CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2001

Lecture 15: Caching: Demand Paged Virtual Memory

15.0 Main Points:

- Concept of paging to disk
- Replacement policies

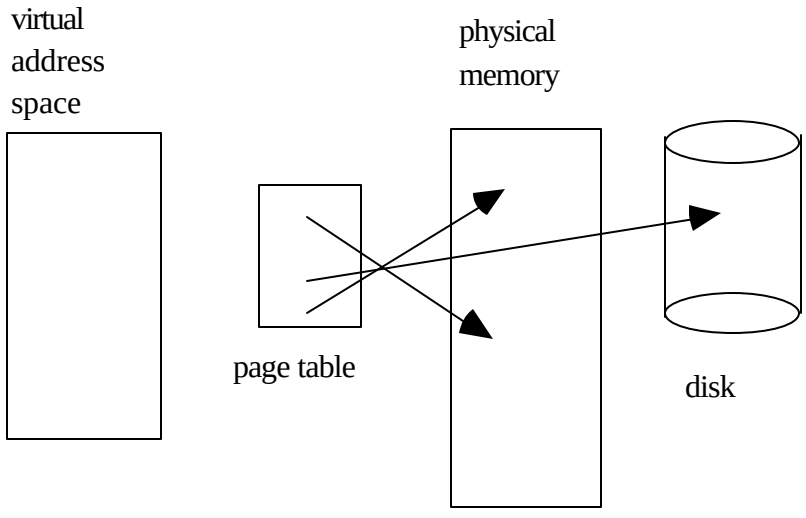
For implementation details, read Levy and Lipman paper!

15.1 Demand Paging

Up to now, all of a job's virtual address space must be in physical memory.

But programs don't use all of their memory all of the time. In fact, there is a 90-10 rule: programs spend 90% of their time in 10% of their code.

Instead, use main memory as a cache for disk. Some pages in memory, some on disk.



Benefits:

- Bigger virtual address space: illusion of infinite memory
- Allow more programs than will fit in memory, to be running at same time

15.2 Demand Paging Mechanism

1. Page table has "present" (valid) bit
   - if present, pointer to page frame in memory
   - if not present, go to disk
2. Hardware traps to OS on reference to invalid page
   - (In MIPS/Nachos, trap on TLB miss, OS checks page table valid bit)
3. OS software:
   a. choose an old page to replace
   b. if old page has been modified, write contents back to disk
   c. change its page table entry and TLB entry
   d. load new page into memory from disk
   e. update page table entry
   f. continue thread

All this is transparent: OS just runs another job in the meantime.

15.2.1 Software-loaded TLB

Instead of having the hardware load the TLB when a translation doesn't match, the MIPS/Snake/Nachos TLB is **software loaded**. Idea is, if have high TLB hit rate, ok to trap to software to fill TLB, even if it's a bit slower.

How do we implement this? How can a process run without access to a page table?

Basic mechanism (just generalization of above):

1. TLB has "present" (valid) bit
   - if present, pointer to page frame in memory
   - if not present, use software page table
2. Hardware traps to OS on reference not in TLB
3. OS software:

a. check if page is in memory
b. if yes, load page table entry into TLB
c. if no, perform page fault operations outlined above
d. continue thread

Paging to disk, or even having software load the TLB – all this is transparent – job doesn't know it happened.

15.2.2 Transparent page faults

Need to transparently re-start faulting instruction.

Hardware must help out, by saving:
1. faulting instruction (need to know which instruction caused fault)
2. processor state

What if an instruction has side effects (CISC processors)?

```
mov (sp)+,10
```

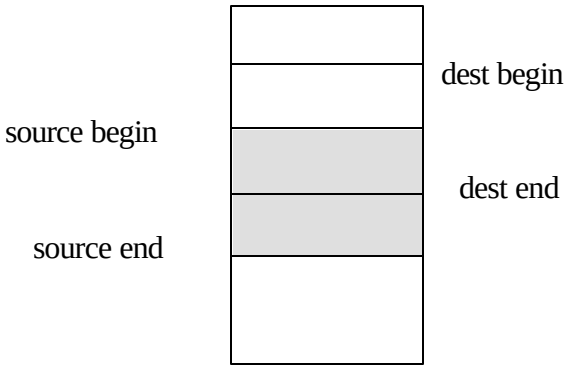
Two options:
- unwind side-effects
- finish off side-effects

Are RISCs easier? What about delayed loads? Delayed branches?

```
ld (sp), r1
```

What if next instruction causes page fault, while load is still in progress? Have to save enough state to allow CPU to restart.

Lots of hardware designers don't think about virtual memory. For example: block transfer instruction. Source, destination can be overlapping (destination before source). Overwrite part of source as instruction proceeds.



No way to unwind instruction!

15.3 Page replacement policies

Replacement policy is an issue with any caching system.

15.3.1 Random?

Typical solution for TLB's. Easy to implement in hardware.

15.3.2 FIFO?

Throw out oldest page. Be fair – let every page live in memory for the same amount of time, then toss it.

Bad, because throws out heavily used pages instead of those that are not frequently used.

15.3.3 MIN

Replace page that won't be used for the longest time into the future.

15.3.4 LRU

Replace page that hasn't been used for the longest time.

If induction works, LRU is a good approximation to MIN. Actually, people don't use even LRU, they approximate it.

15.3.5 Example

Suppose we have 3 page frames, and 4 virtual pages, with the reference string:
A B C A B D A D B C B (virtual page references)

What happens with FIFO?

Ref String slot	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

Page faults in physical memory, with FIFO replacement

What about MIN?

Ref String s slot	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

Page faults in physical memory, with MIN replacement

What about LRU? Same decisions as MIN, but won't always be this way!

When will LRU perform badly? When next reference is to the least recently used page.

Reference string: A B C D A B C D A B C D

Ref String s slot	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

Page faults in physical memory, with LRU replacement

Same behavior with FIFO! What about MIN?

Ref String s slot	A	B	C	D	A	B	C	D	A	B	C	D
1	A									B		
2		B					C					
3			C	D								

Page faults in physical memory, with MIN replacement

15.3.6 Does adding memory always reduce the number of page faults?

Yes, for LRU, MIN. No, for FIFO (Belady's anomaly)

Ref String s slot	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	
Ref String s slot	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

Example of Belady's Anomaly with FIFO replacement

With FIFO, contents of memory can be completely different with different number of page frames.

By contrast, with LRU or MIN, contents of memory with X pages is a subset of contents with X +1 pages. So with LRU or MIN, having more pages never hurts.

15.4 Implementing LRU

15.4.1 Perfect

Timestamp page on each reference
Keep list of pages ordered by time of reference

15.4.2 Clock algorithm

Approximate LRU (approx to approx to MIN)

Replace **an** old page, not **the** old**est** page

Clock algorithm: arrange physical pages in a circle, with a clock hand.

1. Hardware keeps **use bit** per physical page frame
2. Hardware sets use bit on each reference

   If use bit isn't set, means not referenced in a long time
3. On page fault:
   - Advance clock hand (not real time)
   - check use bit
     1 -> clear, go on
     0 -> replace page

Will it always find a page or loop infinitely? Even if all use bits are set, it will eventually loop around, clearing all use bits -> FIFO

What if hand is moving slowly?

Not many page faults and/or find page quickly

What if hand is moving quickly?

Lots of page faults and/or lots of reference bits set.

One way to view clock algorithm: crude partitioning of pages into two categories: young and old. Why not partition into more than 2 groups?

15.4.3 Nth Chance

Nth chance algorithm: don't throw page out until hand has swept by n times

OS keeps counter per page – # of sweeps

On page fault, OS checks use bit:

1 => clear use and also clear counter, go on

0 => increment counter, if < N, go on

else replace page

How do we pick N?

Why pick large N? Better approx to LRU.

Why pick small N? More efficient; otherwise might have to look a long way to find free page.

Dirty pages have to be written back to disk when replaced. Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?

Common approach:

- clean pages – use N = 1
- dirty pages – use N = 2 (and write-back to disk when N=1)

15.4.4 State per page table entry

To summarize, many machines maintain four bits per page table entry:

- **use**: set when page is referenced, cleared by clock algorithm
- **modified**: set when page is modified, cleared when page is written to disk
- **valid**: ok for program to reference this page
- **read-only**: ok for program to read page, but not to modify it (for example, for catching modifications to code pages)

15.4.5 Do we really need a hardware-supported "modified" bit ?

No. Can emulate it (BSD UNIX). Keep two sets of books:

(i) pages user program can access without taking a fault

(ii) pages in memory

Set (i) is a subset of set (ii). Initially, mark all pages as read-only, even data pages. On write, trap to OS. OS sets modified bit, and marks page as read-write.

When page comes back in from disk, mark read-only.

15.4.6 Do we really need a hardware-supported "use" bit?

No. Can emulate it, exactly the same as above:

(i) Mark all pages as invalid, even if in memory.

(ii) On read to invalid page, trap to OS.

(iii) OS sets use bit, and marks page read-only.

(iv) On write, set use and modified bit, and mark page read-write.

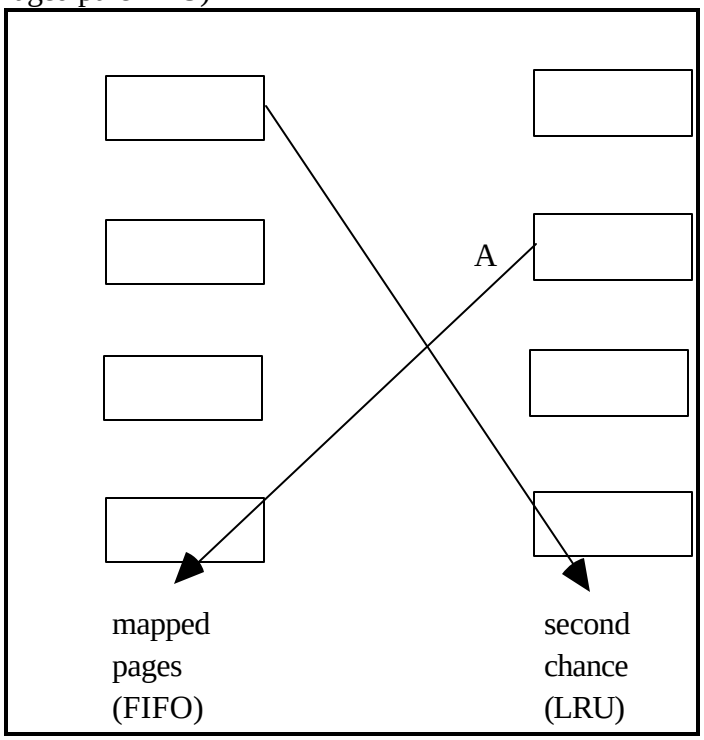
(v) When clock hand passes by, reset use bit and mark page as invalid.

But remember, clock is just an approximation of LRU. Can we do a better approximation, given that we have to take page faults on some reads and writes to collect use information? Need to identify an old page, not the oldest page!

VAX/VMS didn't have a use or modified bit, so had to come up with some solution.
Idea was to split memory in two parts – mapped and unmapped:

i. Directly accessible to program (marked as read-write)
(managed FIFO)

ii. Second-chance list (marked as invalid, but in memory)
(managed pure LRU)



Reference to page on second chance list

On page reference:

if mapped, access at full speed
otherwise page fault:
if on second chance list, mark read-write
move first page on FIFO list onto end of second chance list
(and mark invalid)
if not on second chance list, bring into memory
move first page on FIFO list onto end of second chance
replace first page on second chance list

How many pages for second chance list?

- if 0, FIFO

- if all, LRU, but page fault on every page reference

Pick intermediate value. Result is:

+ few disk accesses (page only goes to disk if it is unused for a long time)

– increased overhead trapping to OS (software/hardware tradeoff)

15.4.7 Does software-loaded TLB need a use bit?

What if we have a software-loaded TLB (as in Nachos)? Two options:

1. Hardware sets use bit in TLB; when TLB entry is replaced, software copies use bit back to page table.
2. Software manages TLB entries as FIFO list; everything not in TLB is second-chance list, managed as strict LRU.

15.4.8 Core map

Page tables map virtual page # -> physical page #

Do we need the reverse? physical page # -> virtual page #?

Yes. Clock algorithm runs through page frames. What if it ran through page tables?

(i) many more entries

(ii) what if there is sharing?

15.5 Thrashing

Thrashing: memory overcommitted, pages tossed out while still needed. When thrashing occurs, the system becomes less able to do useful work because it is spending too many resources to swap pages in and out of memory.

Example: One program, touches 50 pages (each equally likely). Have only 40 physical page frames

If have enough pages, 200 ns/ref
If have too few pages, assume every 5th page reference, page fault

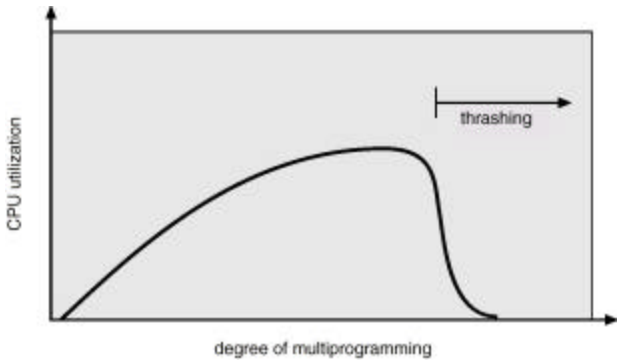
4 refs x 200 ns
1 page fault x 10 ms for disk I/O
_____

result: 5 ref, 10 ms + 800ns => 2 ms/ref

Problem: system doesn't know what it's getting into.

Log more and more users into the system, eventually:

total # of pages needed > # of pages available



So what do you do about this?

1. One process alone too big? Change program so it needs less memory, has better locality. For example, split matrix multiply up into smaller sub-matrix multiplies, that each fit into memory.
2. Several jobs?
   - figure out needs/process (working set)
   - run only groups that fit (balance sets)
     kick other processes out from memory

Remember: issue here is not total size of process, but rather total number of pages being using at the moment.

How do we figure needs/process out?

Working set (Denning, MIT, mid-60's)

- Informally, collection of pages process is using right now
- Formally, set of pages job has referenced in last T seconds.

How do we pick T?

- 1 page fault = 10 msec
- 10 msec = 2 million instructions

So T needs to be a lot bigger than 1 million instructions.

How do you figure out what working set is?

(a) Modify clock algorithm, so that it sweeps at fixed intervals. Keep idle time/page – how many sec since last reference

(b) With second chance list – how many seconds since got put on 2nd chance list

Now that you know how many pages each program needs, what to do?

Balance set

1. if all fit? done
2. if not? throw out fat cats. Bring them back eventually.

What if T is too big?

- waste memory; too few programs fit in memory

What if T is too small?

- thrashing

Too big is better than too small!

15.6 Fairness

On a page fault, do you consider all pages in one pool, or only those of the process that caused the page fault?

Global replacement (UNIX) – all pages in one pool.

More flexible – if my process needs a lot, and you need a little, I can grab pages from you. Problem – one turkey can ruin whole system (want to favor jobs that need only few pages!)

Per-process (VMS) – give each a separate pool, for example, a separate clock for each process. Less flexible.

Example:

- intermittent interactive job (emacs)
- batch job (compilation)

When compilation is over, emacs pages have to be brought back in, and no history information.