Memory

Language Support for Dynamic Allocation

These days dynamic allocation is common in languages

Memory

Language Support for Dynamic Allocation

These days dynamic allocation is common in 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 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 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`)

And several other approaches!

Memory

Physical Memory

Dynamic Partitioning

So we need to to be dynamic: create and allocate a partition as needed

Memory

Physical Memory

Dynamic Partitioning

So we need to to be dynamic: create and allocate a partition 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

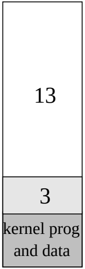
Physical Memory

We can allocate sequentially, moving up memory

Memory

Physical Memory

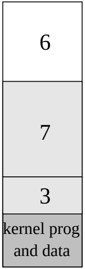
We can allocate sequentially, moving up memory



Memory

Physical Memory

We can allocate sequentially, moving up memory



6

7

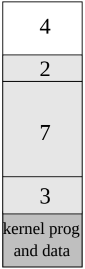
3

kernel prog and data

Memory

Physical Memory

We can allocate sequentially, moving up memory



4

2

7

3

kernel prog
and data

Memory

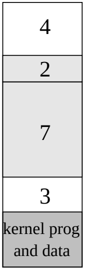
Physical Memory

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

Memory

Physical Memory

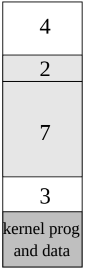
The problem is when a process ends and we get memory back: it creates holes



Memory

Physical Memory

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



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

Memory

Physical Memory

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

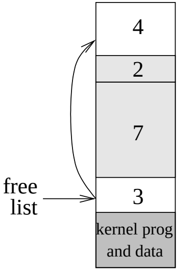
Physical Memory

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

Memory

Physical Memory

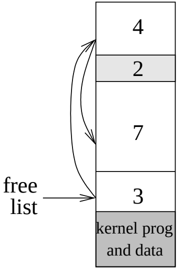
We need to keep a list of free blocks so we can track free space: a *freelist*



Memory

Physical Memory

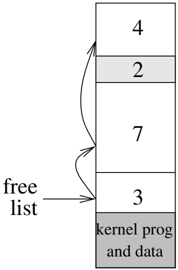
When a block is freed, put it in the freelist. It helps to keep the freelist sorted in address order:



Memory

Physical Memory

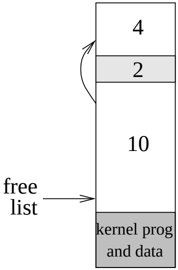
When a block is freed, put it in the freelist. It helps to keep the freelist sorted in address order:



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

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

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

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

Memory

Physical Memory

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 the allocation

Memory

Physical Memory

Note that fragments are created in two ways:

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

Memory

Physical Memory

Note that fragments are created in two ways:

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

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

Memory

Physical Memory

If we can’t find a big enough free space, we can consider *compaction* of memory using a technique called *garbage collection*

Memory

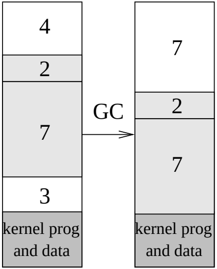
Physical Memory

If we can’t find a big enough free space, we can consider *compaction* of memory using a technique called *garbage collection*

The OS stops all running processes (i.e., stops scheduling processes); shifts their code and data around to close up the gaps; then lets the processes continue (i.e., starts scheduling again)

Memory

Physical Memory



Memory

Physical Memory

GC is not often used in general-purpose OSs

Memory

Physical Memory

GC is not often used in general-purpose OSs

- it is a very expensive operation to move all these blocks around

Memory

Physical Memory

GC is not often used in general-purpose OSs

- it is a very expensive operation to move all these blocks around
- this takes a lot of time away from running of processes

Memory

Physical Memory

GC is not often used in general-purpose OSs

- it is a very expensive operation to move all these blocks around
- this takes a lot of time away from running of processes
- the pause while things are moved is bad for interactive and real-time behaviour

Memory

Physical Memory

GC is not often used in general-purpose OSs

- it is a very expensive operation to move all these blocks around
- this takes a lot of time away from running of processes
- the pause while things are moved is bad for interactive and real-time behaviour
- the erratic nature of when GCs are needed leads to unpredictable behaviour from the OS

Memory

Physical Memory

GC is not often used in general-purpose OSs

- it is a very expensive operation to move all these blocks around
- this takes a lot of time away from running of processes
- the pause while things are moved is bad for interactive and real-time behaviour
- the erratic nature of when GCs are needed leads to unpredictable behaviour from the OS
- given the right kind of hardware support, better solutions completely avoiding the need for GC are possible

Memory

Physical Memory

GC *is* successfully used in user languages, e.g., Lisp, Java

Memory

Physical Memory

GC *is* successfully used in user languages, e.g., Lisp, Java

There are ways of implementing GC to avoid the stop-and-copy (ephemeral GC), or mitigating the overhead (generational GC) but even so it is not popular for OSs

Memory

Physical Memory

Notice that all these rely on *relocatable* processes, namely ones that don’t refer to specific locations in memory

□

Memory

Physical Memory

Notice that all these rely on *relocatable* processes, namely ones that don't refer to specific locations in memory

Note that the code in a process now can't use an absolute "jump to memory location 42", but must use a relative "jump by *n* bytes"

□

Memory

Physical Memory

Notice that all these rely on *relocatable* processes, namely ones that don’t refer to specific locations in memory

Note that the code in a process now can’t use an absolute “jump to memory location 42”, but must use a relative “jump by *n* bytes”

And similarly for referencing data in memory

Memory

Physical Memory

Notice that all these rely on *relocatable* processes, namely ones that don’t refer to specific locations in memory

Note that the code in a process now can’t use an absolute “jump to memory location 42”, but must use a relative “jump by *n* bytes”

And similarly for referencing data in memory

Unless we know in advance where our process is going to be placed in memory, we cannot have code that has fixed absolute addresses in it

□

Memory

Physical Memory

Notice that all these rely on *relocatable* processes, namely ones that don't refer to specific locations in memory

Note that the code in a process now can't use an absolute "jump to memory location 42", but must use a relative "jump by *n* bytes"

And similarly for referencing data in memory

Unless we know in advance where our process is going to be placed in memory, we cannot have code that has fixed absolute addresses in it

These issues develop when we move to *virtual* memory later, but in general code should not assume it lives in a given place in memory

□