Interprocess Communication

Look at Unix primitives. Also covered in Tanenbaum chapter on Unix.

How does one process communicate with another process?

- semaphores  *signal* notifies *waiting* process

- message passing  processes send and receive messages.

- software interrupt  process notified *asynchronously*

Software Interrupts

Similar to hardware interrupt; processes interrupt each other through software operations. Important to realize that interrupts are asynchronous! Stops execution and then restarts.

Examples:

- user types “attention” or “interrupt” key (ctl-C or DEL)

- child process completes

- an alarm scheduled by the process has expired

- resource limit exceeded (e.g., disk quota, CPU time, etc.)

- programming errors such as accessing invalid data, divide by zero

- `SendInterrupt(pid, num)`  sends an interrupt of type *num* to process *pid*. In Unix this routine is *kill()*.

- `HandleInterrupt(num, handler)`  specifies that user supplied routine *handler* should be invoked when interrupt of type *num* occurs. In Unix this routine is *signal()*.

  Typical handlers:

  ignore

  terminate (perhaps with core dump of virtual space)

  user supplied interrupt handler
/* signal.C */

#include <iostream.h>
#include <signal.h>
#include <unistd.h>

int n;

main(int argc, char **argv)
{
    void InterruptHandler(int), InitHandler(int);

    n = 0;

    signal(SIGINT, InterruptHandler); /* signal 2 */
    signal(SIGHUP, InitHandler); /* signal 1 */
    while (1) {
        n++;
        sleep(1);
    }
}

void InterruptHandler(int signum)
{
    cout << "Received " << signum << ", value of n is " << n << "\n";
    exit(0);
}

void InitHandler(int signum)
{
    cout << "Received " << signum << ", resetting the value of n to zero\n";
    n = 0;
}
% g++ -o signal signal.C
% ./signal
./signal
^C            (interrupt character)
Received 2, value of n is 3
% ./signal &
[1] 32363
% kill -1 %1
Received 1, resetting the value of n to zero
% kill -2 %1
% Received 2, value of n is 16
[1]    Done    . ./signal
Pipes

In Unix, a pipe is a unidirectional, stream communication abstraction. Show a picture!! One process writes to the “write end” of the pipe, and a second process reads from the “read end” of the pipe.

The command interpreter is responsible for setting up a pipe. For instance, upon entering:

```
% ls | more
```

the shell would:

1. create a pipe.
2. create a process for the `ls` command, setting stdout to the write side of the pipe.
3. create a process for the `more` program, setting stdin to the read side of the pipe.

A pipe consists of (keep using the same picture showing the pipe as a buffer)

- two descriptors, one for reading, one for writing.
- reading from the pipe advances the read pointer
- writing to the pipe advances the write pointer
- example of the bounded-buffer problem:
  - operating system buffers data in the pipe (Unix pipe 4096 bytes (4K))
  - operating system blocks reads of empty pipe
  - operating system blocks writes to full pipe
- pipe data consists of unstructured character stream
Pipes unify input and output. When a process starts up, it *inherits* open file descriptors from its parent.

- by convention, file descriptor 0 is standard input
- file descriptor 1 is standard output
- file descriptor 2 is standard error

Thus, when a process reads from standard input, it doesn’t know (or care!) whether it is reading from a file or from another process.

Likewise, output written to standard output might go to a terminal, a file, or another process.

System calls:

- `count = read(fd, buffer, nbytes)` reads from a file descriptor, `scanf/cin` built on top of.
- `count = write(fd, buffer, nbytes)` writes to a file descriptor, `printf/cout` built on top of.
- `error = pipe(rgfd)` creates a pipe. `rgfd` is an array of two file descriptors. Read from `rgfd[0]`, write to `rgfd[1]`. 
/* pipe.C */

#include <iostream.h>
#include <unistd.h>

#define DATA "hello world"
#define BUFSIZE 1024

int rgfd[2];  // file descriptors of streams */

/* NO ERROR CHECKING, ILLUSTRATION ONLY!!!! */

main()
{
    char sbBuf[BUFSIZE];

    pipe(rgfd);
    if (fork()) { /* parent, read from pipe */
        close(rgfd[1]);  // close write end */
        read(rgfd[0], sbBuf, BUFSIZE);
        cout << "-->
" << sbBuf << '\n';
        close(rgfd[0]);
    }
    else { /* child, write data to pipe */
        close(rgfd[0]);  // close read end */
        write(rgfd[1], DATA, sizeof(DATA));
        close(rgfd[1]);
        exit(0);
    }
}

For the following, which is the parent and which is the child? (parent should read from pipe so “more” is the parent process). Last to complete.

% ls | more