Virtual Memory

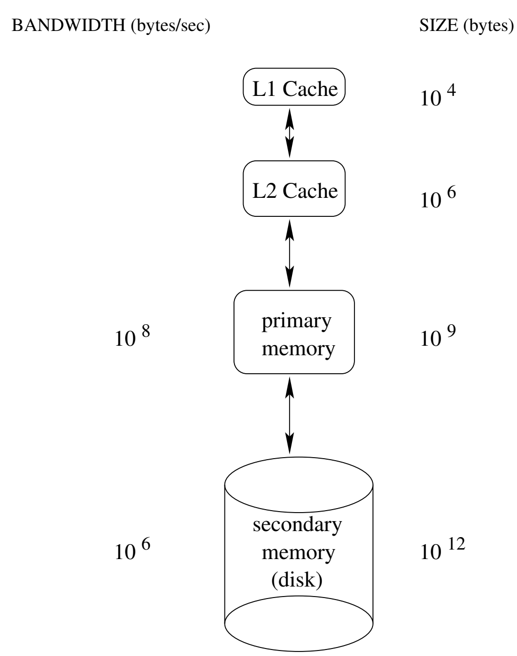
Goals:

- Allow virtual address spaces that are larger than the physical address space.
- Allow greater multiprogramming levels by using less of the available (primary) memory for each process.

Method:

- Allow pages (or segments) from the virtual address space to be stored in secondary memory, as well as primary memory.
- Move pages (or segments) between secondary and primary memory so that they are in primary memory when they are needed.

The Memory Hierarchy



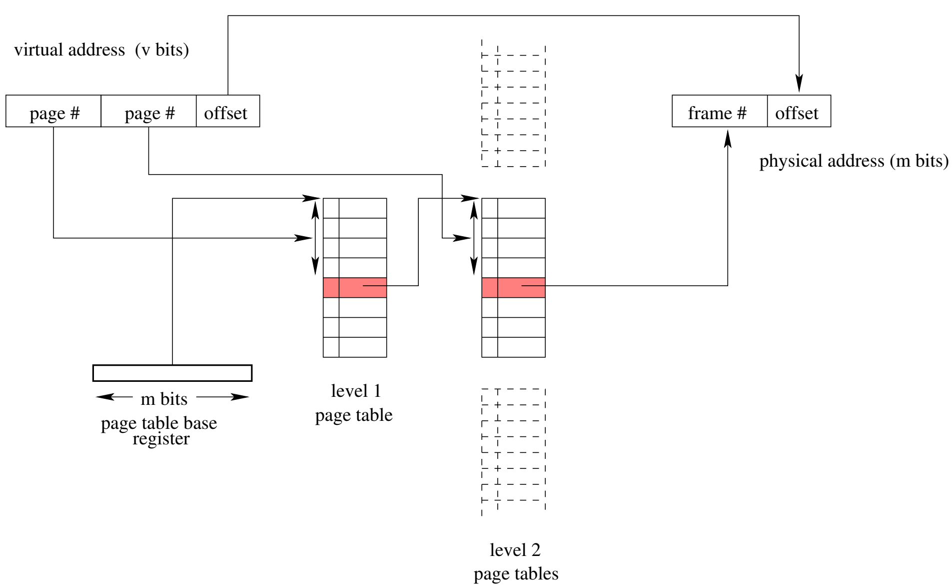
Large Virtual Address Spaces

- Virtual memory allows for very large virtual address spaces, and very large virtual address spaces require large page tables.
- example: 2^{48} byte virtual address space, 8Kbyte (2^{13} byte) pages, 4 byte page table entries means

$$\frac{2^{48}}{2^{13}}2^2 = 2^{37} \text{ bytes per page table}$$

- page tables must be in memory and physically contiguous
- some solutions:
  - multi-level page tables - page the page tables
  - inverted page tables

Two-Level Paging



Inverted Page Tables

- A normal page table maps virtual pages to physical frames. An inverted page table maps physical frames to virtual pages.
- Other key differences between normal and inverted page tables:
  - there is only one inverted page table, not one table per process
  - entries in an inverted page table must include a process identifier
- An inverted page table only specifies the location of virtual pages that are located in memory. Some other mechanism (e.g., regular page tables) must be used to locate pages that are not in memory.

Paging Policies

When to Page?:
Demand paging brings pages into memory when they are used. Alternatively, the OS can attempt to guess which pages will be used, and *prefetch* them.

What to Replace?:
Unless there are unused frames, one page must be replaced for each page that is loaded into memory. A *replacement policy* specifies how to determine which page to replace.

Similar issues arise if (pure) segmentation is used, only the unit of data transfer is segments rather than pages. Since segments may vary in size, segmentation also requires a *placement policy*, which specifies where, in memory, a newly-fetched segment should be placed.

Paging Mechanism

- A *valid* bit (V) in each page table entry is used to track which pages are in (primary) memory, and which are not.

  $V = 1$**:** valid entry which can be used for translation

  $V = 0$**:** invalid entry. If the MMU encounters an invalid page table entry, it raises a *page fault* exception.

- To handle a page fault exception, the operating system must:
  - Determine which page table entry caused the exception. (In NachOS, and in real MIPS processors, the MMU places the offending virtual address into the `BadVAddrReg` register.)
  - Ensure that that page is brought into memory.

  On return from the exception handler, the instruction that resulted in the page fault will be retried.

- If (pure) segmentation is being used, there will a valid bit in each segment table entry to indicate whether the segment is in memory.

Optimal Page Replacement

- There is an optimal page replacement policy for demand paging.
- The OPT policy: replace the page that will not be referenced for the longest time.

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	a	a	a	a	a	a	c	c	c
Frame 2		b	b	b	b	b	b	b	b	b	d	d
Frame 3			c	d	d	d	e	e	e	e	e	e
Fault ?	x	x	x	x			x			x	x	

- OPT requires knowledge of the future.

A Simple Replacement Policy: FIFO

- the FIFO policy: replace the page that has been in memory the longest
- a three-frame example:

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	d	d	d	e	e	e	e	e	e
Frame 2		b	b	b	a	a	a	a	a	c	c	c
Frame 3			c	c	c	b	b	b	b	b	d	d
Fault ?	x	x	x	x	x	x	x			x	x	

Other Replacement Policies

- FIFO is simple, but it does not consider:

  Recency of Use: when was a page last used?

  Frequency of Use: how often a page has been used?

  Cleanliness: has the page been changed while it is in memory?

- The *principle of locality* suggests that usage ought to be considered in a replacement decision.
- Cleanliness may be worth considering for performance reasons.

Locality

- Locality is a property of the page reference string. In other words, it is a property of programs themselves.
- *Temporal locality* says that pages that have been used recently are likely to be used again.
- *Spatial locality* says that pages “close” to those that have been used are likely to be used next.

In practice, page reference strings exhibit strong locality. Why?

Least Recently Used (LRU) Page Replacement

- LRU is based on the principle of temporal locality: replace the page that has not been used for the longest time
- To implement LRU, it is necessary to track each page’s recency of use. For example: maintain a list of in-memory pages, and move a page to the front of the list when it is used.
- Although LRU and variants have many applications, LRU is often considered to be impractical for use as a replacement policy in virtual memory systems. Why?

The “Use” Bit

- A *use bit* (or *reference bit*) is a bit found in each page table entry that:
  - is set by the MMU each time the page is used, i.e., each time the MMU translates a virtual address on that page
  - can be read and updated by the operating system
- Page table entries in NachOS include a use bit.

The use bit provides a small amount of efficiently-maintainable usage information that can be exploited by a page replacement algorithm.

The Clock Replacement Algorithm

- The clock algorithm (also known as “second chance”) is one of the simplest algorithms that exploits the use bit.
- Clock is identical to FIFO, except that a page is “skipped” if its use bit is set.
- The clock algorithm can be visualized as a victim pointer that cycles through the page frames. The pointer moves whenever a replacement is necessary:

```
while use bit of victim is set
   clear use bit of victim
   victim = (victim + 1) % num_frames
choose victim for replacement
victim = (victim + 1) % num_frames
```

Frequency-based Page Replacement

- Another approach to page replacement is to count references to pages. The counts can form the basis of a page replacement decision.
- Example: LFU (Least Frequently Used)
  Replace the page with the smallest reference count.
- Any frequency-based policy requires a reference counting mechanism, e.g., MMU increments a counter each time an in-memory page is referenced.
- Pure frequency-based policies have several potential drawbacks:
  - Old references are never forgotten. This can be addressed by periodically reducing the reference count of every in-memory page.
  - Freshly loaded pages have small reference counts and are likely victims - ignores temporal locality.

Page Cleanliness: the Dirty Bit

- A page is *dirty* if it has been changed since it was loaded into memory.
- A dirty page is more costly to replace than a clean page. (Why?)
- The MMU identifies dirty pages by setting a *dirty bit* in the page table entry when the contents of the page change. Operating system clears the dirty bit when it cleans the page.
- The dirty bit potentially has two roles:
  - Indicates which pages need to be cleaned.
  - Can be used to influence the replacement policy.

Enhanced Second Chance Replacement Algorithm

- Classify pages according to their use and dirty bits:

  (0,0): not recently used, clean.

  (0,1): not recently used, dirty.

  (1,0): recently used, clean

  (1,1): recently used, dirty

- Algorithm:
  1. Sweep once looking for (0,0) page. Don't clear use bits while looking.
  2. If none found, look for (0,0) or (0,1) page, this time clearing "use" bits while looking.

Page Cleaning

- A dirty page must be cleaned before it can be replaced, otherwise changes on that page will be lost.
- *Cleaning* a page means copying the page to secondary storage.
- Cleaning is distinct from replacement.
- Page cleaning may be *synchronous* or *asynchronous*:

  synchronous cleaning: happens at the time the page is replaced, during page fault handling. Page is first cleaned by copying it to secondary storage. Then a new page is brought in to replace it.

  asynchronous cleaning: happens before a page is replaced, so that page fault handling can be faster.

  – asynchronous cleaning may be implemented by dedicated OS *page cleaning threads* that sweep through the in-memory pages cleaning dirty pages that they encounter.

Prefetching

- Prefetching means moving virtual pages into memory before they are needed, i.e., before a page fault results.
- The goal of prefetching is *latency hiding*: do the work of bringing a page into memory in advance, not while a process is waiting.
- To prefetch, the operating system must guess which pages will be needed.
- Hazards of prefetching:
  - guessing wrong means the work that was done to prefetch the page was wasted
  - guessing wrong means that some other potentially useful page has been replaced by a page that is not used
- most common form of prefetching is simple sequential prefetching: if a process uses page x, prefetch page $x + 1$.
- sequential prefetching exploits spatial locality of reference

Page Size Tradeoffs

- larger pages mean:
  - + smaller page tables
  - + better TLB "coverage"
  - + more efficient I/O
  - - greater internal fragmentation
  - - increased chance of paging in unnecessary data

Belady's Anomaly

- FIFO replacement, 4 frames

Num	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	a	a	a	e	e	e	e	d	d
Frame 2		b	b	b	b	b	b	a	a	a	a	e
Frame 3			c	c	c	c	c	c	b	b	b	b
Frame 4				d	d	d	d	d	d	c	c	c
Fault?	x	x	x	x			x	x	x	x	x	x

- FIFO example on Slide 9 with same reference string had 3 frames and only 9 faults.

More memory does not necessarily mean fewer page faults.

Stack Policies

- Let $B(m, t)$ represent the set of pages in a memory of size m at time t under some given replacement policy, for some given reference string.
- A replacement policy is called a *stack policy* if, for all reference strings, all m and all t:

$$B(m, t) \subseteq B(m + 1, t)$$

- If a replacement algorithm imposes a total order, independent of memory size, on the pages and it replaces the largest (or smallest) page according to that order, then it satisfies the definition of a stack policy.
- Examples: LRU is a stack algorithm. FIFO and CLOCK are not stack algorithms. (Why?)

Stack algorithms do not suffer from Belady's anomaly.

Global vs. Local Page Replacement

- When the system's page reference string is generated by more than one process, should the replacement policy take this into account?

  Global Policy: A global policy is applied to all in-memory pages, regardless of the process to which each one "belongs". A page requested by process X may replace a page that belongs another process, Y.

  Local Policy: Under a local policy, the available frames are allocated to processes according to some memory allocation policy. A replacement policy is then applied separately to each process's allocated space. A page requested by process X replaces another page that "belongs" to process X.

Detailed TLB and Paging Example

- Assume TLB and global page replacments are done using a round-robin algorithm with the next entries to replace being 1 and 3, repectively. Also assume a 4 KB page size. What happens in hardware and the kernel if program A performs the following operations in sequence: read 0x329, write 0x429, read 0x3691, read 0x271a, write 0x1741?

TLB

	VPN	PFN	D	U	V	RO
0	0	3	0	0	1	0
1	1	2	1	1	0	1
2	3	5	1	1	0	0

Page Table A

	VPN	PFN	D	U	V	RO
0	0	3	0	0	1	0
1	1		0	0	0	1
2	2		0	0	0	0
3	3	5	0	0	1	0
4	4	0	0	0	1	0

Page Table B

	VPN	PFN	D	U	V	RO
0	0	1	0	0	1	0
1	1	2	0	0	1	1
2	2					
3	3					
4	4					
5	5	4	1	1	0	0
6	6					
7	7	4	0	0	1	0

CoreMap

	VPN	PGM
0	4	A
1	0	B
2	1	B
3	0	A
4	7	B
5	3	A

How Much Memory Does a Process Need?

- Principle of locality suggests that some portions of the process's virtual address space are more likely to be referenced than others.
- A refinement of this principle is the *working set model* of process reference behaviour.
- According to the working set model, at any given time some portion of a program's address space will be heavily used and the remainder will not be. The heavily used portion of the address space is called the *working set* of the process.
- The working set of a process may change over time.
- The *resident set* of a process is the set of process pages that are located in memory.

> According to the working set model, if a process's resident set includes its working set, it will rarely page fault.

Resident Set Sizes (Example)

```
PID   VSZ  RSS COMMAND
805 13940 5956 /usr/bin/gnome-session
831  2620  848 /usr/bin/ssh-agent
834  7936 5832 /usr/lib/gconf2/gconfd-2 11
838  6964 2292 gnome-smproxy
840 14720 5008 gnome-settings-daemon
848  8412 3888 sawfish
851 34980 7544 nautilus
853 19804 14208 gnome-panel
857  9656 2672 gpilotd
867  4608 1252 gnome-name-service
```

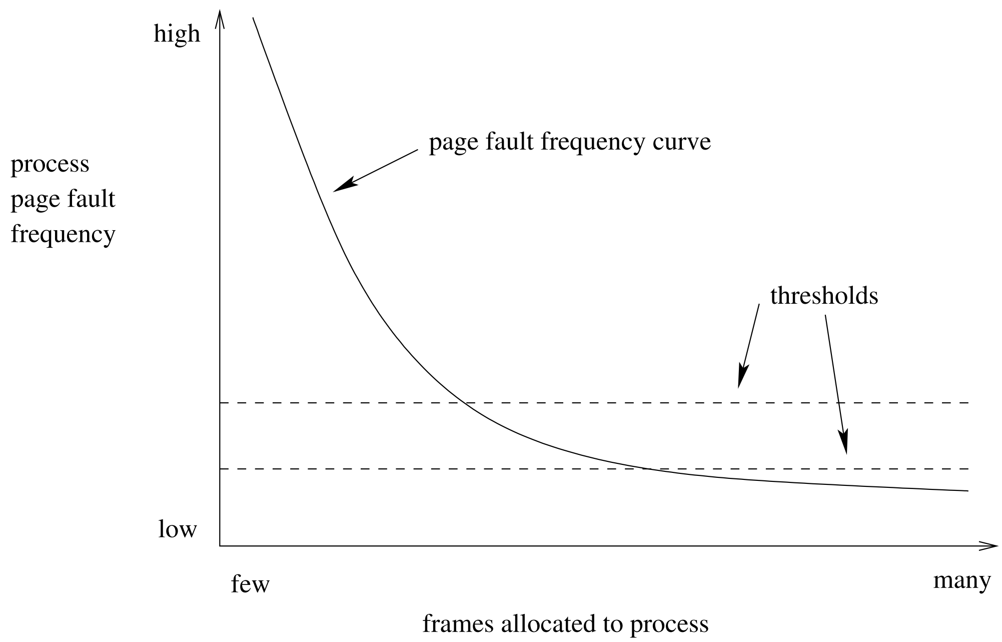
Refining the Working Set Model

- Define $WS(t, \Delta)$ to be the set of pages referenced by a given process during the time interval $(t - \Delta, t)$. $WS(t, \Delta)$ is the working set of the process at time t.
- Define $|WS(t, \Delta)|$ to be the size of $WS(t, \Delta)$, i.e., the number of *distinct* pages referenced by the process.
- If the operating system could track $WS(t, \Delta)$, it could:
  - use $|WS(t, \Delta)|$ to determine the number of frames to allocate to the process under a local page replacement policy
  - use $WS(t, \Delta)$ directly to implement a working-set based page replacement policy: any page that is no longer in the working set is a candidate for replacement

Page Fault Frequency

- A more direct way to allocate memory to processes is to measure their *page fault frequencies* - the number of page faults they generate per unit time.
- If a process's page fault frequency is too high, it needs more memory. If it is low, it may be able to surrender memory.
- The working set model suggests that a page fault frequency plot should have a sharp "knee".

A Page Fault Frequency Plot



Thrashing and Load Control

- What is a good multiprogramming level?
  - If too low: resources are idle
  - If too high: too few resources per process
- A system that is spending too much time paging is said to be *thrashing*. Thrashing occurs when there are too many processes competing for the available memory.
- Thrashing can be cured by load shedding, e.g.,
  - Killing processes (not nice)
  - Suspending and *swapping out* processes (nicer)

Swapping Out Processes

- Swapping a process out means removing all of its pages from memory, or marking them so that they will be removed by the normal page replacement process. Suspending a process ensures that it is not runnable while it is swapped out.
- Which process(es) to suspend?
  - low priority processes
  - blocked processes
  - large processes (lots of space freed) or small processes (easier to reload)
- There must also be a policy for making suspended processes ready when system load has decreased.