Chapter 8

Pipes

Pipes

- Pipes are kernel data structures for inter-process communication
- int *pipe*(int *filedes*[2]);
- ***Linux:*** *pipe* creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. *filedes*[0] is for reading, *filedes*[1] is for writing.
- "The ***POSIX*** standard does not specify what happens if a process tries to write to *filedes*[0] or read from *filedes*[1]."
- ***Solaris:*** The *pipe*() function creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. The files associated with *fildes*[0] and *fildes*[1] are streams and are both opened for reading and writing.

Pipes — PUP Example 3.20 (USP Program 6.3)

```
#include <stdio.h> <stdlib.h> <unistd.h> <fcntl.h>
void main(void)
{  int fd[2]; pid_t childpid;

   pipe(fd);
   if ((childpid = fork()) == 0) {  /* ls as child */
      dup2(fd[1], STDOUT_FILENO);
      close(fd[0]); close(fd[1]);
      execl("/usr/bin/ls", "ls", "-l", NULL);
      perror("The exec of ls failed");
   } else {                    /* sort as parent */
      dup2(fd[0], STDIN_FILENO);
      close(fd[0]); close(fd[1]);
      execl("/usr/bin/sort", "sort", "-n", "+4", NULL);
      perror("The exec of sort failed");                  }
exit(0); }
```


Named Pipes (FIFOs)

- Named pipes (FIFOs) are pipes turned into file system objects.
- A named pipe is a *special file* with access regulated via file system permissions:

  `prw-------  1 kahl users  0 Jan 30 00:08 /tmp/fifo1`
- Data is passed though the FIFO by the kernel without writing it to the file system.
- Normally, opening the FIFO blocks until the other end is opened also.
- When a process tries to write to a FIFO that is not opened for read on the other side, the process is sent a *SIGPIPE* signal.

PUP Program 3.3 skeleton (USP Prog. 6.5)

```
void main (int argc, char *argv[])
{  mode_t fifo_mode = S_IRUSR | S_IWUSR;
   int fd, status;  char buf[BUFSIZE]; unsigned strsize;
   mkfifo(argv[1], fifo_mode); /* create FIFO with u=rw */
   if (fork() == 0) {  /* The child writes */
      fprintf(stderr, "Child[%d] about to open\n", getpid());
      fd = open(argv[1], O_WRONLY);
      sprintf(buf,"written by child[%d]\n", getpid());
      strsize = strlen(buf) + 1;
      write(fd, buf, strsize);
      fprintf(stderr, "Child[%d] is done\n", getpid());
   } else {                    /* The parent does a read */
      fprintf(stderr, "Parent[%d] about to open\n",getpid());
      fd = open(argv[1], O_RDONLY | O_NONBLOCK);
      fprintf(stderr,"Parent[%d] about to read\n",getpid());
      while ((wait(&status)== -1) && (errno == EINTR)) {}
      read(fd, buf, BUFSIZE);
      fprintf(stderr, "Parent[%d] got: %s\n", getpid(), buf);
  }}
```


Client-Server Communication Using FIFOs

- Writes of up to *PIPE_BUF* bytes are **atomic**
  - ⇒ one FIFO can receive (short) requests from several clients
- Reads have **no** atomicity properties
  - ⇒ each reader neads one dedicated FIFO

I/O Blocking

How to monitor both the keyboard and the net in one process?

- *read*() on the keyboard hangs until signalled
- A byte on the net does not send a signal …
- *read*() on the net hangs until signalled
- A normal key press does not send a signal, either …
- *getc*(3) etc. are implemented on top of *read*(2)
- ???

Nonblocking I/O

In the man page for *read*(2) we find under “ERRORS”:

EAGAIN Non-blocking I/O has been selected using *O_NONBLOCK* and no data was immediately available for reading.

Checking the man page for open(2):

O_NONBLOCK or *O_NDELAY*

When possible, the file is opened in non-blocking mode. Neither the open nor any subsequent operations on the file descriptor which is returned will cause the calling process to wait. For the handling of FIFOs (named pipes), see also fifo(4). This mode need not have any effect on files other than FIFOs.

select() — #include <sys/select.h>

```
int select(int n, fd_set *readfds, fd_set *writefds,
    fd_set *exceptfds, struct timeval *timeout);
```

Three independent sets of descriptors are watched.

- Those listed in *readfds* will be watched to see if characters become available for reading (more precisely, to see if a read will not block — in particular, a file descriptor is also ready on end-of-file)
- Those in *writefds* will be watched to see if a write will not block
- Those in *exceptfds* will be watched for exceptions

On exit, the sets are modified **in place** to indicate which descriptors actually changed status.

Timeout using select() — from Linux man page

```
#include <stdio.h> <sys/time.h> <sys/types.h> <unistd.h>
int main(void) {
  fd_set rfds; struct timeval tv; int retval;

  FD_ZERO(&rfds);   /* Watch stdin (fd 0) */
  FD_SET(0, &rfds); /*  to see when it has input. */
  tv.tv_usec = 0;
  tv.tv_sec = 5;    /* Wait up to five seconds. */

  retval = select(1, &rfds, NULL, NULL, &tv);
       /* Don't rely on the value of tv now! */
  if (retval) printf("Data is available now.\n");
       /* FD_ISSET(0, &rfds) will be true. */
  else  printf("No data within five seconds.\n");
  return 0;}
```


Avoiding Suspension on Individual *read* and *write* Calls

Problem:

- Normal *read* and *write* **block** until I/O possible
- Program may need to do other things while I/O impossible
- Program may need perform I/O where it **first** becomes possible

Different solutions:

- Open with *O_NONBLOCK* and "**poll manually**"
- Use *select*
- Use *poll*
- Open with *O_ASYNC* and perform I/O in signal handlers
- Use multiple threads (carefully …)

Concurrent Reading — 1

```
#include <stdio.h>                          /* interleavingRead1.c */
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <errno.h>
int main(void) { pid_t childpid; int i,k,n,fd; char buf[2];

 if ((childpid = fork()) == -1) { perror("fork"); return 1; }
 if ((fd = open("test",O_RDONLY)) == -1)
   { perror("couldn't open"); return 1; }

 k = (childpid == 0) ? 1 : 10; /* distinguish parent and child */
 for (i=0; i<10; i++) {
   while((n = read(fd, buf, 1)) == -1 && (errno == EINTR)) {}
   printf("%2d: %2d --- %c\n", k, k * i, buf[0]);
   usleep((10 + k) * 20000);                          }
 return 0; }
```

Concurrent Reading — 2

```
#include <stdio.h>                          /* interleavingRead2.c */
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <errno.h>
int main(void) { pid_t childpid; int i,k,n,fd; char buf[2];

  if ((fd = open("test",O_RDONLY)) == –1)
    { perror("couldn't open"); return 1; }
  if ((childpid = fork()) == –1) { perror("fork"); return 1; }

  k = (childpid == 0) ? 1 : 10; /* distinguish parent and child */
  for (i=0; i<10; i++) {
    while((n = read(fd, buf, 1)) == –1&& (errno == EINTR)) {}
    printf("%2d: %2d --- %c\n", k, k * i, buf[0]);
    usleep((10 + k) * 20000);                      }
  return 0; }
```


Asynchronous I/O

- Standard **non-blocking I/O**: Input processed after successful return of non-blocking calls to *read*(), or return from *select*()
- **Asynchronous I/O with signals:** input processed in signal handler
  - Applications in real-time processing
  - May use signal queueing
  - Set up with *ioctl* (and *fcntl*)
- **New Asynchronous I/O according to POSIX:AIO**
  - Issue asynchronous requests; inspect results later
  - Allows to specify signals, or work without signals.

POSIX:AIO

- int *aio_read*(struct *aiocb* **aiocbp*);

  requests an asynchronous *read*
- int *aio_write*(struct *aiocb* **aiocbp*);

  requests an asynchronous *write*
- int *aio_error*(const struct *aiocb* **aiocbp*);

  returns the error status for the asynchronous I/O request with control block pointed to by *aiocbp*
- *ssize_t aio_return*(struct *aiocb* **aiocbp*);

  returns the final return status for the asynchronous I/O request with control block pointed to by *aiocbp*