100];
  unlink("aPipe");  /* Remove named pipe if it already exists */
  mknod("aPipe", S_IFIFO, 0);  /* Create name pipe */
  chmod("aPipe", 0660);  /* Change its permissions */
  fd = open("aPipe", O_RDONLY);  /* Open it for reading */
   while(readLine(fd, str) );  /* Display received messages */
     printf("%s\n", str);
  close(fd);   /* Close pipe */
 }
/*********************************************/
readLine( fd, str )
int fd;
char* str;
/* Read s single NULL-terminated line into str from fd */
/* Return 0 when the end of input is reached and 1 otherwise */
{
   int n;
     do /* Read characters until NULL or end of input */
      {
        n = read( fd, str, 1);  /* Read one character */
      }
```

```
        while ( n>0 && *str++ != NULL );
       return ( n> 0 );  /* Return false if end of input */
     }
```

Writer Program

```
 #include <stdio.h>
 #include <fcntl.h>
 /*****************************************************/
 main()
  {
   int fd, messageLen, i;
   char message[100];
   /* Prepare message */
   sprintf( message, "Hello from PID %d", getpid() );
   messageLen = strlen( message ) +1;
   do /* Keep trying to open the file until successful */
   {
     fd = open( "aPipe", O_WRONLY ); /*Open named pipe for writing */
     if ( fd == -1 ) sleep(1);  /* Try again in 1 second */
   } while ( fd == -1 );
  for ( i=1; i<=3; i++)  /* Send three messages */
   {
     write( fd, message, messageLen );  /* Write message down pipe */
     sleep(3);   /* Pause a while */
   }
  close(fd);   /* Close pipe descriptor */
}
```