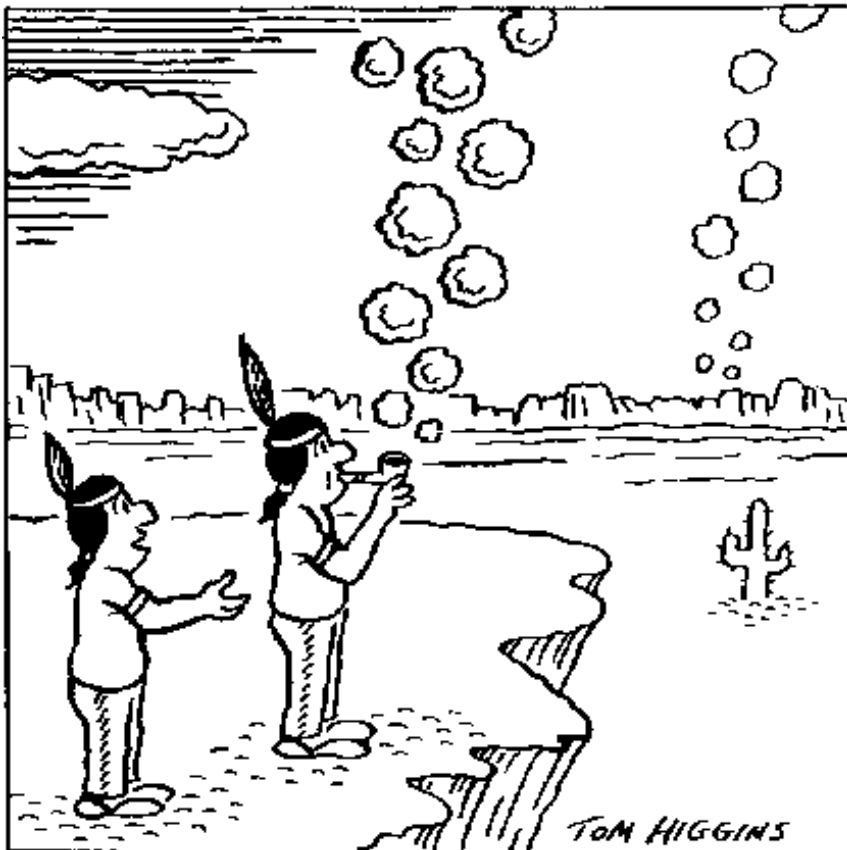




Interprocess communication



“The best way to predict the future
is to invent it.”

Alan Kay



Communicating Processes

- Processes can be started using the exec system calls
- Processes can operate in parallel using the fork system call followed by exec
- Processes sometimes wish to cooperate and exchange information during execution



Communicating Processes

- Why multiple processes?
 - Synchronous activities
 - Asynchronous events
- Signals
- Pipes
- Files - select
- Shared memory (tutorial)



File Descriptors

- low level I/O is performed on file descriptors that are small integers indicating an open file
- when process is started file descriptor 0 is standard input, 1 is standard output, 2 is standard error output
- low level system call functions operate on file descriptors



I/O system calls

- low level I/O functions include:
 - creat
 - open, close
 - read, write
 - ioctl
- eg read 100 characters from standard input into array “buffer”

```
read(0,buffer,100)
```



pipe

```
int pipe(int filedes[2]);
```

- `filedes` is a two element array of integers that is filled in with two file descriptors
- `filedes[0]` is for reading
- `filedes[1]` is for writing
- data written into `filedes[1]` can be subsequently read from `filedes[0]`
- the `pipe` function returns 0 on success, -1 on failure



Using pipes

- a parent process can communicate with a child by creating a pipe before the fork
- the parent can then write data to `fdes[1]` and the child can read `fdes[0]`
- the system has a small amount of buffering
- if the buffer is filled, the writer is suspended until the reader has read some data



Using files

- Process A talks to Process B via a named file.
- A: Open file, write data, close file
- B: Open file, read data, close file
- How can they synchronise read/write?



Using files

Process A

- wait for signal
- `int fd = open("shared.txt", O_WRONLY);`
- write some data
- `close(fd)`
- send signal



Using files

Process B

- Busy loop for flag, triggered by signal
- `int fd = open("shared.txt", O_RDONLY);`
- read some data
- `close(fd)`
- Send signal back



Using files

- Requires setup signal
- What is the event?
 - File has been opened/closed
 - File has been read/written
- Use `select()` !
 - `man 2 select`



Using files with select()

- select() monitors multiple file descriptors
- The file descriptors are specified by the file descriptor set.
- select() wait for an I/O event for any of the file descriptors in the set
 - There is data available to read from the file
 - There is space available to write to the file



Using files with select()

- Macros are used to define the set of file descriptors
 - FD_ZERO(), FD_SET(), FD_CLR(), FD_ISSET()
- There are three descriptor sets

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



Blocking I/O

Each file descriptor access can cause the processes to block

- pipe/file write will block until space available
- pipe/file read will block until data available
- Processes cannot do other useful work. May miss events, deadlines
 - Realtime systems may not afford to wait for I/O



Blocking I/O

Synchronous sharing of data of

- They each are waiting for signal interrupt, cannot do useful work
- E.g. wait for 100 bytes, but only received 46
- It is possible to open a file in NON BLOCKING MODE



Blocking I/O

- `open("bigfile.bin", O_NONBLOCK | O_RDONLY);`
- Change mode of existing fd (stdin, stdout, stderr)

```
fcntl(fd, F_SETFL,  
      fcntl(fd, F_GETFL) |  
      O_NONBLOCK);
```

Use existing
flags



Summary

- For simple communications between processes a pipe can be used
- For communications where a process needs to be interrupted, signals can be used
- For large amounts of shared data:
 - files are helpful
 - `select()` and `epoll()`
 - non-blocking
 - shared memory
- picture acknowledgement:

<http://www.pipes.org/Ephemeris>