Session 4: Interprocess Communication

Dr John Fawcett

What’s the problem?

We have done such a good job of preventing our processes from interfering with each other that there is no way for two processes to work together! E.g. a webserver process cannot talk to a database server process.

Without compromising security, we need to find a way for processes to talk to each other.

- Today: processes running on the same computer
- Tomorrow: processes running on different computers

Existing concepts

Common pattern:

When we want to add a new concept to an operating system, we make it look like something that programmers already know how to use.

Two mechanisms are used to present interprocess communication to user processes.

1. File handles
2. Virtual memory

File handles

When a program wants to access a file stored on the disk, how does that work?

1. fh = fopen("/path/to/my/file", "rw");
2. int x;
3. fread(fh, &x, sizeof(int));
4. fclose(fh);

What's happening here?

File handles

When a program wants to access a file stored on the disk, how does that work?

1. fh = fopen("/path/to/my/file", "rw");
2. int x;
3. fread(fh, &x, sizeof(int));
4. fclose(fh);

What's happening here?

1. fopen() is a function implemented by the operating system. The arguments are the path name of the file we want to access, and the type(s) of access we require: read and write in this example. The return value, stored into 'fh', is a *handle* allowing us to access this later.

File handles

When a program wants to access a file stored on the disk, how does that work?

1. fh = fopen("/path/to/my/file", "rw");
2. int x;
3. fread(fh, &x, sizeof(int));
4. fclose(fh);

What's happening here?

2. We declare a region of memory large enough to hold a single integer, and call it 'x'.

File handles

When a program wants to access a file stored on the disk, how does that work?

1. fh = fopen("/path/to/my/file", "rw");
2. int x;
3. fread(fh, &x, sizeof(int));
4. fclose(fh);

What's happening here?

3. Using the operating system's fread() function, we fill the bytes of memory for variable x by loading data from the file. We specify which file we want to read from using the handle provided by fopen() and stored in the variable called 'fh'.

File handles

When a program wants to access a file stored on the disk, how does that work?

1. fh = fopen("/path/to/my/file", "rw");
2. int x;
3. fread(fh, &x, sizeof(int));
4. fclose(fh);

What's happening here?

4. fclose() is also implemented by the operating system. It tells the operating system that we have finished using a file handle so the operating system can write any changes we might have made back to the disk.

Communicating between processes

One way to communicate between processes is for one of them to write into files in a particular directory on the hard disk, and for the other process to read from those files.

Pros:

\+ Receiving process does not have to be running at the same time as the sender: the disk provides *buffering*.

Cons:

- Slow.
- Sender does not know whether the message got through now, or will ever be read.
- Anyone with access to the disk can read the messages.

The abstraction of file handles

Processes never have access to the disk. fread(), fwrite() have *blocking* semantics.

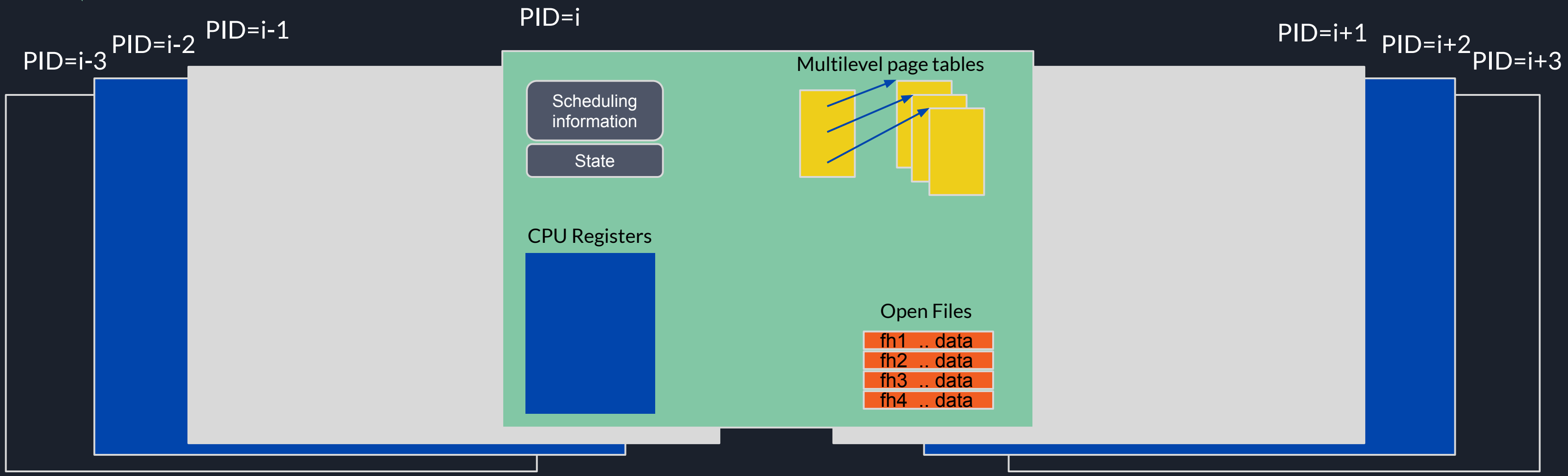
Permissions are checked when handles are opened (not when subsequently used).

A file handle is a short integer code that indexes the *process open file table*, in the process control block.

The process open file table holds the information required by the operating system:

- where we're up to in reading each open file (the *seek position*);
- mode of the handle (read-only, read-write, append-only, etc.);
- where the data is stored on disk.

Process open file table



Pipes

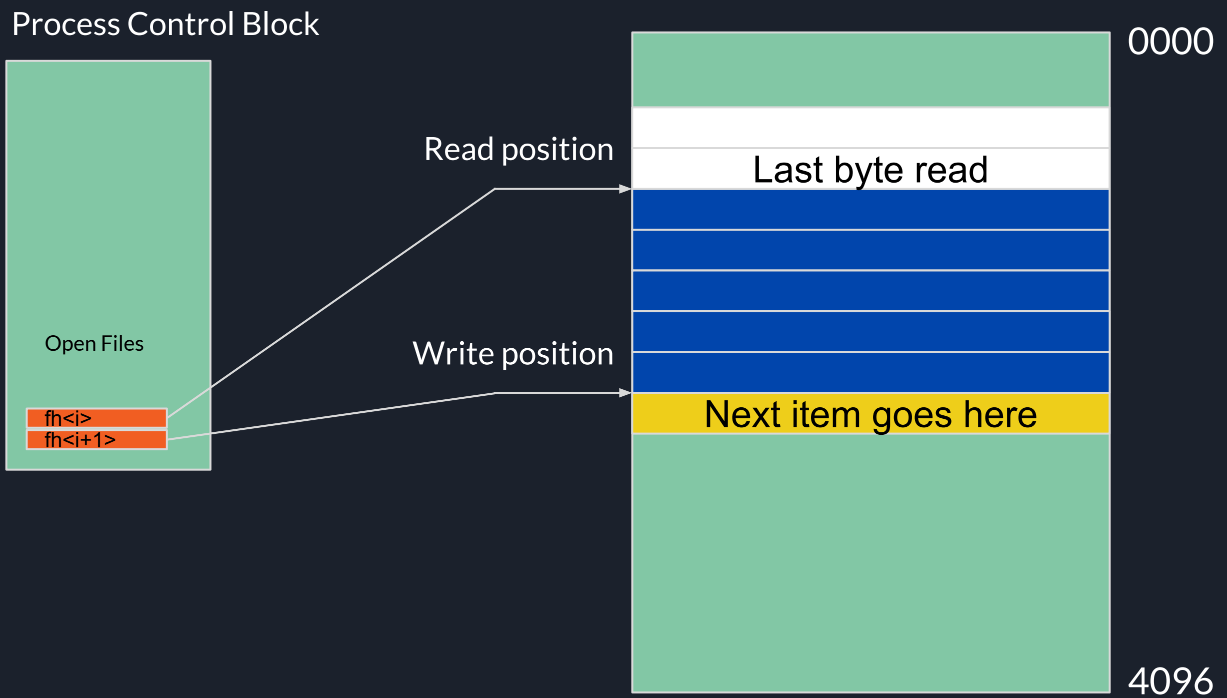
A pipe is a form of interprocess communication that looks to a process like two file handles: one read-only and one append-only.

Instead of storing data on the disk, the operating system stores the data in one page of physical memory.

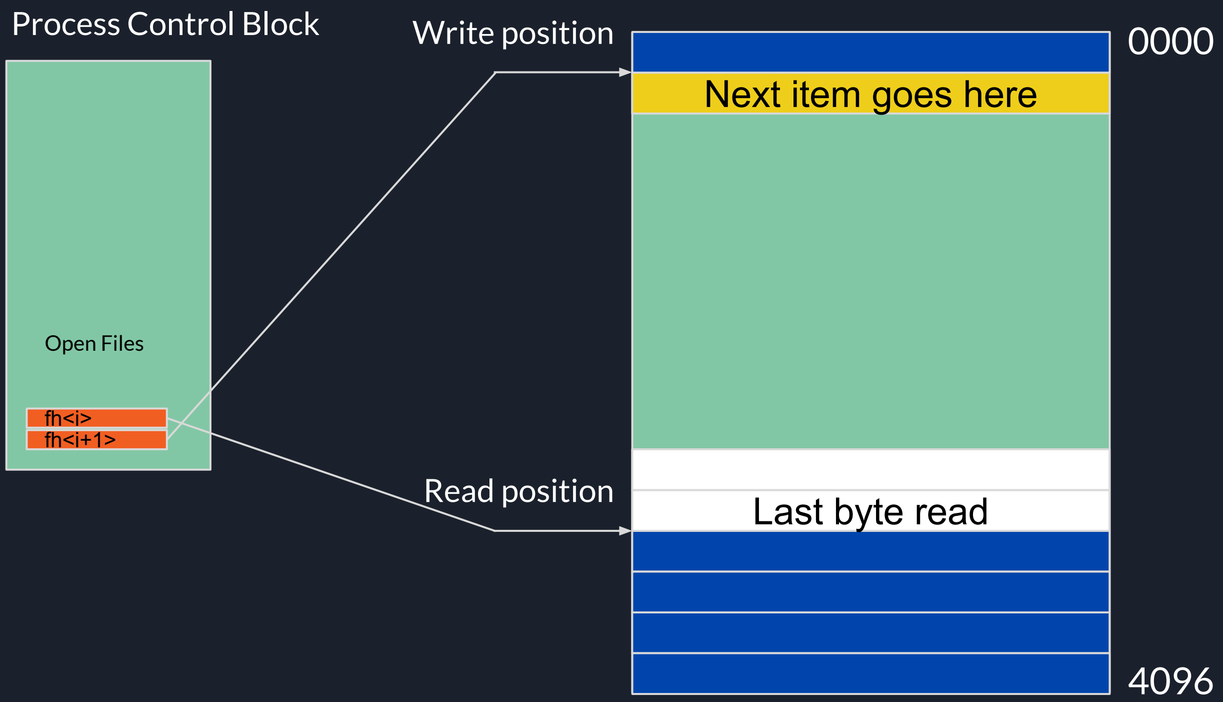
The memory is used (circularly) as a queue: write bytes at one end, read bytes from the other, and each byte can be read only once.

It’s also *blocking*: a process will be blocked if it tries to write when the queue is full, and if it tries to read when the queue is empty.

Pipes


Process Control Block
Open Files
fh<i>
fh<i+1>
Read position
Write position
Last byte read
Next item goes here
0000
4096

Pipes ("circular" queue)



Talking to yourself?

At the moment, both the read and write file handles are in the same process. Rather than allowing interprocess communication, so far, we only have a way for a process to talk to itself (which it could using its own memory – we don't need a pipe for that).

Remember how we create new processes: an existing process calls *fork()* to create a second process that is a copy of itself, including the read and write file handles for the pipe!

When a process wants to talk to another, it…

1. Creates a pipe
2. Calls fork()
3. The child process closes the write handle and the parent process closes the read handle (or vice-versa). For two-way communication, use two pipes!

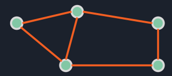
Serialising data

We can only send bytes so, when we want to send a structured message, we need to encode the message as bytes. This is called *serialisation*. Reversing that is *deserialisation*.

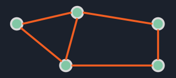
Example: to send a list of numbers from one process to another, we need to say how many numbers there are first, and then send the numbers.

serialise([10,6,-19,3]) ⇒ 4,10,6,-19,3

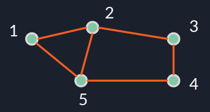
length then data

serialise() ⇒ ?

Serialising data

serialise() ⇒ ?

There are 5 vertices in this graph. The vertices are identical so we can number them in any way we like.



There are 5 vertices and the edges are [(1,2), (1,5), (2,5), (2,3), (5,4), (3,4)].

One encoding is 5, 6, 1,2, 1,5, 2,5, 2,3, 5,4, 3,4.

Notice that ‘6’ is the number of pairs of numbers. This is OK because the receiver will be expecting a list of edges, which are pairs.

Characteristics of pipes

Slow: we have to copy data (twice).

Complex / Slow: we have to serialise our data into bytes, then deserialise at the receiving end.

Slow: we have to use the operating system (and the interrupt mechanism) to send messages in each direction, which is a lot of overhead.

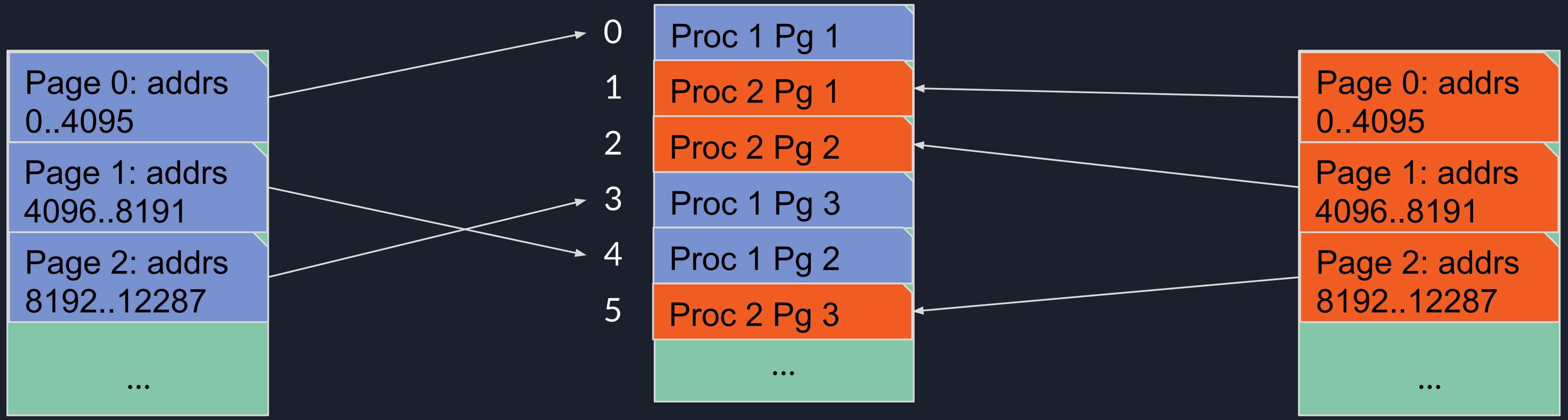
One-way, or two-way if you create two.

Very secure: private and can only be used between processes with a common ancestor.

Virtual memory

The second mechanism we could use is the virtual memory system.

Recall this diagram (from Session 2)...



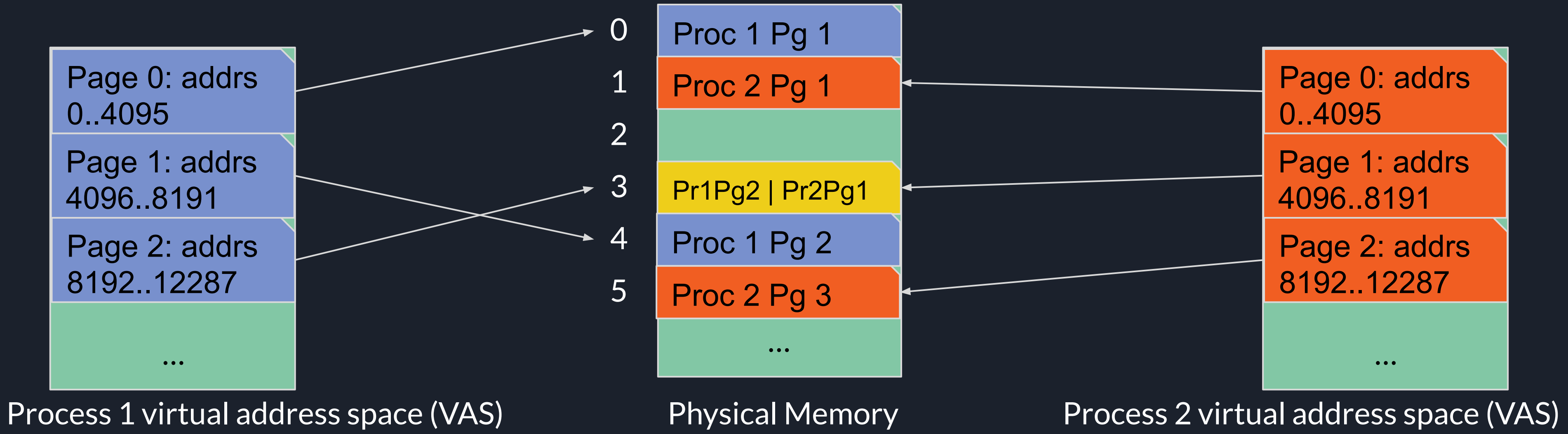
Process 1 virtual address space (VAS)

Physical Memory

Process 2 virtual address space (VAS)

Shared memory

A *shared page* is one physical page that is mapped into the virtual address space of two or more processes. The yellow page in frame 3 is mapped to addresses 8192..12287 in process 1 and to 4096..8191 in process 2. If process 1 writes to address 8192, process 2 can read that data from its virtual address 4096.



Characteristics of shared memory

Fast: we don’t have to copy data (twice).

Fast: we don’t need to serialise data structures to send them as bytes, then rebuild a copy of them. We are simply sharing the original data structure.

Fast: we only need to use the interrupt mechanism to set-up the shared page, not to send messages subsequently.

Simple: we do not need to ‘flatten’ data structures containing pointers. Provided the pointers are relative (+12 bytes), not absolute (at addr=6,000), the bytes will be interpreted in the same way by all sharing processes.

Two-way or one-way (just set-up the page table permissions), and private (secure).

Now it’s time for you to have a go

Exercise sheet!