# Inter-Process Communication

## Shared Memory

Next: shared memory

# Inter-Process Communication

## Shared Memory

Next: shared memory

In early computers, all memory was shared between processes: one process could easily write to the memory allocated to another process

# Inter-Process Communication

## Shared Memory

Next: shared memory

In early computers, all memory was shared between processes: one process could easily write to the memory allocated to another process

This is generally a bad idea, so is now prevented by the kernel (recall MMUs and read/write flags)

# Inter-Process Communication

## Shared Memory

Next: shared memory

In early computers, all memory was shared between processes: one process could easily write to the memory allocated to another process

This is generally a bad idea, so is now prevented by the kernel (recall MMUs and read/write flags)

On the other hand, access to memory is very fast, so we might want to use it for IPC

# Inter-Process Communication

## Shared Memory

Next: shared memory

In early computers, all memory was shared between processes: one process could easily write to the memory allocated to another process

This is generally a bad idea, so is now prevented by the kernel (recall MMUs and read/write flags)

On the other hand, access to memory is very fast, so we might want to use it for IPC

Just like using files: A writes to memory, B reads from it

# Inter-Process Communication

## Shared Memory

Next: shared memory

In early computers, all memory was shared between processes: one process could easily write to the memory allocated to another process

This is generally a bad idea, so is now prevented by the kernel (recall MMUs and read/write flags)

On the other hand, access to memory is very fast, so we might want to use it for IPC

Just like using files: A writes to memory, B reads from it

Again, this goes against the original design of an OS, so must be carefully set up and controlled

# Inter-Process Communication

## Shared Memory

And, also just like files we have the issues of

# Inter-Process Communication

## Shared Memory

And, also just like files we have the issues of

- Which area of memory to use? A well-known area, or per-process areas?

# Inter-Process Communication

## Shared Memory

And, also just like files we have the issues of

- Which area of memory to use? A well-known area, or per-process areas?
- How does B know when data has arrived? Memory is “always there” unlike files which can be created and removed; so when looking at memory it can be hard to know if you are reading the data you want or some previous junk that happened to be lying around

# Inter-Process Communication

## Shared Memory

And, also just like files we have the issues of

- Which area of memory to use? A well-known area, or per-process areas?
- How does B know when data has arrived? Memory is “always there” unlike files which can be created and removed; so when looking at memory it can be hard to know if you are reading the data you want or some previous junk that happened to be lying around
- So A might write a special value to a specific memory location to flag that the data is complete; but again B must poll this location to see when this is done

# Inter-Process Communication

## Shared Memory

And, also just like files we have the issues of

- Which area of memory to use? A well-known area, or per-process areas?
- How does B know when data has arrived? Memory is “always there” unlike files which can be created and removed; so when looking at memory it can be hard to know if you are reading the data you want or some previous junk that happened to be lying around
- So A might write a special value to a specific memory location to flag that the data is complete; but again B must poll this location to see when this is done
- The memory protections must be set properly to allow only the authorised processes to read or write it

# Inter-Process Communication

## Shared Memory

The speed of shared memory means that it is very good for IPC, as long as it is supported by further mechanisms like signals or semaphores to flag when data is ready

# Inter-Process Communication

## Shared Memory

The speed of shared memory means that it is very good for IPC, as long as it is supported by further mechanisms like signals or semaphores to flag when data is ready

More on shared memory when we get to memory management

The speed of shared memory means that it is very good for IPC, as long as it is supported by further mechanisms like signals or semaphores to flag when data is ready

More on shared memory when we get to memory management

**Exercise** Compare shared memory and pipes

# Inter-Process Communication

## And Others

There are several other IPC mechanisms in use

# Inter-Process Communication

## And Others

There are several other IPC mechanisms in use

Including

# Inter-Process Communication

## And Others

There are several other IPC mechanisms in use

Including

- signals: like interrupts, but at the software level

# Inter-Process Communication

## And Others

There are several other IPC mechanisms in use

Including

- signals: like interrupts, but at the software level
- semaphores: a way for processes to synchronise (e.g., have one wait for another)

# Inter-Process Communication

## And Others

There are several other IPC mechanisms in use

Including

- signals: like interrupts, but at the software level
- semaphores: a way for processes to synchronise (e.g., have one wait for another)
- software buses: a software-level messaging system, extensively used in GUIs to transfer information between windows, e.g., cut-and-paste

# Inter-Process Communication

## And Others

There are several other IPC mechanisms in use

Including

- signals: like interrupts, but at the software level
- semaphores: a way for processes to synchronise (e.g., have one wait for another)
- software buses: a software-level messaging system, extensively used in GUIs to transfer information between windows, e.g., cut-and-paste

**Exercise** Read about these

# Inter-Process Communication

So, which IPC mechanism to choose?

# Inter-Process Communication

So, which IPC mechanism to choose?

As always, it depends on the application

# Inter-Process Communication

So, which IPC mechanism to choose?

As always, it depends on the application

The best way to choose is to have lots of experience of using them

# Inter-Process Communication

So, which IPC mechanism to choose?

As always, it depends on the application

The best way to choose is to have lots of experience of using them

- The level your program is at: low or high?

# Inter-Process Communication

So, which IPC mechanism to choose?

As always, it depends on the application

The best way to choose is to have lots of experience of using them

- The level your program is at: low or high?
- The amount of data to be communicated: just a bit or a huge datafile?

# Inter-Process Communication

So, which IPC mechanism to choose?

As always, it depends on the application

The best way to choose is to have lots of experience of using them

- The level your program is at: low or high?
- The amount of data to be communicated: just a bit or a huge datafile?
- What is available?

# Inter-Process Communication

So, which IPC mechanism to choose?

As always, it depends on the application

The best way to choose is to have lots of experience of using them

- The level your program is at: low or high?
- The amount of data to be communicated: just a bit or a huge datafile?
- What is available?
- What your boss tells you to use

# Inter-Process Communication

So, which IPC mechanism to choose?

As always, it depends on the application

The best way to choose is to have lots of experience of using them

- The level your program is at: low or high?
- The amount of data to be communicated: just a bit or a huge datafile?
- What is available?
- What your boss tells you to use
- and so on

# Memory

We now turn to the next major topic: memory management

# Memory

We now turn to the next major topic: memory management

In the earliest computers the purpose of memory management was to share out a very limited resource, but it was soon found that inter-process *protection* was vital

# Memory

We now turn to the next major topic: memory management

In the earliest computers the purpose of memory management was to share out a very limited resource, but it was soon found that inter-process *protection* was vital

Both needs are still true, particularly the limited aspect: you might have 16GB in your PC, but it's not enough!

# Memory

We now turn to the next major topic: memory management

In the earliest computers the purpose of memory management was to share out a very limited resource, but it was soon found that inter-process *protection* was vital

Both needs are still true, particularly the limited aspect: you might have 16GB in your PC, but it's not enough!

Gates' Law: programs double in size every 18 months

# Memory

We now turn to the next major topic: memory management

In the earliest computers the purpose of memory management was to share out a very limited resource, but it was soon found that inter-process *protection* was vital

Both needs are still true, particularly the limited aspect: you might have 16GB in your PC, but it’s not enough!

Gates’ Law: programs double in size every 18 months

(Really Wirth’s Law: Software is decelerating faster than hardware is accelerating)

# Memory

## Physical Memory

We first consider how processes (code and data) should be laid out in memory

We first consider how processes (code and data) should be laid out in memory

This is called *physical* memory layout to distinguish it from *virtual* memory, which comes later

# Memory

## Physical Memory

Memory in a process might be allocated or freed at several points

# Memory

## Physical Memory

Memory in a process might be allocated or freed at several points

- Allocation only at process initialisation. Called *static* allocation. Featured in the earliest OSs

# Memory

## Physical Memory

Memory in a process might be allocated or freed at several points

- Allocation only at process initialisation. Called *static* allocation. Featured in the earliest OSs
- Allocation while the process is running. Called *dynamic* allocation. Early systems did not support this and you had to know in advance how much memory your process would need at initialisation

# Memory

## Physical Memory

Memory in a process might be allocated or freed at several points

- Allocation only at process initialisation. Called *static* allocation. Featured in the earliest OSs
- Allocation while the process is running. Called *dynamic* allocation. Early systems did not support this and you had to know in advance how much memory your process would need at initialisation
- Freeing while the process is running

# Memory

## Physical Memory

Memory in a process might be allocated or freed at several points

- Allocation only at process initialisation. Called *static* allocation. Featured in the earliest OSs
- Allocation while the process is running. Called *dynamic* allocation. Early systems did not support this and you had to know in advance how much memory your process would need at initialisation
- Freeing while the process is running
- Freeing at process end

But also the kernel needs memory:

# Memory

## Physical Memory

But also the kernel needs memory:

- Allocation and freeing within the kernel. The kernel has to be dynamic otherwise it would be very difficult to get started, e.g., creating processor control blocks

# Memory

## Physical Memory

Early operating systems were not dynamic

# Memory

## Physical Memory

Early operating systems were not dynamic

So they could only run a fixed number of processes

# Memory

## Physical Memory

Early operating systems were not dynamic

So they could only run a fixed number of processes

And the processes were of a fixed size

# Memory

## Physical Memory

Early operating systems were not dynamic

So they could only run a fixed number of processes

And the processes were of a fixed size

Reflecting this, early computer languages did not support dynamic allocation, e.g., FORTRAN, every array must be of a fixed size, specified in the source code

# Memory

## Physical Memory

Early operating systems were not dynamic

So they could only run a fixed number of processes

And the processes were of a fixed size

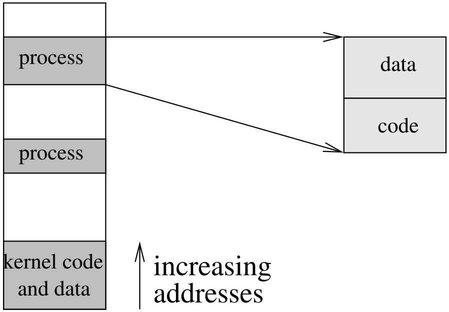
Reflecting this, early computer languages did not support dynamic allocation, e.g., FORTRAN, every array must be of a fixed size, specified in the source code

Dynamic allocation for both kernel and the processes was soon introduced in OSs, but computer languages took a while to catch up with the new facility

# Memory

## Physical Memory

Physical memory in an early computers looked something like this:




Memory Layout

# Memory

## Physical Memory

Remember the kernel itself needs code and data space

# Memory

## Physical Memory

Remember the kernel itself needs code and data space

A gap above the kernel area allows for dynamic allocation of memory to itself

# Memory

## Physical Memory

Remember the kernel itself needs code and data space

A gap above the kernel area allows for dynamic allocation of memory to itself

But the earliest systems had no dynamic behaviour at all, both OS and programs were completely static

# Memory

## Physical Memory

Remember the kernel itself needs code and data space

A gap above the kernel area allows for dynamic allocation of memory to itself

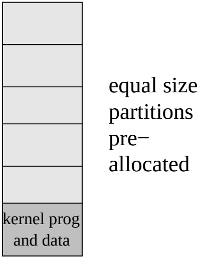
But the earliest systems had no dynamic behaviour at all, both OS and programs were completely static

Again, some early languages (FORTRAN, again) did not have a stack, and thus no recursion

## Partitioning

## Partitioning

The earliest and simplest memory layout is a static system called *partitioning*, where areas are allocated at boot time

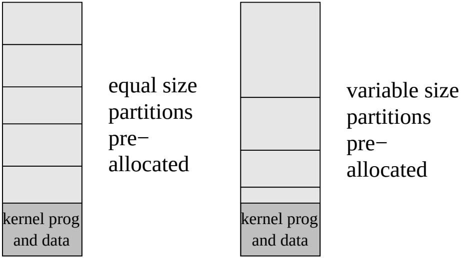


# Memory

## Physical Memory

### Partitioning

The earliest and simplest memory layout is a static system called *partitioning*, where areas are allocated at boot time




A process is loaded into the smallest free partition it will fit into

# Memory

## Physical Memory

If you don’t have dynamic allocation even in the kernel (e.g., for allocating new PCBs), then having fixed partitions is ideal

Equal size is easy to implement, but usually causes wasted space when a process does not fill its allocation

# Memory

## Physical Memory

If you don't have dynamic allocation even in the kernel (e.g., for allocating new PCBs), then having fixed partitions is ideal

Equal size is easy to implement, but usually causes wasted space when a process does not fill its allocation

And it can't cope with larger processes—you would have to reboot the computer with re-configured partitions

# Memory

## Physical Memory

If you don't have dynamic allocation even in the kernel (e.g., for allocating new PCBs), then having fixed partitions is ideal

Equal size is easy to implement, but usually causes wasted space when a process does not fill its allocation

And it can't cope with larger processes—you would have to reboot the computer with re-configured partitions

Variable size is not much harder to implement, but its efficiency depends heavily on the choice of partition sizes as ideally they should match the expected process sizes

# Memory

## Physical Memory

Partitioning is a good arrangement if you only run a fixed set of applications that you know in advance, e.g., a stock manager plus a payroll system plus a employees record system

# Memory

## Physical Memory

Partitioning is a good arrangement if you only run a fixed set of applications that you know in advance, e.g., a stock manager plus a payroll system plus a employees record system

IBM's OS/360 (mid 1960s) had three partitions: one for spooling punched cards to disk; one for spooling disk to printers; and one to run jobs

## Overlays

**Overlays**

In early systems, if a process was too big to fit in the memory allocated, the programmer could use *overlays*

**Overlays**

In early systems, if a process was too big to fit in the memory allocated, the programmer could use *overlays*

This is where only *part* of the process code is loaded into memory at once: only partly *resident*

**Overlays**

In early systems, if a process was too big to fit in the memory allocated, the programmer could use *overlays*

This is where only *part* of the process code is loaded into memory at once: only partly *resident*

If a non-resident part of the process is needed, the programmer must know this and include code to load the needed part of the process into memory, overwriting a part of the process they do not need at the moment

**Overlays**

In early systems, if a process was too big to fit in the memory allocated, the programmer could use *overlays*

This is where only *part* of the process code is loaded into memory at once: only partly *resident*

If a non-resident part of the process is needed, the programmer must know this and include code to load the needed part of the process into memory, overwriting a part of the process they do not need at the moment

If that part of the process is needed again later, the programmer has to reload the code

# Memory

## Physical Memory

This works, at the cost of some speed of execution, but only if you are an excellent programmer who can keep track on what parts of code are loaded at any particular time

# Memory

## Physical Memory

This works, at the cost of some speed of execution, but only if you are an excellent programmer who can keep track on what parts of code are loaded at any particular time

A similar trick works with data: but with newly generated data you have to save it somewhere (e.g., disk) first, before overwriting it, so that it can be loaded back in later, when needed

# Memory

## Physical Memory

This works, at the cost of some speed of execution, but only if you are an excellent programmer who can keep track on what parts of code are loaded at any particular time

A similar trick works with data: but with newly generated data you have to save it somewhere (e.g., disk) first, before overwriting it, so that it can be loaded back in later, when needed

This trick of swapping memory back and forth to the disk gets a big boost later

# Memory

## Physical Memory

We need to fit a process into a single contiguous chunk of memory as we can't spread it amongst several areas since

# Memory

## Physical Memory

We need to fit a process into a single contiguous chunk of memory as we can’t spread it amongst several areas since

- it will be very complicated for the OS to keep track of which areas of memory are allocated to which process

# Memory

## Physical Memory

We need to fit a process into a single contiguous chunk of memory as we can't spread it amongst several areas since

- it will be very complicated for the OS to keep track of which areas of memory are allocated to which process
- more importantly, you can't split code up in this way, having one instruction in one place and the next instruction somewhere else entirely

# Memory

## Physical Memory

We need to fit a process into a single contiguous chunk of memory as we can't spread it amongst several areas since

- it will be very complicated for the OS to keep track of which areas of memory are allocated to which process
- more importantly, you can't split code up in this way, having one instruction in one place and the next instruction somewhere else entirely
- similarly for data: we will have to keep track of what data is where

# Memory

## Physical Memory

We need to fit a process into a single contiguous chunk of memory as we can’t spread it amongst several areas since

- it will be very complicated for the OS to keep track of which areas of memory are allocated to which process
- more importantly, you can’t split code up in this way, having one instruction in one place and the next instruction somewhere else entirely
- similarly for data: we will have to keep track of what data is where

(But when we come to virtual memory later we shall see that exactly this *is* possible with modern hardware!)

# Memory

## Language Support for Dynamic Allocation

These days dynamic allocation is common in programming languages

# Memory

## Language Support for Dynamic Allocation

These days dynamic allocation is common in programming languages

- Implicit memory management, e.g., Java. Where the language controls the creation and deletion of objects

```
bigobject x; // memory is allocated for x
x = foo();   // that memory is now inaccessible
```

# Memory

## Language Support for Dynamic Allocation

These days dynamic allocation is common in programming languages

- Implicit memory management, e.g., Java. Where the language controls the creation and deletion of objects

  ```
  bigobject x; // memory is allocated for x
  x = foo();   // that memory is now inaccessible
  ```

- Explicit memory management, e.g., C. Where the programmer controls the creation and deletion of objects (`malloc` and `free`)

# Memory

## Language Support for Dynamic Allocation

These days dynamic allocation is common in programming languages

- Implicit memory management, e.g., Java. Where the language controls the creation and deletion of objects

  ```
  bigobject x; // memory is allocated for x
  x = foo();   // that memory is now inaccessible
  ```

- Explicit memory management, e.g., C. Where the programmer controls the creation and deletion of objects (`malloc` and `free`)

Amongst several other approaches

# Memory

## Language Support for Dynamic Allocation

These days dynamic allocation is common in programming languages

- Implicit memory management, e.g., Java. Where the language controls the creation and deletion of objects

```
bigobject x; // memory is allocated for x
x = foo();   // that memory is now inaccessible
```

- Explicit memory management, e.g., C. Where the programmer controls the creation and deletion of objects (`malloc` and `free`)

Amongst several other approaches

So an OS must be able to support this

## Dynamic Partitioning

**Dynamic Partitioning**

We need to to be dynamic: the first step is being able to create and allocate a partition of the appropriate size as needed

**Dynamic Partitioning**

We need to to be dynamic: the first step is being able to create and allocate a partition of the appropriate size as needed

A lot more complicated to implement, but this allows the process (i.e., the job submission) to say how big a partition it needs and the OS allocates just that

# Memory

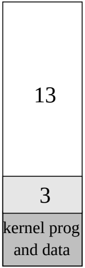
## Physical Memory

We can allocate sequentially, moving up memory

# Memory

## Physical Memory

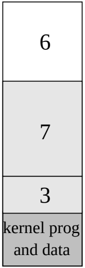
We can allocate sequentially, moving up memory



# Memory

## Physical Memory

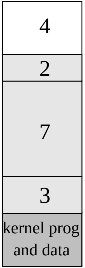
We can allocate sequentially, moving up memory



# Memory

## Physical Memory

We can allocate sequentially, moving up memory



# Memory

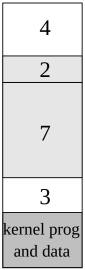
## Physical Memory

The problem is when a process ends and we get the memory back: it creates holes

# Memory

## Physical Memory

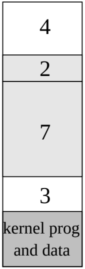
The problem is when a process ends and we get the memory back: it creates holes



# Memory

## Physical Memory

The problem is when a process ends and we get the memory back: it creates holes




We have space enough to run a process of size 5, but nowhere to put it

This is a general problem, called *fragmentation* and is very difficult to solve effectively

# Memory

## Physical Memory

This is a general problem, called *fragmentation* and is very difficult to solve effectively

The more processes come and go, the worse the fragmentation gets

# Memory

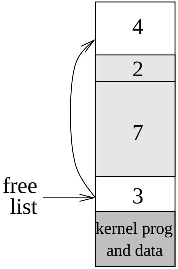
## Physical Memory

We need to keep a list of free blocks so we can track free space: a *freelist*

# Memory

## Physical Memory

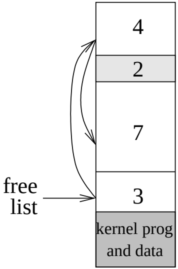
We need to keep a list of free blocks so we can track free space: a *freelist*



# Memory

## Physical Memory

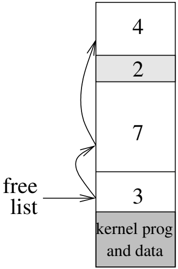
When a block is freed, put it in the freelist. It helps to keep the freelist sorted in address order:



# Memory

## Physical Memory

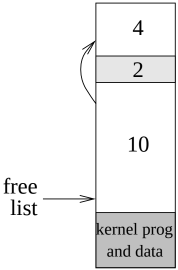
When a block is freed, put it in the freelist. It helps to keep the freelist sorted in address order:



# Memory

## Physical Memory

Slightly more clever is to *coalesce* physically adjacent blocks



# Memory

## Physical Memory

When we want some space, we search the freelist

# Memory

## Physical Memory

When we want some space, we search the freelist

We don’t want to waste space, so after choosing a big enough block we slice off the chunk we need and return the unused part to the freelist

# Memory

## Physical Memory

When we want some space, we search the freelist

We don't want to waste space, so after choosing a big enough block we slice off the chunk we need and return the unused part to the freelist

But there might be several blocks on the freelist that we could use: which one to choose?

# Memory

## Physical Memory

When we want some space, we search the freelist

We don’t want to waste space, so after choosing a big enough block we slice off the chunk we need and return the unused part to the freelist

But there might be several blocks on the freelist that we could use: which one to choose?

Strategies for choosing blocks include:

# Memory

## Physical Memory

When we want some space, we search the freelist

We don’t want to waste space, so after choosing a big enough block we slice off the chunk we need and return the unused part to the freelist

But there might be several blocks on the freelist that we could use: which one to choose?

Strategies for choosing blocks include:

- Best Fit. Find the *smallest* available big enough hole. Slow as we always have to search the entire freelist and results in lots of small fragments that are effectively useless as they are too small to be allocated

# Memory

## Physical Memory

- First Fit. Use the *first* available big enough hole. Initially faster than Best Fit and tends to leave larger and more useful fragments. But fragments tend to be created near the front of the freelist, so we have to search further and further each time

# Memory

## Physical Memory

- First Fit. Use the *first* available big enough hole. Initially faster than Best Fit and tends to leave larger and more useful fragments. But fragments tend to be created near the front of the freelist, so we have to search further and further each time
- Worst Fit. Find the *biggest* available big enough hole. Strangely this works out better than you think. Slicing chunks off bigger blocks tends to leave larger fragments that are more likely to be useful. Marginally faster than Best Fit as we have larger and therefore fewer blocks in the freelist to search through

# Memory

## Physical Memory

- Next Fit. Continue looking from where we last allocated and take the next available big enough hole. Fast, and improves on First Fit by spreading small fragments across memory

# Memory

## Physical Memory

- Next Fit. Continue looking from where we last allocated and take the next available big enough hole. Fast, and improves on First Fit by spreading small fragments across memory
- And many others

# Memory

## Physical Memory

- Next Fit. Continue looking from where we last allocated and take the next available big enough hole. Fast, and improves on First Fit by spreading small fragments across memory
- And many others

There are plenty of other memory management systems (e.g., Buddy memory allocation; Slab allocation; etc.) targeting the fragmentation problem

Note that fragments are created in two ways:

# Memory

## Physical Memory

Note that fragments are created in two ways:

- when carved off a bigger block in an allocation

# Memory

## Physical Memory

Note that fragments are created in two ways:

- when carved off a bigger block in an allocation
- when returned at process exit

Note that fragments are created in two ways:

- when carved off a bigger block in an allocation
- when returned at process exit

The second generally gives us larger fragments, but both need to be addressed

# Memory

## Physical Memory

Allocation of physical memory is **still a problem** in current machines where certain kinds of hardware need large contiguous chunks of physical memory, e.g., GPUs