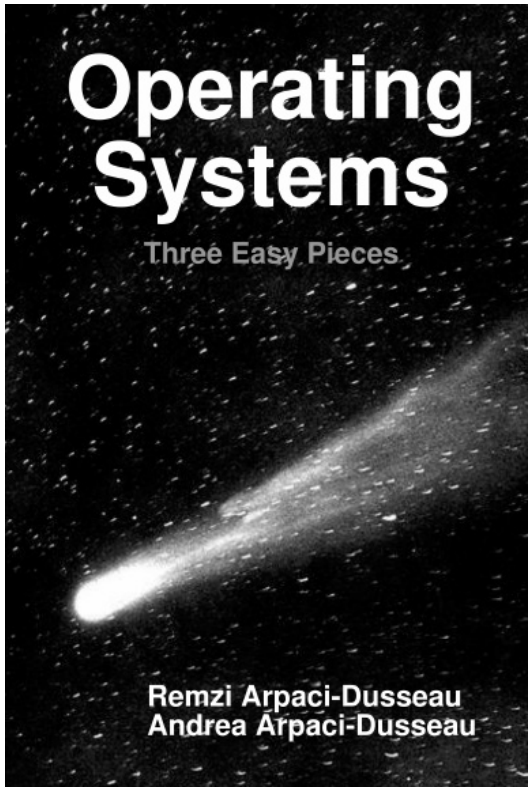
Where we are: further virtualisation of memory

- **how page faults enable ‘illusions’**
- virtual memory, by demand paging
- page replacement, frame allocation

References in the OSTEP book for this week



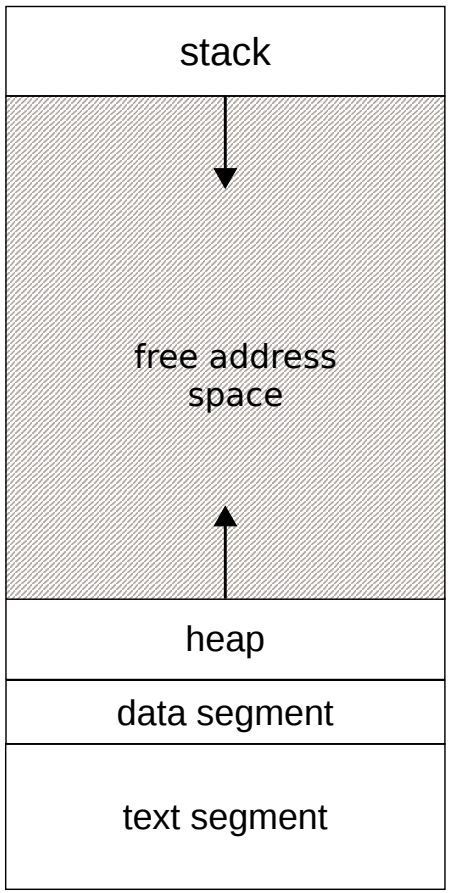
Page faults 21.2, 21.3

Illusions chapter 21 introduction

Demand paging chapter 21 (all); 22.10

Page replacement, frame allocation chapter 22

A picture we've seen before. . .



Each process has a virtual address space.

Usually, much of it is unused (unmapped)!

MMU hardware performs *address translation*

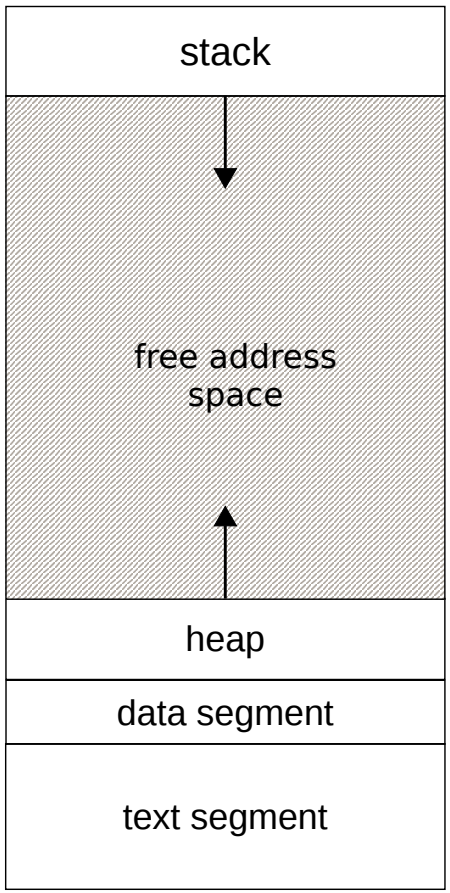
Access to an unmapped address causes a *page fault*

- a hardware interrupt
- causing the OS to regain control
- . . . and do what?

Simple answer: *kill the process*

- 'Segmentation fault' (a misnomer nowadays)

A picture we've seen before. . .

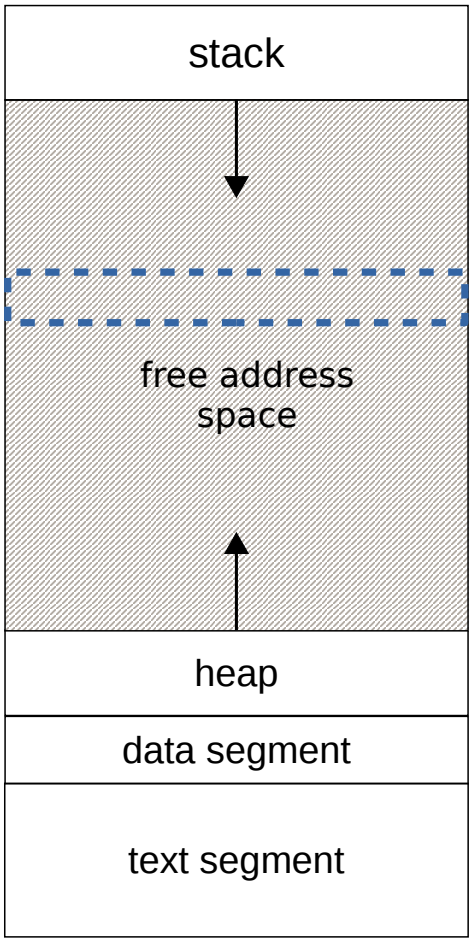


Instead of killing the process when a page fault is raised, the OS can choose to 'do something'!

Page faults create a *point of interposition* in the OS, between user programs and hardware

The OS can create 'useful illusions' by handling page faults in creative ways. Examples:

- lazy allocation, lazy loading. . .
- virtual memory via *page swapping* (a.k.a. just *paging* or *swapping*): the illusion that the machine has more memory than the physical memory in the machine
- memory-mapped files: the illusion that files on disk are resident in the address space



OS may choose to *reserve* some virtual address space *without* allocating phys. memory there.

Say we run a process *as if* there *is* memory there.

Accessing that ‘memory’ causes a *page fault*.

To *handle* the page fault, the OS might. . .

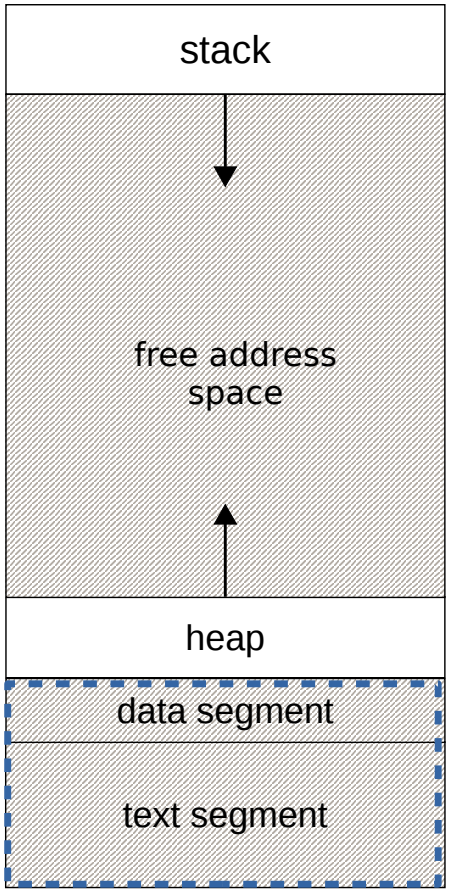
- allocate physical memory (lazy allocation)
- allocate memory and populate with file data (memory-mapped file; lazy loading)
- keep emulating the access (rarer. . .)

Then resume the process!

A simple 'just abort' page fault handler on x86 (from InfOS)

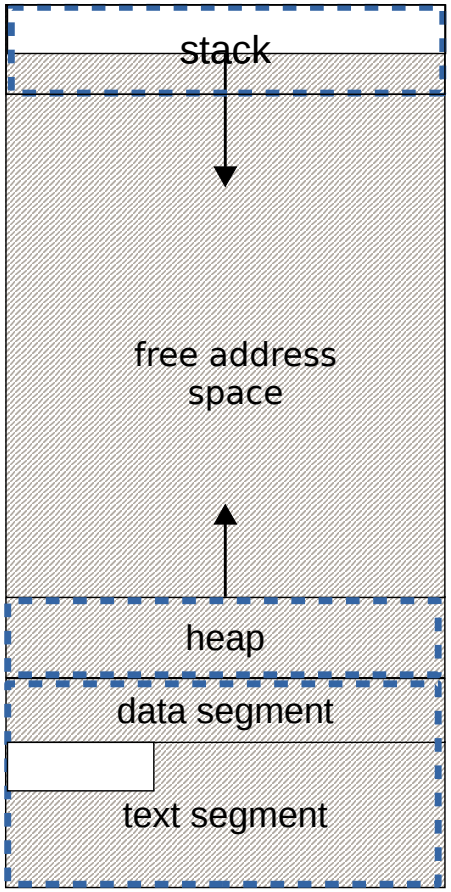
```
1  static void handle_page_fault(const IRQ *irq, void *priv)
2  {
3    // Retrieve the fault_address from control register.
4    uint64_t fault_address;
5    asm volatile("mov %%cr2, %0" : "=r"(fault_address));
6
7    if (current_thread == NULL) {
8      // If there is no current_thread, then this fault happened REALLY early. We must abort.
9      syslog.messagef(LogLevel::FATAL, "*** PAGE FAULT @ vaddr=%p", fault_address);
10     arch_abort();
11   }
12   // If there is a current thread, abort it.
13   syslog.messagef(LogLevel::WARNING, "*** PAGE FAULT @ vaddr=%p rip=%p proc=%s",
14     fault_address, current_thread->context().native_context->rip,
15     current_thread->owner().name().c_str());
16   // TODO: support passing page faults into threads.
17   current_thread->owner().terminate(-1);
18 }
```

(for illustration! details not important!)



We could make *all* allocation in the process lazy.

- stack: allocate/grow as used
- heap: allocate/grow as used?
- segments: copy from disk only as needed



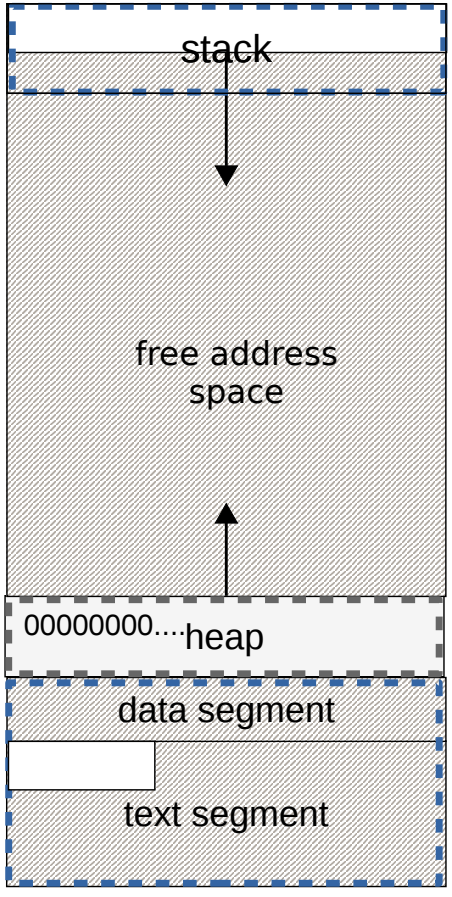
We could make *all* allocation in the process lazy.

- stack: allocate/grow as used
- heap: allocate/grow as used?
- program segments: copy as needed from disk

. . . all occurring in page-sized chunks.

This is not too far from reality! Exceptions:

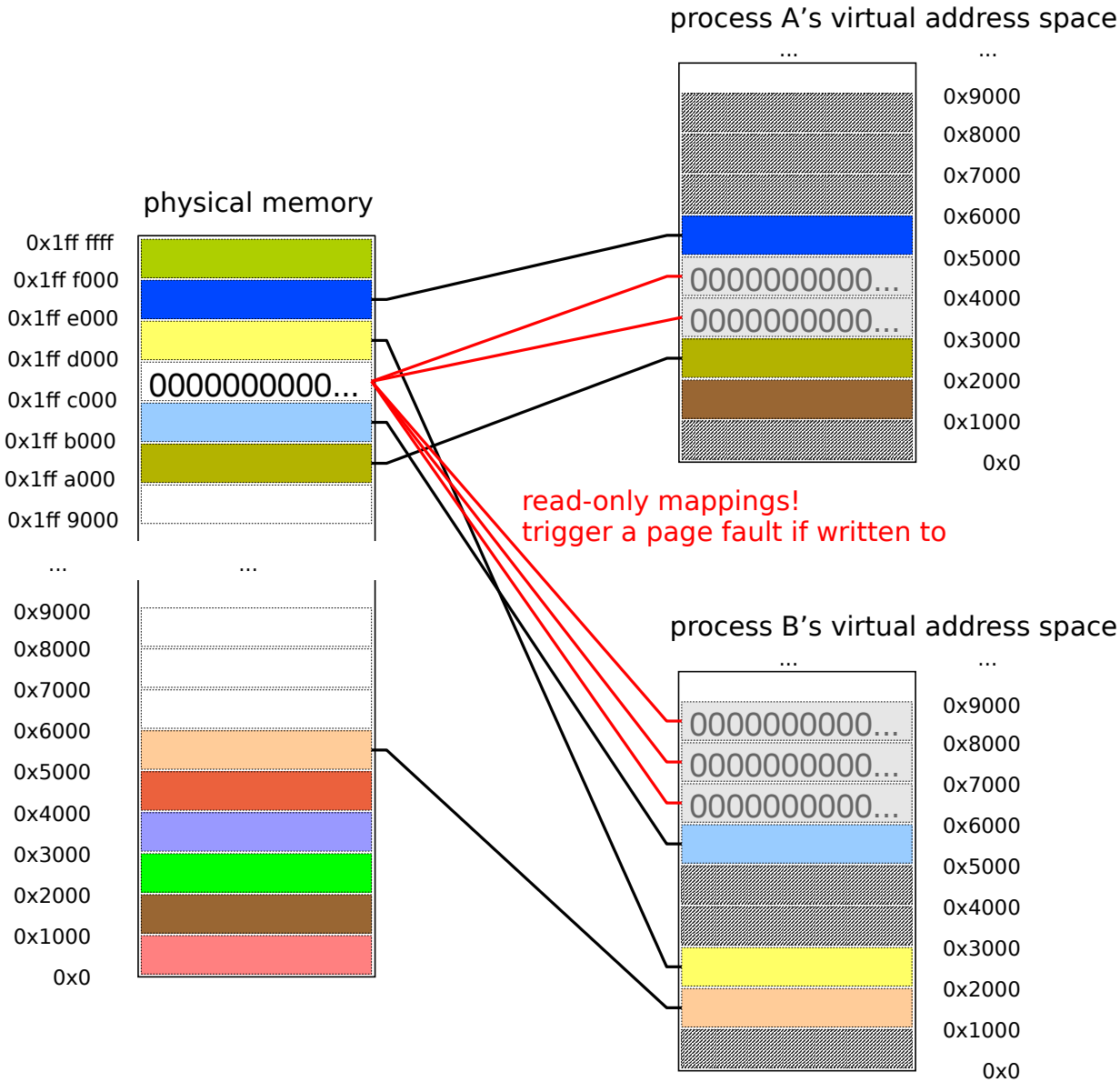
- the heap mapping **doesn't** work like this. . .
- it is made/grown explicitly by system call
- . . . **not** grown lazily by page fault.
- (Why? We'll see later in the course.)

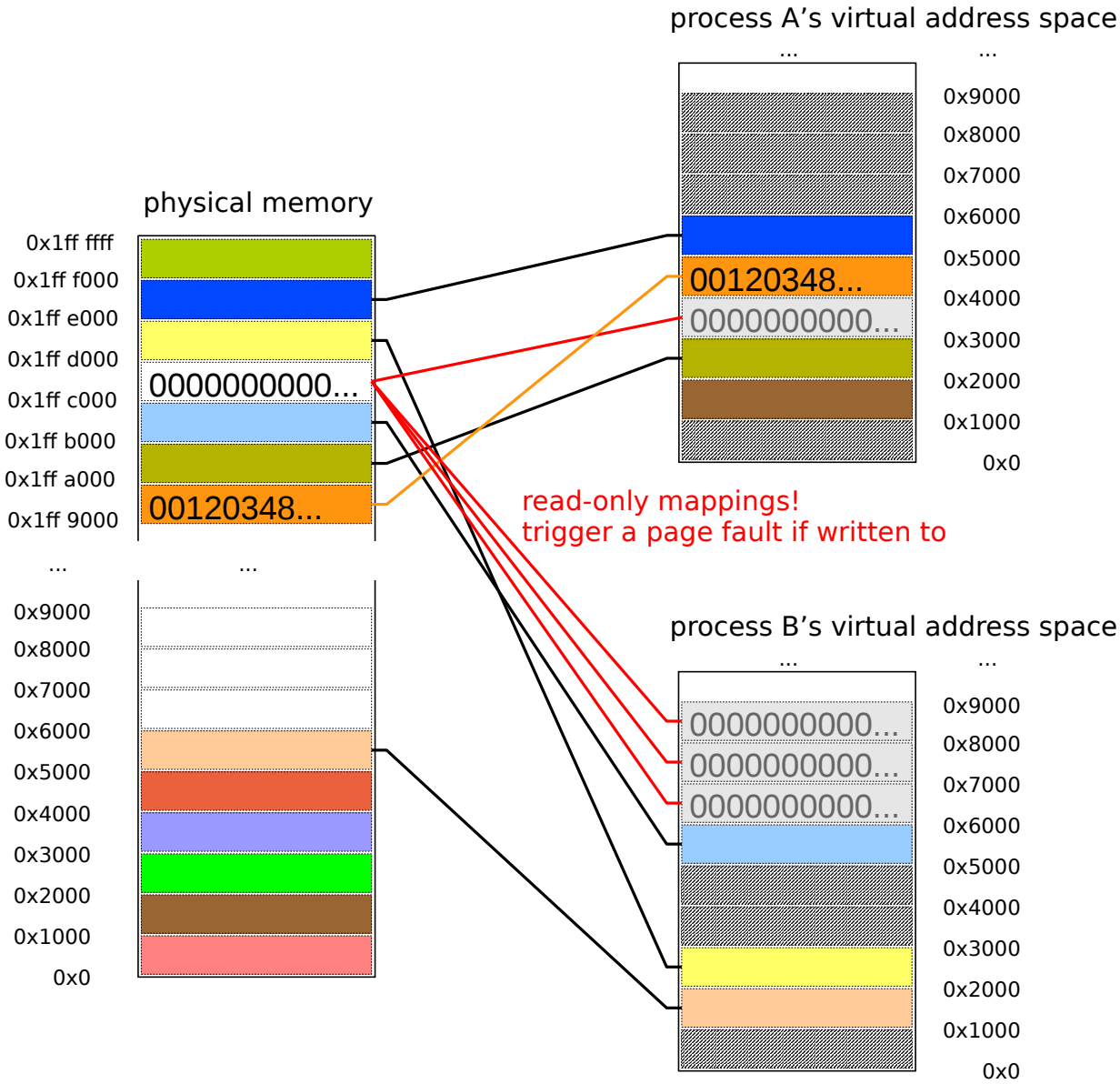


Besides lazy allocation, other tricks exist. . .

Lazy zeroing:

- Illusion of a large *zero-initialized* region
- by mapping *read-only* the same *zero frame*
- . . . lots of times over!
- *On first write*, fault handler maps fresh frame
- **good:** allocation appears fast
- **good:** allows *sparse* use of memory
- . . . i.e. $O(n)$ zeroes fit in $O(1)$ memory, and
- . . . zeroes are *readable* at all times


process A's virtual address space
physical memory
0x1ff ffff
0x1ff f000
0x1ff e000
0x1ff d000
0x1ff c000
0x1ff b000
0x1ff a000
0x1ff 9000
0000000000...
0x9000
0x8000
0x7000
0x6000
0x5000
0x4000
0x3000
0x2000
0x1000
0x0
read-only mappings!
trigger a page fault if written to
process B's virtual address space


process A's virtual address space
physical memory
0x1ff ffff
0x1ff f000
0x1ff e000
0x1ff d000
0x1ff c000
0x1ff b000
0x1ff a000
0x1ff 9000
0000000000...
00120348...
0x9000
0x8000
0x7000
0x6000
0x5000
0x4000
0x3000
0x2000
0x1000
0x0
read-only mappings!
trigger a page fault if written to
process B's virtual address space

What just happened?

The process tried to *write* to the read-only zero mapping!

```
mov $0x00120348,(%rax)  ; %rax points to our zero-init'd memory area
```

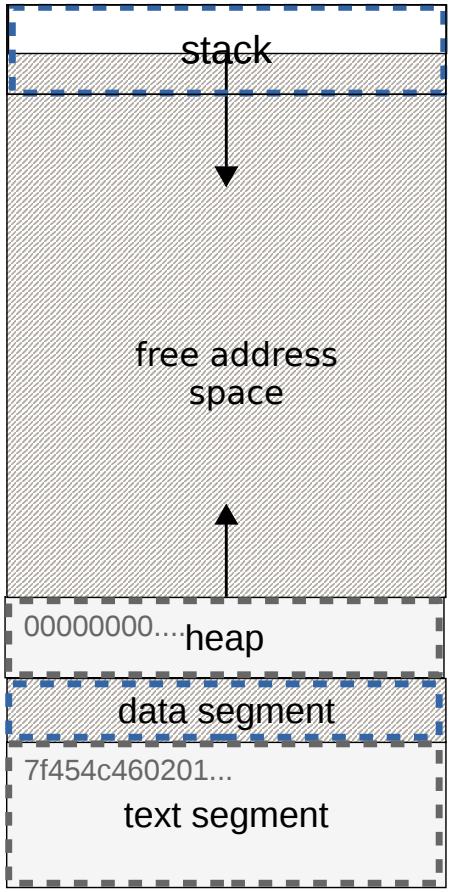
In the *page fault handler*, the operating system. . .

1. Considered the *faulting address* and *which address space* (process)
2. Determined (how?) that this area was allocated '*copy on write*'
3. Allocated a fresh frame
4. Map it at the faulting page
5. . . . i.e. *replacing* the previous mapping of shared zero frame
6. Copied the shared contents (zeroes) into the new frame
7. Resumed the process at the faulting instruction. . .
8. . . . which now succeeded!

The program is oblivious to all this. . .

A more elaborate page fault handler (still InfOS!)

```
1  static void handle_page_fault(const IRQ *irq, void *priv)
2  {
3    // Retrieve the fault_address from control register.
4    uint64_t fault_address;
5    asm volatile("mov %%cr2, %0" : "=r"(fault_address));
6    if (current_thread == NULL) {
7      // If there is no current_thread, then this fault happened REALLY early. We must abort.
8      syslog.messagef(LogLevel::FATAL, "*** PAGE FAULT @ vaddr=%p", fault_address);
9      arch_abort();
10   }
11   // If there is a current thread, look up what we should do
12   syslog.messagef(LogLevel::WARNING, "*** PAGE FAULT @ vaddr=%p rip=%p proc=%s",
13     fault_address, current_thread->context().native_context->rip,
14     current_thread->owner().name().c_str());
15   /* Is there a non-zero cookie in that PTE? */
16   VMA& vma = current_thread->owner().vma();
17   uint32_t cookie;
18   bool success = vma.get_pte_cookie(fault_address, cookie);
19   if (success && cookie)
20   {
21     syslog.messagef(LogLevel::DEBUG, "page fault looks like a ..."
```



Lazy zeroing is really a special case of *lazy copying*, known as **Copy on Write** (CoW)

Any *shared data*, not just zeroes, can be shared by 'many-to-one' read-only mappings. . .

Then copy *at the time of write* (if it ever happens!)

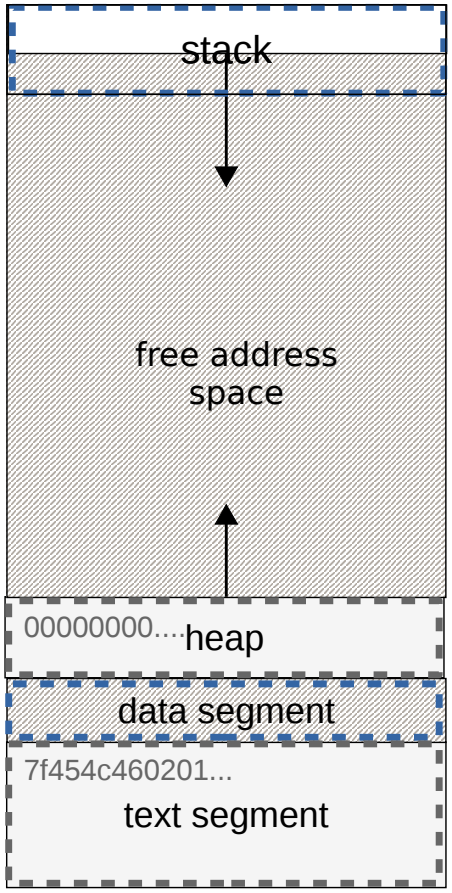
Often used for *text segments* in executable files.

Programs rarely write to their own text. . .

→ usually all *processes* that are instances of the same *program* can share it, saving memory.

(In fact, the data is not only lazily copied within memory, but also lazily brought in from disk. . .)

Not just *lazily allocating* physical memory; also *taking it back*!



Memory is often a scarce resource.

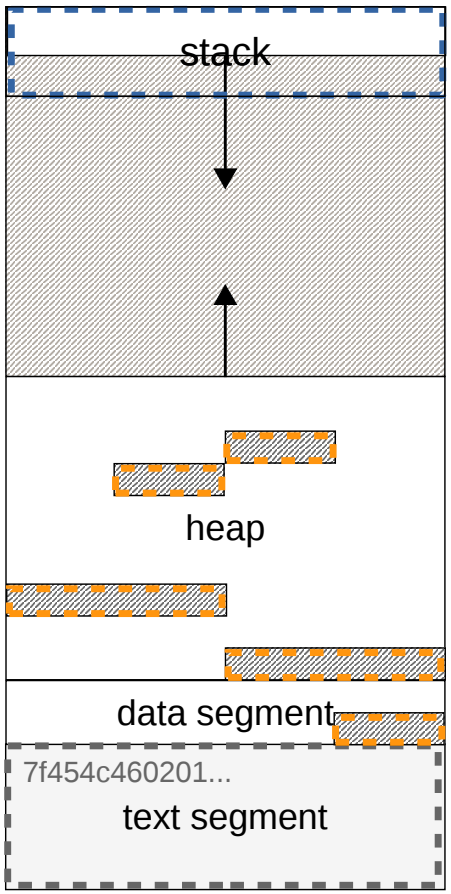
Can we create an *illusion* of having more of it?

- . . . to run more processes at once
- . . . to run *bigger* processes

Yes, by *swapping* data from and to disk

. . . using page faults to trigger re-loading.

Not just *lazily allocating* physical memory; also *taking it back*!



Not all memory is being used at once. . .

We swap data to and from disk

. . . using page faults to trigger re-loading.

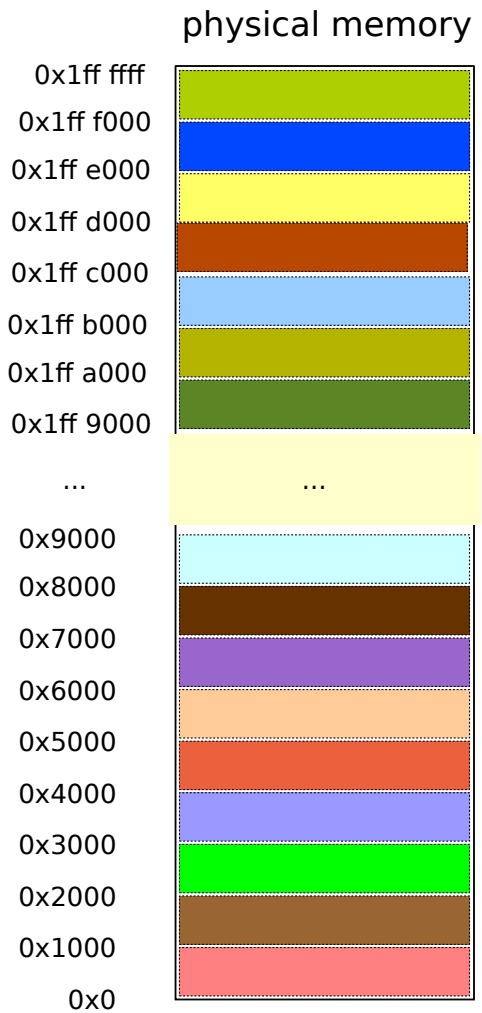
Not-recently-used pages are *replaced* or *evicted*

. . . suspiciously like in a cache!

Where we are: further virtualisation of memory

- how page faults enable ‘illusions’
- **virtual memory, by demand paging**
- page replacement, frame allocation

Imagine our physical memory is full up!



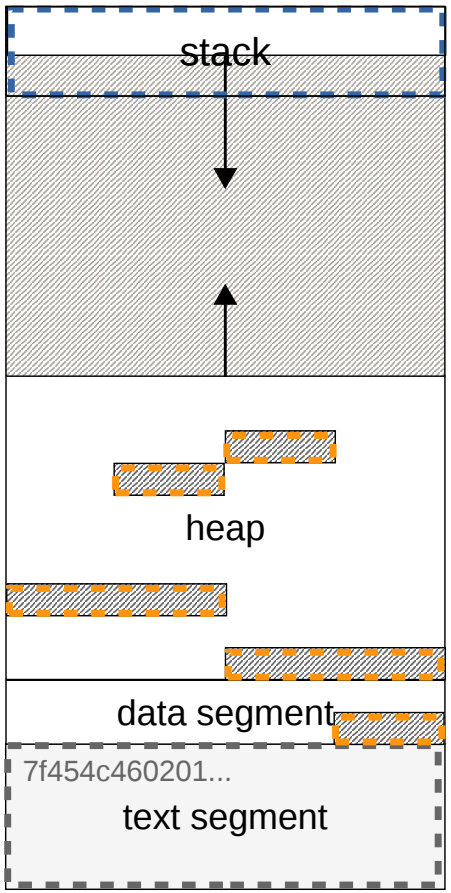
Every single page is being used for something.

Or *is it*?

Every frame is *allocated* to a process. . .

But they can't be using everything at once. . .

Not all physical memory is being used *at once*. . .



‘Virtual memory’ properly refers
to a specific illusion, *implemented using* virtual address translation

The illusion is of the computer
having more memory available to processes than it actually has physical memory.

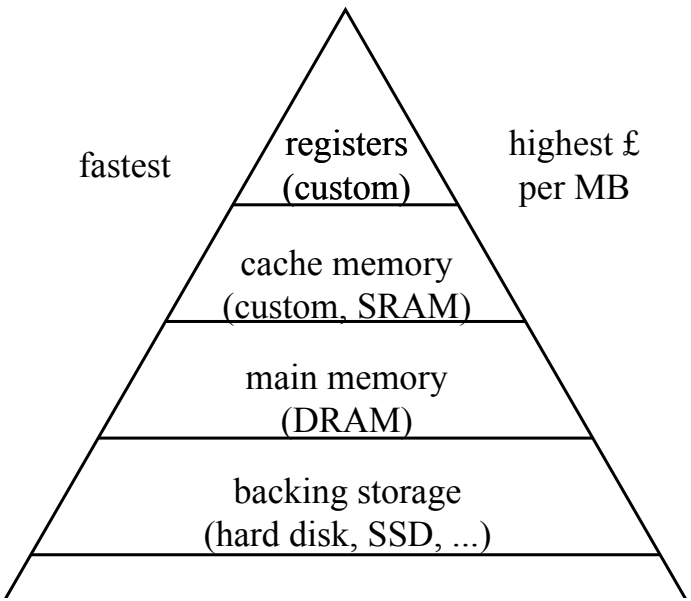
(So far I have been very careful, usually, to say ‘virtual address space’ or ‘virtual addressing’, not ‘virtual memory’.)

We move data to a *swap device*, i.e. a disk or other slow device further down the *storage hierarchy*

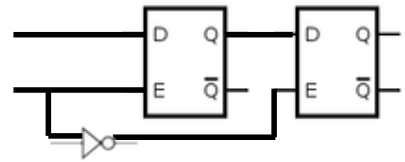
Storage hierarchy

To get the best *performance per £*, we build systems using a mix of technologies

- Top of hierarchy: small but fast
- Lower: bigger but slower



Flip-flops: registers; a few kB



SRAM: cache memory; a few MB

DRAM: main memory; a few GB

Hard disk: persistent storage; a few TB

Virtual memory as an extension of the storage hierarchy

Combine main memory
with reserved space on disk ('swap area')

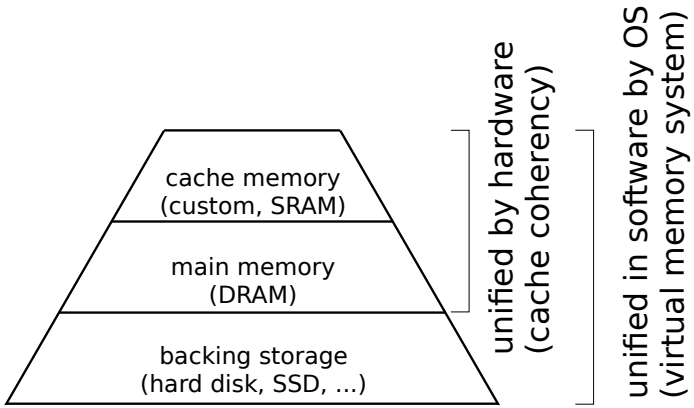
DRAM:
main memory;
a few GB

To free up physical memory, pages not (currently) needed can be *swapped out* to disk, not taking up DRAM

Hard disk:
persistent
storage;
a few TB

. . . and brought back on a faulting access

Illusion: more memory than physical DRAM!



Unlike cache memory, swap space
is managed by software (OS), not hardware

Interlude: disks are *block devices*

What is a 'disk' anyway? Disks are *block* devices, meaning they read and write data in large-ish units at a time, typically between 512 bytes and 4kB

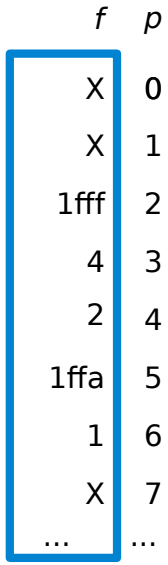
Unlike *memory*, they are *persistent* but are not byte- or word-addressable (only *block*-addressable).

As exposed by the *disk controller*, via which the operating system communicates with the disk, a disk is a linear sequence of blocks.

Operating systems commonly provide a layer of abstraction above this: the *filesystem*, a tree of files (leaf nodes) and directories (branch nodes). The OS stores this tree as a block-level data structure on the disk.

As alternatives to holding a filesystem, block devices can be used as *swap areas* and also *databases* (a 'relational' alternative to filesystems)

How can we implement swapping?



Page tables *as defined by the hardware*
are no longer sufficient to keep track of what's in our virtual address space. . .

As far as the hardware knows, page tables exist
only to map each page number to either a *frame number*
or to 'invalid' (nothing mapped)

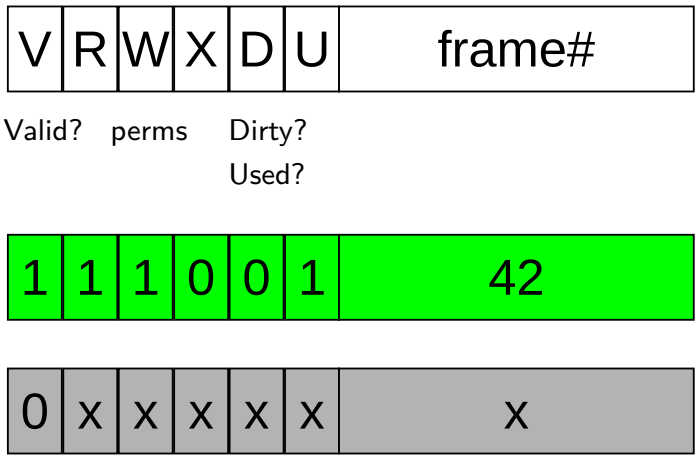
But we now might have:

- 'page swapped out to disk block 42'
- 'page read-in from disk block 123 was deferred'

. . . and also e.g.:

- 'read-only page is to be treated *copy-on-write*'

Two alternatives



The cheap and nasty option: stash extra information in *invalid* page table entries.

If hardware sees an PTE whose valid bit is *not* set, it will just ignore it.

So as the OS, we're free to use the other bits however we like.

The realistic option: in the OS, keep a separate data structure

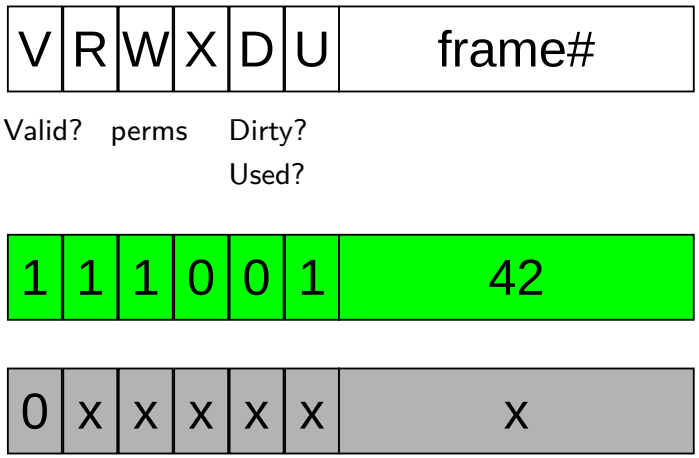
- e.g. Linux has a red-black tree of *virtual memory areas* (VMAs) *
- allows lazy creation of page table entries themselves (!)

Your lab exercise this week:

- implement the cheap & nasty option ('stash a cookie' in the PTE)
- use it for lazy loading (what to stash and when?)

* wart: InfOS uses 'VMA' differently, to refer to the whole virtual address space. It's better to say 'VAS' for that. Oh well.

Two alternatives



If we opt to stash data in the PTE, it is entirely up to us (the OS) what that data is and how it is encoded.

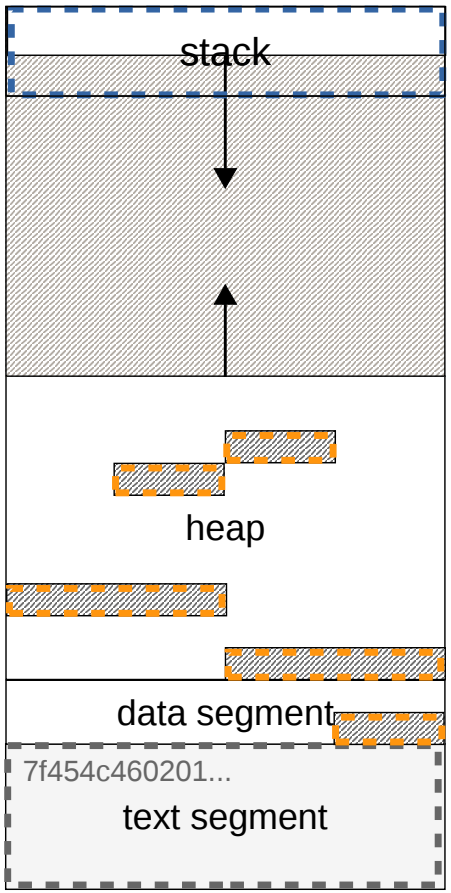
But it must be able to encode whatever we will need to know when a page fault comes in, so we can handle the fault.

E.g. what disk block the data is at, but also things like what permissions apply, whether copy-on-write applies, etc.

It doesn't have to involve a *block* number at all! Something more abstract might sometimes be appropriate.

In the lab, study the *page fault handler*, since it has already been written with these choices already made.

Demand-paged virtual memory

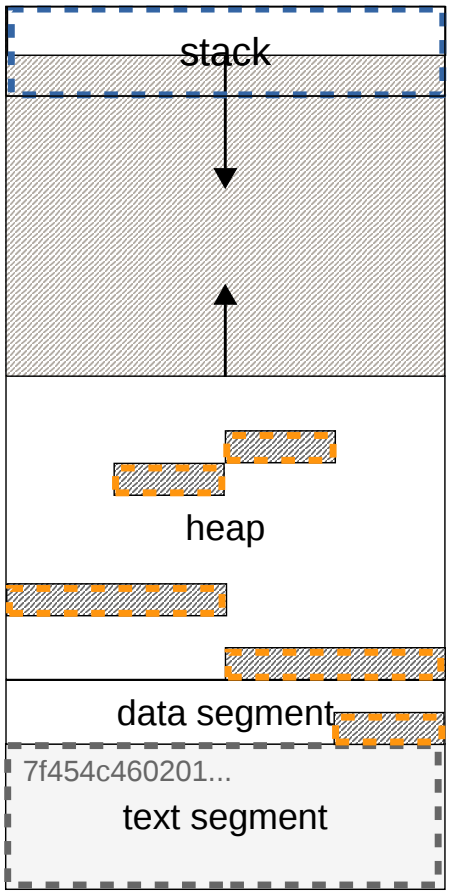


We've now seen most of the ingredients of a modern *virtual memory system*.

These systems are often described as *demand-paged*, meaning

- they transfer page-sized chunks of memory
- . . . *on demand*, i.e. when a fault comes in
- . . . the *lazy* approach!

Demand-paged virtual memory



Being lazy is often a good choice
for performance, but it's not straightforward.

- fault handling makes processes wait on I/O
- frequent swapping *really* hurts performance
- being more eager (less lazy) could save time
- . . . *if* we can accurately
  predict which other pages will be needed!

Proactively transferring
data *before* it is needed is called *prefetching*.

Modern systems combine demand
paging with judicious prefetching, often based on
heuristics (simple) or *predictive models* (fancier).

Some further ingredients

Block cache: for programs accessing *files* on disk, it's good to keep an in-memory cache of those blocks (disk is *much* slower than memory).

. . . but be careful: don't also cache the *swap file*, if we are using a file for our demand-paging swap area!

Memory-mapped files: it's a convenient programming abstraction to have *files on disk* appear as *data in memory*. We can demand-page the file data from/to disk.

. . . a kind of inverse of when we swap out memory to disk (≈ 'file-mapped memory'). Don't additionally cache the file in the block cache!

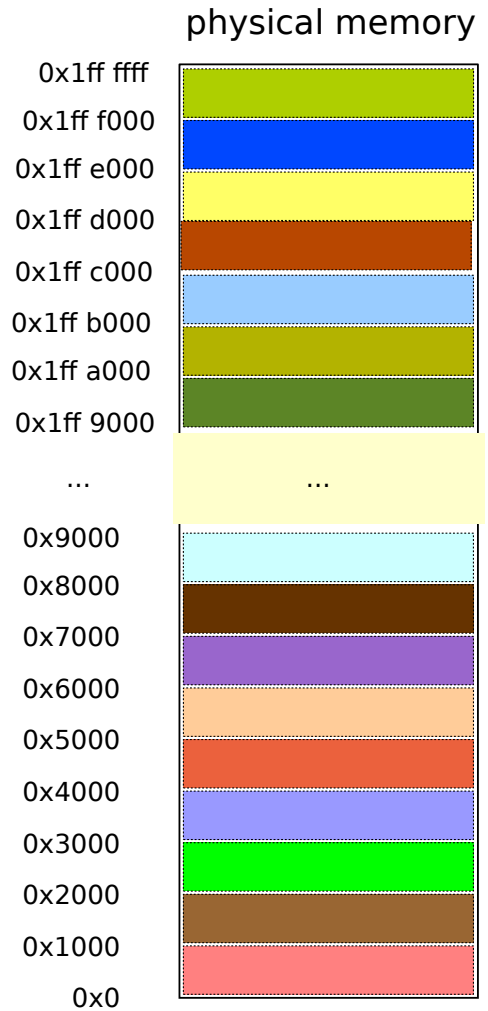
Resolving interactions between these: a *unified virtual memory system* treats cacheable disk I/O and swappable memory as (roughly) the same thing, avoiding the 'don'ts' we just described.

Where we are: further virtualisation of memory

- how page faults enable ‘illusions’
- virtual memory, by demand paging
- **page replacement, frame allocation**

Using resources wisely

Virtual memory becomes useful when physical memory is full.*



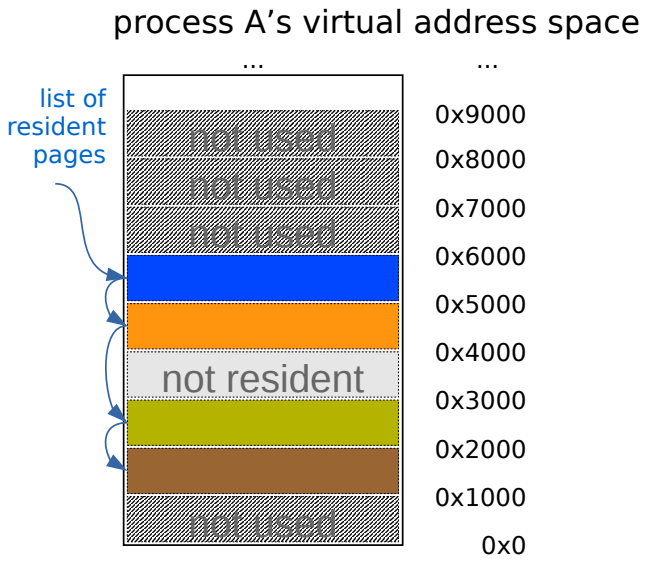
When this happens, we have difficult decisions.

- Which page(s) are the best choice for swapping out to disk, in order to free up memory? This is called **page replacement**.
- How many frames of physical memory does each process get? This is called **frame allocation**.

* It's unwise to let physical memory get *completely* full at any one time. What if the OS needs to allocate some memory for internal use, just to make progress itself, e.g. in order to carry out page replacement?

Page replacement

If a process needs more memory and is at its maximum frame allocation, pick one of *its own* pages to replace. But which?



Page replacement is analogous to replacement in cache memory (which you have studied)

. . . but implemented in software, in the OS.

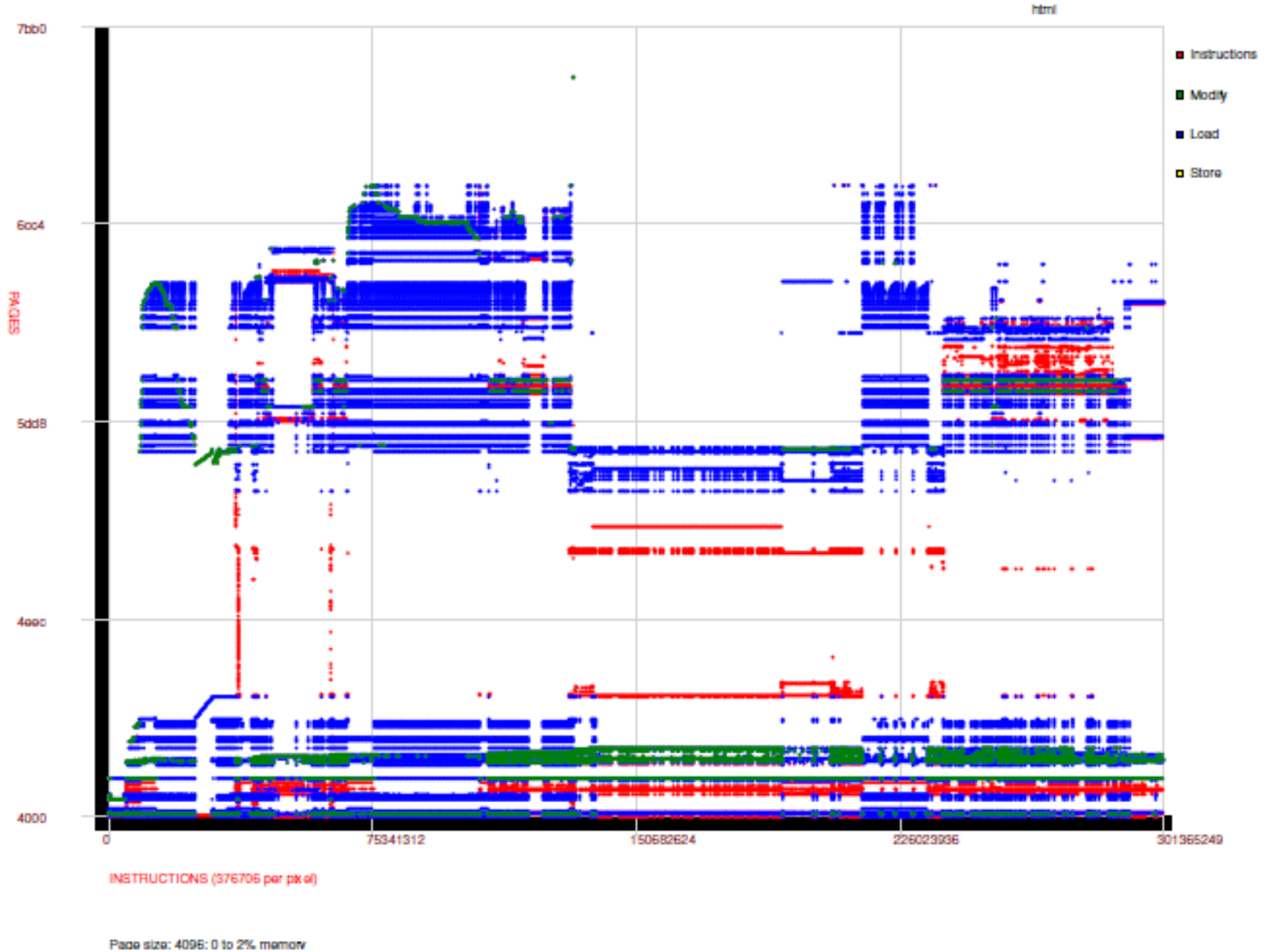
Imagine that each resident page is conceptually stored in a linked list (kept by the OS). Several common replacement policies. . .

FIFO: treat list as a queue, pick from the head ('old end'; 'least recently added'!)

Random: pick randomly

Least recently used (LRU): like FIFO, but somehow keep the list sorted in order of *use timestamp*. When did the process last access this page?

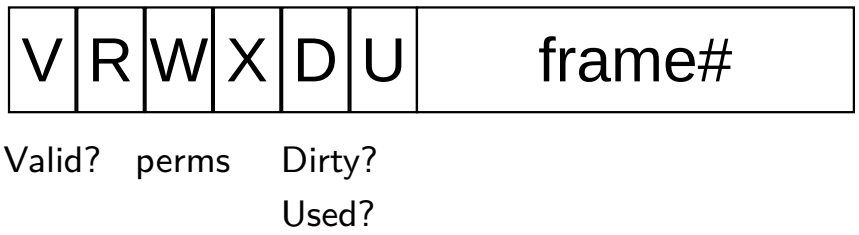
Why LRU is a good idea: locality of reference



Source: Adrian McMenamin, 'Applying Working Set Heuristics to the Linux Kernel', MSc thesis, Birkbeck College, London, 2011

Why we can't implement LRU: hardware

Hardware doesn't give us timestamps! It just gives us (roughly) this.



Let's make
use of the Used bit, sometimes
called the Referenced bit.

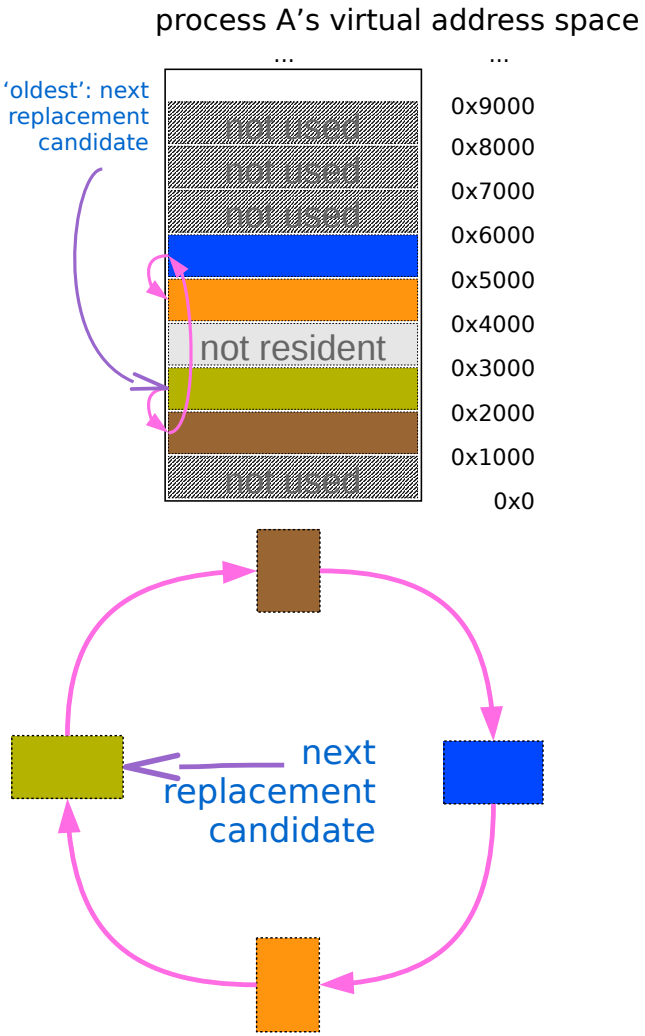
The hardware sets this bit whenever this page is used.*
It additionally sets the Dirty bit whenever the page is written to.

The OS periodically *clears* all Used bits. So if the bit is set, we know the page was used *recently*. (But we don't know *how recently*, relative to others that also have the bit set.)

* Strictly speaking, the bit gets set in the TLB entry, then asynchronously flushed to the PTE in main memory.

Our best approximation of LRU

We need to pick a page, preferring those with a clear Used bit. To break ties, we can at best degenerate to Random or FIFO.



A common approach: **'second-chance FIFO'**. Walk the list from oldest to newest, like FIFO. If we see a non-Used page, choose it. If we see a Used page, give it a 'second chance': clear its U bit and keep walking, *sending it to the back ('new end') of the list*.

To optimise the implementation, instead use a circular list with a 'current position' pointer, from where we resume our search each time we need to pick. This is called **Clock** and avoids an 'expensive' send-to-back list manipulation.

Frame allocation

efficiency of one process's execution

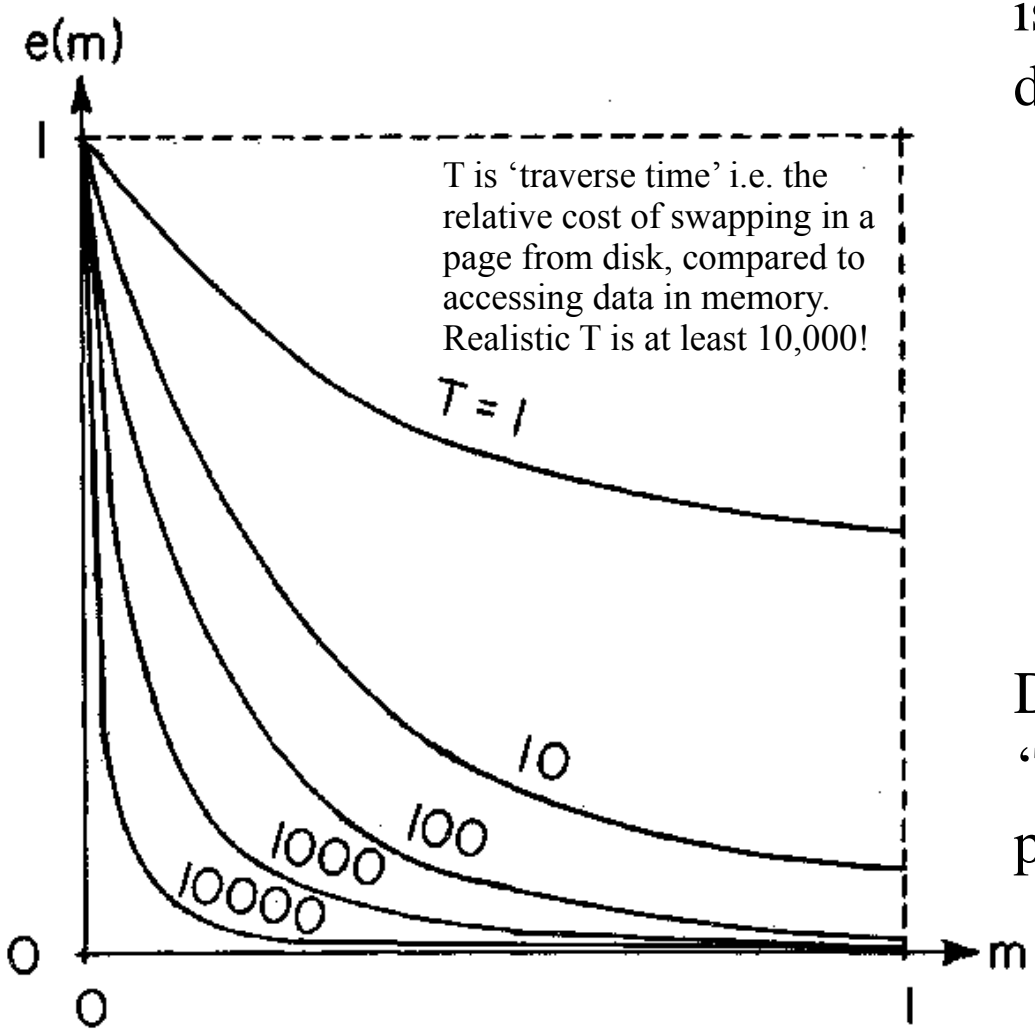


FIGURE 4—Efficiency

probability of a 'miss'
i.e. of needing to swap in a page on any memory access

When physical memory is scarce, how many frames do we give to each process?

- Taking a page fault and swapping in a page from disk is *really slow*
- If done frequently, performance hits the floor

Diagram from P.J. Denning, 'Thrashing: its causes and prevention', Proc. AFIPS, 1968

Frame allocation

efficiency of one process's execution

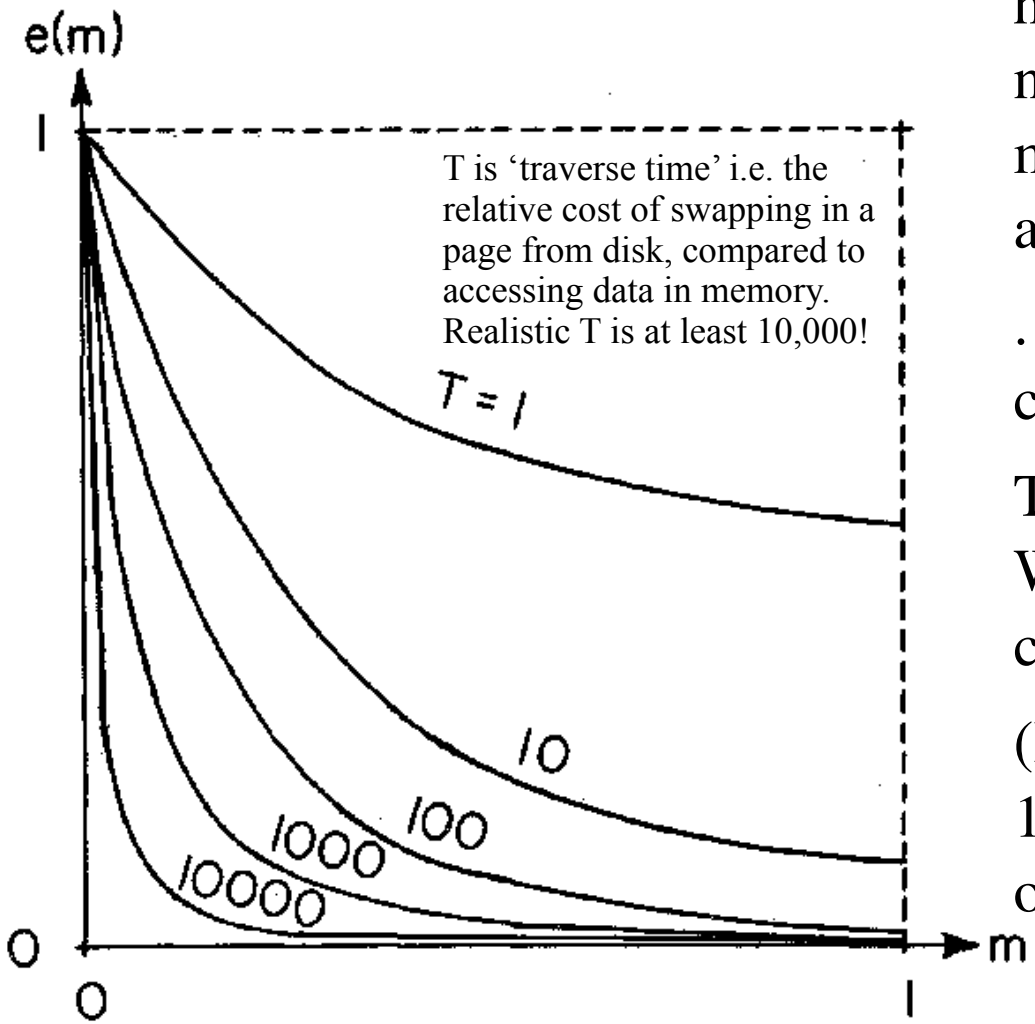


FIGURE 4—Efficiency

probability of a 'miss'
i.e. of needing to swap in a page on any memory access

Conceptually, each process has a set of pages that it is 'using now', i.e. that it needs in order to make uninterrupted progress over a small time interval (say 100ms)

. . . its *working set*; size of this is called the working set size (WSS)

To estimate WSS: count *Used bits* after clearing them (say) 100ms ago

(Imaging a cutting a vertical 100ms slice through the 'locality of reference' plot on p164.)

Frame allocation

efficiency of one process's execution

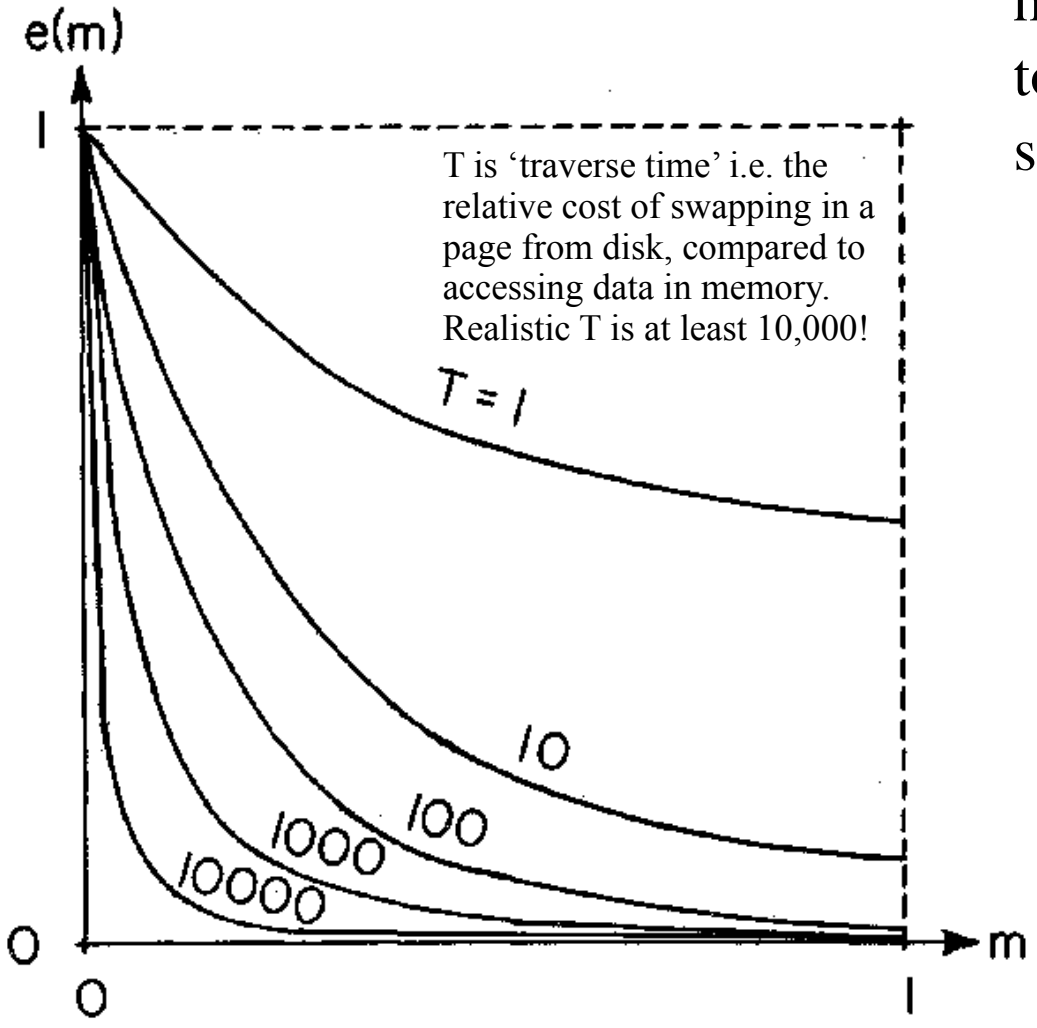


FIGURE 4—Efficiency

probability of a 'miss'
i.e. of needing to swap in a page on any memory access

When physical memory is scarce, how many frames do we give to each process? In short, 'equal shares' is not a good policy!

- when all processes get less than their WSS, system throughput will drop to nearly zero!
- This is called *thrashing*: #runnable plummets and the system is spending ~all its time waiting on I/O

Frame allocation

efficiency of one process's execution

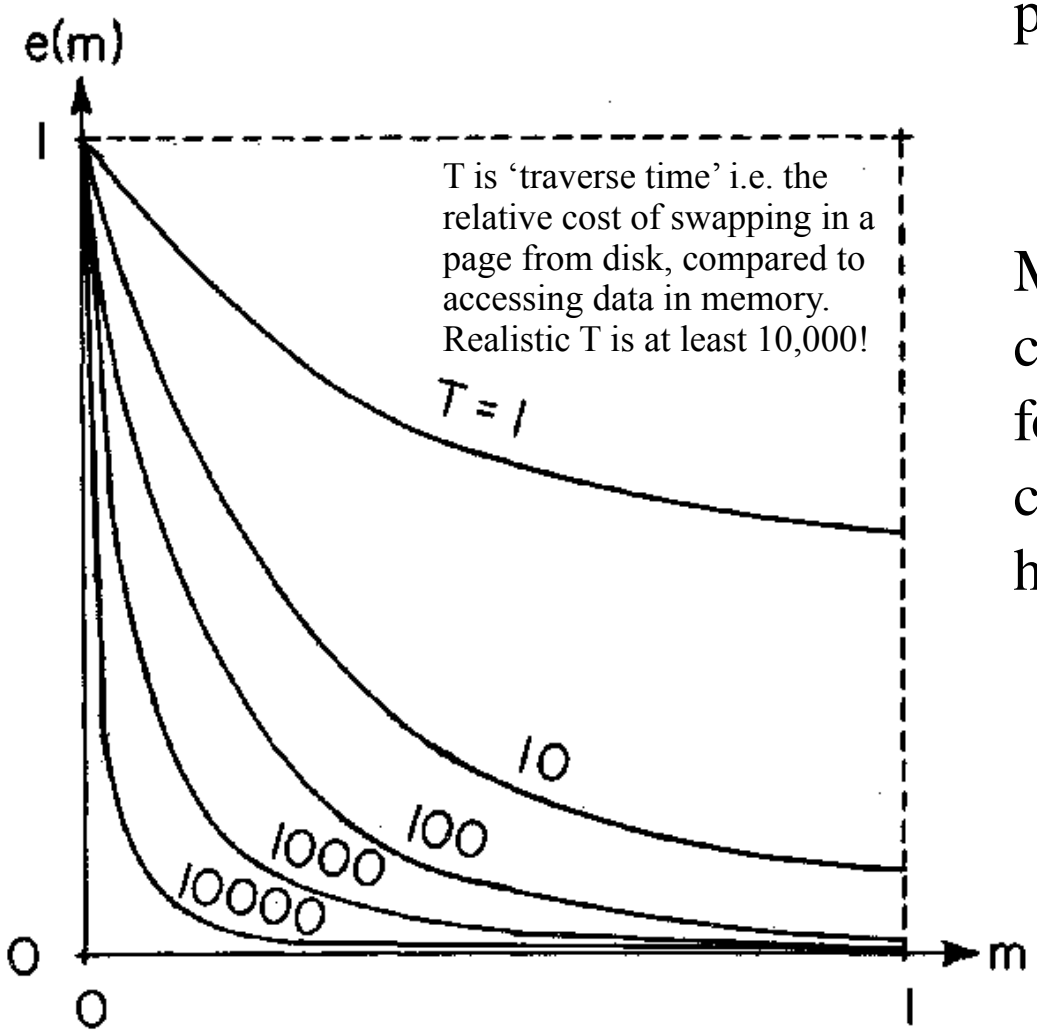


FIGURE 4—Efficiency

probability of a 'miss'
i.e. of needing to swap in a page on any memory access

Better to give *some* processes at least WSS frames.

- Others will have to wait!

Many other complementary strategies exist for managing out-of-memory conditions; e.g. Linux has 'OOM killer' (big hammer!)