Operating Systems Functions

Steven Hand

8 lectures for CST Ib and Diploma

Lent Term 2000

Handout 2

Memory Management

- Limited physical memory (DRAM), need space for:
  - operating system image
  - processes (text, data, heap, stack, ...)
  - I/O buffers
- Memory management subsystem deals with:
  - Support for address binding (i.e. loading, dynamic linking).
  - Allocation of limited physical resources.
  - Protection & sharing of 'components'.
  - Providing convenient abstractions.
- Quite complex to implement:
  - processor-, motherboard-specific.
  - trade-offs keep shifting.

Logical vs Physical Addresses (1)

Old systems directly accessed [physical] memory, which caused some problems, e.g.

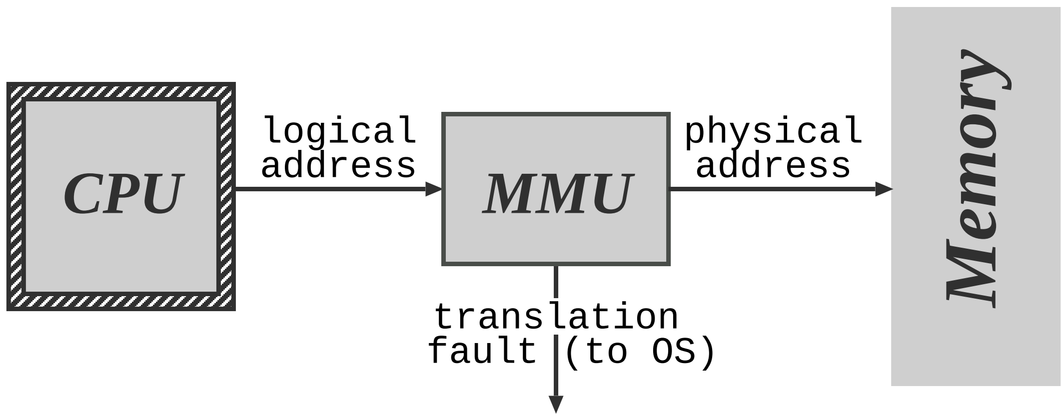
- Contiguous allocation:
  - need large lump of memory for process
  - with time, get [external] fragmentation

  ⇒ require expensive compaction

- Address binding (i.e. dealing with *absolute* addressing):
  - “`int x; x = 5;`” → “`movl $0x5, ????`”
  - compile time ⇒ must know load address.
  - load time ⇒ work every time.
  - what about swapping?

Can avoid lots of problems by separating concept of *logical* (virtual) and *physical* (“real”) addresses.

Logical vs Physical Addresses (2)

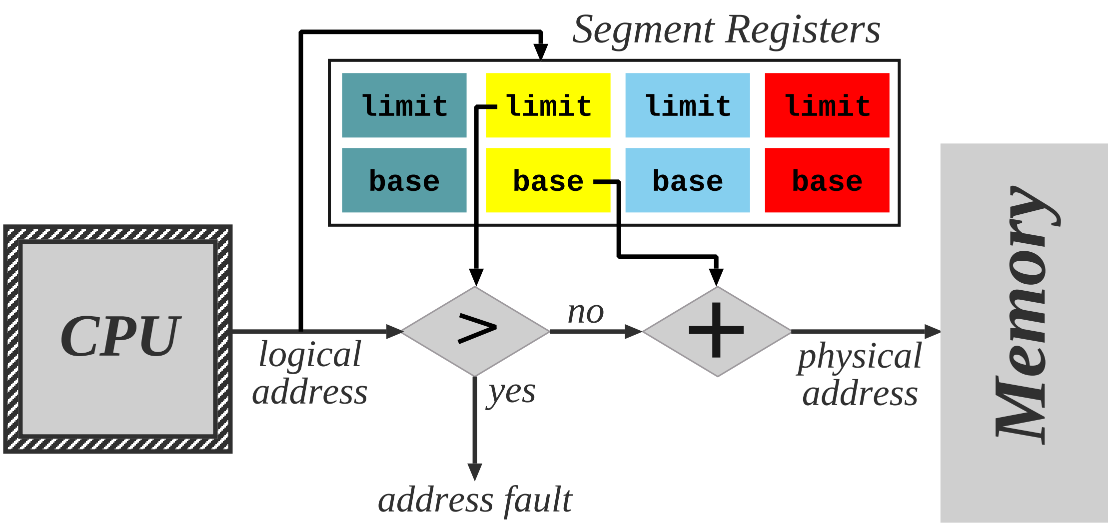


Run time mapping from logical to physical addresses.
If make this *per process* then:

- Each process has own *address space*.
- Allocation problem split:
  - virtual address allocation easy.
  - allocate physical memory 'behind the scenes'.
- Address binding solved:
  - bind to logical addresses at compile-time.
  - bind to real addresses at load time/run time.

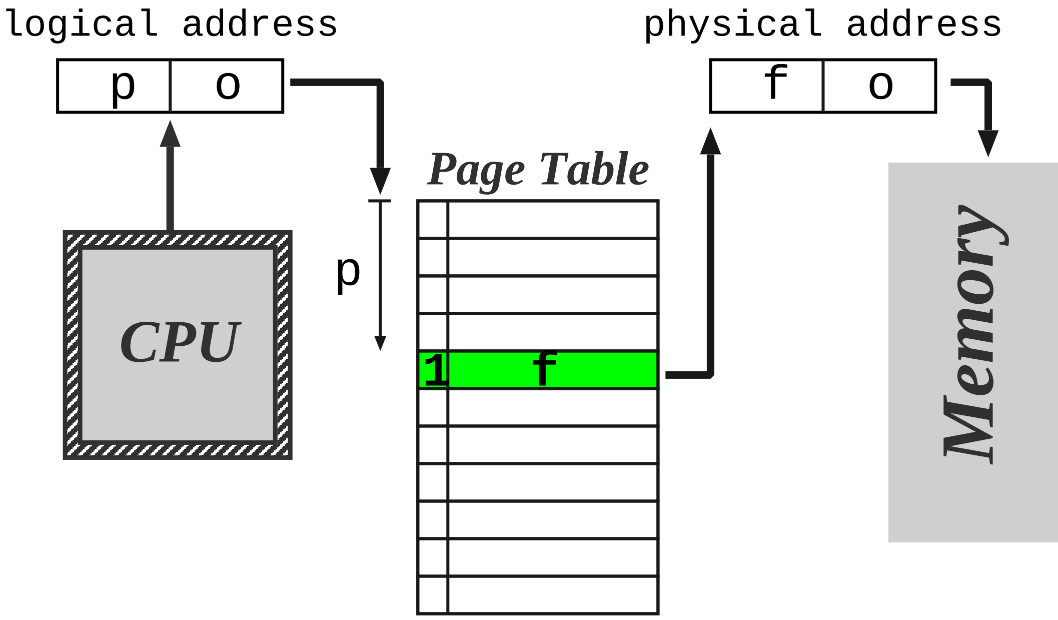
Two variants: segmentation and paging.

Segmentation



- MMU has a set (≥ 1) of segment registers.
- CPU issues tuple (s, o):
  1. MMU selects segment s.
  2. Checks $o \leq$ `limit`.
  3. If ok, forwards `base`$+o$ to memory controller.
- Nice logical view (protection & sharing)
- Problem: still have [external] fragmentation.

Paging



1. Physical memory: f frames each 2^s bytes.
2. Virtual memory: p pages each 2^s bytes.
3. *Page table* maps $\{0, \ldots, p-1\} \rightarrow \{0, \ldots, f-1\}$
4. Allocation problem has gone away!

Typically have $p \gg f \Rightarrow$ add *valid* bit to say if a given page is represented in physical memory.

Problem: now have *internal* fragmentation.

Problem: protection/sharing now per page.

Segmentation versus Paging

	logical view	allocation
Segmentation	✔	✘
Paging	✘	✔

⇒ try combined scheme.

- E.g. *paged segments* (Multics, OS/2)
  - divide each segment s_i into $k = \lceil l_i / 2^n \rceil$ pages, where l_i is the limit (length) of the segment.
  - have page table per segment.

  ✘ high hardware cost / complexity.

  ✘ not very portable.

- E.g. *software segments* (most modern OSs)
  - consider pages $[m, \ldots, m + l]$ to be a segment.
  - OS must ensure protection / sharing kept consistent over region.

  ✘ loss in granularity.

  ✔ relatively simple / portable.

Translation Lookaside Buffers

Typically #pages large $\Rightarrow$ page table lives in memory.

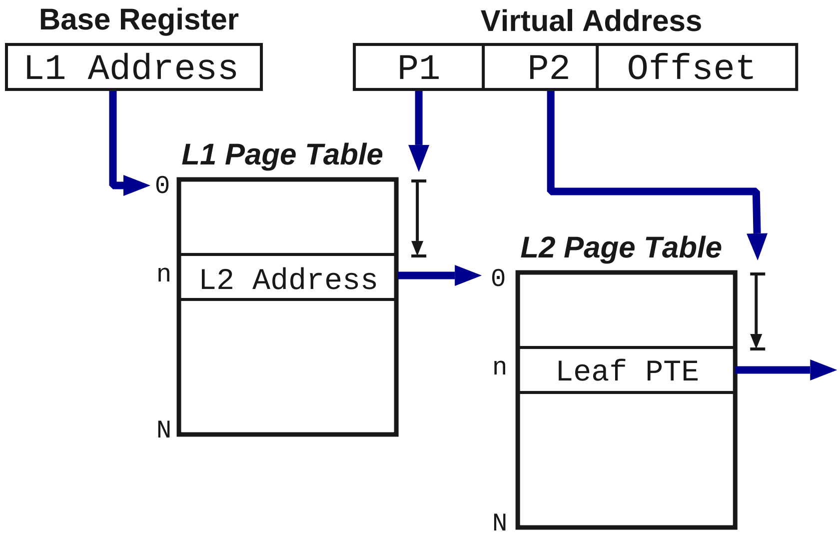
Add *TLB*, a fully associative cache for mapping info:

- Check each memory reference in TLB first.
- If miss $\Rightarrow$ need to load info from page table:
  - may be done in h/w or s/w (by OS).
  - if full, replace entry (usually h/w)
- Include protection info $\Rightarrow$ can perform access check in parallel with translation.
- Context switch requires [expensive] flush:
  - can add process tags to improve performance.
  - "global" bit useful for wide sharing.
- Use *superpages* for large regions.
- So TLB contains n entries something like:

Tag			2^k bits		2^k bits					Data
V	G	PTag	Page Number		Frame Number	S	K	R	W	X

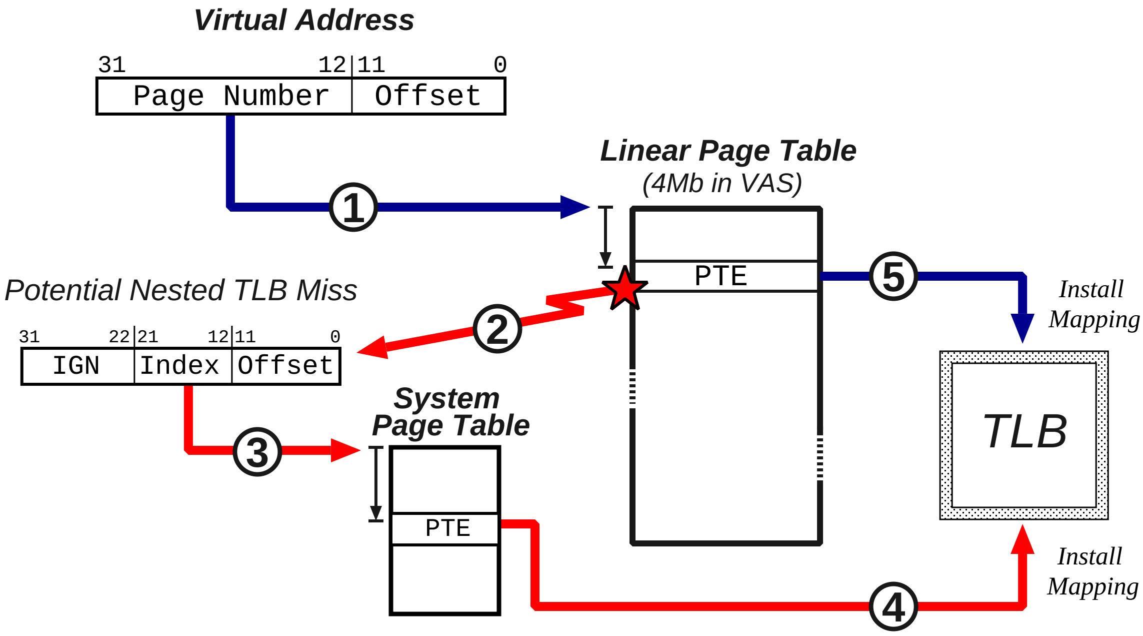
Most parts also present in *page table entries* (PTEs).

Multi-Level Page Tables



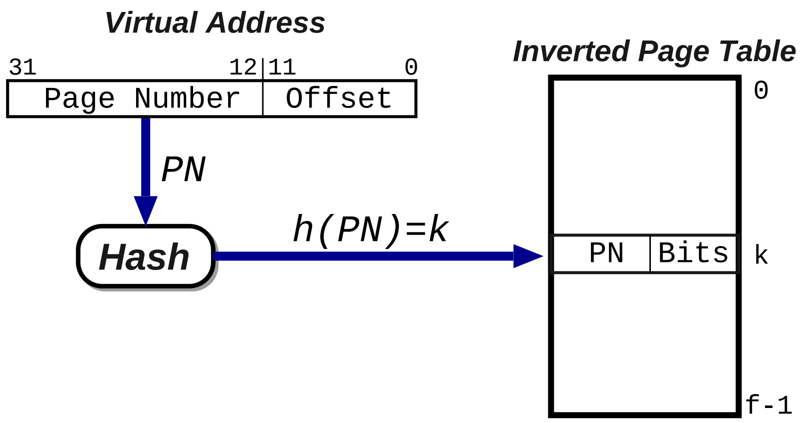
- Modern systems have 2^{32} or 2^{64} byte VAS $\Rightarrow$ have between 2^{22} and 2^{42} pages (and hence PTEs).
- Solution: use N-ary tree (N large, 256–4096).
- Keep PTBR per process and context switch.
- Advantages: easy to implement; cache friendly.
- Disadvantages:
  - Potentially poor space overhead.
  - Inflexibility: superpages, residency.
  - Require $d \geq 2$ memory references.

Linear Page Tables



- Modification of MPTs:
  - pages of LPT translated on demand
  - i.e. stages ②, ③ and ④ not always needed.
- Advantages:
  - can require just 1 memory reference.
  - (initial) miss handler simple.
- But doesn't fix sparsity / superpages.
- *Guarded page tables* (≈ tries) claim to fix these.

Inverted Page Tables



- Recall $f \ll p \Rightarrow$ keep entry per *frame*.
- Then table size bounded by physical memory!
- IPT: frame number is $\mathtt{h(pn)}$

✔ only one memory reference to translate.

✔ no problems with sparsity.

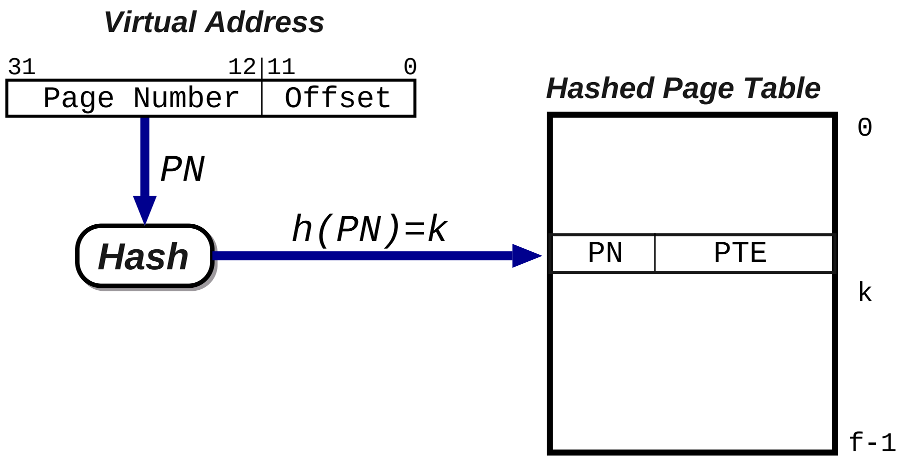
✔ can easily augment with process tag.

✘ no option on which frame to allocate

✘ dealing with collisions.

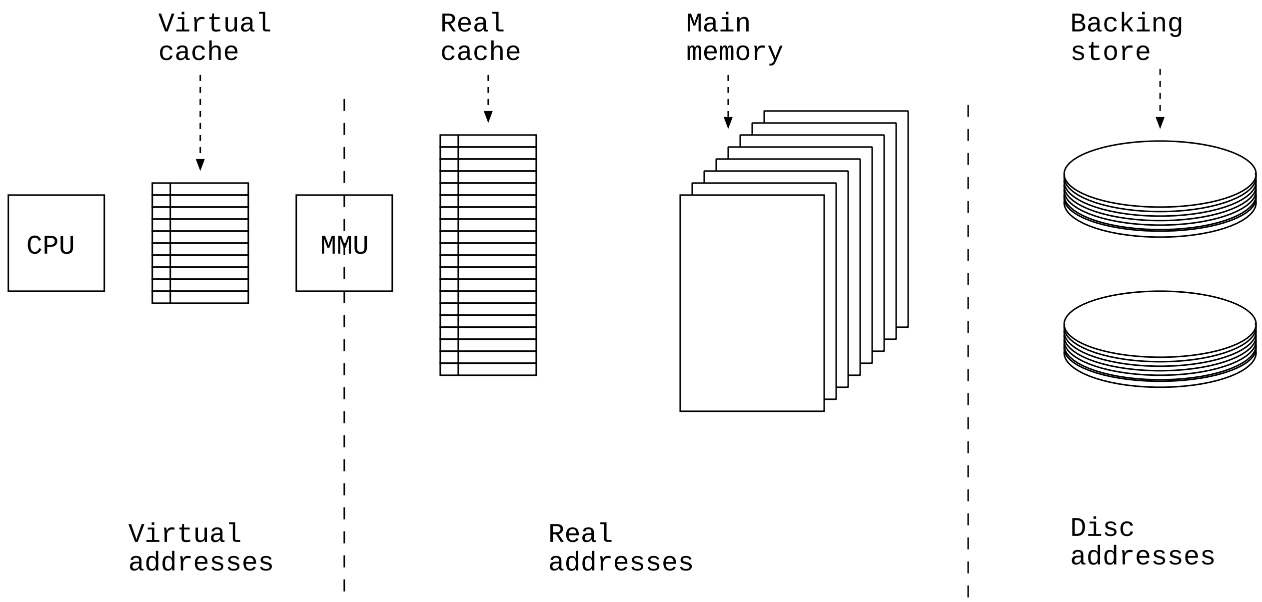
✘ cache unfriendly.

Hashed Page Tables



- HPT: simply extend IPT into proper hash table.
- i.e. make frame number explicit.
  - ✔ can map to any frame.
  - ✔ can choose table size.
  - ✘ table now bigger.
  - ✘ sharing still hard.
  - ✘ still cache unfriendly, no superpages.
- Can solve these last with *clustered page tables*.

Memory Hierarchy



Memory hierarchy on modern machine composed of some subset of:

- CPU registers
- virtual cache(s)
- physical cache(s)
- main memory
- backing store

Consider cost of context switch in such systems.

Swapping

- If the number of processes exceeds total physical memory, then process can be swapped out to secondary store (e.g. disk).
- This makes space for a second process to enter memory
- When the original process is resumed it is swapped back into memory
- Can be used to preempt low priority tasks for high priority tasks
- When a process is rolled back in to memory then it must be positioned at same physical address if load/compile time relocation is used.
- If runtime relocation is used then need to change mapping in MMU to reflect new base and limit of process.

Swapping (2)

How it works:

1. OS maintains a ready queue of processes on disk which are ready to be executed.
2. When OS decides to run a process it calls dispatcher to check whether process is in memory.
3. If not the dispatcher may need to swap out a currently resident process and swap in the required process.

Note that:

- can cause very large context switch times $\Rightarrow$ need to make execution time long relative to swap time.
- user program needs to keep OS informed of how much memory it is using.
- What if have pending I/O on a process to be swapped?
- Standard swapping used on few systems in practice (too slow, too difficult to implement).
- Modified swapping used on Unix; starts automatically if memory use reaches a threshold.

Virtual Memory

- We have under our belts a range of techniques for implementing memory management
- Unfortunately most of these (as stated thus far) require the **entire process** to be in memory to execute
- VM allows execution of processes which need not be entirely in memory
- Big improvement on manual scheme (overlays)
- Commonly implemented by **demand paging**
- On architectures with paged segmentation also uses paging
- On purely segmented architectures can use **demand segmentation**

Demand Paging

- Processes reside on disk
- To execute a process we swap it in in a **lazy** fashion
- Need to modify process page table to show which pages are in memory
- Use a **valid/invalid** bit scheme
- Access to invalid page causes a **page fault**

Algorithm

1. Check whether this was an invalid memory access by process
2. Invalid reference $\Rightarrow$ kill process; otherwise page in the desired page
3. Find a free frame in memory (from free frame list)
4. Start disk I/O to read desired page into the new frame

5. When I/O finished modify process page table and process control table to show new page valid

6. Restart the process on the instruction which faulted

- Pure demand paging – never bring a page in until required

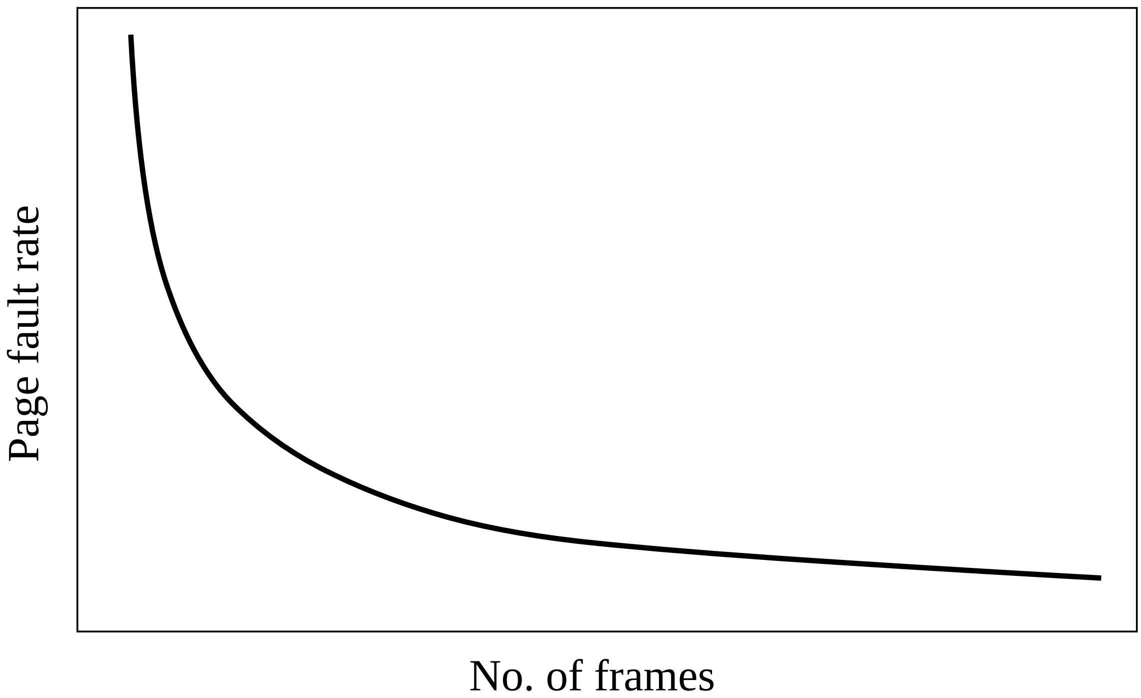
- Hardware support for paging is same as for paging with swapping

Performance Issues

- Compute the effective memory access time for demand paged system
- Let $p = \mathbb{P}$(Page Fault) (hopefully close to 0), $m =$ memory access time
- Effective access time $= (1-p) \times m + p \times t$, where $t =$ page fetch time composed of
  1. Trap into OS
  2. Save user process state (registers, stack etc)
  3. Determine whether fault is for legal page; find its disk location
  4. Schedule I/O from disk to free frame
     (a) Disk queueing time
     (b) Disk seek time & latency
     (c) Transfer time
  5. (while waiting reschedule CPU to allow another proc to run)
  6. On disk I/O complete interrupt, save current process state
  7. Correct page and process tables with new page information

8. Mark process as runnable
9. Restore user process state and restart instruction

- Often better to copy all pages to swap at process startup eg: BSD 4.3



Page Replacement

- We have assumed that a fault occurs for a page at most once
- Memory is limited and processes cannot simply grow forever
- Need to discard unused pages if total demand for pages exceeds phys memory size
- Page replacement: find a currently unused frame and free it:
  1. Find the desired replacement page on disk
  2. Select a free frame to use for incoming page
     (a) if there is a free frame use it
     (b) otherwise select a victim page to free
     (c) write the victim page back to disk, mark it as invalid in its process page tables
  3. Read desired page into freed frame
  4. Restart user process
- Overhead can be reduced by adding a 'dirty' bit to pages (or to frames)

Page Replacement Algorithms

- FIFO
  - Keep a queue of pages to replace - earliest in is first out
  - Performance difficult to predict - no idea whether page replaced will be used again or not
  - Discard is independent of page use frequency
- Optimal Algorithm
  - Replace page which will not be used for longest period of time
  - Can only be done on statically determined reference strings
  - Serves as a good comparison for other algorithms
- Least Recently Used
  - Notice that OPT works on basis of when page will be **used** whereas FIFO works on when the page was brought into memory
  - LRU replaces frame which has not been used for longest
  - Problem is to determine the LRU ordering

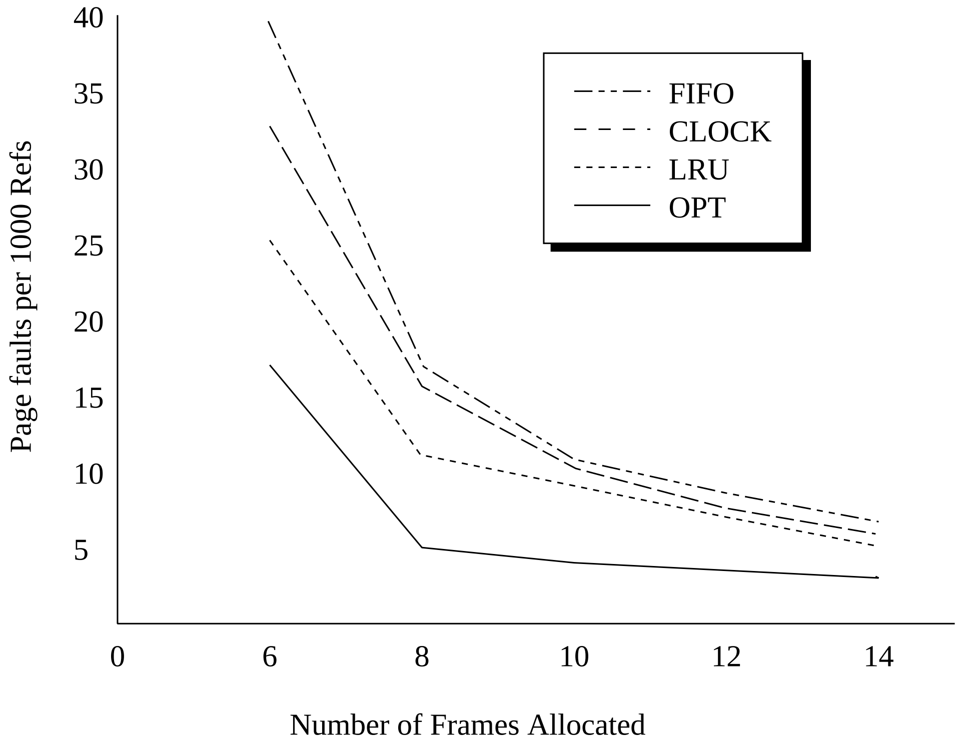
Implementation – Counters

- Give each page table entry a time of use field and give CPU a logical clock (counter)
- When page is referenced the PT entry is updated to clock value
- Replace page with smallest time value
- Requires a sort to find minimum of page clock values
- Also adds a write to memory (PT) every page reference
- What about clock overflow ?

Implementation – Stack

- Keep a stack of pages (doubly linked list) with MRU page on top
- Discard from bottom of stack
- Requires changing pointers per reference
- Appropriate with microcoded support
- Still very slow without extensive hardware support

Performance Comparison



LRU Approximations

- Many systems provide help in form of a **reference bit** updated by hardware whenever page is touched

- Allows us to determine set of active pages after some time

- Can discard unused pages (or not recently used)

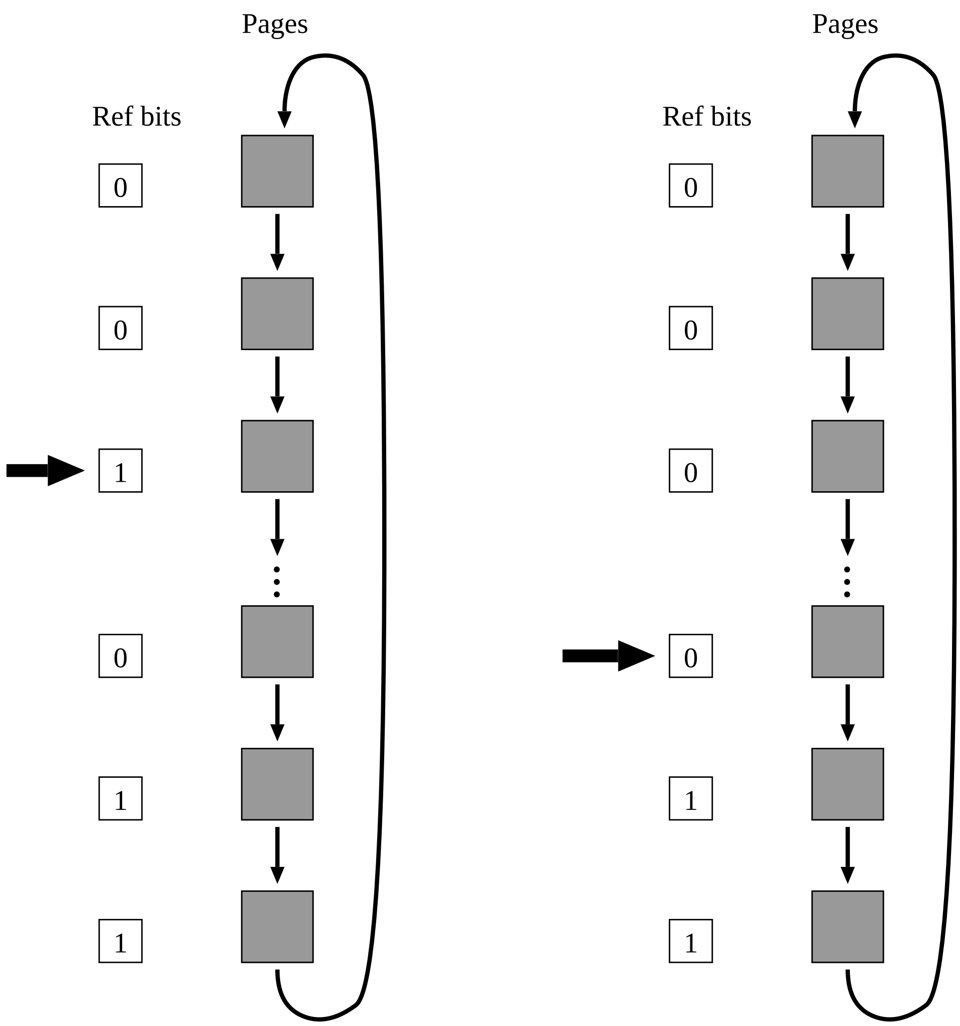
- Improvement if record reference bits periodically: **additional reference bits** algorithm, e.g.
  - OS maintains 8-bit value for each page; initially zero.
  - Periodically (e.g. 100ms) shift reference bit onto high order bit of the byte.
  - Select lowest value (or one of) to replace

- Second Chance: use only reference bit
  - Use FIFO to select candidate page for replacement
  - Before discard check its reference bit
  - If reference bit is 0: discard
  - If reference bit is 1: reset reference bit and add page to FIFO queue with time = current time.

– A page given a second chance is the last to be replaced

– If reference bit is always being set, page will never be replaced

– Often called *clock* since can consider current pointer to be a hand sweeping around ...



- Enhanced Second Chance:
  - Consider both reference bit and the page modify bit – gives 4 (ordered) pairs
  - (0,0) - best page to replace
  - (0,1) - not recently used but modified – next best
  - (1,0) - recently used but clean – probably code in use
  - (1,1) - recently used and modified – bad choice for replacement
- If no h/w provided reference bit can emulate:
  - to clear "reference bit", mark page no access.
  - if referenced ⇒ trap, update permissions, resume.
  - to check if referenced, check if not still no access.
  - can use sim. scheme for modified bit.

Other Schemes

- Counting Algorithms — keep counter of number of refs to each page
  - LFU: replace page with smallest count
  - MFU: replace highest count because low count $\Rightarrow$ most recently brought in.
- Page Buffering Algorithms:
  - Keep a min. number of victims in a free pool
  - New page goes into a frame on the free list, before writing out victim.
  - Alternatively remember page contents of pages in free pool
- (Pseudo) MRU:
  - Consider access of e.g. large array.
  - Page to replace is one application has *just finished with*, i.e. most recently used.
  - e.g. track page faults and look for sequences.
  - discard the k^{th} in victim sequence.
- Application-specific:
  - provide hook for app. to suggest replacement.
  - must be careful with denial of service ...

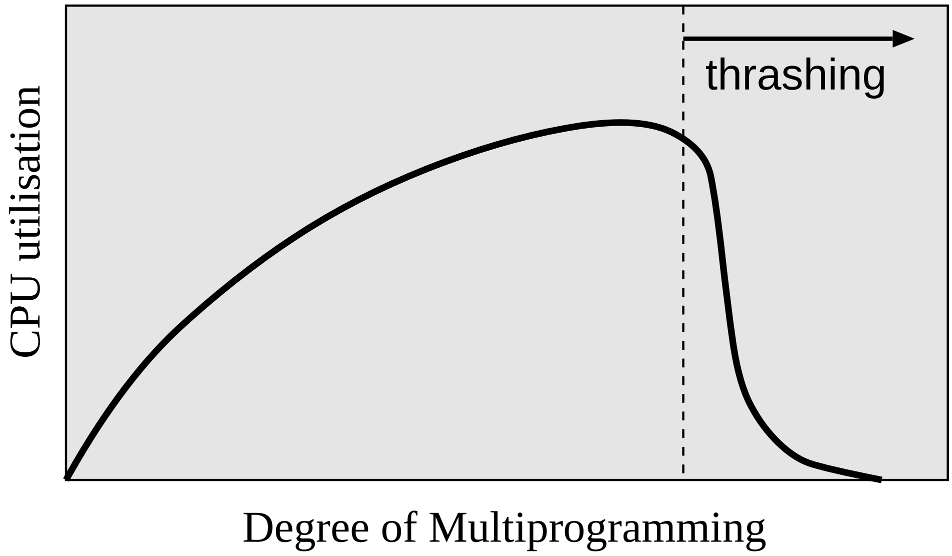
Frame Allocation

- Aims to solve the problem of how many (of the available) frames to give to each process
- Can we page OS code and data ?
- Need a minimum number of free frames - required by instruction set archictecture
- Sometimes also care *which* frames we give to which process ("page colouring")

Allocation Algorithms

- Obvious choice is to split m frames over n processes as m/n, with $m\%n$ in free pool
- Alternatively allocate in a proportional fashion - scale allocation by process sizes
- When new process created each running process loses some proportion of its frames
- When there is competition for frames we can choose between global and local allocation/replacement

Thrashing



- If a process has too few (below min-free) frames then it must be swapped out (eg low priority task)
- If a process continually page faults then **thrashing** results

How does it occur ?

1. Kernel monitors CPU utilisation; if it is too low, increase MPL by starting a new process
2. Say we are using global page replacement — a process begins to demand more pages, taking from other processes

3. But the other processes need those pages, so they fault to bring them back in

4. Number of runnable processes drops (since they're all waiting on I/O)

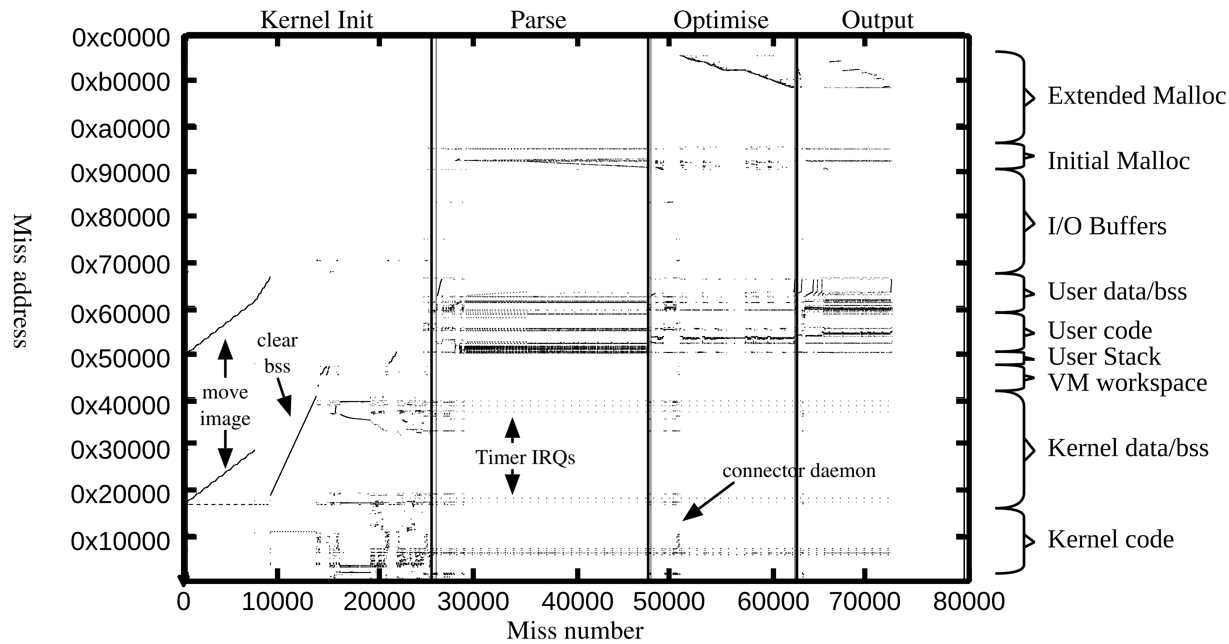
5. CPU utilisation drops

6. GOTO 1

- To prevent thrashing need to give process as many pages as it "needs"

- How do we know what that is?

Locality of Reference



Locality of reference: in a short time interval, locations referenced by a program tend to be grouped into a few regions in its address space.

- procedure being executed
- ... sub-procedures
- ... data access
- ... stack variables

Note: have locality in both space and time.

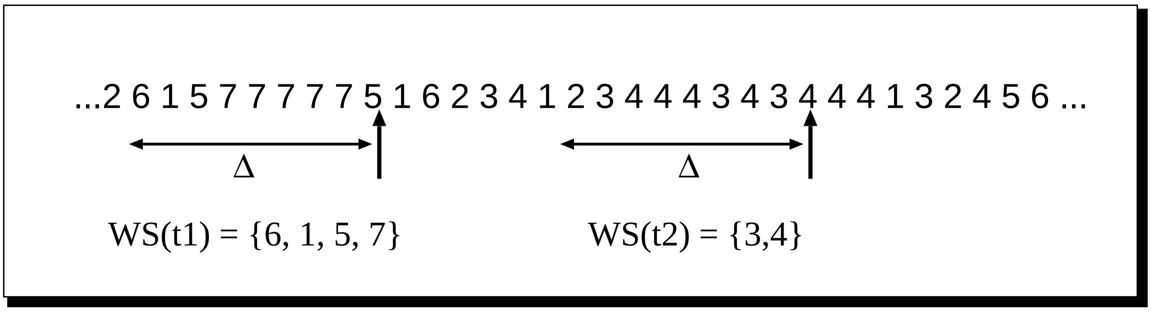
Working Set

Problem of more “simultaneously” accessed pages than physical pages:

Define the *Working Set* (Denning ACM SIGOPS 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*, and in each of which get locality.
- OS can try to prevent thrashing by maintaining sufficient pages for current phase.
- In general can be used as a scheme to determine allocation for each process.

Calculation of Working Set



- Define window size Δ of most recent page refs
- If a page is "in use" it is in the working set
- Gives an approximation to locality of program
- Given the size of the working set for each process WSS_i, can compute total frame demand D
- If $D > m$ we are in danger of thrashing – suspend a process
- Alternatively use page fault frequency (PFF) of process

Prepaging

- 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 (eg after suspension)
- When process is started can pre-page by adding its frames to free list

Page Sizes

- How do we select a page size (given no hardware constraints)?
- Typical values are 512 to 16K bytes
- Trade off size of PT and degree of fragmentation due to page size
- Historical trend towards larger page sizes
- Today many processors (e.g. alpha, x86, ARM) have *multiple* page sizes: tricky for O/S to use ...

Other Performance Issues

- Program structure
- Language choice
- I/O Interlock - need to lock some pages in memory during DMA
- Can lock a page brought in for a low priority process
- VM is the anithesis of (hard) RT systems work — must lock all pages.
- For SRT, trade-offs may be available (e.g. self-paging in Nemesis).

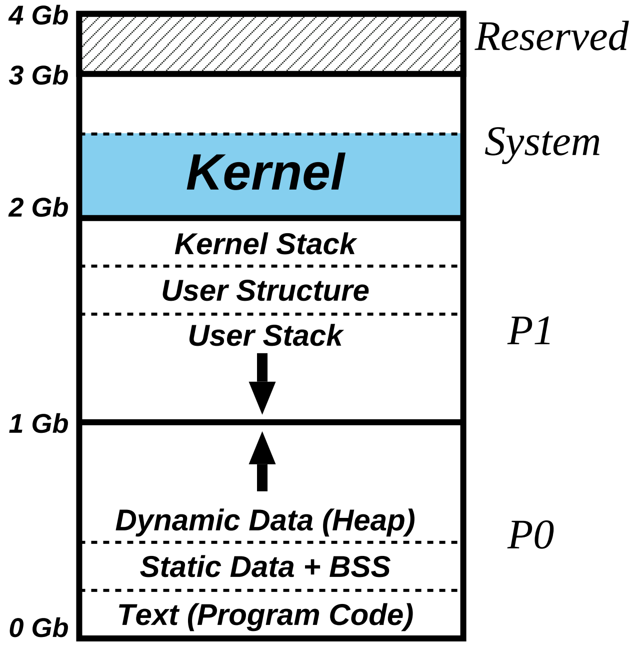
Demand Segmentation

- Infrequently used, OS/2 is an example on the 286
- No hardware support for paging - but supports segments
- OS keeps a segment descriptor table which provides info on what is in memory etc
- On a segment fault, the required segment is brought in, replacing some other segment
- May need to compact memory to fit in a segment which is coming in

Case Study 1: Unix

- Swapping allowed from very early on.
- Kernel Per-process info. split into two kinds:
  - `proc` and `text` structures always resident.
  - page tables, `user` structure and kernel stack could be swapped out.
- Swapping performed by special process: the *swapper* (usually process `0`).
  - periodically awaken and inspect processes on disk.
  - choose one waiting longest time and prepare to swap in.
  - victim chosen by looking at scheduler queues: try to find process blocked on I/O.
  - other metrics: priority, overall time resident, time since last swap in (for stability).
- From 3BSD / SVR2 onwards, implemented demand paging.
- Today swapping only used when dire shortage of physical memory.

Unix: Address Space



4.3 BSD Unix address space borrows from VAX:

- 0Gb–1Gb: segment $P0$ (text/data, grow upward)
- 1Gb–2Gb: segment $P1$ (stack, grows downward)
- 2Gb–3Gb: *system* segment (for kernel).

Address translation done in hardware LPT:

- System page table always resident.
- $P0$, $P1$ page tables in system segment.
- Segments have page-aligned length.

Unix: Page Table Entries

	VLD 31	30 27	26	FOD 25	FS 24	23 21	20 0
Resident	1	AXS	M	0	x	xxx	Frame Number
Demand Zero	0	AXS	x	1	0	xxxx..........xxxx	
Fill from File	0	AXS	x	1	1	Block Number	
Transit/Sampling	0	AXS	M	0	x	xxx	Frame Number
On backing store	0	AXS	x	0	x	0000..........0000	

- PTEs for valid pages determined by h/w.
- If valid bit not set ⇒ use up to OS.
- BSD uses *FOD* bit, *FS* bit and the *block number*.
- First pair are "fill on demand":
  - DZ used for BSS, and growing stack.
  - FFF used for executables (text & data).
  - Simple pre-paging implemented via *klusters*.
- Sampling used to simulate reference bit.
- Backing store pages located via *swap map*(s).

Unix: Paging Dynamics

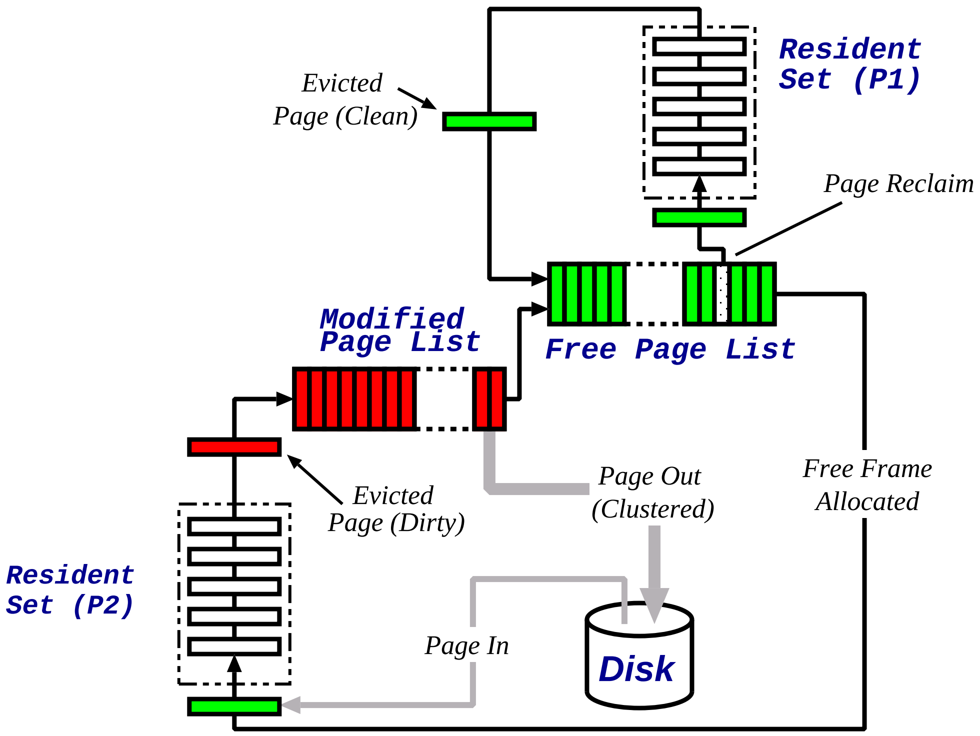
- Physical memory managed by *core map*:
  - array of structures, one per *cluster*.
  - records if free or in use and [potentially] the associated page number and disk block.
  - *free list* threads through core map.
- Page replacement carried out by the *page dæmon*.
  - every 250ms, OS checks if "enough' (viz. `lotsfree`) physical memory free.
  - if not $\Rightarrow$ wake up page dæmon.
- Basic algorithm: global [two-handed] clock:
  - hands point to different entries in core map
  - first check if can replace front cluster; if not, clear its "reference bit" (viz. mark invalid).
  - then check if back cluster referenced (viz. marked valid); if so given second chance.
  - else flush to disk (if necessary), and put cluster onto end of free list.
  - move hands forward and repeat ...
- System V Unix uses an almost identical scheme ...

Case Study 2: VMS

- VMS released in 1978 to run on the VAX-11/780.
- Aimed to support a wide range of hardware, and a job mix of real-time, timeshared and batch tasks.
- This led to a design with:
  - A *local* page replacement scheme,
  - A *quota* scheme for physical memory, and
  - An aggressive *page clustering* policy.
- First two based around idea of *resident set*:
  - simply the set of pages which a given process currently has in memory.
  - each process also has a *resident-set limit*.
- Then during execution:
  - pages faulted in by pager on demand.
  - once hit limit, choose victim from resident set.

  $\Rightarrow$ minimises impact on others.
- Also have swapper for extreme cases.

VMS: Paging Dynamics



- Basic algorithm: simple [local] FIFO.
- Suckful ⇒ augment with software "victim cache":
  - Victim pages placed on tail of FPL/MPL.
  - On fault, search lists before do I/O.
- Lists also allow aggressive *page clustering*:
  - if $|\text{MPL}| \geq$ `hi`, write ($|\text{MPL}| -$ `lo`) pages.
  - Get ~100 pages per write on average.

VMS: Other Issues

- Modified page replacement:
  - introduce *callback* for privileged processes.
  - prefer to retain pages with TLB entries.
- Automatic resident set limit adjustment:
  - system counts #page faults per process.
  - at quantum end, check if rate > `PFRATH`.
  - if so and if "enough" memory ⇒ increase RSL.
  - *swapper trimming* used to reduce RSLs again.
  - NB: real-time processes are exempt.
- Other system services:
  - `$SETSWM`: disable process swapping.
  - `$LCKPAG`: lock pages into memory.
  - `$LKWSET`: lock pages into resident set.
- VMS still alive: recent versions updated to support 64-bit address space.

Other VM Techniques

Once have MMU, can (ab)use for other reasons

- Assume OS provides:
  - system calls to change memory protections.
  - some way to "catch" memory exceptions.
- This enables a large number of applications.
- e.g. concurrent garbage collection:
  - mark unscanned areas of heap as no-access.
  - if mutator thread accesses these, trap.
  - on trap, collector scans page(s), copying and forwarding as necessary.
  - finally, resume mutator thread.
- e.g. incremental checkpointing:
  - at time t atomically mark address space read-only.
  - on each trap, copy page, mark r/w and resume.
  - ✔ no significant interruption.
  - ✔ more space efficient

Single Address Space Operating Systems

- Emerging large (64-bit) address spaces mean SAS plausible once more:
- Separate concerns of “what we can see” and “what we are allowed to access”.
- Advantages: easy sharing (unified addressing).
- Problems:
  - address binding issues return.
  - cache/TLB setup for MVAS model.
- Distributed shared virtual memory:
  - turn a NOW into a SMP.
  - how seamless do you think this is?
- Persistent object stores:
  - support for pickling & compression?
  - garbage collection?
- Sensible use requires restraint ...