FIFOs

- also called named pipes
- first in first out
- half-duplex
- associated with a file in the file system
- can be used between unrelated processes

Creating a FIFO

- created by calling mkfifo

```
#include <sys/types.h>
#include <sys/stat.h>
```

- int mkfifo(const char *pathname, mode_t mode);
- /* return 0 on success, -1 on error & set errno */

- mode specifies the permissions on the FIFO
- mkfifo will fail if the FIFO exists already; in that case, errno is set to EEXIST
- a FIFO may also be created by the mkfifo command (e.g. mkfifo –m 0666 FIFO)
- FIFOs cannot be used on NFS-mounted filesystems

Opening a FIFO

- open can be used to open a FIFO for reading or writing
- a FIFO should be opened either read-only or write-only
- there is a O_NONBLOCK flag to open
  fd = open (FIFO, O_WRONLY | O_NONBLOCK);
- fcntl can be used to enable O_NONBLOCK after the file is already open
- if O_NONBLOCK is not set (the default), then an open will block if:
  - we open a FIFO for reading but no process has it open for writing; or
  - we open a FIFO for writing but no process has it open for reading

- if O_NONBLOCK is set,
  - an open for read-only will return immediately
  - an open for write-only will return –1 with errno set to ENXIO if no process has the FIFO open for reading

Reading & Writing

- use read & write, just as for a pipe
- the behaviour of reading from & writing to a FIFO is the same as that for a pipe
- reading from a FIFO that no process has open for writing: read returns 0 to indicate end-of-file (when all data has been read)

- otherwise, reading from an empty FIFO:

    - O_NONBLOCK not set: blocks until data is available for FIFO is no longer open for writing
    - O_NONBLOCK set: returns an error with errno set to EAGAIN

- the macro PIPE_BUF specifies the maximum amount of data that can be written atomically to a FIFO
- writing to a FIFO that is not open for reading by any process generates the SIGPIPE signal; errno is set to EPIPE if read returns (e.g. if signal is ignored)
- otherwise, writing to a FIFO:
    - O_NONBLOCK not set: write may block
    - O_NONBLOCK set:

        - number of bytes to write is less than or equal to PIPE_BUF:
            - if there is enough room in the FIFO, all bytes are transferred
            - if there is not enough room: write returns –1 with errno set to EGAIN (because write must be atomic)
        - number of bytes to write is greater than PIPE_BUF:
            - if FIFO is full, write returns –1 with errno set to EAGAIN
            - if there is room in the FIFO, write transfers & returns the number of bytes the FIFO can hold

<u>Other operations</u>

- use close to close the FIFO
- use unlink to unlink the FIFO
- seeking within a FIFO is not allowed; if lseek is called on a FIFO, it will return –1 with errno set to ESPIPE

Pipes

- generally half-duplex i.e. data flows in one direction
    - some systems provide full-duplex pipes but POSIX only specifies half duplex ones
- first in, first out
- can only be used between processes that share a common ancestor
    - typically, a process creates a pipe before it calls fork; the parent & child then use the pipe to communicate

<u>Creation</u>

- a pipe is created by the pipe function

```
#include <unistd.h>
int pipe(int filedes[2]); /* returns 0 if OK, -1 on error */
```

- the address of an s reading while filedes[1] is open for writing
- filedes [0] is open for reading while filedes [1] is open for writing
- pipe returns 0 on success & -1 on error; possible errors include file table overflow (ENFILE) & too many open files (EMFILE)

<u>Writing to a pipe</u>

- use write to write to a pipe
- writing to a pipe that is not open for reading by any process generates the SIGPIPE signal

    - the default action of the signal is to terminate the process
    - if the signal is ignored or if it is caught & the signal handler returns, write returns –1 & errno is set to EPIPE (broken pipe)

- the macro PIPE_BUF (in <limit.h> or <sys/param.h>) specifies the maximum number of bytes a pipe may hold
- each write operation appends data to the end of the pipe
- a write operation of PIPE_BUF bytes or less is guaranteed to be atomic; it will not be interleaved with other write requests to the same pipe
- if a process writes to a pipe & there is enough space (to guarantee atomicity if necessary), the data is sent down the pipe & the call returns immediately
- if there is not enough space & the O_NONBLOCK flag is not set (the default) for the pipe, the process will block until data is read from the pipe
- if there is not enough space but the O_NONBLOCK flag is set, then

    - if the number of bytes in the write request is less than or equal to PIPE_BUF, write returns –1 with errno set to EAGAIN

   - otherwise, part of the data may be written & write returns the number of bytes written, or if the pipe is full, write returns –1 with errno set to EAGAIN.

- The O_NONBLOCK flag may be set using the fcntl function

Example:

```
#include <fcntl.h>
…
int     flags;

if ((flags= fcntl(fd, F_GETFL, 0)) < 0) {  /* get original flag */
          perror("F_GETGL");
          exit(1);
}
flags |= O_NONBLOCK ;      /* turn on O_NONBLOCK  */
if (fcntl(fd, F_SETFL, flags) < 0) {      /* set new flag */
          perror("F_SETFL");
          exit (2);
}
```

<u>Reading from a pipe</u>

- use read to read from a pipe
- when reading from a pipe whose write end has been closed, read returns 0 to indicate end-of-file when all data has been read
   - technically, read returns 0 when there are no more writers to the pipe; there can be more than one process writing to a pipe

   - if the O_NONBLOCK flag is not set, a process attempting to read from an empty pipe will block until data is available or until the write end is closed

   - if the O_NONBLOCK flag is set, a read operation on an empty pipe will return –1 with errno set to EAGIN
   - if we ask to read more data than is in the pipe, only the available data is returned & read returns the number of bytes read

<u>Closing a pipe</u>

- use close to close the ends of a pipe
- the effect on read & write when an end of a pipe is closed is given above

<u>dup2</u>

```
#include <unistd.h>

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

/* returns newfd if OK, -1 on error & sets errno */

- dup2 makes newfd a copy of the file descriptor oldfd, closing newfd first if necessary
- if oldfd equals newfd, dup2 returns newfd without closing it

<u>popen & pclose</u>
#include <stdio.h>

file * popen(const char *cmdstring. const car * type);
/* returns file pointer if OK, NULL on error */

int pclose (FILE *fp);
/* returns termination status of cmdstring, or –1 on error */

- popen creates a pipe, forks a child, closes the unused ends of the pipe & then execs a shell to execute cdstring: it returns a file pointer
- if type is “r”, the returned file pointer is connected to the standard output of cmdstring & we can read from it
- if type is “w”, the returned file pointer is connected to the standard input of cmdstring & we can write to it
- since popen execs a shell to executed cmdstring, cmdstring can contain special characters
- use pclose to close the file pointer obtained by popen; it waits for the command to terminate & returns the termination status of the shell.

Processes

- A program is an executable file residing in a disk file
- A process is an executing instance of a program
- Each process is identified by a unique nonnegative integer, its process ID (PID)

Process Creation

- A new process is created when an existing process calls the fork () system call
- the existing process is called the parent process
- the new process is called the child process
- the child process is a copy of its parent
- processes form a hierarchy; each process has 1 parent & may have 1 or more child processes
- the init process (with PID 1) is usually the first user process created when the system boots

Process Attributes

Each process has a number of attributes; some are :
PID - unique process ID number
PPID - parent process ID number
PGID - process group ID (PID=PGID for group leader)
SID - session ID (PID=PGID=SID for session leader
TTY - associated terminal device (controlling terminal)
TPGID- terminal process group ID
RUID -real user id
EUID -effective user id
RGID -real group id
EGID -effective group id

- The TPGID is the PGID of the foreground process group associated with the controlling terminal

Destroying Processes

- We can kill a process by sending it a signal with the kill command kill [- signal] pid
- A signal is a software interrupt
- Signals are numbered & are given names
- Some common signals: SIGINT, SIGTERM, SIGQUIT, SIGKILL
- Some signals can be caught or ignored by a process so that sending them will not terminate the process
- However, SIGKILL (usually signal number 9) cannot be caught or ignored; so it can be used to kill a runaway process (kill –KILL pid)
- Killing the login shell is equivalent to logging out

- You can only kill processes that you own
- The superuser can kill any process (except a zombie)

Process Management

- Use the ps command to display process states
  % ps
  % ps –aux
  % ps –axj

- use nice to start a process at a specific priority
  nice [+|-n] command
- n is the offset from the default nice number
- lower number is better priority

- use nohup to run a command immune to hang-ups (so that command continues to run in the background after you log out)
  % nohup make >& log&
- if stdout is a tty, it & stderr are redirected to the file nohup.out

- use at to schedule a job

% at 1:30pm today
mail awei < infile
^D
Job 4 will be executed using /bin/csh
% at –1

date	owner	queue	job#
11:30:00 02/07/97	awei	c	4

- standard output & standard error output from an "at" job is mailed to the user.

Sessions & Process Groups

- In a typical situation, processes connected by pipes belong to the same process group
- Processes in the same login session belong to the same session

Example

Consider the following 2 commands issued in the same login session:

% process1 | process2 | process3 & (a background process group; can have several)
% process4 | process5 (a foreground process group)

The 2 process groups belong to the same session.

User & group ids of a process

- Real user & group IDs identify who is actually executing the program & the group he belongs to
- The effective user & group IDs determine file access permissions
- Usually the effective user/group ID of the process is the same as its real user/group ID
- For set-user-ID programs, the real user ID is the person executing the program, but the effective user ID is the owner of the program
- Similarly for set-group-ID program
- Use chmod to create set-user-ID/set-group-ID programs

Daemons

- a long-running background process
- Don't have a controlling terminal
- Typically a server process that waits for a client to contact it, requesting some kind of service

Example
The ftpd daemon is responsible for handling FTP requests; the client is ftp

Process Termination

- A process can terminate normally (e.g. by calling exit ()) or abnormally (e.g. by calling abort ())
- in either case, a termination status is generated
- the parent process can obtain the termination status by calling wait () or waitpid ()
- a process will block if it calls wait () /waitpid() before its child has terminated
- if the parent terminates before its child processes, the init process inherits those child processes (i.e. it becomes their parent)

Zombies

- When a process terminates, most of its resources (e.g. memory) are released
- The kernel has to keep some info of the process so that the info is available when the parent calls wait () or waitpid()
- The info kept includes the PID, termination status & the amount of CPU time taken by the process
- A zombie is a process that has terminated but whose parent hasn't "waited" for it

Process Life Cycle

what happens when we run xedit:

- current shell forks a copy of itself
- child process execs the xedit program (by calling of the exec functions)
    - when a process execs a program, it is completely replaced by the new program & the new program starts executing
- if xedit is run in the foreground, the parent calls wait() or waitpid() & blocks
- otherwise we get back the command prompt immediately

what happens when edit terminates:

- if it was running in the foreground, wait() or waitpid() returns & we get back the command prompt wait() or waitpid()
- If it was running in the background, the parent shell will be notified by a signal: the parent will then call wait() or waitpid()