To improve CPU utilization in a multiprogramming environment we need multiple programs in main memory at the same time.

Basic CPUs and Physical Memory
- CPU <-> cache <-> Physical memory
  - CPU stall going to main memory
  - cache speedups

Address Binding
- compile time
- load time (relocatable code)
- execution time
- logical (CPU) vs physical (memory) addresses
  - MMU -- changes logical into physical
- PIC (position independent code)

Dynamic Linking and shared libraries
- Needs PIC code
- Usually needs advanced hardware
- Protection from other processes, dynamic address control
Memory Management Techniques

Swapping
- Copying out memory to a "backing store"
- Early systems used "drum storage"
- Disk is used now, both rotating and solid state
- End of quantum, swap out while running another process.
- Larger memory means less swapping
- Thrashing -- spending most of time swapping
- Mobile OSes don’t usually support swapping
- Give other applications memory warnings ...

Contiguous Memory Allocation
- Full memory footprint of programs stored contiguously
- Protection from other processes
  - Base and limit registers (Kernel mode access only)
  - Min and Max for current process
  - real address vs base+offset
- Partition memory for processes
  - Fixed sized partitions
  - Variable sized partitions
Partition algorithms (variable sizes)
- First fit
- Best fit
- Worst fit
- Fragmentation
  - external -- variable partitions
  - internal -- fixed sized or blocked allocation

Paging -- a better solution
- Non-contiguous memory allocation
- Uses special hardware to do address mapping

Basic Method
- Physical memory -- divided up into equal sized frames
- Logical memory -- divided up into pages (same size as frames)
- MMU maps between logical memory (pages) to physical (frames)
- Page table: maps page to frame
- Physical memory can have more frames than logical has pages
- Page table changed between processes (kernel mode only)
Blitz as an example of paging

Memory
- Byte addressable, 32 bit addresses
- 32 bit aligned
- Big Endian
- Page size is 8K bytes
- Page Table Registers
  - PTBR - Page Table Base Register (32 bits)
    - loaded with "ldptbr" instruction
  - PTLR - Page Table Length Register (32 bits) in bytes
    - loaded with "ldptlr" instruction
- Paging is turned on by "Paging Enabled" bit in Status Register set to 1
- Physical memory:
  - Low addresses -- interrupt table
  - High addresses -- memory mapped I/O devices
- When paging is on, logical address are broken up as:
  - 31-24: high order bits (8 bits)
  - 23-13: page number bits (11 bits)
  - 12-00: offset (13 bits)
Review:
- 31-24: high order bits (8 bits)
- 23-13: page number bits (11 bits)
- 12-00: offset (13 bits)

Physical addresses are generated from logical addresses as:
- high order bits ignored (16M max mapped address space)
- Page number selects a 32 bit entry from the page table
- frame number in the page table is concatenated with offset
- result is physical memory address

Page table entry format:
- 31-13: frame number
- 12-04: ignored (may be used by OS)
- 3: Dirty bit (1 => updated)
- 2: Referenced bit (1 => referenced)
- 1: Writable bit (1 => writable, 0=>read only)
- 0: Valid bit

Memory cycle: R or W, logical address
- May cause one of several exceptions: Page Invalid, Page Read-Only, Address exception
Blitz as an example of paging (page 3)

Referenced and Dirty bit
- set by the MMU as part of the operation
  - Referenced => used for either a read or write
  - Dirty => page has had a store to it

More Complex Paging Hardware
- Multi-level page tables: (32 bit x86 series, old NS 32532)
  - Page size 4k, 32 bit addresses
  - 1024 entries in each page table page
- Virtual Address
  - 11-00: offset
  - 21-12: entry B
  - 31-22: entry A
- Register has a pointer to Table A (Page directory)
- Entry A: 1K entries pointing to entry B tables (Page Table)
- Entry B: 1K entries pointing to frames
- Filled out page tables ... 1025 4K pages: 4,198,400 bytes
- Entries similar to blitz: bits 31-12, frame number,
  - 11-7 unused (mostly), 6 dirty, 5 referenced, 4 cache disable
  - 3 write-through, 2 user/supervisor, 1 R/W, 0 valid
RISC-V Summary

- **SV 32** -- very similar to i386, 4K pages, 2 level page tables
  - 1K entries per page table, 32 bits/entry
  - ANY PTE can be a "leaf"
  - 4K page and a 4M page (must be 4M aligned, super page)
- **SV 39** -- Add another level, 3 levels
  - 4k base page, 64 bit entries -> 512 entries per page table
  - ANY PTE can be a "leaf", 4K, 2M, and 1G pages! (aligned)
- **SV 48** -- Add another level, 4 levels
  - 4k base page, 64 bit entries -> 512 entries per page table
  - any PTE can be a "leaf", 4k, 2M, 1G and 512G pages. (aligned)
- **Ref:** [https://riscv.org/specifications/privileged-isa](https://riscv.org/specifications/privileged-isa) pg 59
Larger Page Tables

Some 64-bit architectures use other techniques:

- **Hashed Page tables**
  - Entry: Virtual Address, Frame, chain address
  - Size of table an issue
- **Clustered page tables** — similar to hashed
  - Each entry points to a cluster of pages
  - 8, 16, or 32 pages in cluster
  - Makes smaller hash tables
- **Inverted Page Tables (Ultra sparc, Power PC)**
  - Issue: regular page tables may take lots of memory
  - Solution: frame table
    - Entry: Process Id, Virtual Address/page
    - VA: \(<\text{processID, Page #, offset}>\)
    - Inverted table is searched for \(<\text{PID, Page#}>\)
    - Inverted table in associative memory or hash table
    - Harder to implement shared memory

Shared pages?

- blitz/i386/RISCV-5 methods
- Hashed and inverted tables
Other hardware support for paging

Problem: Where to store page tables?
- In special registers?
  - Process switch requires one to reload registers
- In memory?
  - Each memory reference needs to look up a page table entry
  - PTBR (Page Table Base Register)
    - Makes process switch much easier, one register
    - double (or more) the time to access memory
- Solution to this ... TLB (Translation Look-aside buffer)
  - High speed associative memory
  - Stores <page number, frame number> pairs
  - Can get it "invalidated" or "flushed"
  - TLB set up by accessing a page table
  - Fewer entries than total pages available
  - Sometimes TLB entries can be "wired down"
  - Some TLBs store <pid, page number, frame number>
    - Can be used by multiple processes concurrently
Segmentation

Another twist of logical addresses to physical addresses
- Idea of various segments
  - e.g. Text, Global, Heap, Stack
  - may expand memory by using unique addresses for each segment
  - e.g. Often know when fetching instructions vs data
- Old example: HP 3000, 63 text segments, 1 or 2 data segments
  - Segments can help do shared libraries
  - Allowed for larger memory space than 16bit addresses allowed
- More recent example: IA-32
  - Up to 16K segments, each segment 4G
    - 8K shared segments, 8K private segments
  - 6 segment registers to allow a process to address multiple segments
  - final physical address, 32 bit
  - doesn’t allow larger physical than logical spaces
Virtual Memory (Chapter 10)

Previous chapter
- multiple processes in memory at the same time
- techniques to share main memory
- page table mechanisms

This chapter -- complete memory view for processes
- how to manage memory (by kernel) for processes

Basic requirement -- instructions/data must be in real memory to use them
- not all data/instructions need to be in memory all the time
  - some unused code may never be needed
- logical memory may be larger than physical memory
  - Old times ... overlays
- Error cases may not be needed
- Complete subsystems may be unused during a particular run
- Programmer allocates 100x100, user uses 10x10
- Allow placing of data/instructions in memory only if needed.
  - Not all of all segments are mapped
  - Allows more processes in main memory at the same time
Virtual Memory -- separation of logical (user) view from physical memory
- Programmer can program with a large VM address
- Programmer can view it as linear and contiguous
- Paging hardware allows for "shared pages"
  - Use to get shared libraries implemented

Demand paging
- A different kind of "swapping"
  - Swapping?
    - Save entire process memory to disk, reload to memory to run
  - Demand paging can be a "lazy swapper"
    - Process doing this is the "pager"
    - Code (text) of program is on disk
    - Can allocate disk space for process "r/w memory" to be saved.
Demand paging (page 2)

- don’t load memory from disk until it is needed
  - How?
    - Page fault => need page X
    - find page X on disk
    - load page X into memory
    - update page table
    - rerun instruction
  - uses the valid bit in the page tables
  - larger page tables (multi-level) demand load page tables
Pure demand paging ...

- start program running without any pages in memory!
  (New program & process )
- not cool ... there are known pages needed
  - instructions at load position
  - initial stack location
  - global data possibly
- performance is an issue for demand paging
- levels of access: cache -> memory -> disk
- times? 10ns, 200ns, 8ms
- effective access time = (1-p)*ma + p*pft
  - ma = memory access time (ignoring cache effects),
  - pft = page fault time, p = fault probability 0 <= p <= 1
- Use ns (nano seconds)
- ma = 200
Demand paging (page 3)

Performance (continued)

- pft?
  - trap, context switch, call page fault function
  - find page, lookup disk file, schedule page load
  - wait for page to be loaded
  - return from trap (context switch ....)

- up to 8 ms (or more!)

- time = (1-p) * 200 + p * 8000000

  = 200 + 7999800*p

- If p = .001 (one out of 1000) => 8.2 microsec memory cycle!

- 10% performance degradation?
  - 220 > 200 + 7999800*p
  - 20 > 7999800*p
  - p < 0.0000025 ( 1 in 400000)
Process creation

Fork():
- Have a running process with a complete memory image
- Options?
  - Copy the entire memory space?
  - Copy on write!
    - On fork, turn all page entries to R/O
    - A process that gets a page fault due to write
      - copy frame
      - set each page table to point to a different one
  - (Harder to do on inverted page tables)
  - Processes share R/O pages
- Advantage of copy-on-write?
  - Don’t copy and then throw away!
- vfork()?
  - suspend parent, let child run using parent’s memory
  - child should immediately call exec().
  - child change of memory will show up in parent!
Exec(): Keep current PCB and so forth, rebuild memory image
Demand Paging and Page Replacement

With demand paging comes something not as expensive as swapping ...
- page removal from frames when out of frames
  - page fault -> need more memory
  - memory is full, need to reuse a frame
  - take a page from some other process
    - Dirty or clean page?
      - clean if possible
don’t have to write it out
don’t have to wait for it to be written
- Algorithm for selecting frame/page to throw out... (page replacement algorithms)
  - most OSes have their own scheme .... but
  - there are standard algorithms to consider
    - FIFO
    - Issues?
May throw out one you need soon
Belady’s anomaly -- more frames increase page fault rate in some cases
Expect more page frames lower fault rate
Optimal Page replacement
   □ Always replace the page that will not be used for the longest period of time.
   □ Doesn’t really exist

Trying to approximate the Optimal Page replacement algorithm
   □ Least recently used (LRU)
      □ page that has not been used for the longest
      □ assume it will not be needed soon
      □ locality of reference in code and data

How to implement?
   □ Hardware support is essential
   □ Counters -- add a memory reference counter to hardware
      □ Access to a page stores counter to that page table entry
      □ Smallest counter in page table is LRU page
   □ Stack -- add a stack to the page table
      □ Each memory access puts the current page on top of stack
      □ Entries are not allowed to be duplicated
      □ Entry at the bottom of stack is LRU page
Problem?
  - Few computers supply previous mentioned hardware support
  - What do they provide?
    - Referenced and Dirty Bits -- like Blitz
LRU approximation algorithms
  - These algorithms assume that the referenced and dirty bits are cleared on load

second-chance
  - Basic algorithm is fifo
  - when a page is selected, check reference bit
    - if 0, replace
    - if 1, add to end of fifo and clear reference bit

enhanced second-chance
  - use referenced and modified, use the pair (r,d)
    - (0,0) not referenced, not modified, good choice
    - (0,1) not referenced, modified, requires a page out also
    - (1,0) referenced, not modified, may be used again
    - (1,1) referenced, modified, most likely in active use, page out required
  - replace the oldest in the lowest non-empty class first
Additional-reference-bits
- keep an extra integer value for each page in memory (8 bits works)
- at a regular interval shift reference bit into extra integer at MSB
  - 100 ms a good time?
  - right shift R -> extra_int, lsb (least significant bit) drops out
  - clear the reference bit
- replace page with smallest extra integer
- vary the number of bits
- extreme case of 1 bit => second-chance algorithm

LFU - Least frequently used
- keep a count of the number of times the page is used
  - hardware counter?
  - reference bit?
- small counts imply not frequently used
- Issue: Initial use page, not used later
  - Solution: count aging

MFU - most frequently used
- idea is that new pages just brought in have not been frequently used
Other paging related ideas

Page-Buffering
- Keep a collection of free frames -- the pool
- page fault -> select page to replace via algorithm
  - get a free frame for new page, start read immediately
  - if old frame is dirty, write it out, then add it back to the pool
- want to keep a minimum number in the free frame pool
- allows process to resume faster than a "write page, load page" operation
- Modification to improve "write times"
  - when paging device is idle, select a modified frame to write out
  - improves the probability that the page is not dirty when selected for page out
- Another tweak -- pages in the free pool "remember" which page they contain
  - A page fault for a page in the free pool requires no I/O to restore
  - works well with FIFO or second-chance
  - works with other paging algorithms
Frame Allocation

How should frames be allocated to processes?
- Equal allocation?
- Proportional allocation?

First ... minimum frame count?
- Instruction length ... can it cross a page
- Data access:
  - number of memory locations per instruction
  - any indirection
  - infinite indirection?
  - limit to 16 levels or so

Equal allocation: m frames, n processes, each gets m/n

Proportional allocation: frames needed total vs frames needed by all processes
- P_i_frames_needed/total_frames_needed * frames_available

Priority allocation: give more frames to high priority processes
Global versus Local allocation
  □ Process i gets a page fault
    □ look only at pages owned by P_i? (local)
    □ look at all possible frames? (global)
  □ Global -- a process can’t control own fault rate
  □ Local -- may not get access to unused memory
    □ large program spending lots of time in small part of program/data
  □ Global used most often

Non-uniform memory access issues
  □ Multi-CPU / Memory Module systems
  □ CPU access faster to some memory
  □ choose frames with minimum latency
Thrashing

- CPU utilization vs degree of multiprogramming
  - At some point, increasing the load decreases the CPU utilization
  - Happens more often in global replacement algorithms
    - spend more time doing paging than CPU work
- Locality
  - a set of frames actively used together
  - a way to help quantify what pages should be in memory
  - may have several localities during the running of a program
  - don’t have enough pages for locality ... the process thrashes

Working Set Model
- Parameter: delta time -- working-set window
- Varies over time
- Find a working set for each process
- Keep each frame allocation to working set
- Helps stop thrashing and increase CPU utilization
- May be able to help detect working set via page fault rate
Memory Mapped Files and Shared Memory

Techniques using paging

- Memory Mapped Files
  - Map file, don’t page in contents
  - Access to file is via memory "reference"
  - Uses pager to get data into memory
  - Automatic write back of "dirty pages"
  - Allows multiple processes to use same file
- CMU: Recoverable Virtual Memory (RVM)
  - Build a data structure in memory
  - Copy goes to disk
  - Run program again, recover data structure

- NetBSD on small files:
  - `mmap(source file); mmap(dest file); memmove(); close();`

Shared Memory

- Use page tables, map same physical frames into logical adr space of 2 or more processes
- Can map as r/w or r/o pages
- SYSV API for shared memory
Kernel Memory and Allocation

Kernel memory is somewhat different than "user memory"
- Still using from limited frame pool
- Hardware may require contiguous memory, e.g. DMA buffers
- Some OSes may not run in a paged mode
- How about a page fault while running in kernel mode?
  - Error for most OSes.
- Read about Buddy System ... not really that good
- Typical allocators
  - subsystem allocates frames
  - may hand out smaller chunks to other parts of the kernel
  - large allocations may be integral number of frames, contiguous
- Typical OS
  - on boot find free frames
  - initialize a kernel memory allocation
  - use "free frames" for both user pages and kernel allocation
  - kernel allocation may interfere with user processes by grabbing frames
- NetBSD -- pmap component maps physical memory
Other issues

Prepaging -- Trying to predict page needs and get the page in memory before use
- May do this for a newly exec()ed process and processes being swapped in.
- Possible problems: guessed wrong, too much prepaging

Page size?
- Often the hardware dictates page size
- Some machines offer several page sizes
  - small pages -> more efficient memory use (fragmentation)
  - larger pages -> less paging
- time required to read/write a given page

TLB reach
- TLB (Translation Lookaside Buffer)
- TLB reach -- amount of memory accessible from the TLB
- page size * number of TLB entries
- would like working set all from the TLB
- Some architectures allow for multiple page sizes
  - means TLB is partially managed by software
I/O and frames
- Common I/O technique, DMA (Direct Memory Access)
- DMA uses real memory addresses
- What if user buffer crosses a page boundary?
  - Don’t do DMA to user memory
  - OR Move pages to be contiguous
- Lock (or pin) a frame in memory for I/O operation
- Lock frames for kernel into memory
  - OSes don’t like to generate page faults themselves!

Kernel access to user data:
- Blitz:
  - Virtual vs Real address
  - Kernel running without mapping, user running with mapping
- Other machines, e.g. i386 or NS32532
  - kernel and user both mapped
  - NetBSD: uiomove() function
  - Boot time: start running at real addresses, switch to virtual

Read section 10.10