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 2GB 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 2GB 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 2GB 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

Memory

Physical 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

Memory
Physical Memory

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, declared 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

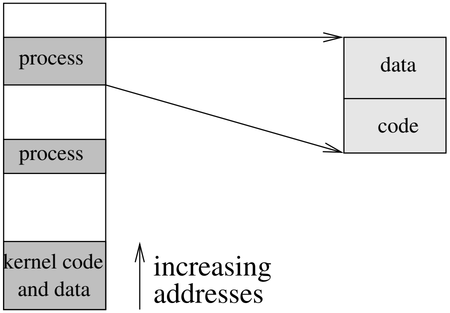
Reflecting this, early computer languages did not support dynamic allocation, e.g., FORTRAN, every array must be of a fixed size, declared in the source code

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

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

Memory
Physical Memory

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

Memory

Physical Memory

Partitioning

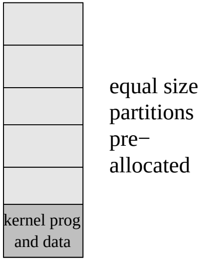
The earliest and simplest memory layout is a static system called *partitioning*, where areas are allocated at boot time

Memory

Physical Memory

Partitioning

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

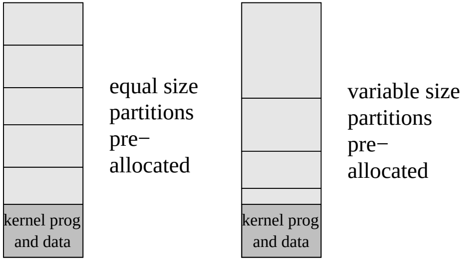


Memory

Physical Memory

Partitioning

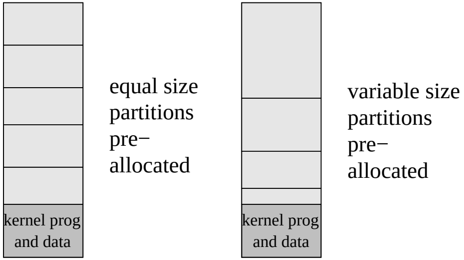
The earliest and simplest memory layout is a static system called *partitioning*, where areas are allocated at boot time


kernel prog and data
equal size partitions pre–allocated
kernel prog and data
variable size partitions pre–allocated

Memory
Physical Memory

Partitioning

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


kernel prog and data
equal size partitions pre–allocated
kernel prog and data
variable size partitions pre–allocated

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

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

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

Variable size is not much harder to implement, but 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

Memory

Physical Memory

Overlays

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

Memory

Physical Memory

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*

Memory

Physical Memory

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 we shall see that exactly this *is* possible with modern hardware!

□