Process Protection

Next, we return briefly to resource protection

Process Protection

Next, we return briefly to resource protection

In particular, protection of resources between users

Process Protection

Userids are used everywhere

- Memory: a chunk of memory has a userid associated. This tells the kernel which processes are allowed to access it

Process Protection

Userids are used everywhere

- Memory: a chunk of memory has a userid associated. This tells the kernel which processes are allowed to access it
- Files: each file has a userid associated. This tells the kernel which processes are allowed to access it

Process Protection

Userids are used everywhere

- Memory: a chunk of memory has a userid associated. This tells the kernel which processes are allowed to access it
- Files: each file has a userid associated. This tells the kernel which processes are allowed to access it
- Similarly for other resources

Process Protection

Userids are used everywhere

- Memory: a chunk of memory has a userid associated. This tells the kernel which processes are allowed to access it
- Files: each file has a userid associated. This tells the kernel which processes are allowed to access it
- Similarly for other resources

We shall see more of this when we get to memory and files

Process Protection

Userids are used everywhere

- Memory: a chunk of memory has a userid associated. This tells the kernel which processes are allowed to access it
- Files: each file has a userid associated. This tells the kernel which processes are allowed to access it
- Similarly for other resources

We shall see more of this when we get to memory and files

Exercise. Find out the userid allocated to you on the Uni's `linux.bath.ac.uk` machine

Process Protection

A new process (usually) inherits the userid of its parent process

Process Protection

A new process (usually) inherits the userid of its parent process

Of course, this lead to another bootstrapping problem: how can a user get a process going in the first place?

Process Protection

A new process (usually) inherits the userid of its parent process

Of course, this lead to another bootstrapping problem: how can a user get a process going in the first place?

If there are no processes running with my userid, how can I ever get a process to be created?

Process Protection

A new process (usually) inherits the userid of its parent process

Of course, this lead to another bootstrapping problem: how can a user get a process going in the first place?

If there are no processes running with my userid, how can I ever get a process to be created?

So there is a distinguished user, variously called the *superuser* or *root* or *administrator*

Process Protection

A new process (usually) inherits the userid of its parent process

Of course, this lead to another bootstrapping problem: how can a user get a process going in the first place?

If there are no processes running with my userid, how can I ever get a process to be created?

So there is a distinguished user, variously called the *superuser* or *root* or *administrator*

This is mostly a **normal user**, but the OS allows it full access to other users' files, processes, etc.

Process Protection

A new process (usually) inherits the userid of its parent process

Of course, this lead to another bootstrapping problem: how can a user get a process going in the first place?

If there are no processes running with my userid, how can I ever get a process to be created?

So there is a distinguished user, variously called the *superuser* or *root* or *administrator*

This is mostly a **normal user**, but the OS allows it full access to other users' files, processes, etc.

In particular, root can suspend or kill any user's processes and read or modify their files

Process Protection

Don’t confuse the root user with kernel mode

Process Protection

Don’t confuse the root user with kernel mode

Root’s processes run in user mode, just like other users’ processes

Process Protection

Don't confuse the root user with kernel mode

Root's processes run in user mode, just like other users' processes

Hardware access is still mediated by the OS, but the **inter-user** protections are not enforced by the OS for root

Process Protection

Don't confuse the root user with kernel mode

Root's processes run in user mode, just like other users' processes

Hardware access is still mediated by the OS, but the **inter-user** protections are not enforced by the OS for root

In the OS there is the equivalent of

if uid_of_process == uid_of_resource or uid_of_process == uid_of_root then
    allow access
else
    disallow access

Process Protection

Critically, root can change the userid of its processes: by doing so it gives away its privileges, but thereby allows a normal user to have a process

Process Protection

Critically, root can change the userid of its processes: by doing so it gives away its privileges, but thereby allows a normal user to have a process

When a user logs in to a system a process, owned by root, starts up, changes its userid to the user, and then starts other processes as that user

Process Protection

Many resources are restricted by the OS so only the superuser can use them: this provides an extra level of protection to resources that are sensitive

Process Protection

Many resources are restricted by the OS so only the superuser can use them: this provides an extra level of protection to resources that are sensitive

For example, shutting down the computer. We can't allow any user process to turn off the computer, so this operation is restricted by the kernel to the root user

Process Protection

Many resources are restricted by the OS so only the superuser can use them: this provides an extra level of protection to resources that are sensitive

For example, shutting down the computer. We can't allow any user process to turn off the computer, so this operation is restricted by the kernel to the root user

Any shutdown program will need to have root ownership and this will be carefully policed by the system

Process Protection

Root is generally trusted by the kernel and allowed access to everyone's resources

Process Protection

Root is generally trusted by the kernel and allowed access to everyone's resources

So root-owned processes can completely trash everyone's programs and data on the machine if they want to

Process Protection

Root is generally trusted by the kernel and allowed access to everyone’s resources

So root-owned processes can completely trash everyone’s programs and data on the machine if they want to

This is why you should keep the use of the administrator account to a minimum

Process Protection

Root is generally trusted by the kernel and allowed access to everyone's resources

So root-owned processes can completely trash everyone's programs and data on the machine if they want to

This is why you should keep the use of the administrator account to a minimum

Doing everyday stuff as administrator is just asking for trouble, and is throwing away many of the protection mechanisms that OSs have developed to provide

Process Protection

This user-level protection is what prevents your processes from interfering with my processes: as we have different userids, the kernel knows to keep them separate

Process Protection

This user-level protection is what prevents your processes from interfering with my processes: as we have different userids, the kernel knows to keep them separate

In particular, if you download an application or Web page that contains a malicious worm or virus, properly working user protection will limit the damage that malware can do to just your files and your processes

Process Protection

This user-level protection is what prevents your processes from interfering with my processes: as we have different userids, the kernel knows to keep them separate

In particular, if you download an application or Web page that contains a malicious worm or virus, properly working user protection will limit the damage that malware can do to just your files and your processes

Not ideal, but better than letting the malware have full reign over the entire machine

Process Protection

A big part of the spread of malware in Windows OSs is the weakness of this kind of barrier to their spread: too many programs run as administrator and this can ultimately cause the entire system to be affected

Process Protection

A big part of the spread of malware in Windows OSs is the weakness of this kind of barrier to their spread: too many programs run as administrator and this can ultimately cause the entire system to be affected

Note that if your OS *requires* the use of a virus checker, this is a strong sign that your OS is not confident in its implementation of process protection

Process Protection

A big part of the spread of malware in Windows OSs is the weakness of this kind of barrier to their spread: too many programs run as administrator and this can ultimately cause the entire system to be affected

Note that if your OS *requires* the use of a virus checker, this is a strong sign that your OS is not confident in its implementation of process protection

Virus scanners address the *symptom*, not the *problem*

Process Protection

Summary: user protection is useful and helpful

Process Protection

Summary: user protection is useful and helpful

So don’t run things as root/administrator unless absolutely necessary

Process Protection

Summary: user protection is useful and helpful

So don’t run things as root/administrator unless absolutely necessary

And don’t confuse it with kernel/user mode

Inter-Process Communication

We now look at how processes communicate amongst themselves

Inter-Process Communication

We now look at how processes communicate amongst themselves

Many processes can be created, process, then exit without needing to refer to any other process

Inter-Process Communication

We now look at how processes communicate amongst themselves

Many processes can be created, process, then exit without needing to refer to any other process

But there are many processes that need to send data to, or receive data from other running processes

Inter-Process Communication

We now look at how processes communicate amongst themselves

Many processes can be created, process, then exit without needing to refer to any other process

But there are many processes that need to send data to, or receive data from other running processes

For example, a new program starting might wish to tell the process managing the display that it wishes to pop up a window on the display

Inter-Process Communication

We now look at how processes communicate amongst themselves

Many processes can be created, process, then exit without needing to refer to any other process

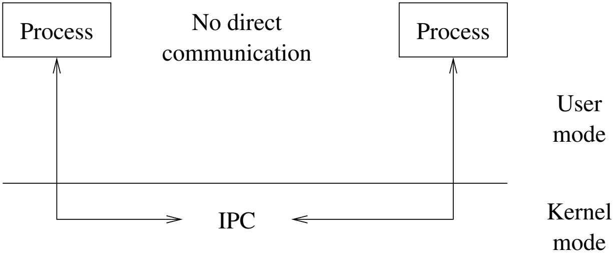
But there are many processes that need to send data to, or receive data from other running processes

For example, a new program starting might wish to tell the process managing the display that it wishes to pop up a window on the display

Or one process has to wait for another to finish some action (e.g., pop up a window) before it can progress itself: this is *synchronisation*

Inter-Process Communication

Inter-Process Communication (IPC) can be achieved in many different ways, but all must be, at base, supported by the OS; recall that by default the kernel tries to stop one process interfering with another



Inter-Process Communication

IPC contradicts this non-interference, and so must be treated very carefully by the kernel

There must be rules and restrictions, or else one process could just blast another process with data, preventing it from doing any useful work

Inter-Process Communication

We shall be looking at

- Files
- Pipes
- Shared memory

as a sample of IPC mechanisms

Inter-Process Communication

Files

A simple way for two processes to communicate is using an existing resource, namely files

Inter-Process Communication

Files

A simple way for two processes to communicate is using an existing resource, namely files

On the face of it this is just

Inter-Process Communication

Files

A simple way for two processes to communicate is using an existing resource, namely files

On the face of it this is just

- Process A wishes to send some data to process B

Inter-Process Communication

Files

A simple way for two processes to communicate is using an existing resource, namely files

On the face of it this is just

- Process A wishes to send some data to process B
- A writes it to a file

Inter-Process Communication

Files

A simple way for two processes to communicate is using an existing resource, namely files

On the face of it this is just

- Process A wishes to send some data to process B
- A writes it to a file
- B reads it

Inter-Process Communication

Files

A simple way for two processes to communicate is using an existing resource, namely files

On the face of it this is just

- Process A wishes to send some data to process B
- A writes it to a file
- B reads it

This seems easy

Inter-Process Communication

Files

A simple way for two processes to communicate is using an existing resource, namely files

On the face of it this is just

- Process A wishes to send some data to process B
- A writes it to a file
- B reads it

This seems easy

But it's much harder than this, of course

Inter-Process Communication

Files

- Which file to use? A and B need to agree on a filename to use, but this is not so easy. They can use a single “well-known” file, but this is problematic if many processes are all writing to the same file simultaneously. For example, C wants to communicate with D at the same time via the same file

Inter-Process Communication

Files

- Which file to use? A and B need to agree on a filename to use, but this is not so easy. They can use a single “well-known” file, but this is problematic if many processes are all writing to the same file simultaneously. For example, C wants to communicate with D at the same time via the same file
- There could be a separate file for each pair of processes, but to agree on a file name A and B must have previously communicated…

Inter-Process Communication

Files

- Which file to use? A and B need to agree on a filename to use, but this is not so easy. They can use a single "well-known" file, but this is problematic if many processes are all writing to the same file simultaneously. For example, C wants to communicate with D at the same time via the same file
- There could be a separate file for each pair of processes, but to agree on a file name A and B must have previously communicated…
- How does B know when data has arrived? B might have to repeatedly poll the file until the data arrives. This doesn't scale well to large numbers of files or processes

Inter-Process Communication

Files

- The file protections must be set properly (recall userids) to allow only the authorised processes to read/write to them

Inter-Process Communication

Files

- The file protections must be set properly (recall userids) to allow only the authorised processes to read/write to them
- Files are quite slow relative to the mechanisms we are going to see later

Inter-Process Communication

Files

- The file protections must be set properly (recall userids) to allow only the authorised processes to read/write to them
- Files are quite slow relative to the mechanisms we are going to see later

In general, files are not used for IPC

Inter-Process Communication

Files

- The file protections must be set properly (recall userids) to allow only the authorised processes to read/write to them
- Files are quite slow relative to the mechanisms we are going to see later

In general, files are not used for IPC

But they should be considered as a choice when *huge* amounts of data need to be transferred

Inter-Process Communication

Files

- The file protections must be set properly (recall userids) to allow only the authorised processes to read/write to them
- Files are quite slow relative to the mechanisms we are going to see later

In general, files are not used for IPC

But they should be considered as a choice when *huge* amounts of data need to be transferred

Exercise Read about the mechanism of choice to transfer the data describing the first ever image of a black hole (April 2019)

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

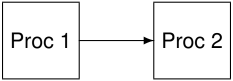
Conceptually, a pipe connects two processes together, taking output from one and feeding it as input to the other

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

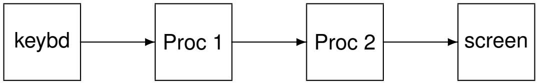


Inter-Process Communication

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



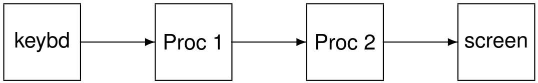
This might be part of a larger pipeline

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

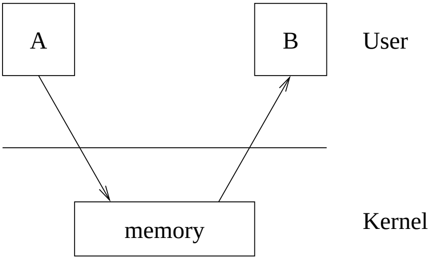
A pipe is implemented as a buffer (chunk of memory) held by the kernel, not directly accessible by user processes

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

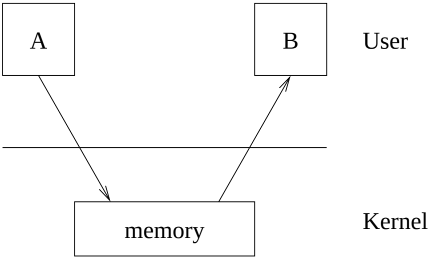
Pipes



Implementation of a Pipe

Inter-Process Communication

Pipes

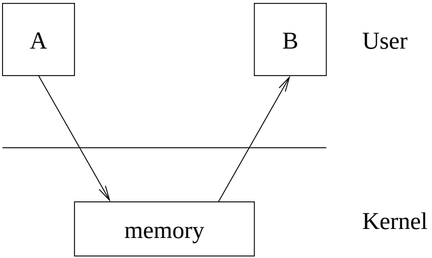


Implementation of a Pipe

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

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: it provides a *command line* interface (CLI)

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: it provides a *command line* interface (CLI)

Popular with Unix derivatives, unpopular with Windows derivatives

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

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

```
% ps | sort
```

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

Pipes are

- simple and efficient

Inter-Process Communication

Pipes

Pipes are

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

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

But also

But also

- are unidirectional

But also

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

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...)

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

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

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

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