M3: Translation, protection, sharing

****

Review -- 1 min

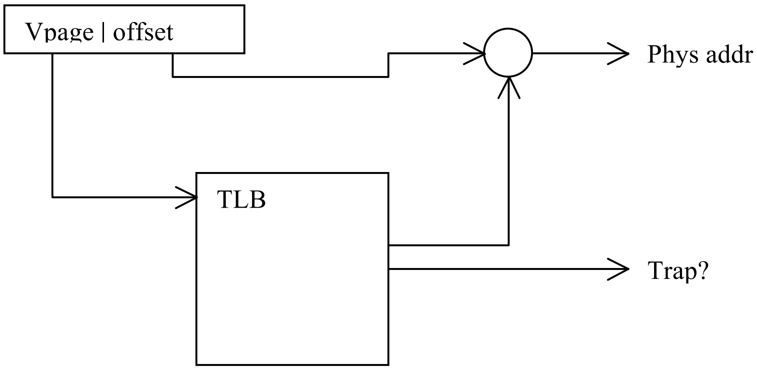
****

Goals of virtual memory:

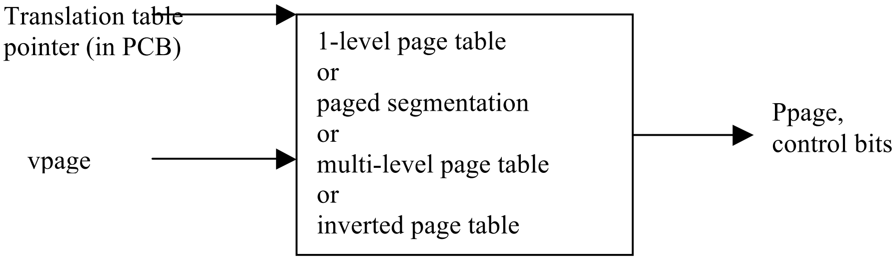
- protection
- relocation
- sharing
- illusion of infinite memory
- minimal overhead
  - space
  - time

Last time: we ended with page table
Evaluation:
Paging
+ simple memory allocation
+ easy to share
- big page tables if sparse address space QUESTION: WHY?

Modern hardware is always:



Memory data structure is opaque object:



****

Outline - 1 min

****

Finish last time:

Scalable schemes:

- Multi-level translation
- Inverted page table
- Others in book: paged page tables, segmented paging, paged segmentation, hashed page tables, …

Quantitative measures:

- Space overhead: internal v. external fragmentation, data structures
- Time overhead: AMAT: average memory access time

Other uses

Copy on write, control bits

Paging to disk

Basic mechanism

Writing and sharing – dirty bit and core map

performance

****

Preview - 1 min

****

Replacement policies

Control bits revisited

****

Lecture - 35 min

****

Finish last time:

Scalable schemes:

- Multi-level translation
- Inverted page table
- Others in book: paged page tables, segmented paging, paged segmentation, hashed page tables, …

Quantitative measures:

- Space overhead: internal v. external fragmentation, data structures

- Time overhead: AMAT: average memory access time

****

Admin - 3 min

Solutions HW2 available

HW 3 out

Project 2 out

****

Lecture - 35 min

****

1. Control bits: Sharing, Copy on write, ...

1.1 Control bits

Once we have our level of indirection, we can do a lot with it.
Mechanism: *control bits* (I've been gliding past that on previous examples)

Control bits -- each page table entry has control bits
e.g.,
"valid" -- useful for sparse page tables
"read" -- possible (but uncommon) to allow execute but not read code
"write" -- useful for sharing code, read only data structures
e.g., code
e.g., kernel can map "information" data structures into user space read only
"execute" -- "no execute" pages are a security feature
"user/kernel" -- page can only be accessed by kernel

Example: x86 page table entry: (from JOS)

```
// Page table/directory entry flags.
#define PTE_P           0x001// Present
#define PTE_W           0x002// Writeable
#define PTE_U           0x004// User
#define PTE_PWT         0x008// Write-Through
#define PTE_PCD         0x010// Cache-Disable
```

```
    #define PTE_A            0x020// Accessed
    #define PTE_D            0x040// Dirty
    #define PTE_PS           0x080// Page Size
    #define PTE_MBZ          0x180// Bits must be zero

    // The PTE_AVAIL bits aren't used by the kernel or interpreted by the
    // hardware, so user processes are allowed to set them arbitrarily.
    #define PTE_AVAIL        0xE00        // Available for software use
#define PTE_AVAIL1      0x200
#define PTE_AVAIL2      0x400
#define PTE_AVAIL3      0x800
```

1.2 Sharing -- read-only sharing

Process 1 has a read/write page at virtual address VA1 that it wants to share with process 2; make this page read-only for process 2 at virtual address VA2 (VA1 != VA2)

What would x86 page tables for P1 and P2 look like?

Can also do this with read+write sharing -- can put shared data structures there and use locks, condition variables to coordinate access

1.3 Copy on write

2 processes want to share data
Each wants to be able to write some of the data
Don't want to see each others' changes

Example: Code sharing (writing is uncommon, but if someone wants to write, let them. Just make sure no one else is affected.)

Example: Fork() -- the fork() system call creates a new process that is identical* (*almost -- see man page for details) to the original

Most pages probably won't change. Some will (must!)

-- one implementation -- make copy on write copy

How to do this:
(1) copy page table,
(2) What do we have to change in page table?
(3) What happens when a process writes

exec() – free all existing pages; create new memory regions (based on executable header info); initialize new memory regions; initialize program counter; ready.

2. memory expansion – paging to disk

Reality: finite physical memory

Abstraction: infinite memory
(OK, not infinite. "Big")

give process the illusion that it has entire virtual address space
2 ideas
(1) sparse address space (don't allocate physical memory for unused regions)
(2) paging to disk (if run out of physical memory, use disk for storage)

Original idea is that we want to be able to run a job that won't fit in memory or run more jobs than will fit

- run large process (> physical memory)
- run many processes ("swap" out processes that are not currently running)

Today: VM paging to disk more important for
- memory mapped files
- sharing executables

How important is virtual memory to expand memory?

“*In operating systems, when you see the word “virtual”, substitute the word “slow””*

VM invented in late 60’s/early 70’s – memory > $10K/MB (today <$0.1) → less important to oversubscribe
70’s – many users per MB 90’s – many MB’s per user
70’s – disk a lot slower than CPU or mem 90’s disk much much slower than CPU or mem

still – convenient – can start up hundreds of shells @1MB each w/o worrying, etc…

in 1970’s inventing VM was quite difficult
Now that we know how to do it, not that hard. So it is worth having around.

Modern use of virtual memory
Memory mapped files. Useful to map a file (on disk) into memory. Access it via load/store rather than system calls to read and write.

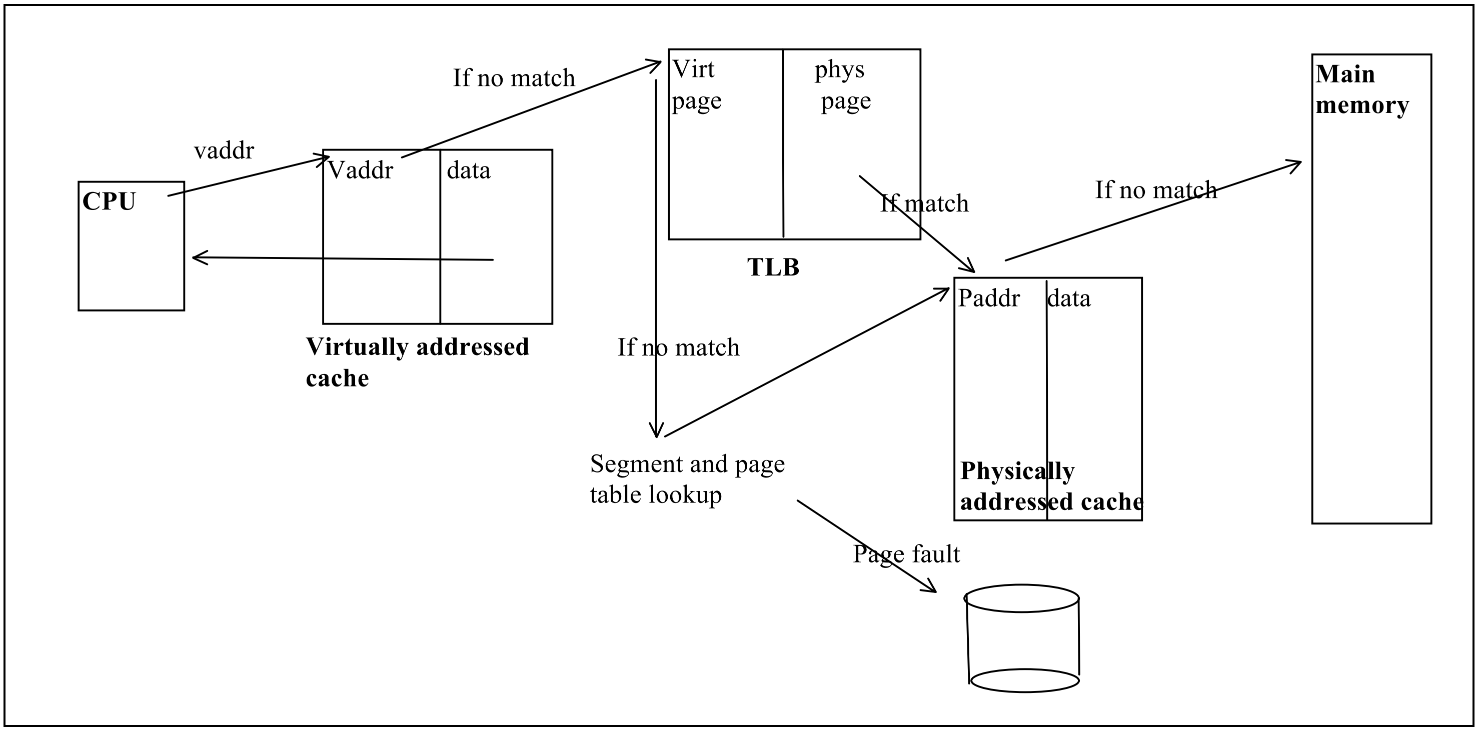
2.1 implementing virtual memory paging to disk (simple)

Basic idea – page table entry:
Frame number | control bits
Add a new control bit “paged”

Valid	paged	frame number
1	0	holds frame number of physical memory
1	1	holds pointer to copy of frame on disk
0	X	address is invalid

page fault – memory reference to data that is currently on disk. Need to bring data from disk into memory to complete reference.

Question: How do you handle a page fault?



How do you handle a “page fault”

1) TLB lookup – miss; fault to OS

2) OS traverses page tables, seg tables, etc. Find page table entry
QUESTION: ok –let’s assume we have page table – what do we see?
QUESTION: How do we know where to go on disk?
-- simple case: swap file --> can put offset in page table
-- modern case: multiple files mapped into memory
--> "memory regions"

3) Find a free physical frame
QUESTION – what would be a good way to implement this?

4) Schedule disk operation to read page into newly allocated frame
QUESTION -- how do we know disk address to read from?

5) when disk read is complete, update page table and TLB

6) restart OS instruction (tricky in general, but knowing what you know about traps/faults how would this work?)

Questions:

1) Book says “invalid = paged to disk” (no “paged” bit). In that case, how do we tell difference between "invalid" page and "paged out" page?
2) on page fault page-in, what consistency issues are there (e.g., what might need to be invalidated in the picture?)
3) on context switch, what consistency issues are there (e.g., what might need to be invalidated in the picture?)

2.2 Modern implementation

Need to support different areas of VM that map to different regions of different files (not just paging everything to swap file)

- executable, memory mapped regions, “anonymous” pages (page to swap file)
- → keep list/tree of virtual memory regions, with
  - start VA
  - size
  - permissions
  - paging file, offset
  - …

3. Paging to disk: implementation issues – writing, sharing, replacement

3.1 Writing

what happens if a page is written?
Write through – send write to immediately lower level (disk)
Write back – send write to lower level when block evicted from higher level
Which one would we use here?

QUESTION: How would you implement write back? How do you know which pages need to be written back and which are OK?

Dirty bit:

- implemented in TLB – when TLB sees a write request to a page, it sets the "dirty bit" in TLB. When evicted from TLB, need to copy dirty bit to page table and core map
- how do you invalidate a dirty page
  - write to disk then mark invalid

3.2 Sharing

Two different virtual addresses can map to the same physical frame

What happens if that frame is paged out (or paged in or moved?)

Solution: core map

coreMap[physPage] is a data structure that tracks info you may want in a replacement policy (e.g., last reference time, dirty, reference count, virtual page to invalidate)

3.3 Replacement

Suppose you have no free pages and want to allocate more memory to a process by sending some page to disk. Which page should you choose?

Basic answer:
LRU: pick page unlikely to be used in near future ~= page not used in recent past

Variations/tweaks

- Prefer replacing clean pages v. dirty (1 disk I/O v. 2)
- Don't wait until you are completely out of pages
  - Pageout daemon might keep 10% free

4. Performance of demand paging

Suppose p is probability of a memory reference causing a page fault

AMAT = (1-p) * memory time + p * page fault time

Problem: memory time O(100ns), disk time O(10ms = 10^7ns)

QUESTION: what does p need to be to ensure that paging hurts performance by less than 10%?
1.1 * t_mem = (1-p)t_mem + p t_disk
p = (.1 t_mem)/(t_mem + t_disk)
~= (.1 * 10^2)/(10^7+10^)
~= 10^-6
At most one access out of 1,000,000 can be a page fault. (Hit rate greater than 99.9999%!)

5. Thrashing

Thrashing – memory overcommitted – pages tossed out while still needed

Example – one program touches 50 pages (each equally likely); only have 40 physical page frames
If have enough pages – 100ns/ref
If have too few pages – assume every 5th reference → page fault
4refs x 100ns
1 page fault x 10ms for disk I/O
→ 5 refs per 10ms + 400ns = 2ms/ref = **20,000x slowdown!!!**

Really important -- even if you don't build OS's, you need to be aware of thrashing.

5.1 Problem: system doesn't know what it is getting in to

Log more and more users into system – eventually:
total number of pages needed > number of pages available

Picture: jobs/sec v. total system throughput

So, what do you do about this?
1) One process alone too big?
Change program so it needs less memory or has better locality

For example, split matrix multiply into smaller sub-matrices that each fit in memory

2) Several jobs?

- figure out needs/process (working set)
- run only groups that fit (balance sets) – kick other processes out of memory
  e.g., suppose you are paging and running at a 10,000x slowdown, if kicking half of the jobs out would get you to stop paging and run at full speed, you trade a 1.5x slowdown for a 10,000x slowdown

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

How do we figure needs/process out?

5.1.1 Working set (denning, MIT mid 60's)

Informally – collection of pages a process is using right now

Formally – set of pages job has referenced in last T seconds

How do we pick T?

1 page fault = 10ms
10ms = 2M instructions
→ T needs to be a lot biger than 1 million instructions

How do you figure out what working set is?

- replacement policy keeps track of time to last access – use information from it (next lecture)

a) modify clock algorithm so that it seeps at fixed intervals (keep idle time/page; how many seconds since last reference)
b) with second chance list – how many seconds since got put on 2nd chance list

Now you know how many pages each program needs. What do you do?

Balance set

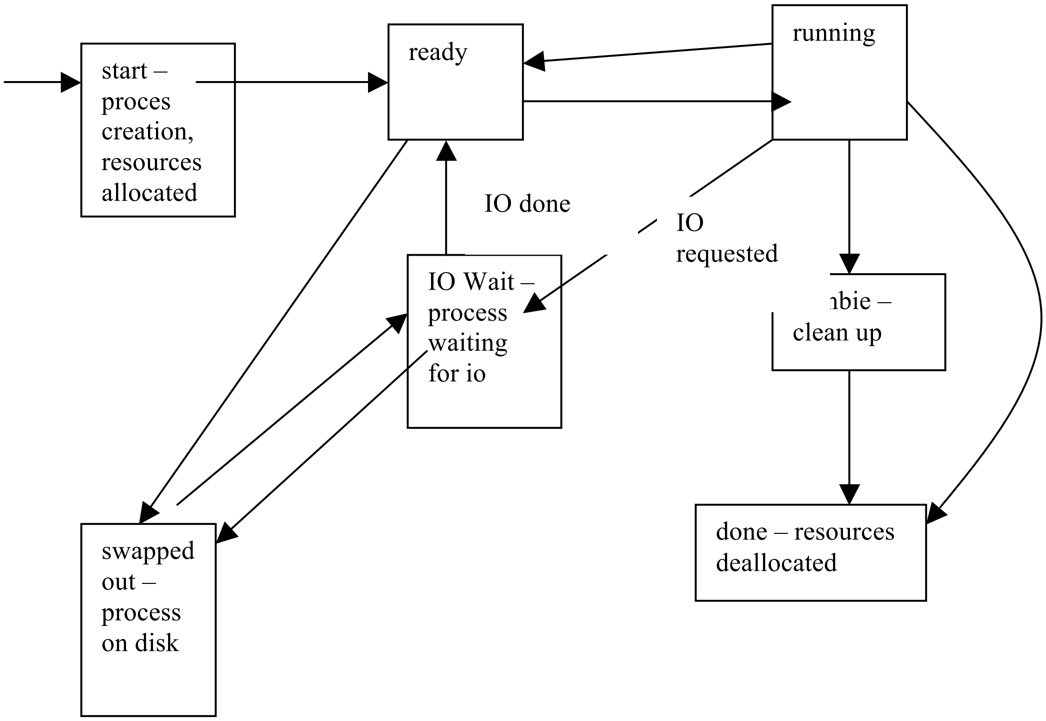
1) if all fits? Done
2) if not? Throw out fat cats; bring them back eventually

What if T too big?
→ waste memory – too few programs fit in memory
What if T too small?
→ thrashing

6. Swapping v. paging

If system low on memory, may be better off moving an entire process to disk and stopping it from running for a while

System stop thrashing -> system runs more efficiently → average (and perhaps all) jobs run faster


start – proces creation, resources allocated
ready
running
IO done
IO requested
IO Wait – process waiting for io
ıbie – clean up
done – resources deallocated
swapped out – process on disk

6.1.1 Need two levels of scheduling

Upper level decides on swapping

- when
- who
- for how long

Lower levels decide who on ready queue actually runs on CPU

Upper level invoked when there are processes swapped out and whenever we need to load more programs than can fit in main memory