[06] PAGING

OUTLINE

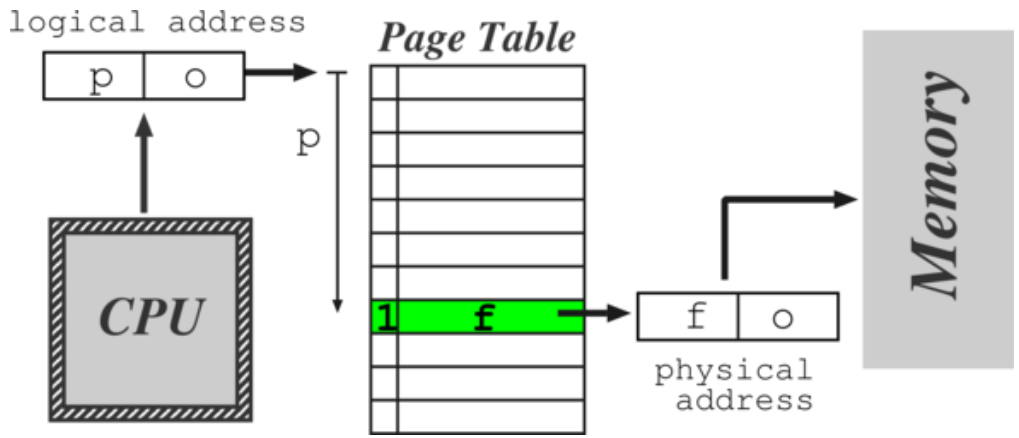
- Paged Virtual Memory
  - Concepts
  - Pros and Cons
  - Page Tables
  - Translation Lookaside Buffer (TLB)
  - Protection & Sharing
- Virtual Memory
  - Demand Paging Details
  - Page Replacement
  - Page Replacement Algorithms
- Performance
  - Frame Allocation
  - Thrashing & Working Set
  - Pre-paging
  - Page Sizes

PAGED VIRTUAL MEMORY

- **Paged Virtual Memory**
  - **Concepts**
  - **Pros and Cons**
  - **Page Tables**
  - **Translation Lookaside Buffer (TLB)**
  - **Protection & Sharing**
- Virtual Memory
- Performance

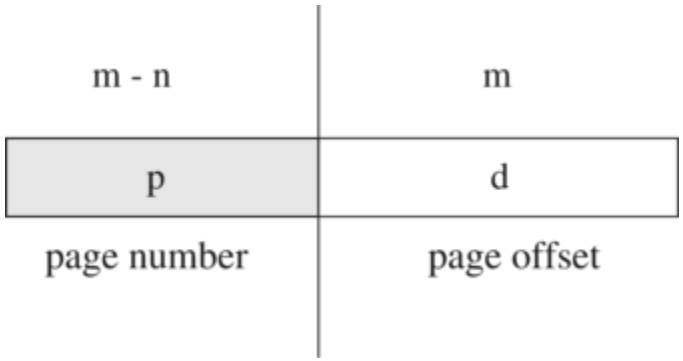
PAGED VIRTUAL MEMORY

Another solution is to allow a process to exist in non-contiguous memory, i.e.,



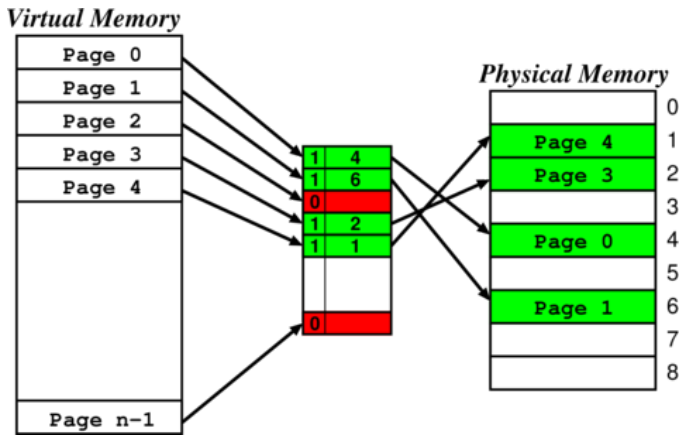
- Divide **physical** memory into **frames**, small fixed-size blocks
- Divide **logical** memory into **pages**, blocks of the same size (typically 4kB)
- Each CPU-generated address is a page number p with page offset o
- Page table contains associated frame number f
- Usually have $|p| \gg |f|$ so also record whether mapping valid

PAGING PROS AND CONS



- Hardware support required — frequently defines the page size, typically a power of 2 (making address fiddling easy) ranging from 512B to 8192B (0.5kB — 8kB)
- Logical address space of $2m$ and page size $2n$ gives $p = m - n$ bits and $o = m$ bits
- Note that paging is itself a form of dynamic relocation: simply change page table to reflect movement of page in memory. This is similar to using a set of *base* + *limit* registers for each page in memory

PAGING PROS AND CONS



- Memory allocation becomes easier but OS must maintain a page table per process
  - No external fragmentation (in physical memory at least), but get internal fragmentation: a process may not use all of final page
  - Indicates use of small page sizes — but there's a significant per-page overhead: the **Page Table Entries** (PTEs) themselves, plus that disk IO is more efficient with larger pages
  - Typically 2 — 4kB nowadays (memory is cheaper)

PAGING PROS AND CONS

- Clear separation between user (process) and system (OS) view of memory usage
  - Process sees single logical address space; OS does the hard work
  - Process cannot address memory they don't own — cannot reference a page it doesn't have access to
  - OS can map system resources into user address space, e.g., IO buffer
  - OS must keep track of free memory; typically in **frame table**
- Adds overhead to context switching
  - Per process page table must be mapped into hardware on context switch
  - The page table itself may be large and extend into physical memory

PAGE TABLES

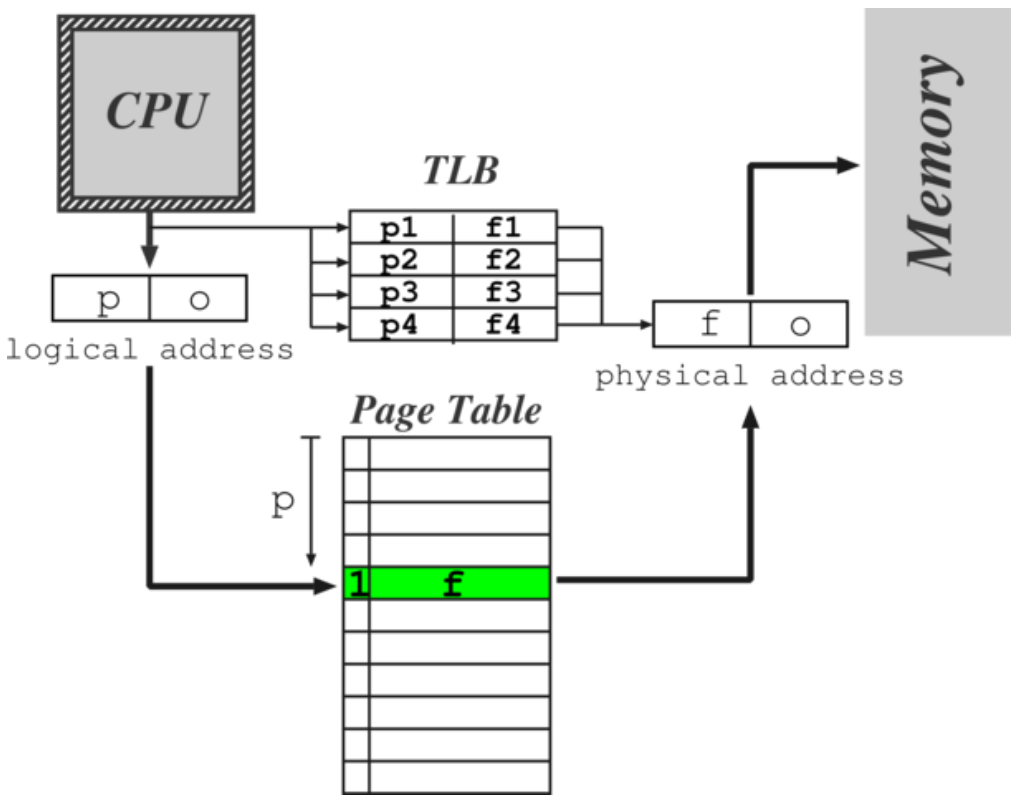
Page Tables (PTs) rely on hardware support:

- Simplest case: set of dedicated relocation registers
  - One register per page, OS loads registers on context switch
  - E.g., PDP-11 16 bit address, 8kB pages thus 8 PT registers
  - Each memory reference goes through these so they must be fast
- Ok for small PTs but what if we have many pages (typically $O(10^6)$)
  - Solution: Keep PT in memory, then just one MMU register needed, the **Page Table Base Register** (PTBR)
  - OS switches this when switching process
- Problem: PTs might still be very big
  - Keep a **PT Length Register** (PTLR) to indicate size of PT
  - Or use a more complex structure (see later)
- Problem: need to refer to memory twice for every "actual" memory reference
  - Solution: use a **Translation Lookaside Buffer** (TLB)

TLB OPERATION

When memory is referenced, present TLB with logical memory address

- If PTE is present, get an immediate result
- Otherwise make memory reference to PTs, and update the TLB
- Latter case is typically 10% slower than direct memory reference



TLB ISSUES

As with any cache, what to do when it's full, how are entries shared?

- If full, discard entries typically Least Recently Used (LRU) policy
- Context switch requires TLB flush to prevent next process using wrong PTEs — Mitigate cost through **process tags** (how?)

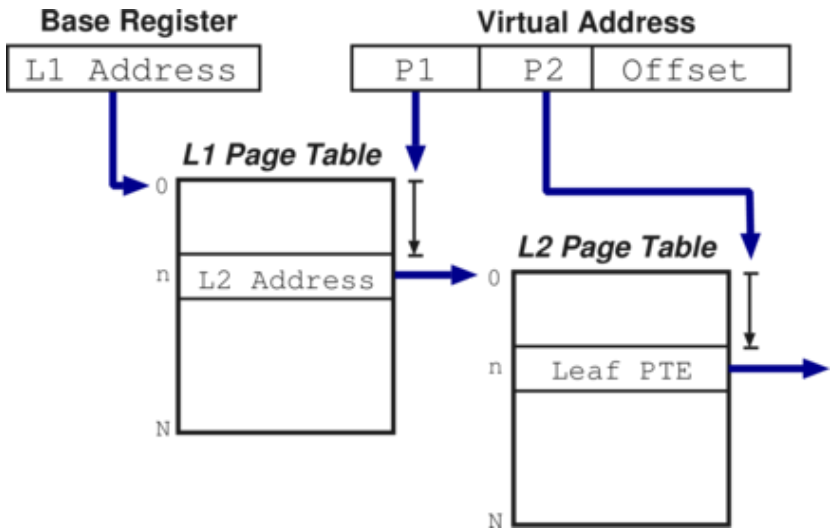
Performance is measured in terms of **hit ratio**, proportion of time a PTE is found in TLB. Example:

- Assume TLB search time of 20ns, memory access time of 100ns, hit ratio of 80%
- Assume one memory reference required for lookup, what is the **effective memory access time**?
  - 0.8 x 120 + 0.2 x 220 = 140 ns
- Now increase hit ratio to 98% — what is the new effective access time?
  - 0.98 x 120 + 0.02 x 220 = 122 ns — just a 13% improvement
  - (Intel 80486 had 32 registers and claimed a 98% hit ratio)

MULTILEVEL PAGE TABLES

Most modern systems can support very large ($2^{32}, 2^{64}$) address spaces, leading to very large PTs which we don't really want to keep all of in main memory

Solution is to split the PT into several sub-parts, e.g., two, and then page the page table:



- Divide the page number into two parts

  e.g., 20 bit **page number**, 12 bit **page offset**

- Then divide the page number into **outer** and **inner** parts of 10 bits each

EXAMPLE: VAX

A 32 bit architecture with 512 byte pages:

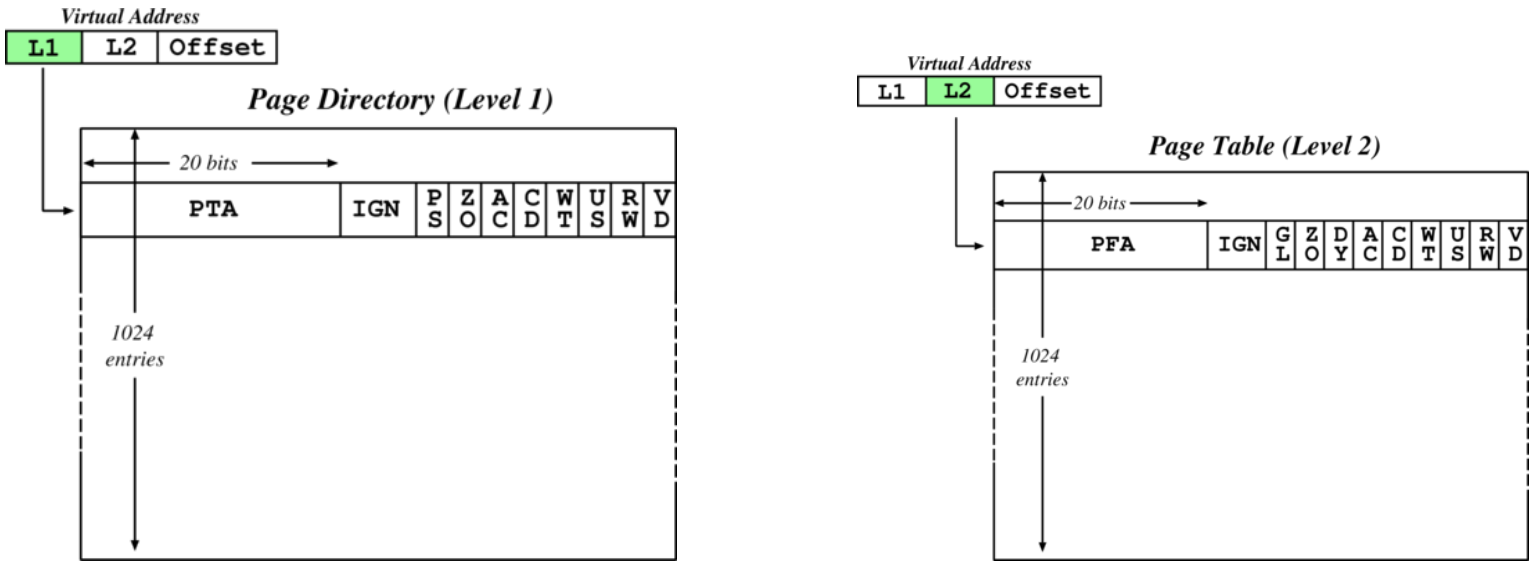
- Logical address space divided into 4 sections of 2^{30} bytes
- Top 2 address bits designate **section**
- Next 21 bits designate **page** within section
- Final 9 bits designate **page offset**
- For a VAX with 100 pages, one level PT would be 4MB; with sectioning, it's 1MB

For 64 bit architectures, two-level paging is not enough: add further levels.

- For 4kB pages need 2^{52} entries in PT using 1 level PT
- For 2 level PT with 32 bit outer PT, we'd still need 16GB for the outer PT

Even some 32 bit machines have > 2 levels: SPARC (32 bit) has 3 level paging scheme; 68030 has 4 level paging

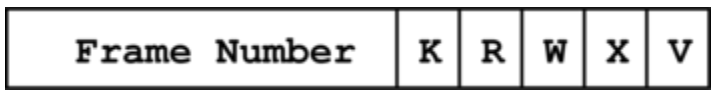
EXAMPLE: X86



Page size is 4kB or 4MB. First lookup to the **page directory**, indexed using top 10 bits. The page directory address is stored in an internal processor register (`cr3`). The lookup results (usually) in the address of a page table. Next 10 bits index the **page table**, retrieving the **page frame address**. Finally, add in the low 12 bits as the **page offset**. Note that the page directory and page tables are exactly one page each themselves (not by accident)

PROTECTION ISSUES

Associate **protection bits** with each page, kept in page tables (and TLB), e.g. one bit for read, one for write, one for execute (`RWX`). Might also distinguish whether may only be accessed when executing in kernel mode, e.g.,



As the address goes through the page hardware, can check protection bits — though note this only gives *page granularity* protection, not byte granularity

Any attempt to violate protection causes hardware trap to operating system code to handle. The entry in the TLB will have a valid/invalid bit indicating whether the page is mapped into the process address space. If invalid, trap to the OS handler to map the page

Can do lots of interesting things here, particularly with regard to sharing, virtualization, ...

SHARED PAGES

Another advantage of paged memory is code/data sharing, for example:

- Binaries: editor, compiler etc.
- Libraries: shared objects, DLLs

So how does this work?

- Implemented as two logical addresses which map to one physical address
- If code is re-entrant (i.e. stateless, non-self modifying) it can be easily shared between users
- Otherwise can use copy-on-write technique:
  - Mark page as read-only in all processes
  - If a process tries to write to page, will trap to OS fault handler
  - Can then allocate new frame, copy data, and create new page table mapping
- (May use this for lazy data sharing too)

Requires additional book-keeping in OS, but worth it, e.g., many hundreds of MB shared code on this laptop. (Though nowadays, see unikernels!)

VIRTUAL MEMORY

- Paged Virtual Memory
- **Virtual Memory**
  - **Demand Paging Details**
  - **Page Replacement**
  - **Page Replacement Algorithms**
- Performance

VIRTUAL MEMORY

Virtual addressing allows us to introduce the idea of **virtual memory**

- Already have valid or invalid page translations; introduce "non-resident designation and put such pages on a non-volatile backing store
- Processes access non-resident memory just as if it were "the real thing"

Virtual Memory (VM) has several benefits:

- **Portability**: programs work regardless of how much actual memory present; programs can be larger than physical memory
- **Convenience**: programmer can use e.g. large sparse data structures with impunity; less of the program needs to be in memory at once, thus potentially more efficient multi-programming, less IO loading/swapping program into memory
- **Efficiency**: no need to waste (real) memory on code or data which isn't used (e.g., error handling)

VM IMPLEMENTATION

Typically implemented via demand paging:

- Programs (executables) reside on disk
- To execute a process we load pages in on demand; i.e. as and when they are referenced
- Also get demand segmentation, but rare (eg., Burroughs, OS/2) as it's more difficult (segment replacement is much harder due to segments having variable size)

DEMAND PAGING DETAILS

When loading a new process for execution:

- Create its address space (page tables, etc)
- Mark PTEs as either invalid or non-resident
- Add PCB to scheduler

Then whenever we receive a page fault, check PTE:

- If due to invalid reference, kill process
- Otherwise due to non-resident page, so "page in" the desired page:
  - Find a free frame in memory
  - Initiate disk IO to read in the desired page
  - When IO is finished modify the PTE for this page to show that it is now valid
  - Restart the process at the faulting instruction

DEMAND PAGING: ISSUES

Above process makes the fault invisible to the process, but:

- Requires care to save process state correctly on fault
- Can be particularly awkward on a CPU with pipelined decode as we need to wind back (e.g., MIPS, Alpha)
- Even worse on on CISC CPU where single instruction can move lots of data, possibly across pages — we can't restart the instruction so rely on help from microcode (e.g., to test address before writing). Can possibly use temporary registers to store moved data
- Similar difficulties from auto-increment/auto-decrement instructions, e.g., ARM
- Can even have instructions and data spanning pages, so multiple faults per instruction; though **locality of reference** tends to make this infrequent

Scheme described above is pure demand paging: don't bring in pages until needed so get lots of page faults and IO when process begins; hence many real systems explicitly load some core parts of the process first

PAGE REPLACEMENT

To page in from disk, we need a free frame of physical memory to hold the data we're reading in — but in reality, the size of physical memory is limited so either:

- Discard unused pages if total demand for pages exceeds physical memory size
- Or swap out an entire process to free some frames

Modified algorithm: on a page fault we:

1. Locate the desired replacement page on disk
2. Select a free frame for the incoming page:
    1. If there is a free frame use it, otherwise select a victim page to free
    2. Then write the victim page back to disk
    3. Finally mark it as invalid in its process page tables
3. Read desired page into the now free frame
4. Restart the faulting process

...thus, having no frames free effectively doubles the page fault service time

PAGE REPLACEMENT

Can reduce overhead by adding a "dirty" bit to PTEs

- Can allow us to omit step (2.2) above by only writing out page was modified, or if page was read-only (e.g., code)

How do we choose our victim page?

- A key factor in an efficient VM system: evicting a page that we'll need in a few instructions time can get us into a really bad condition
- We want to ensure that we get few page faults overall, and that any we do get are relatively quick to satisfy

We will now look at a few **page replacement algorithms**:

- All aim to minimise page fault rate
- Candidate algorithms are evaluated by (trace driven) simulation using reference strings

PAGE REPLACEMENT ALGORITHMS

FIRST-IN FIRST-OUT (FIFO)

Keep a queue of pages, discard from head. Performance is hard to predict as we've no idea whether replaced page will be used again or not: eviction is independent of page use frequency. In general this is very simple but pretty bad:

- Can actually end up discarding a page currently in use, causing an immediate fault and next in queue to be replaced — really slows system down
- Possible to have more faults with increasing number of frames (**Belady's anomaly**)

OPTIMAL ALGORITHM (OPT)

Replace the page which will not be used again for longest period of time. Can only be done with an oracle or in hindsight, but serves as a good baseline for other algorithms

LEAST RECENTLY USED (LRU)

Replace the page which has not been used for the longest amount of time. Equivalent to OPT with time running backwards. Assumes that the past is a good predictor of the future. Can still end up replacing pages that are about to be used

Generally considered quite a good replacement algorithm, though may require substantial hardware assistance

But! How do we determine the LRU ordering?

IMPLEMENTING LRU: COUNTERS

- Give each PTE a time-of-use field and give the CPU a logical clock (counter)
- Whenever a page is referenced, its PTE is updated to clock value
- Replace page with smallest time value

Problems:

- Requires a search to find minimum counter value
- Adds a write to memory (PTE) on every memory reference
- Must handle clock overflow

Impractical on a standard processor

IMPLEMENTING LRU: PAGE STACK

- Maintain a stack of pages (doubly linked list) with most-recently used (MRU) page on top
- Discard from bottom of stack

Problem:

- Requires changing (up to) 6 pointers per [new] reference (max 6 pointers)
- This is very slow without extensive hardware support

Also impractical on a standard processor

APPROXIMATING LRU

Many systems have a reference bit in the PTE, initially zeroed by OS, and then set by hardware whenever the page is touched. After time has passed, consider those pages with the bit set to be **active** and implement **Not Recently Used** (NRU) replacement:

- Periodically (e.g. 20ms) clear all reference bits
- When choosing a victim to evict, prefer pages with clear reference bits
- If also have a **modified** or **dirty bit** in the PTE, can use that too

Referenced?	Dirty?	Comment
no	no	best type of page to replace
no	yes	next best (requires writeback)
yes	no	probably code in use
yes	yes	bad choice for replacement

IMPROVING THE APPROXIMATION

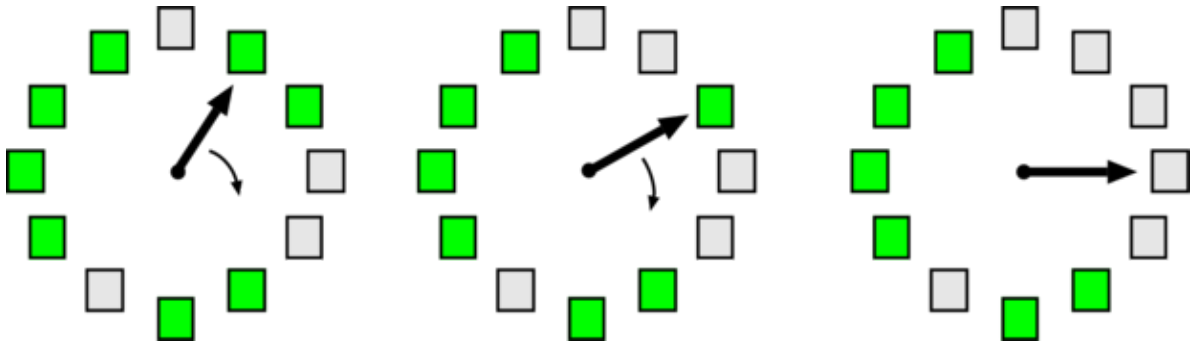
Instead of just a single bit, the OS:

- Maintains an 8-bit value per page, initialised to zero
- Periodically (e.g. 20ms) shifts reference bit onto high order bit of the byte, and clear reference bit

Then select lowest value page (or one of) to replace

- Keeps the history for the last 8 clock sweeps
- Interpreting bytes as `u_ints`, then LRU page is min(*additional_bits*)
- May not be unique, but gives a candidate set

FURTHER IMPROVMENT: SECOND-CHANCE FIFO



- Store pages in queue as per FIFO
- Before discarding head, check reference bit
- If reference bit is 0, discard else reset reference bit, and give page a second chance (add it to tail of queue)

Guaranteed to terminate after at most one cycle, with the worst case having the second chance devolve into a FIFO if all pages are referenced. A page given a second chance is the last to be replaced

IMPLEMENTING SECOND-CHANCE FIFO

Often implemented with a circular queue and a current pointer; in this case usually called **clock**

If no hardware is provided, reference bit can emulate:

- To clear "reference bit", mark page no access
- If referenced then trap, update PTE, and resume
- To check if referenced, check permissions
- Can use similar scheme to emulate modified bit

OTHER REPLACEMENT SCHEMES

Counting Algorithms: keep a count of the number of references to each page

- **Least Frequently Used** (LFU): replace page with smallest count
  - Takes no time information into account
  - Page can stick in memory from initialisation
  - Need to periodically decrement counts
- **Most Frequently Used** (MFU): replace highest count page
  - Low count indicates recently brought in

PAGE BUFFERING ALGORITHMS

- Keep a minimum number of victims in a free pool
- New page read in before writing out victim, allowing quicker restart of process
- Alternative: if disk idle, write modified pages out and reset dirty bit
  - Improves chance of replacing without having to write dirty page

(Pseudo) MRU: Consider accessing e.g. large array.

- The page to replace is one application has just finished with, i.e. most recently used
- Track page faults and look for sequences
- Discard the kth in victim sequence

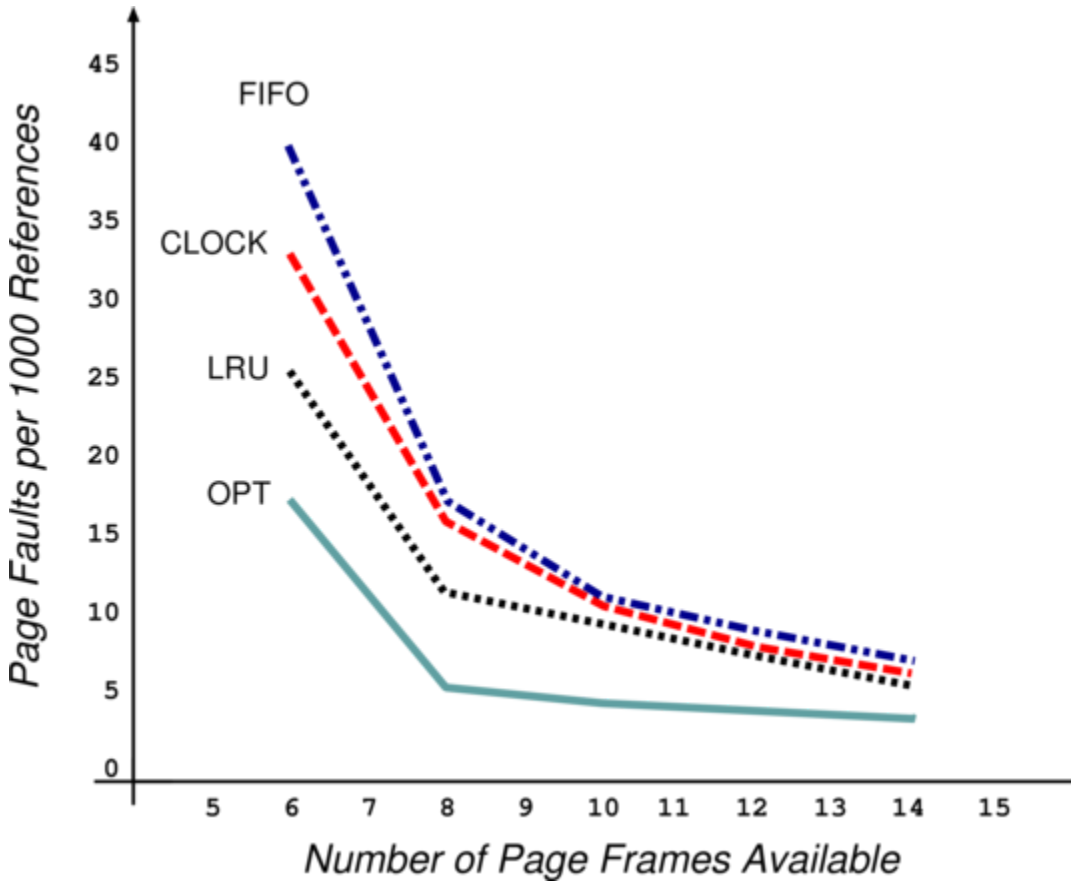
Application-specific: stop trying to second-guess what's going on and provide hook for application to suggest replacement, but must be careful with denial of service

PERFORMANCE

- Paged Virtual Memory
- Virtual Memory
- **Performance**
  - **Frame Allocation**
  - **Thrashing & Working Set**
  - **Pre-paging**
  - **Page Sizes**

PERFORMANCE COMPARISON

This plot shows page-fault rate against number of physical frames for a pseudo-local reference string (note offset *x* origin). We want to minimise area under curve. FIFO could exhibit *Belady's Anomaly* (but doesn't here). Can see that getting frame allocation right has major impact — much more than which algorithm you use!



FRAME ALLOCATION

A certain fraction of physical memory is reserved per-process and for core OS code and data. Need an allocation policy to determine how to distribute the remaining frames. Objectives:

- Fairness (or proportional fairness)?
  - E.g. divide m frames between n processes as m/n, remainder in free pool
  - E.g. divide frames in proportion to size of process (i.e. number of pages used)
- Minimize system-wide page-fault rate?
  - E.g. allocate all memory to few processes
- Maximize level of multiprogramming?
  - E.g. allocate min memory to many processes

Could also allocate taking process priorities into account, since high-priority processes are supposed to run more readily. Could even care which frames we give to which process ("page colouring")

FRAME ALLOCATION: GLOBAL SCHEMES

Most page replacement schemes are global: all pages considered for replacement

- Allocation policy implicitly enforced during page-in
- Allocation succeeds iff policy agrees
- *Free frames* often in use so steal them!

For example, on a system with 64 frames and 5 processes:

- If using fair share, each processes will have 12 frames, with four left over (maybe)
- When a process dies, when the next faults it will succeed in allocating a frame
- Eventually all will be allocated
- If a new process arrives, need to steal some pages back from the existing allocations

FRAME ALLOCATION: COMPARISON TO LOCAL

Also get **local page replacement schemes**: victim always chosen from within process

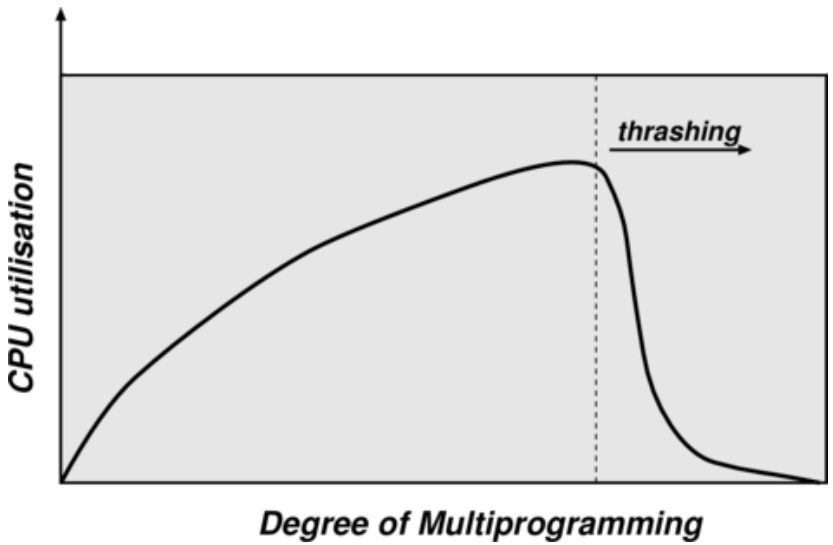
In global schemes the process cannot control its own page fault rate, so performance may depend entirely on what other processes page in/out

In local schemes, performance depends only on process behaviour, but this can hinder progress by not making available less/unused pages of memory

Global are optimal for throughput and are the most common

THE RISK OF *THRASHING*

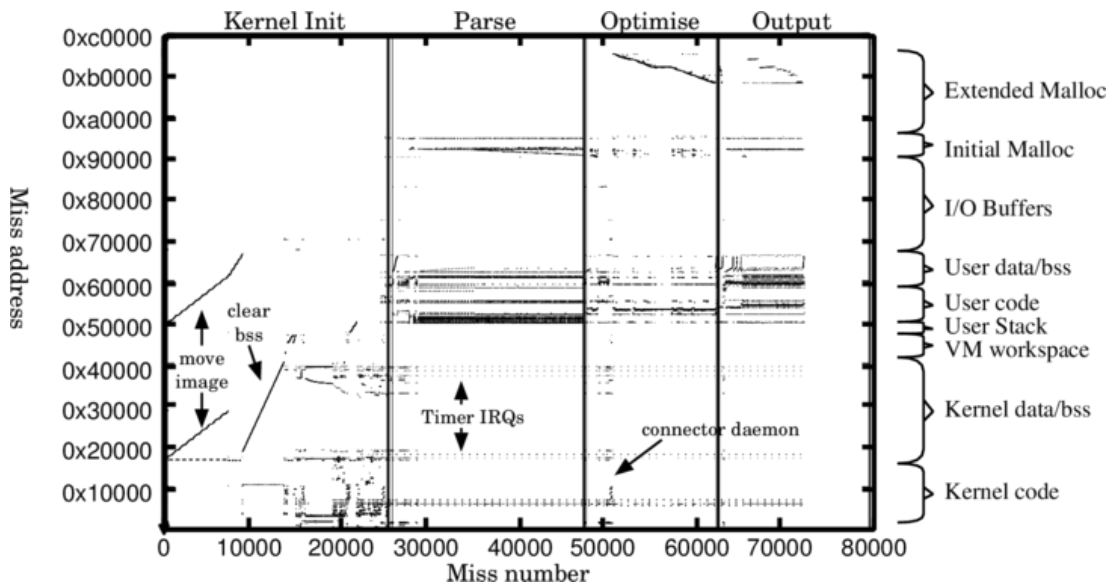
More processes entering the system causes the **frames-per-process** allocated to reduce. Eventually we hit a wall: processes end up stealing frames from each other, but then need them back so fault. Ultimately the number of runnable processes plunges



A process can *technically* run with minimum-free frames available but will have a very high page fault rate. If we're very unlucky, OS monitors CPU utilisation and increases level of multiprogramming if utilisation is too low: machine dies

Avoid thrashing by giving processes as many frames as they "need" and, if we can't, we must reduce the MPL — a better page-replacement algorithm will not help

LOCALITY OF REFERENCE



In a short time interval, the locations referenced by a process tend to be grouped into a few regions in its address space:

- Procedure being executed
- Sub-procedures
- Data access
- Stack variables

AVOIDING THRASHING

We can use the locality of reference principle to help determine how many frames a process needs:

- Define the **Working Set** (WS) (Denning, 1967)

  Set of pages that a process needs in store at "the same time" to make any progress

- Varies between processes and during execution
- Assume process moves through phases
- In each phase, get (spatial) locality of reference
- From time to time get phase shift

CALCULATING WORKING SET

Then OS can try to prevent thrashing by maintaining sufficient pages for current phase:

- Sample page reference bits every, e.g., 10ms
- Define window size Δ of most recent page references
- If a page is "in use", say it's in the working set
- Gives an approximation to locality of program
- Given the size of the working set for each process WSS_i, sum working set sizes to get total demand D
- If $D > m$ we are in danger of thrashing — suspend a process

This optimises CPU util but has the need to compute WSS_i (moving window across stream). Can approximate with periodic timer and some page reference script. After some number of intervals (i.e., of bits of state) consider pages with count < 0 to be in WS. In general, a working set can be used as a scheme to determine allocation for each process

PRE-PAGING

- Pure demand paging causes a large number of PF when process starts
- Can remember the WS for a process and pre-page the required frames when process is resumed (e.g. after suspension)
- When process is started can pre-page by adding its frames to free list
- Increases IO cost: How do we select a page size (given no hardware constraints)?

PAGE SIZES

- Trade-off the size of the PT and the degree of fragmentation as a result
- Typical values are 512B to 16kB — but should be reduce the numbers of queries, or ensure that the window is covered
- Larger page size means fewer page faults
  - Historical trend towards larger page sizes
  - Eg., 386: 4kB, 68030: 256B to 32kB

So, a page of 1kB, 56ms for 2 pages of 512B but smaller page allows us to watch locality more accurately. Page faults remain costly because CPU and memory much much faster than disk

SUMMARY

- Paged Virtual Memory
  - Concepts
  - Pros and Cons
  - Page Tables
  - Translation Lookaside Buffer (TLB)
  - Protection & Sharing
- Virtual Memory
  - Demand Paging Details
  - Page Replacement
  - Page Replacement Algorithms
- Performance
  - Frame Allocation
  - Thrashing & Working Set
  - Pre-paging
  - Page Sizes