Inter-Process Communication

Pipes

A *pipe* is an IPC mechanism provided by some OSs

Inter-Process Communication

Pipes

A *pipe* is an IPC mechanism provided by some OSs

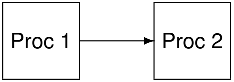
Conceptually, a pipe connects two processes together, taking output from one and feeding it as input to the other

Inter-Process Communication

Pipes

A *pipe* is an IPC mechanism provided by some OSs

Conceptually, a pipe connects two processes together, taking output from one and feeding it as input to the other

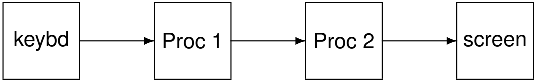


Inter-Process Communication

Pipes

A *pipe* is an IPC mechanism provided by some OSs

Conceptually, a pipe connects two processes together, taking output from one and feeding it as input to the other



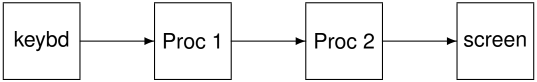
This might be part of a larger pipeline

Inter-Process Communication

Pipes

A *pipe* is an IPC mechanism provided by some OSs

Conceptually, a pipe connects two processes together, taking output from one and feeding it as input to the other



This might be part of a larger pipeline

And the pipes go via the kernel, not directly between processes

Inter-Process Communication

Pipes

Pipes have a fixed size: 4096 bytes is common

Inter-Process Communication

Pipes

Pipes have a fixed size: 4096 bytes is common

A writes to the pipe, B reads from the pipe and they do so independently of each other

Inter-Process Communication

Pipes

Pipes have a fixed size: 4096 bytes is common

A writes to the pipe, B reads from the pipe and they do so independently of each other

This is like the way we pass data via files

Inter-Process Communication

Pipes

Pipes have a fixed size: 4096 bytes is common

A writes to the pipe, B reads from the pipe and they do so independently of each other

This is like the way we pass data via files

But pipes also provide synchronisation

Inter-Process Communication

Pipes

A writes bytes into the pipe: if the pipe gets full, A is blocked by the OS until space is freed up by B reading some

Inter-Process Communication

Pipes

A writes bytes into the pipe: if the pipe gets full, A is blocked by the OS until space is freed up by B reading some

B reads bytes from the pipe: if the pipe gets empty, B is blocked by the OS until bytes are available by A writing some

Inter-Process Communication

Pipes

A writes bytes into the pipe: if the pipe gets full, A is blocked by the OS until space is freed up by B reading some

B reads bytes from the pipe: if the pipe gets empty, B is blocked by the OS until bytes are available by A writing some

Thus the scheduling of A and B can be affected

Inter-Process Communication

Pipes

A writes bytes into the pipe: if the pipe gets full, A is blocked by the OS until space is freed up by B reading some

B reads bytes from the pipe: if the pipe gets empty, B is blocked by the OS until bytes are available by A writing some

Thus the scheduling of A and B can be affected

Bytes are read out in the same order they were written in: FIFO

Inter-Process Communication

Pipes

A writes bytes into the pipe: if the pipe gets full, A is blocked by the OS until space is freed up by B reading some

B reads bytes from the pipe: if the pipe gets empty, B is blocked by the OS until bytes are available by A writing some

Thus the scheduling of A and B can be affected

Bytes are read out in the same order they were written in: FIFO

Note there are two kinds of communication here: (1) the data, and (2) synchronisation on production/consumption of the data

Inter-Process Communication

Pipes

A pipe is implemented as a buffer (chunk of memory) held by the kernel, not directly accessible by user processes

Inter-Process Communication

Pipes

A pipe is implemented as a buffer (chunk of memory) held by the kernel, not directly accessible by user processes

A write to or read from the pipe involves a syscall

Inter-Process Communication

Pipes

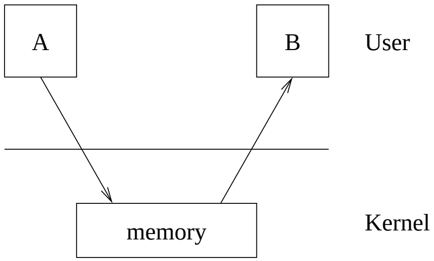
A pipe is implemented as a buffer (chunk of memory) held by the kernel, not directly accessible by user processes

A write to or read from the pipe involves a syscall

This is how the kernel can control blocking A and B, making sure A does not overfill the buffer and making sure B is not reading data that is not there

Inter-Process Communication

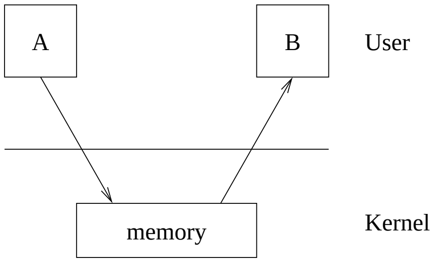
Pipes



Implementation of a Pipe

Inter-Process Communication

Pipes

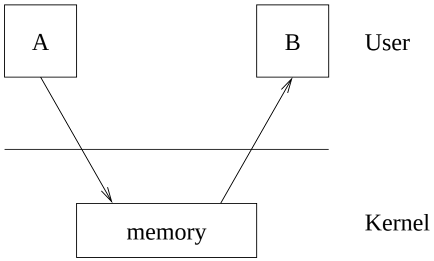


Implementation of a Pipe

If A wants to write to the pipe, it makes a system call: the kernel can check for space in the buffer and block A if necessary

Inter-Process Communication

Pipes



Implementation of a Pipe

If A wants to write to the pipe, it makes a system call: the kernel can check for space in the buffer and block A if necessary

Symmetrically for B reading from the pipe

Inter-Process Communication

Pipes

Pipes are supported well by Unix and are very easy to create and use when using a shell

Aside

A *shell* is just a program that waits for you to type something and then possibly creates some new processes according to what you typed: the *command line* interface

Popular with Unix derivatives, unpopular with Windows derivatives

Inter-Process Communication

Pipes

Pipes are supported well by Unix and are very easy to create and use when using a shell

```
% ps | sort
```

Inter-Process Communication

Pipes

Pipes are supported well by Unix and are very easy to create and use when using a shell

```
% ps | sort
```

The % is the shell prompt; `ps` is the “list processes” command; `sort` is a sorting program; the `|` is the notation for a pipe in this shell

Inter-Process Communication

Pipes

Pipes are supported well by Unix and are very easy to create and use when using a shell

```
% ps | sort
```

The % is the shell prompt; `ps` is the “list processes” command; `sort` is a sorting program; the `|` is the notation for a pipe in this shell

So this displays a sorted list of processes

Inter-Process Communication

Pipes

Pipes are also easy to create within programs: see the POSIX function `pipe`

Aside

POSIX is a *library* standard: it contains a list of standard functions that provide a simple and uniform front end to OS syscalls (amongst doing useful other things) and describes their expected behaviours, e.g., `open`, `close` and `sqrt`

Aside

POSIX is a *library* standard: it contains a list of standard functions that provide a simple and uniform front end to OS syscalls (amongst doing useful other things) and describes their expected behaviours, e.g., `open`, `close` and `sqrt`

This helps portability between OSs by hiding some OS specific details (e.g., details of syscalls)

Aside

POSIX is a *library* standard: it contains a list of standard functions that provide a simple and uniform front end to OS syscalls (amongst doing useful other things) and describes their expected behaviours, e.g., `open`, `close` and `sqrt`

This helps portability between OSs by hiding some OS specific details (e.g., details of syscalls)

Unix derivatives are usually mostly compliant, Windows less so

Aside

POSIX is a *library* standard: it contains a list of standard functions that provide a simple and uniform front end to OS syscalls (amongst doing useful other things) and describes their expected behaviours, e.g., `open`, `close` and `sqrt`

This helps portability between OSs by hiding some OS specific details (e.g., details of syscalls)

Unix derivatives are usually mostly compliant, Windows less so

Warning: remember some people regard such systems libraries as part of the OS

Aside

POSIX is a *library* standard: it contains a list of standard functions that provide a simple and uniform front end to OS syscalls (amongst doing useful other things) and describes their expected behaviours, e.g., `open`, `close` and `sqrt`

This helps portability between OSs by hiding some OS specific details (e.g., details of syscalls)

Unix derivatives are usually mostly compliant, Windows less so

Warning: remember some people regard such systems libraries as part of the OS

Even though they live and operate in user mode

Pipes

A typical sequence in a program is for a process to create a pipe then create a child process (i.e., ask the kernel to create a pipe then ask the kernel to make a new process)

Pipes

A typical sequence in a program is for a process to create a pipe then create a child process (i.e., ask the kernel to create a pipe then ask the kernel to make a new process)

(After a bit of technical fiddling) the pipe is now ready to use for IPC between parent and child

Inter-Process Communication

Pipes

Pipes are

Inter-Process Communication

Pipes

Pipes are

- simple and efficient

Inter-Process Communication

Pipes

Pipes are

- simple and efficient
- easy to use from programs and from a shell

Inter-Process Communication

Pipes

Pipes are

- simple and efficient
- easy to use from programs and from a shell
- a powerful way of combining processes and programs

Inter-Process Communication

Pipes

Pipes are

- simple and efficient
- easy to use from programs and from a shell
- a powerful way of combining processes and programs
- used a great deal

Inter-Process Communication

Pipes

But also

Inter-Process Communication

Pipes

But also

- are unidirectional

Inter-Process Communication

Pipes

But also

- are unidirectional
- technical detail: are only between *related* processes. Often one is the parent of the other

Inter-Process Communication

Pipes

But also

- are unidirectional
- technical detail: are only between *related* processes. Often one is the parent of the other
- can trivially create deadlocks if you use them carelessly (A creates a child process B with two pipes A$\rightarrow$B and B$\rightarrow$A. . .)

Inter-Process Communication

Pipes

Pipes are so useful there have been a couple of extensions:

Inter-Process Communication

Pipes

Pipes are so useful there have been a couple of extensions:

- Named Pipes: these can can be shared by unrelated processes (but have the naming problem that IPC using files have)

Inter-Process Communication

Pipes

Pipes are so useful there have been a couple of extensions:

- Named Pipes: these can can be shared by unrelated processes (but have the naming problem that IPC using files have)
- Sockets: pipes between processes on different machines. The basis of the Internet

Inter-Process Communication

Sockets

A socket allows bidirectional IPC between two processes (pipes are unidirectional for mostly historical reasons)

□

Inter-Process Communication

Sockets

A socket allows bidirectional IPC between two processes (pipes are unidirectional for mostly historical reasons)

The processes may be on the same or widely remote machines

□

Inter-Process Communication

Sockets

A socket allows bidirectional IPC between two processes (pipes are unidirectional for mostly historical reasons)

The processes may be on the same or widely remote machines

The technical issues behind implementing sockets are clearly much more complicated than basic pipes, but they present the same kind of FIFO, byte oriented, blocking channel

□

Inter-Process Communication

Sockets

A socket allows bidirectional IPC between two processes (pipes are unidirectional for mostly historical reasons)

The processes may be on the same or widely remote machines

The technical issues behind implementing sockets are clearly much more complicated than basic pipes, but they present the same kind of FIFO, byte oriented, blocking channel

We shall see some of those issues later in this Unit

□

Inter-Process Communication

Sockets

A socket allows bidirectional IPC between two processes (pipes are unidirectional for mostly historical reasons)

The processes may be on the same or widely remote machines

The technical issues behind implementing sockets are clearly much more complicated than basic pipes, but they present the same kind of FIFO, byte oriented, blocking channel

We shall see some of those issues later in this Unit

A lot of the modern world is built on top of sockets!

□