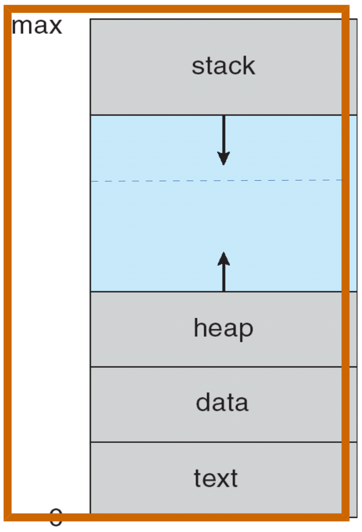
Operating Systems – Supplementary Notes

Episode 1 – The Phantom Memory



This is what a process's memory space looks like.

The stack grows **downwards**. Roughly, it is where temporary variables, method parameters, and return addresses are stored

Local variables to the method are pushed onto the stack

The memory is taken and released by **decrementing** the stack pointer at the start of the method; and **incrementing** it by the same amount at the end.

- To call a method:
  - Push any parameter values onto the stack
  - Use 'call' to push the **return address** (the address of the next line of code in the caller) onto the stack, and then jump to the code for the method to call
  - Inside the method:
    - Take/release memory from the stack for local variables
    - Use 'ret' to pop the return address off the stack, and jump back to there
  - Increment the stack pointer to free up the memory used by the parameters

The stack has a **finite size** – see the dotted line on the diagram. If you write a method that calls itself in an infinite loop, you can hit this limit – known as a **stack overflow**.

Variables on the stack **cannot survive once the method is over**. In short, this is what the heap is for: it is used **whenever you write the word `new`**. It starts as one large block of free memory – a single hole of available memory. As space is allocated, parts of this become used. As space is deallocated, previously used parts of it become free again. Inevitably, the memory is then divided into a mixture of free regions, and used regions, next to each other.

The list of where the free/used memory is, is stored in memory as a **doubly linked list** where each node in the list is a **memory control block** placed immediately before some area of memory. As the slides note, doubly linked is important, to allow adjacent free regions to be merged.

First-fit/best-fit/worst-fit choose which free region to use, when a request for some memory is made. e.g. `new Geoff()`, where storing an object of type `Geoff` needs 32 bytes of memory: where should that 32 bytes be stored?

Fragmentation

- External fragmentation: e.g. 100 bytes of memory are requested; but there is no **contiguous** block of memory (a single hole) of size 100, even if the total size of all the holes added together, is 100.
- Internal fragmentation: a memory allocation request is made; more memory is given. This can happen if the leftover space in a hole is not enough for a memory control block. You will have seen this in practice when writing your solution for the week 2 programming exercise.

Object pools

To help avoid external fragmentation, one can use an **object pool**. Instead of creating space for small objects one at once, create space for several of them in one go; and hand these out each time new is written. If the pool is empty, create some more. If everything in the pool has been deallocated, the pool can be deleted.

This can substantially reduce the allocation overheads. Have a look at the lecture slides and the tutorial questions for the outline of why this is the case.

Memory paging

If a process's memory needs change whilst it is running (loading/saving documents, ...) there is no easy way to decide how much memory it gets whilst it is running.

Memory paging avoids this problem by using a **page table**. The memory of the machine is split into **frames** of a fixed size (e.g. 4KB). A process' page table then says for each **page** of memory, which frame of memory is it stored in.

Logical memory addresses are divided into two parts when using paging:

- A page number, p (used as the index into the page table)
- An offset, d (how many bytes into this page do you want to read)

Address binding, courtesy of good-advice mallard, turns this **logical memory address** into a **physical memory address**: for the logical memory address (p,d), the physical memory address is page_table[p] + d.

The memory within a single page itself: it's a block of memory, split into free/used regions, so the same techniques covered on the previous page, apply within pages too.

Translation Lookaside Buffer (TLB)

The page table lives in memory. This means in principle, address binding needs two memory accesses. The TLB is a cache of 'page_table[p]' for some page numbers p. If the frame number for p is usually in the TLB, then most of the time, we only need one memory access. If one has the probability of a TLB hit, one can do a weighted average to find out the expected memory access time – see the slide for details.

Virtual memory

If a computer runs out of memory, we have a problem. Killing processes would be inelegant. Instead, use the copious amounts of disk space computers have (compared to how much RAM they have) to swap out pages to disk when they're not in use; and swap them back in when they're in use again. This is in a nutshell what virtual memory does.

How do we determine what page to swap out? We choose a **victim frame** using some algorithm. In the week 4 programming exercise you implemented the **second chance** algorithm. Other options are first-in first-out, least recently used, and various others.

But, these algorithms aren't perfect: they look at the past memory accesses, to try to guess what is now no longer in use. As with the stock market, though, the past is not necessarily a predictor of the future. Ideally we want to swap out the page that isn't going to be used for the longest amount of time. Otherwise, it will need to be swapped back in again soon, and swapping takes time (conventional hard disks are easily 100000x slower than RAM).

The **optimal** page replacement algorithm is a theoretical solution to this: it looks into the future and chooses as a victim the page that isn't going to be used for the longest amount of time. It's theoretical because one actually cannot look into the future. But at least it lets us know how good the other algorithms are doing.

(The only exception to not knowing the future, is prefetching: programmers can write hints into their code about what memory it will access next. This was sketched out on a later slide.)

The valid bit (and other bits too)

Q: How do we know which pages are in memory, and which aren't?

A: By storing a 'valid bit' on the page table. That is, the page table now contains a frame address, and a valid bit, for each page number. Accessing a page whose valid bit is false, triggers a **page fault** – the **page fault handler** in the operating system kicks in, to implement virtual memory. The process then eventually gets the data it wanted off that page, without knowing any of this has happened.

Cool trick: the valid bit takes no extra space, as we can use the bottom bit of the frame address (which was always zero anyway) to store the valid bit. See the lecture slides for an illustration of this.

We can use this trick to store all sorts of bits – the dirty bit, and for security, read-only/execute-only bits. Or, if using the second-chance algorithm, the reference bit.

Thrashing

Virtual memory relies on being able to choose victim frames that contain pages of memory that are not presently in use.

A process' **locality** (or **working set**) is the pages it is currently using (i.e. is likely to access in the next short while).

If the total size of the processes' localities, across all processes on the machine, exceeds the amount of physical memory of the machine, we have a problem: victims are being chosen that are immediately or shortly afterwards need to be swapped back in from disk again. This quickly causes swapping time to account for most of the runtime of the system: processes end up sitting on the 'waiting for IO' queue, so CPU usage falls to near-zero whilst the hard disk clicks away like mad.