Signal Management (Cont'd)

```
Check for pending, blocked signals
SYNOPSIS
       #include <signal.h>

       int sigpending (set)
       sigset_t *set;

   where:
       set    A structure to which the list of
              signals are to be written

   returns:     0 on success or -1 (generally no
                error is possible, but
                implementation defined)
DESCRIPTION
       The sigpending() function shall store the
       set of signals that are blocked from delivery
       and pending for the calling process, in the
       space pointed to by the argument set.
```

Signal Management (Cont'd)

```
EXAMPLE:
   sigset_t   pending_sigs, take_sig_mask;
   struct sigaction new, old;

   if(sigpending (&pending_sigs) == -1){
     perror("sigpending failed: ");
     exit(1);
   }

   if(sigismember (&pending_sigs, SIGINT)){
      /** SIGINT has been delivered but the  **/
      /** block bit for SIGINT is set so no  **/
      /** action has been taken yet...may    **/
      /** want to install a specific handler **/
      /** and turn off the block bit to      **/
      /** handle the signal now........      **/

      sigemptyset (&take_sig_mask);
      sigaddset (&take_sig_mask, SIGINT);
      sigaction(SIGINT, &new, &old);
      sigprocmask(SIG_UNBLOCK, &take_sig_mask, NULL);
      pause();
      sigprocmask(SIG_BLOCK, &take_sig_mask, NULL);
                       .
                       .
                       .
    }
```

Signal Management (Cont'd)

Wait for a signal to arrive

```
SYNOPSIS
       int pause(void)

SYNOPSIS
       #include <signal.h>

       int sigsuspend (sigmask)
       sigset_t *sigmask;

   where:
       sigmask   A structure containing a set of
                 signals, which temporarily replaces
                 the current block mask and allows
                 the process to sleep until some
                 signal, other than any in the new
                 mask, arrives

   returns:      -1 with errno EINTR (cannot succeed)
                 BUT the original signal mask IS
                 restored upon return with sigsuspend()
```

Signal Management (Cont'd)

```
sigsuspend(), pause()    cont'd

EXAMPLE:
   void  handler(int signum)
   { return; }

   sigset_t   suspended_blocked_sigs, take_sig_mask;
   struct sigaction new, old;

   sigfillset (suspended_blocked_sigs);
   sigfillset (take_sig_mask);
   sigdelset(suspended_blocked_sigs, SIGALRM);
   new.sa_handler = handler;
   new.sa_mask = take_sig_mask;
   new.sa_flags = 0;
   if(sigaction(SIGALRM, &new, &old) == -1){
         perror("can't set signals: ");
         exit(1);
   }
   alarm(5);
   sigsuspend(suspended_blocked_sigs);
   /**** when alarm clock rings, reset signals ****/
   /**** how would use of pause() be different ****/

   if(sigaction(SIGALRM, &old, NULL) == -1){
         perror("can't set signals: ");
         exit(2);
   }
```

Signal Management (Cont'd)

```
Set the real-time alarm clock
SYNOPSIS
       #include <unistd.h>

       unsigned int alarm (sec)
       unsigned int sec;

   where:
       sec   The number of wall-clock seconds to
             wait before sending SIGALRM to the
             caller.  If 0, cancel the alarm

  returns:   time remaining on the clock or 0

EXAMPLE:
   void  handler(int signum)
   { /** handle alarm timeout **/ }
   int time_in_code;

   if(sigaction(SIGALRM, &new, &old) == -1){
         perror("can't set signals: ");
         exit(1);
   }
   alarm(5);
   /*** do code which may encounter black hole ***/
   time_in_code = 5 - alarm(0);
```

Signal Management (Cont'd)

```
Set the real-time, virtual or profile alarm clock
SYNOPSIS
    #include <sys/time.h>

    int setitimer(int which,
                  struct itimerval *value,
                  struct itimerval *ovalue);

   where:
       which   ITIMER_REAL, ITIMER_VIRTUAL, or
               ITIMER_PROF
       value   Name of pointer to structure for
               storing timer value
       ovalue  Name of pointer to structure for
               storing old timer value
DESCRIPTION
       The system provides each process with three
       interval timers, defined in sys/time.h.  The
       getitimer call stores the current value of the
       timer specified by which into the structure
       pointed to by value.  The setitimer call sets
       the value of the timer specified by which to
       the value specified in the structure pointed
       to by value, and if ovalue is not NULL, stores
       the previous value of the timer in the
       structure pointed to by ovalue.

  returns:   0 on success, or -1
```

Signal Management (Cont'd)

`setitimer()` cont'd

The three timers are:

ITIMER_REAL	Decrements in real time. Delivers SIGALRM signal.
ITIMER_VIRTUAL	Decrements in process virtual time. Delivers SIGVTALRM signal.
ITIMER_PROF	Decrements both in process virtual time and during system call execution. Delivers SIGPROF signal.

```
struct itimerval{
 struct timeval it_interval; /* timer interval */
 struct timeval it_value;    /* current value  */
};

struct timeval{
 long    tv_sec;              /* seconds        */
 long    tv_usec;             /* micro seconds  */
};
```

Signal Management (Cont'd)

setitimer()   cont'd

EXAMPLE:

```
void  handler(int signum)
{ /** handle virtual alarm timeout **/ }

struct sigaction new, old;
struct itimerval *value;
struct itimerval *ovalue;

new.sa_handler = handler;
new.sa_mask = take_sig_mask;
new.sa_flags = 0;

value.it_interval.tv_sec=0;
value.it_interval.tv_usec=0;
value.it_value.tv_sec=3;
value.it_value.tv_usec=500000;

if(sigaction(SIGVTALRM, &new, &old) == -1){
      perror("can't set signals: ");
      exit(1);
}
setitimer(ITIMER_VIRTUAL, &value, &ovalue);

/** do work to time, take time-out in 3.5 sec **/
/** stop clock and get return for measurement **/

value.it_value.tv_sec=0;
value.it_value.tv_usec=0;
setitimer(ITIMER_VIRTUAL, &value, &ovalue);
printf("code ran for %d secs and %d micro-secs\n",
 ovalue.it_value.tv_sec, ovalue.it_value.tv_usec);
```

Signal Management (Cont'd)

A non-local go to with signal state

SYNOPSIS

```
#include <setjmp.h>

int sigsetjmp (sigjmp_buf env, int savemask);

void siglongjmp (sigjmp_buf env, int val);
```

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program. They work as do the setjmp() and longjmp() routines but if the savemask argument is non zero, they also save and restore a process's signal mask upon reactivation of a check point. Since the sigaction() call normally blocks further presentation of a signal when in a handler, longjumps out of a handler leave the signal taken blocked until explicitly unblocked by the program. To avoid this, these functions allow the signal mask established during a check point to be restored when that point is reactivated, thereby automatically adjusting the mask for the just taken signal. They are POSIX library routines (3C) from libc, NOT system calls.

Interprocess Communication

Since the traditional UNIX process model has been represented with a single thread of execution, and since the system call interface is basically synchronous, many applications a built from more than one process. When multiple process applications are constructed, it is generally necessary to provide a mechanism for the application processes to exchange information. UNIX provides the following IPC support:

- Half-duplex local-host stream communication is supported by **pipes**
  - Un-named pipes have supported IPC in UNIX from the beginning, but they are based on inheritance, and only work well between parent and child or siblings.
  - Named pipes use the name space of the file system to allow any collection of process to connect, but they still are limited by a half-duplex paradigm
- The BSD UNIX Domain Sockets (UDS) provide a full-duplex local-host stream communication facility with either an un-named or named capability.
- The System V IPCs provide named, local-host communication with:
  - A full-duplex datagram message queue facility
  - Shared memory segment support
  - Semaphore synchronization primitives
- The BSD Internet Socket facility provides a full-duplex, named, interhost stream and datagram communication facility

Pipe Based Interprocess Communication

Pipes are fundamentally one-way communication paths which allow one process to write into a pipe and another process to read from a pipe. They support a logical stream of data (generally implemented in the kernel using a message passing STREAMS pseudo driver), with no physical boundaries. A reader must understand how the data in a pipe are organized in order to extract information correctly, since no boundaries are embedded between logical messages.

- Un-named pipes are created using the **pipe()** system call. Typically they are created by a process prior to forking a child, and the resulting allocated pipe file descriptors (two channels, one open for RDONLY and the other for WRONLY) are then inherited by the created child. The child will use either the *write* side channel and leave the *read* side for the parent, or visa-versa.

- Named pipes are created using the **mknod()** system call. Such a pipe is represented by the allocation of a type **p** inode in the file system, which can be opened anytime after creation by any process wishing to communicate using the pipe, and having appropriate credentials to open the object. The **open()** system call used must indicate one of O_RDONLY, O_WRONLY or O_RDWR, and will block the caller until another process calls open() on the named pipe with a compatible mode.

Pipe Based IPC (Cont'd)

```
IPC with un-named pipe
SYNOPSIS
       int  pipe (fildes)
       int  fildes[2];

 where:
   fildes     Address of an array of two
              file descriptors; fildes[0]
              will hold the smaller channel
              number which will be available
              for reading only, while the
              fildes[1] element will hold the
              larger channel which will be
              open for writing only.

 returns:     0 on success, or -1
```

Pipe Based IPC (Cont'd)

```
pipe()    cont'd
EXAMPLE:
/*** A process which talks to itself  ***/
main()
{
  int   parray[2], nread;
  char  buf[100];

  if(pipe(parray) == -1){
    perror("pipe");
    exit(1);
  }
  if(write(parray[1], "hello",6) == -1){
    perror("write");
    exit(1);
  }
  switch(nread = read(parray[0], buf, sizeof(buf))){
    case -1:  perror("read");
              exit(1);
    case  0:  printf("EOF encountered \n");
              exit(1);
    default:  printf("read %d bytes: %s \n",
                                   nread, buf);
  }
}
The output:
read 6 bytes: hello
```

Using The Standard Tools With Pipes

A one-way communication example:

- The following code assumes a parent process which will carry out some processing on each named object in the current working directory
- Since there is a *naive* UNIX command known as the **ls** command, which will generate the names of all named objects in the current working directory, the application will create a child to run the **ls** command and pipe the results back to the parent application

Using The Standard Tools With Pipes (Cont'd)

EXAMPLE:

```
main()
{
 int pchan[2], pid, done,i;
 char buf[100];

 if(pipe(pchan) == -1){
   perror("pipe");
   exit(1);
 }
 switch( pid = fork() ){
  case -1:  perror("fork");
            exit(2);
  case  0:  close(1);
            if( dup(pchan[1] != 1 ){
              perror("dup");
              exit(3);
            }
            close(pchan[0]);
            close(pchan[1]);
            execl( "/bin/ls", "ls", NULL );
            perror("execl");
            exit(4);
 }
```

Using The Standard Tools With Pipes (Cont'd)

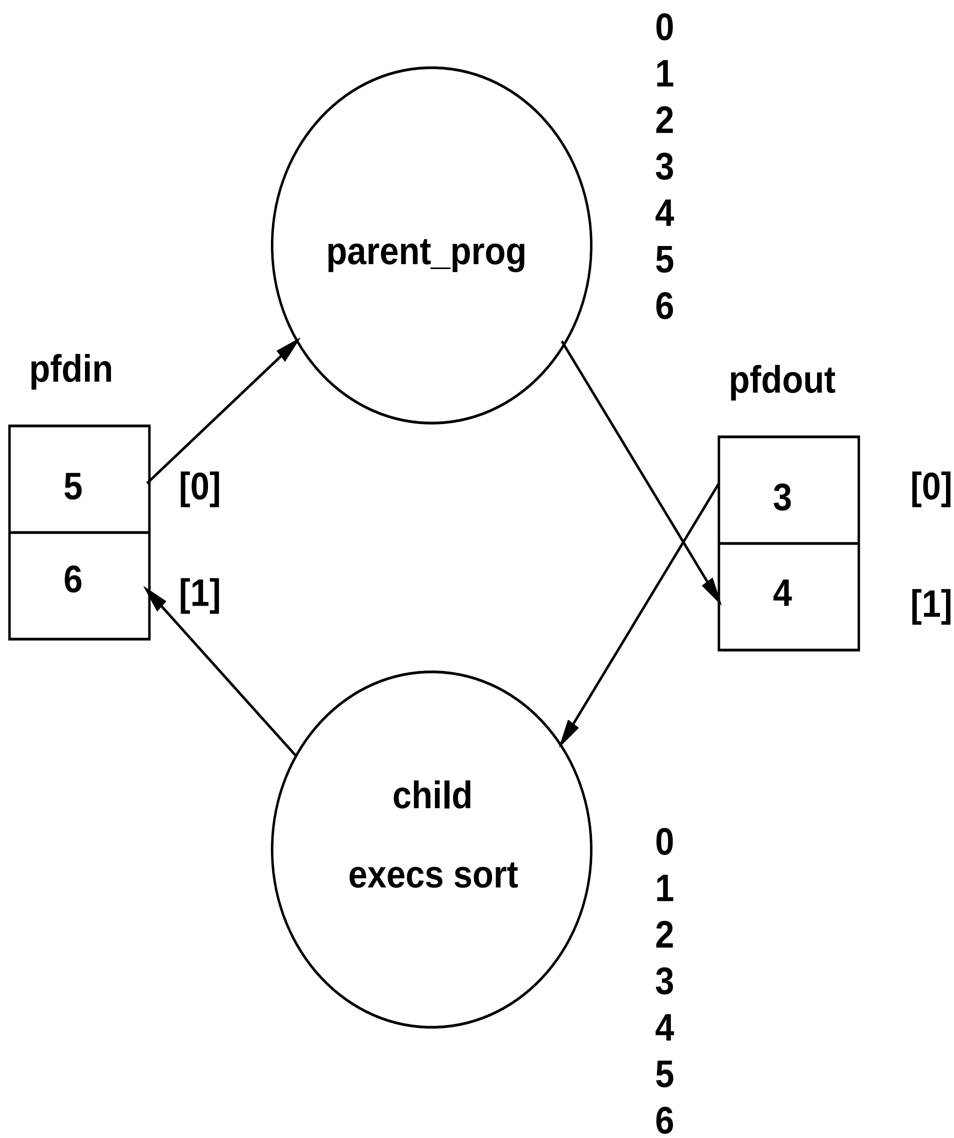
```
/***  THE NEXT STATEMENT IS CRITICAL !!!  ***/
 close(pchan[1]);
 done = 0;
 while (1){
  for( i=0; i<100; i++ ){
    if(read(pchan[0], &buf[i], 1) != 1){
      close(pchan[0]);
      done = 1;
      break;                         /* BREAK FOR  */
    }
    if (buf[i] == '\n'){
      buf[i] = '\0';
      break;                         /* BREAK FOR  */
    }
  }
  if(done) break;                   /* BREAK WHILE */

/*  FILE OBJECT NAME NOW A STRING IN BUF      */
/*  ANY PROCESSING WITH NAME DONE HERE       */

 printf("file object name is %s \n", buf);
 }
}
```

Using The Standard Tools With Pipes

A two-way communication example:

- the following example provides a function called fsort which provides a process the ability to create a child, load the child with the UNIX **sort** tool, pipe process data to the sort tool for sorting and read back the sorted data
- while the shell does not provide a two way piping syntax, it is quite easy to do from an application. The system programmer must be careful however, since deadlock can be a problem in this situation

A Two-Way Communication Example



Using The Standard Tools With Pipes (Cont'd)

EXAMPLE:

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

  main()
  {

    int     pfdout[2], pfdin[2], fd, nread;
    char    buf[512];

    if(pipe(pfdout) == -1 || pipe(pfdin) == -1)
    {
            perror("pipe");
            exit(1);
    }

    switch(fork())
    {
      case -1:      perror("fork");
                    exit(2);
```

Using The Standard Tools With Pipes (Cont'd)

```
  case  0:      if(close(0) == -1)
                {
                      perror("pipe");
                      exit(1);
                }
                if(dup(pfdout[0]) != 0)
                {
                      perror("dup");
                      exit(1);
                }
                if(close(1) == -1)
                {
                      perror("pipe");
                      exit(1);
                }
                if(dup(pfdin[1]) != 1)
                {
                      perror("dup");
                      exit(1);
                }
                if(close(pfdout[0]) == -1 ||
                   close(pfdout[1]) == -1 ||
                   close(pfdin[0])  == -1 ||
                   close(pfdin[1])  == -1 )
                {
                      perror("close");
                      exit(1);
                }
                execlp("grep", "grep", "123", NULL);
                      perror("execlp");
                      exit(1);
}
```

Using The Standard Tools With Pipes (Cont'd)

```
if(close(pfdout[0]) == -1 || close(pfdin[1]) == -1)
{
            perror("close");
            exit(1);
}
if((fd = open("data", O_RDONLY, 0)) == -1)
{
            perror("open");
            exit(1);
}
while((nread = read(fd, buf, sizeof(buf))) != 0)
{
            if(nread == -1)
            {
              perror("read");
              exit(1);
            }
            if(write(pfdout[1], buf, nread) == -1)
            {
              perror("write");
              exit(1);
            }
}
if(close(fd) == -1 || close(pfdout[1]) == -1)
{
            perror("close");
            exit(1);
}
```

Using The Standard Tools With Pipes (Cont'd)

```
while((nread = read(pfdin[0], buf, sizeof(buf))) != 0)
{
            if(nread == -1)
            {

              perror("read");
              exit(1);
            }
            if(write(1, buf, nread) == -1)
            {
              perror("write");
              exit(1);
            }
}
close(pfdin[0]);
}
```

Pipe Based IPC (Cont'd)

```
Duplicating file descriptors (channel numbers)
SYNOPSIS
       int  dup (fildes)
       int  fildes;
   where:
       fildes      A valid, active file descriptor
SYNOPSIS
       int  dup2 (old_fildes, new_fildes)
       int  old_fildes;
       int  new_fildes;
   where:
       old_fildes  A valid, active file descriptor
       new_fildes  Another file descriptor
DESCRIPTION
       Dup return the smallest unopen file descriptor
       and duplicates that descriptor to point to
       fildes.  Dup2 combines the functionality of
       the dup and close operations.  If old_fildes is
       an active, valid descriptor and new_fildes is a
       valid descriptor (active or not), new_fildes is
       made a duplicate of old_fildes.  If old_fildes
       and new_fildes already refer to the same
       object pointer, no changes occur.  In all other
       situations in which new_fildes is active, it is
       closed before being made a duplicate of
       old_fildes.  The close-on-exec flag is set so
       the descriptor remains open across exec(2)
       operations.
```

Pipe Based IPC (Cont'd)

Named pipes can be created with the **mknod()** call as previously discussed. There is also a *libc* library convenience reoutine called *mkfifo()* which call the mknod() system call for you:

```
Make a named pipe
SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>

       int mkfifo (const char *path, mode_t mode);

DESCRIPTION
       The mkfifo routine creates a new FIFO special
       file named by the pathname pointed to by path.
       The mode of the new FIFO is initialized from
       mode.  The file permission bits of the mode
       argument are modified by the process's file
       creation mask [see umask(2)].  The FIFO's owner
       id is set to the process's effective user id.
       The FIFO's group id is set to the process's
       effective group id, or if the S_ISGID bit is
       set in the parent directory then the group id of
       the FIFO is inherited from the parent.  mkfifo()
       calls the system call mknod() to make the file.
```