

Memory Systems

Why Memory Management?

- better resource usage
- independence and protection
- inaction leading to eviction
- liberate apps from resource limitations
- allow sharing when necessary

Goals

- speed up memory accesses
- keep overhead low (key in all OS efforts)
- minimal hardware support

Evolution (Hardware)

- user's view of memory?
 - program plus OS
- fence register provided by hardware
 - loaded by OS when program loaded in mem
 - logical addresses translated on the fly
- base and bound registers
 - static relocation (at load time)
- base and limit registers
 - dynamic relocation (at run time)

Evolution (OS)

- swapping jobs in and out (resource usage)
 - where in memory swapped in?
- partition memory into fixed size regions
 - bundle in processor scheduling with memory requirement
 - particularly suitable for batch environments with large FORTRAN jobs
 - issues to worry about
 - what if more memory needed during execution?
 - fragmentation (internal and external)

Evolution (OS) Contd...

- variable size partitions
 - table in OS indicating available memory
 - dynamically allocate memory and update table
 - when program terminates or does a free-mem
 - update table to indicate unused memory (“holes”)
 - holes scattered all through memory
 - how to allocate?
 - first fit
 - best fit
 - fragmentation possible?

Evolution (OS) Contd...

- external fragmentation severe with first fit
- how to reduce fragmentation?
 - compaction
 - only with dynamic relocation
 - expensive...so do it rarely
 - combine it with swapping

Paging

- up to now contiguous memory requirement
- break up user's logical view of memory into fixed size pages
- break up physical memory into fixed size blocks called "page frame"
- each physical page frame backs a particular logical page
- contiguous memory requirement confined to a page not entire program image

Hardware for Paging

- page table
 - maps logical to physical
 - where to keep?
 - registers?
 - consider 32 bit addresses
 - in memory
 - what is needed in hardware then?

Memory Allocation with Paging

- what to do on program startup?
- what to do to satisfy dynamic requests?
 - list of physical page frames
 - allocate to satisfy page requests
- fragmentation possible?
- how many page tables?
- what happens on context switch?

- Process control block (PCB)

- state of a running program

```
enum state_type {new, ready, running, waiting, halted};  
typedef struct _control_block {  
    enum state_type state;  
    address PC;  
    int reg_file[NumRegs];  
    struct control_block *next_pcb;  
    int priority;  
    address page_table;  
    .....  
} control_block;
```

Virtual Memory

- allows execution of programs that may not be completely in memory
- how to enable that?
 - what hardware?
 - what software?
- memory overlays (archaic technique)
 - hardware: base and limit registers
- demand paging (current technique)
 - hardware: program support for instruction restart

Demand Paging

- hardware support
 - provide page fault exception
 - restart from fault on instruction address
 - restart from a fault on data address
 - consider indirect addressing
 - impact on pipelined processor design
 - come back to this shortly....
- OS overhead for demand paging
 - pagefault handler time (several instructions)
 - swap in/out page (order of milliseconds)
 - restart process (context switch time)

Handling Page Faults

- locate desired page on disk
 - where is this info kept?
- find a free pframe
 - select a victim page
 - how to choose?
 - global vs. paging against oneself?
 - write back if necessary
 - fixup page tables (faulting and victim procs)
- read in page
- restart faulting process

Page Replacement Algorithms

- norm is global page replacement
- Criterion
 - low page fault rate
 - what if you observe excessive page fault rate despite a smart algorithm?
- FIFO
 - victim is longest resident page
 - what hardware?
 - easy to implement (FIFO queue is sufficient)
 - leads to anomalous behavior (see Fig 9.9 of S&G)

- Belady's min
 - victim is page not referenced for the longest time in future
 - what hardware?
 - provably optimal
 - what is the catch?
 - ideal as a gold standard
- True LRU
 - victim is longest unused page
 - what hardware?
 - what software?
 - what needs to happen on every memory access?

- True LRU
 - hardware
 - stack maintained by CPU
 - top of stack -- MRU page
 - bottom of stack -- LRU page
 - every memory reference
 - update stack
 - software
 - upon page fault
 - pick LRU page as victim
 - how does it access the hardware stack?
 - cost
 - prohibitive...in the critical path of pipelined CPU

- Approximate LRU
 - let's say we want to track references only at page level
 - hardware
 - reference bit per page frame
 - set by hardware; cleared by software
 - how to choose victim?
 - can we do better?
 - bit vector per pframe in software (i.e. OS)
 - periodically get a snapshot
 - h/w bit -> MSB of bit vector; right shift bit vector
 - victim is one with smallest ref count

- optimizing/overlapping paging I/O
 - keep a low water mark for free frames
 - daemon kicks in to restore low water mark
 - i.e. don't wait till page fault to look for a free page
 - keep victim pages in pool of dirty pages to be written out
 - write out pages when paging device free
 - victim page mostly “clean” when selected
 - remember original mapping for victims on free pool
 - restore mapping if fault address matches remembered mapping (no I/O)

Reducing Page Faults

- CPU scheduler increases multiprogramming when utilization drops
 - good idea?
 - what is wrong with this policy?

Thrashing

- more paging activity than real execution
- CPU scheduling policy is flawed if it only looks at utilization
- how to prevent thrashing?
- principle of locality
 - bring into memory current locus of program activity
 - no more page faults for this process until this locus of locality exited
- how to determine loci of program activities?

Working Set

- Approximate loci of localities for a program
- working set window of a process
 - set of consecutive page references (say 10000)
- working set size (WSS) of a process
 - the set of pages touched in the window
- total memory pressure
 - sum of WSS_i over all i
- mem pressure greater than physical mem?
- when can you increase degree of multiprogramming?

Determining WSS

- Approximate using timer interrupt and ref bits
 - sample ref bits every delta time units
 - record pages with ref bits turned on
 - clear ref bits
- use page fault rate as a measure of thrashing
 - high and low water marks for page fault rate
 - increase/decrease pool of free frames when actual page fault rate goes above/below high/low water marks

Other Considerations

- Pre-paging
 - remember working set of swapped out process
 - bring in WS when swapped in
- I/O interlock
 - DMA uses physical addresses
 - what if the page designated for I/O is replaced by the paging daemon?
 - solutions
 - double buffering...bad idea
 - lock down physical pages for I/O

Speeding up Address Translation

- pipelined processor and memory translation
 - five stage pipeline (IF, RR, EXEC, MEM, RW)
 - IF or MEM stage
 - VPN to PPN - first memory access (1 cycle)
 - PPN to data - second memory access (2 cycles)
 - need to shrink this down to one cycle
 - how?
 - use principle of locality

- Translation Lookaside Buffer (TLB)
 - hardware in CPU
 - caches recent address translations in hardware
 - every entry
 - $\langle \text{VPN}, \text{PPN}, \text{access-rights}, \text{VALID/INVALID} \rangle$
 - CPU's VA
 - $\langle \text{VPN}, \text{offset} \rangle$
 - lookup TLB
 - if VALID && access-legal
 - » PA is $\langle \text{PPN}, \text{offset} \rangle$
 - » present PA to memory
 - » memory completes the CPU request (read/write)
 - » memory access completed in 1 cycle!
 - » pipelined CPU happy...end of story....
 - else?

- Translation miss (first approach)
 - hardware's problem
 - CPU uses PTBR to look up page table
 - remember page table entry is $\langle \text{PPN}, \text{access-rights}, \text{V/I} \rangle$
 - gets the translation
 - if VALID && access-legal
 - » update TLB (i.e. create a new entry in TLB)
 - » PA is $\langle \text{PPN}, \text{offset} \rangle$
 - » present PA to memory
 - » memory completes the CPU request (read/write)
 - » how many cycles to complete memory access?
 - » what is the pipelined CPU doing all these cycles?
 - else?
 - » memory exception
 - » now it's software's problem...

- Translation miss (second approach)
 - software's problem all the way...
 - raise translation miss exception
 - what does the OS handler do?
 - use PCB to glean PT address
 - get PPN corresponding to VPN
 - update TLB
 - hardware has to provide instruction to facilitate this...
 - reschedule the process
 - do we need PTBR anymore?

- Uniqueness of TLB lookup?
 - process tag with each entry
 - $\langle \text{pid, VPN, PPN, access-rights, V/I} \rangle$
 - flush TLB on context switch
 - all of it?
 - consider what happens when handler runs
 - plain old instructions...nothing special
 - memory being accessed?
 - handler code
 - page table entries
 - can the handler have a translation miss?
 - It better not be for the handler code itself...
 - page tables themselves may not be paged in.....
 - how to resolve such translation misses?

- virtual address partitioned
 - user
 - system
 - allocate page tables from this partition
 - one system page table plus several user page tables
 - system page table always in physical memory
- correspondingly partition TLB
 - what to throw out on context switch?

- revisit address translation
 - <user's VPN, offset>
 - translation miss fault
 - could lead to page fault as well if user page table not in physical memory
 - <system's handler code VPN, offset>
 - translation miss fault
 - could only be for user page table entry
 - where is this address from?
 - corresponding page table entry guaranteed to be in memory?
 - translation miss for the handler **SHOULD NEVER RESULT** in page fault for the page table entry!
 - it's OK to have page fault to bring in to bring in user page table to memory