Interprocess Communication

- Single processes have limits on
  - Memory usage
  - Execution time
  - Disk quotas
- Processes can be executing on different machines that want to share information
- Need to be able to pass data between different processes
- Have already used some high level, complex forms of IPC, namely MPI, pthreads and OpenMP
- Want to look at more primative types

Signals

- Signals are a primative method to tell processes when an asynchronous event occurs
- Processes can send each other signals using the *kill(...)* system call
- You can only send signals to processes that are running with the same user id (unless you are root)
- Other signals are sent by the operating system when errors or events occur
  - Invalid memory access
  - Lack of system resources
  - Terminal interaction
  - Tracing process execution

Standard Signals

- HUP — hang up from a controlling terminal
- INT — interrupt from keyboard
- QUIT — quit (CTRL-C)
- TRAP — breakpoint reached
- BUS — bad memory access
- FPE — floating point exception
- ILL — illegal instruction
- KILL — terminate process
- SEGV — invalid memory reference
- STOP — suspend execution
- CONT — continue execution
- TSTP — stop from the keyboard (CTRL-Z)

Handling signals

- Each process has its own signal table which tells it how to deal with signals
- The default is to call *exit()*
- Using the *signal(...)* system call a user can designate another function to be executed instead of *exit()*
- The function in the table is executed immediately on receipt of the signal (once the program leaves kernel mode)
- Once the signal has been handled (assuming exit hasn't been called) the process continues executing as if nothing happened

Signal Uses

- Used by the scheduler to organise process execution
- Used by debuggers to control the process execution
- Print out current process status or important data structures
- Programs such as web servers, mail servers etc. often use the HUP signal to force a re-read of the configuration files

File descriptors

- Usually when we want to open a file for reading or writing we use `fopen(...)` and then `fprintf(...)` and `fscanf(...)` or `fgets(...)`
- This gives us a pointer to a `FILE` structure
- However the system calls for I/O use a file descriptor rather than a file pointer
  - *int open(const char *pathname, int flags);*
  - *int close(int fd);*
  - *ssize_t read(int fd, void *buf, size_t count);*
  - *ssize_t write(int fd, const void *buf, size_t count);*
- In the UNIX world, everything gets treated like a file — files, network access, directories, disks
- Use the same low level system calls to access them

File descriptor table

- Each process has a table which provides a mapping between numbers and open files
- The file descriptor is the id in the table
- By default every process has three open file descriptors at startup
  - 0 — stdin
  - 1 — stdout
  - 2 — stderr
- These are the file descriptor numbers

Pipes

- The next simplest form of IPC are pipes
- Found in all flavours of Unix (and other OSes)
- The allow data transfer in a FIFO manner
- Two kinds of pipes
  - Named pipes
  - Unnamed pipes
- Named pipes are created using `mknod`
- Unnamed pipes are created using `pipe`

Named pipes

- Named pipes are accessed like files
- Use `open`, `read`, `write`, `close`
- They are accessible to all processes with relevant permissions to access it
- The problem is that you need to know where in the filesystem the named pipe is

```
% mknod mypipe p
% cat /etc/passwd > mypipe
```

```
% cat < mypipe
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bi
⋮
```

Unnamed pipes

- Unnamed pipes are only accessible by related processes
- Normally a process calls `pipe` and then uses `fork` to create a child to which it wants to send data
- `pipe(int fd[2])` creates a pair of file descriptors
  - fd[0] is used for reading
  - fd[1] is used for writing

```
int fd[2];
pipe(fd);
pid = fork();
if(pid)
      write(fd[1], string, strlen(string));
else
      read(fd[0], buffer, buflen);
```

Unnamed pipes

- Unnamed pipes occur relatively frequently
- `ls -al | sort -n +4 | less`
- Almost all unix pipes are uni-directional
- To get two-way communication you need to make two pipes
- Normally you close the unused ends of the pipes in the relevant process
- Using `dup` with pipes can lead to a nice interaction with stdin and stdout

Shared Memory

- Comes from SystemV Unix
- Processes can mark regions of their heap as shareable
- Another process can attach to and use this region as part of their own address space
- Once attached successfully, the second process uses the memory as normal — no further system calls are required
- Each shared memory segment has a key associated with it
- These are unique across the system

Shared Memory

- Creating a shared region
- `shmid = shmget(32, 128*1024, 0777|IPC_CREATE);`
- Attaching a shared region to the address space of another (or the same) process
- `addr = shmat(shmid, 0, 0);`
- Detaching the region
- `shmdt(addr);`
- Controlling the region
- `shmctl(shmid, IPC_RMID, 0);`

Shared Memory

- The key to be used has to be shared between the processes somehow — chicken and egg problem
- The kernel cannot know which processes are allowed access to a shared memory segment
- The sharer can only grant read/write access as in file access
- Rogue processes can just guess the right ids and corrupt data

Sockets

- As outlined, pipes and shared memory both have drawbacks and limitations
- To overcome some of these we introduce sockets
- First appeared in BSD Unix
- Goals
  - Transparency — communication shoudn't depend on whether the processes are on the same machine or not
  - Efficiency — avoid needing server processes to regulate the communication
- Sockets rely on the TCP/IP protocols for communication

OSI Model

- An international standard that separates the different levels of abstraction into layers
- Each layer performs a well defined function
- The model has seven layers
  - Application — SMTP, HTTP etc.
  - Presentation — encoding of data
  - Session — manage dialogue control
  - Transport — ensures pieces are presented in order
  - Network — routing across the network
  - Data link — error free local transmission
  - Physical — electronics of high and low voltage
- This is a reference model — nobody actually implements this to manage their communication

TCP/IP basics

- The TCP/IP suite of protocols implements the network and transport layers of the OSI model
- There are two modes of operation
  - connection oriented, reliable (TCP)
  - connectionless, unreliable (UDP)
- TCP (Transmission control protocol) breaks up your message into small chunks for transfer over the network and re-assembles them in the right order at the other end
- UDP (User datagram protocol) allows small messages to be delivered quickly but with no checking for out of order, missing parts, duplicate arrivals etc.

Network addresses

- To allow programs to talk to each other, each application and machine must be individually addressable
- Each address consists of two parts
  - IP address to identify the machine
  - TCP/UDP port to identify the application
- IPv4 addresses are of the form 134.226.113.65. This gives 2^{32} addresses
- TCP/UDP ports are a number between 1 and 65536
- A new version of the IP protocol, IPv6, uses 128 bits and so gives 2^{128} addresses - approx 1×10^{28} addresses per person on the planet
- The DNS service provides a mapping between IP addresses and human memorable host names (woodstock.tchpc.tcd.ie)

Sockets

- The normal method for network programming is to use BSD sockets
- The end points (IP address and port) are represented by a socket
- The *socket(...)* system call creates a socket which is a file descriptor. It is used by all the rest of the networking calls
- Next we need to choose which port we are using
- If the program is the server then we need to explicity bind to a specific port using the *bind(...)* call
- If the program is a client then we normally don't care which port we use and leave it to the OS to pick one

Sockets

Server

- Use *listen(. . .)* to indicate that new connections are to be accepted by the socket
- Connections are received one at a time using *accept(. . .)*

Client

- Use *connect(. . .)* to establish communication to the server process

Sockets

- Once a connection has been established processes may use *read/write* or more commonly the *send/recv* functions
- Sockets are bi-directional — no need to open two sockets like we needed with pipes
- A socket is just a file descriptor. We can use *fdopen(...)* to get a file handle
- Can then use *fprintf(...)* and *fscanf(...)* to send and receive messages