Pipes

Inter-Process Communication (IPC)

- Chapter 12.1-12.3
- Data exchange techniques between processes:
  - message passing: files, pipes, sockets
  - shared-memory model (not the default … not mentioned in Wang, but we'll still cover in this, a few weeks)
- Limitations of files for inter-process data exchange:
  - slow!
- Limitations of pipes:
  - two processes must be running on the same machine
  - two processes communicating must be "related"
- Sockets overcome these limitations (*we'll cover sockets in the next lecture*)

File Descriptors Revisited

- Section 11.1-2
- Used by low-level I/O
  - `open(), close(), read(), write()`
- declared as an integer
  `int fd ;`
- Not the same as a "file stream", `FILE *fp`
- streams and file descriptors *are* related (see following slides)

Pipes and File Descriptors

- A fork'd child inherits file descriptors from its parent
- It's possible to alter these using **`fclose()`** and **`fopen()`**:

```
fclose( stdin );
FILE *fp = fopen( "/tmp/junk", "r" );
```

- One could exchange two entries in the fd table by closing and reopening both streams, but there's a more efficient way, using **`dup()`** or **`dup2()`** (*…see next slide*)

`dup()` and `dup2()` (12.2)

```
newFD = dup( oldFD );
if( newFD < 0 ) { perror("dup"); exit(1); }
```

or, to force the newFD to have a specific number:

```
returnCode = dup2( oldFD, newFD );
if(returnCode < 0) { perror("dup2"); exit(1);}
```

- In both cases, **`oldFD`** and **`newFD`** now refer to the same file
- For **`dup2()`**, if **`newFD`** is open, it is first automatically closed
- Note that **`dup()`** and **`dup2()`** refer to fd's and *not* streams
  - A useful system call to convert a stream to a fd is

```
int fileno( FILE *fp );
```

`pipe()` (12.2)

- The **`pipe()`** system call creates an internal system buffer and two file descriptors: one for reading and one for writing

- With a pipe, typically want the stdout of one process to be connected to the stdin of another process … this is where **`dup2()`** becomes useful (*see next slide and figure 12-2 for examples*)

- Usage:

```
int fd[2];
pipe( fd );  /* fd[0] for reading; fd[1] for writing */
```

`pipe()`/`dup2()` example

```
/* equivalent to "sort < file1 | uniq" */
int fd[2];
FILE *fp = fopen( "file1", "r" );
dup2( fileno(fp), fileno(stdin) );
fclose( fp );
pipe( fd );
if( fork() == 0 ) {
   dup2( fd[1], fileno(stdout) );
   close( fd[0] );  close( fd[1] );
   execl( "/usr/bin/sort", "sort", (char *) 0 );  exit( 2 );
} else {
   dup2( fd[0], fileno(stdin) );
   close( fd[0] );  close( fd[1] );
   execl( "/usr/bin/uniq", "uniq", (char *) 0 );  exit( 3 );
}
```

`popen()` and `pclose()` (12.1)

- **`popen()`** simplifies the sequence of:
  - generating a pipe
  - forking a child process
  - duplicating file descriptors
  - passing command execution via an exec()
- Usage:

```
FILE *popen( const char *command,
             const char *type );
```

- Example:

```
FILE *pipeFP;
pipeFP = popen( “/usr/bin/ls *.c”, “r” );
```

Sockets

What are sockets? (12.5)

- Sockets are an extension of pipes, with the advantages that the processes don't need to be related, or even on the same machine
- A socket is like the end point of a pipe -- in fact, the UNIX kernel implements pipes as a pair of sockets
- Two (or more) sockets must be connected before they can be used to transfer data
- Two main categories of socket types … we'll talk about both:
  - the <u>UNIX domain</u>: both processes on same machine
  - the <u>INET domain</u>: processes on different machines
- Three main types of sockets: **`SOCK_STREAM`**, **`SOCK_DGRAM`**, and **`SOCK_RAW`** … we'll only talk about **`SOCK_STREAM`**

Connection-Oriented Paradigm

SERVER		CLIENT
Create a socket **`socket()`**		Create a socket **`socket()`**
Assign a *name* to the socket **`bind()`**		
Establish a queue for connections **`listen()`**		
Extract a connection from the queue **`accept()`**	established	Initiate a connection **`connect()`**
`read()`		**`write()`**
`write()`		**`read()`**

Example: server.c

- *FILE* “***server.c***” … highlights:

```
socket( AF_UNIX, SOCK_STREAM, 0 );
serv_adr.sun_family = AF_UNIX;
strcpy( serv_adr.sun_path, NAME );
bind( orig_sock, &serv_adr, size );
listen( orig_sock, 1 );
accept( orig_sock, &clnt_adr, &clnt_len );

read( new_sock, buf, sizeof(buf) );

close( sd );
unlink( the_file );
```

Example: client.c

- *FILE "**`client.c`**"* … highlights:

```
socket( AF_UNIX, SOCK_STREAM, 0 );
serv_adr.sun_family = AF_UNIX;
strcpy( serv_adr.sun_path, NAME );

connect(orig_sock, &serv_adr, size );

write( new_sock, buf, sizeof(buf) );

close( sd );
```

- Note: **`server.c`** and **`client.c`** need to be linked with the **`libsocket.a`** library (ie: **`gcc -lsocket`**)

The INET domain

- The main difference is the **`bind()`** command … in the UNIX domain, the socket name is a *filename*, but in the INET domain, the socket name is a *machine name* and *port number*:

```
static struct sockaddr_in serv_adr;
memset( &serv_adr, 0, sizeof(serv_adr) );
serv_adr.sin_family      = AF_INET;
serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_adr.sin_port        = htons( 6789 );
```

- Need to open socket with **`AF_INET`** instead of **`AF_UNIX`**
- Also need to include **`<netdb.h>`** and **`<netinet/in.h>`**

The INET domain (cont.)

- The client needs to know the machine name and port of the server

```
struct hostent  *host;
host = gethostbyname( "eddie.cdf" );
```

- Note: need to link with **`libnsl.a`** to resolve **`gethostbyname()`**

- see course website for:
  - **`server.c, client.c`** UNIX domain example
  - **`server2.c, client2.c,`** INET domain example