Lab: Interprocess communication

17 February 2005

This lab will help to illustrate inter-process communication, and kernel support for *pipes,* or communication channels. To complete this exercise, you will probably want to have a few pages of scrap paper handy.

We will simulate the execution of several processes on a very simple Central Processing Unit (CPU). The CPU has four registers:

ACC	the 'accumulator'
PC	the program counter
BAS	the base register
LIM	the limit register

The accumulator is used to hold temporary results during computation. The program counter keeps track of which instruction we are executing in a given program. Finally, the base and limit registers are used for protected memory management, as discussed in class.

Apart from the CPU, we will need to represent a small chunk of memory. Draw the memory as an array M, with each index marked, like this:

M[0]	
M[1]	
M[2]	
M[3]	
M[4]	
M[5]	
M[6]	
M[7]	
M[8]	
M[9]	

Finally, we will need a *process control block* (PCB) for each process. This time, we are extending the PCB with an array of *I/O descriptors.* These are pointers to kernel resources that the process may use for communication. The PCB now looks like this:

Process ID	
ACC	
PC	
BAS	
LIM	
IO[0]	
IO[1]	
IO[2]	

We will need to reserve some amount of memory for the operating system itself, as buffers for the communication channels. Each buffer will have just two memory locations. After writing two pieces of data to the buffer, it becomes full.

The I/O descriptors of each process, then, point to the memory locations of these buffers in the OS memory. When a process wants to send data to the buffer, it specifies the I/O channel in the system call. The OS will then copy the value into the buffer. If the buffer is already full, then the process will have to *wait* until space becomes available. Therefore, *each* channel has a wait queue for the PCBs of processes that are waiting for space to open up in the buffer.

A similar thing happens when a process tries to read from a communication channel. If the buffer is empty, there is nothing to read. So the process will have to *wait* until some data become available in the buffer. When processes must wait to both send and receive, it is called *blocking I/O.*

Below are reminders of how the non-communication parts work, from the last lab. On the reverse side, there are four programs we will simulate.

Dispatch

1. Select the process at the front of the ready queue.
2. Copy the contents of its PCB into the corresponding CPU registers.
3. Let it run! You will know which instruction to execute next by looking at the value of the PC. □

Timer interrupt

When the timer interrupt occurs, do the following:

1. Finish whatever instruction you are currently executing.
2. Copy the values of all 4 registers into the PCB of the current process.
3. Move the PCB to the end of the ready queue.
4. Do the **Dispatch** procedure, above. □

Here are the four programs. A program that only sends data, but does not receive is called a *producer*. A program that receives but doesn't send is called a *consumer*. A program that receives and sends is called a *filter*.

In all of these programs, `IO[1]` will be used as an output channel, and `IO[0]` as input. But remember, each process has its own I/O descriptors, so the *output* channel of P1 can easily be hooked to the *input* channel of P2. And voilà, they can communicate!

Program: rolldice (a producer)

This program simulates rolling a 6-sided die *n* times. `IO[1]` is initialized with the address of the output mailbox, and `M[0]` is initialized with *n*.

```
1. if M[0] = 0 then PC ← 6
2. M[0] ← M[0] - 1
3. ACC ← random(1,6)
4. send(IO[1], ACC)
5. PC ← 1
6. send(IO[1], -1)
7. exit()
```

Program: sum (a consumer)

This program adds all the values received from the mailbox `IO[0]` and leaves the sum in `M[0]`.

```
1. M[0] ← 0
2. ACC ← receive(IO[0])
3. if ACC = -1 then PC ← 6
4. M[0] ← M[0] + ACC
5. PC ← 2
6. exit()
```

Program: odd (a filter)

A filter is both a producer and a consumer – it transforms input data to output. This program reads values from mailbox `IO[0]` and writes just the *odd* values to mailbox `IO[1]`.

```
1. ACC ← receive(IO[0])
2. if ACC is even then PC ← 1
3. send(IO[1], ACC)
2. if ACC != -1 then PC ← 1
4. exit()
```

Program: howmany (a consumer)

This program adds all the values received from the mailbox `IO[0]` and leaves the sum in `M[0]`.

```
1. M[0] ← 0
2. ACC ← receive(IO[0])
3. if ACC = -1 then PC ← 6
4. M[0] ← M[0] + 1
5. PC ← 2
6. exit()
```

System calls

- `random`(*i*,*j*) returns a random number between *i* and *j*, inclusive.
- `send`(*box*, *val*) enqueues the value into the mailbox. If the mailbox is full, it *blocks* until there is space. (The process must transition to a wait queue *for that mailbox*.) As soon as the value is sent, if another process was waiting to receive, it receives the value right away and transitions to *ready* state.
- `receive`(*box*) dequeues and returns a value from the mailbox. If the mailbox is empty, it *blocks* until there is a value available. As soon as the value is received, if another process is waiting to send, it now has the opportunity to send its value and move back to the *ready* state.

Experiments

Try this process combination: (`rolldice 10 | sum`). The pipe character (`|`) means you set up a fixed-size mailbox in kernel memory and initialize `IO[1]` of the left process and `IO[0]` of the right process to point to it. After you are confident with how it works, try this one: (`rolldice 12 | odd | howmany`).