Computer systems architectures  
CM12002

Russell Bradford

2023/24

### 1. Memory

#### Virtual Memory: Paging

At last we can talk about paging

*Paging* is copying pages to and from disk

Suppose there is a memory access

If there is a TLB hit, the memory access continues at full speed

On a TLB soft miss the usual case is that the page is still resident in physical memory (not been swapped out), so it’s just a matter of updating the TLB to refer to it by copying the page table entry into the TLB

And then the access can continue as for a hit

### 2. Memory

#### Virtual Memory: Paging

But if the page has been swapped out (“paged out”), then its contents need to be read back from disk: thus a large cost in this case

When the TLB is full and the process want to access a different page, one entry in the TLB needs to be removed: but which?

Note there are two separate issues here:

* which entry in the TLB to remove when the TLB table is full
* which page in physical memory to swap out when physical memory is full

**Exercise** Read about some of the algorithms to choose which TLB entry to remove

### 3. Memory

#### Virtual Memory

The first time a (virtual) page is touched by a process it will cause a (major) page fault

The OS needs to allocate a new physical page for this process

Allocation of new pages is facilitated by the fixed page size: just find *any* unallocated page in physical memory and set it as the physical page mapping from the requested virtual page

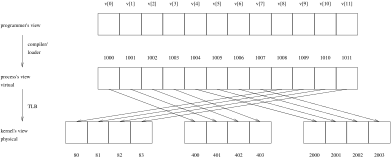
This simplification over earlier allocation methods is the big benefit of using pages

The first page on the freelist of pages is always suitable. No need to search, no size fit issues, no fragmentation issues

### 4. Memory

#### Virtual Memory

A single large datastructure (e.g., a vector, which you normally think of as a contiguous region of memory) in your process might actually be spread, in chunks, all over the place in physical memory



### 5. Memory

#### Virtual Memory

Similarly for code: a chunk of code spanning multiple pages may well be distributed all over physical memory

Code or data might be contiguous in the virtual address space, but definitely not contiguous in the physical address space

### 6. Memory

#### Virtual Memory

OSs often use *lazy* page allocation: don’t allocate anything until the process actually accesses a page, so physical memory is only actually allocated on a page fault when we know we really need it

<+(0)->If process requests 10GB and only uses 1GB, this is not a problem: only 1Gb will be mapped in the page table

And the process’s virtual size can easily be bigger than the physical memory size, either through unmapped or swapped pages

### 7. Memory

#### Virtual Memory

The cost is kept low though the use of the TLB, but remember a page fault is relatively expensive

And swapping is orders of magnitude slower still: we want to avoid swapping if at all possible

This is something in the hands of the programmer: don’t use memory stupidly!

### 8. Memory

#### Virtual Memory

Note that the terms “paging” and “swapping” are near-indistinguishable these days

Swapping used to mean entire processes

Then *segments* (certain large areas) of memory

Now just pages are swapped

Note that when swapping a page back into memory, it doesn’t matter where in physical memory we put it : the page table/TLB ensures the process sees it in the same virtual place

### 9. Memory

#### Virtual Memory

Exercise. Think about the difference between vectors and linked lists in terms of virtual memory and TLBs

Exercise to think about: the page tables in memory can grow so large they need to be swapped themselves…

### 10. Memory

#### Virtual Memory

Examples. A “Hello world” program in C, Java, Python and Perl

|  | C | Java | Python | Perl |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| Resident size KB | 430 | 16500 | 4300 | 1850 |
| Minor Fault | 150 | 3800 | 1200 | 530 |
| Major Fault | 0 | 0 | 0 | 0 |
| Context switch | 2 | 150 | 8 | 4 |

In Linux 3.11.10; 8GB memory

Numbers are approximate and vary on runs due to scheduling

### 11. Memory

#### Virtual Memory

**Shared Memory**

And now shared memory is very easy: just let the TLB do the mapping of virtual pages from different processes to the *same* physical pages

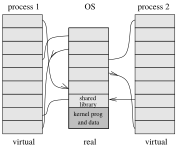
And also the use of virtual memory can let us *share* code between processes

There are many libraries of code to do mundane things like read or writing to files, formatted printing, drawing on the screen and so on

If 10 processes are in memory, each of them using the library read function, does that mean there are 10 copies of the code for read scattered about in memory?

### 12. Memory

#### Virtual Memory



Shared Libraries

### 13. Memory

#### Virtual Memory

Now the OS can load the code for read just once and direct all other processes to use that single copy

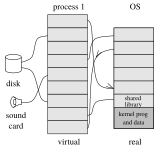
This reduces memory usage, reduces pages faults and has other beneficial properties (see caching)

**Exercise** Read about how virtual memory and *copy on write* allows processes to share data, too

### 14. Memory

#### Virtual Memory

**Exercise** Read about how virtual memory can be used to map memory accesses into access of peripherals, like sound cards, disk and network cards



Memory map

### 15. Filesystems

We now turn to files and filesystems

Current technology has main memory being limited in size (a few gigabytes) and *volatile*: the values disappear when you remove the power

To be able to manipulate more data and to make it *persistent* we turn to larger, but slower, devices like disks

And to organise everything we need *filesystems*

### 16. Filesystems

Note: not all applications want to use filesystems, in particular enterprise databases like to have direct access to disks themselves in order to optimise access for their very specific needs

Some people have experimented with making ordinary applications use DB-like access, mostly to a resounding failure

In general, a filesystem is what people want: a simple, efficient way of accessing their data

### 17. Filesystems

Another note: a filesystem is just an organisation of data, and doesn’t need to be associated with *disks*

Filesystems can be found whenever we have large amounts of data that needs organising

USB keys, iPods, phones, …

It’s even occasionally useful to have a filesystem *in memory*, again as an organisational mechanism

### 18. Filesystems

Yet another note: and it’s not necessary that the object or objects behind the filesystem store data

We can have it so that reading from one particular file actually returns keystrokes from the keyboard

Or writing to another file is actually sending sound to a soundcard

In fact, a Unix philosophy is “all devices are files”

This makes accessing devices incredibly easy for the programmer: just read and write a file

Exercise. Compare with using virtual memory to do the same

### 19. Filesystems

But, for now, we shall think of files in the traditional sense

A *file* is simply a named chunk of data stored somehow on a disk

Humans like easy names like prog.c, so there needs to be a mechanism to convert names to the place on disk where the data is stored

And when we have thousands or millions of files, meaning thousands or millions of names, we need some way of organising the names (even before we have thought of organising the data itself!)

### 20. Filesystems

#### Names

Notice the distinction between the *name* and the *data*

This is *very* important and the distinction runs throughout computer science

The same name can refer to different data (otherwise the whole thing would be useless, we could never fix bugs in prog.c)

Different names can refer to the same data. We tend to forget that, in real life, we can use different names to refer to the same thing: “Lewis Carroll” and “Charles Dodgson”

All but the simplest filesystems allow the same file to have multiple filenames

### 21. Filesystems

#### Names

For the philosophers:

It is possible to have a thing without a name (so how can we refer to it?)

It is possible to have a name without a thing it refers to

It is possible for names to have names

Exercise: read the introduction to the poem “Haddocks’ Eyes”, in “Through the Looking-Glass” by Lewis Carroll and explain the relevance

And explain the use of quotes ’“’ in the above

### 22. Filesystems

#### Names

So names need to be organised; this is usually (but not always) done as a simple *hierarchy*

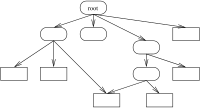
Rather than simply presenting all filenames to the user (a *flat* filesystem), we gather together related files and put them into a *directory*. Also called a *folder*

A directory is just a collection of (names of) files, but it allows us to simplify our thought processes

And (names of) directories can be collected in other directories and so on until we get to the top of the hierarchy, the *root*

### 23. Filesystems

#### Names



Files can appear at all levels But always within a directory In some systems, a file can be in more than one directory Generally, directories can only be within exactly *one* directory, for implementation reasons Directories can be empty

### 24. Filesystems

#### Names

The namespace hierarchy makes referring to a file easy

A Unix example: /usr/bin/ls refers to a file named ls inside a directory named bin inside a directory named usr which is in the root directory

The /s separate the names

The root directory is referred to as /, though its actual name is empty

Other OSs have similar ideas, but use different separators

Files can have multiple names: we might find that /usr/local/bin/dir refers to the same file as /usr/bin/ls

### 25. Filesystems

#### Names

The directory hierarchy forms a *directed acyclic graphs* (DAG)

This means: no loops

No loops means we can simply traverse the whole hierarchy and never get stuck in a loop; and no unconnected loops if we delete a directory

We might find the same file twice, though

This is a tradeoff of flexibility vs. ease of system implementation

### 26. Filesystems

#### Names

To make referring to files even easier, each process has a *current working directory* (cwd)

This is just a prefix, stored in the PCB for each process, so that whenever the process asks for a file by an incomplete filename (not starting with a /), the kernel glues the cwd prefix on to the given name and uses that full name instead

So, with a cwd of /u/cs/1/cs1abc a process that asks for file prog.c gets file /u/cs/1/cs1abc/prog.c

With a cwd of /u/cs/1/cs1def a process that asks for file prog.c gets file /u/cs/1/cs1def/prog.c

### 27. Filesystems

#### Names

This is how different processes can refer to the same name prog.c but get different files

The cwd is a convenience for the programmer and may be changed as often as you like (cd, chdir)