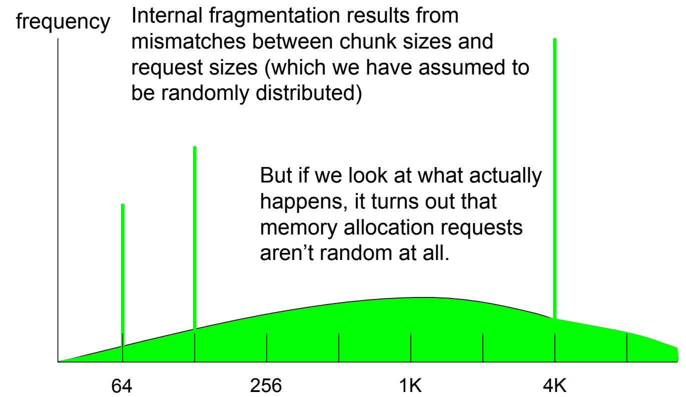
Swapping and Relocation

- 5D Advanced allocation techniques
- 5H Memory compaction
- 3F Execution state model
- 6A Swapping to 2ndary storage
- 5E Dynamic segment relocation
- 6B Paging Memory-Management Units

Memory Allocation Requests



(memory allocation requests)

- memory requests are not well distributed
  - some sizes are used much more than others
- many key services use fixed-size buffers
  - file systems (for disk I/O)
  - network protocols (for packet assembly)
  - standard request descriptors
- these account for much transient use
  - they are continuously allocated and freed

Special Buffer Pools

- if there are popular sizes
  - reserve special pools of particular size buffers
  - allocate/free matching requests from those pools
- benefit: improved efficiency
  - much simpler than variable partition allocation
  - reduces (or eliminates) external fragmentation
- but ... we must know how much to reserve
  - too little: buffer pool will become a bottleneck
  - too much: we will have a lot of idle space

Buffer Pools – Slab Allocation

- requests are not merely for common sizes
  - they are often for the same data structure
  - or even assemblies of data structures
- initializing and demolition are expensive
  - many fields and much structure are constant
- stop destroying and reinitializing
  - recycle data structures (or assemblies)
  - only reinitialize the fields that must be changed
  - only disassemble to give up the space

Balancing Between Competing Pools

- many different special purpose pools
  - demand for each changes continuously
  - memory needs to migrate between them
- we need another *dynamic equilibrium* process
  - configurable/managed free space margins
    - maximum allowable free space per service
    - services must return space beyond these limits
    - graceful response to changing loads
  - claw-back call-back from OS to services
    - OS requests services to free all available memory
    - does not require returning space unless it is needed

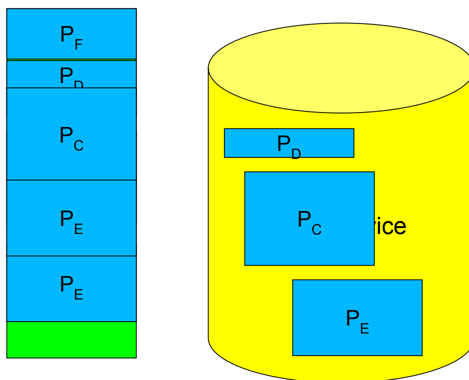
Memory Management

1. allocate/assign physical memory to processes
   – explicit requests: malloc (sbrk)
   – implicit: program loading, stack extension
2. manage the virtual address space
   – instantiate virtual address space on context switch
   – extend or reduce it on demand
3. manage migration to/from secondary storage
   – optimize use of main storage
   – minimize overhead (waste, migrations)

What to do when coalescing fails

- garbage collection is just another way to free
  – it neither helps nor hurts fragmentation
- ongoing activity can starve coalescing
  – chunks reallocated before neighbors become free
- we could stop accepting new allocations
  – convoy on memory manager would trash throughput
- we need a way to rearrange active memory
  – re-pack all processes in one end of memory
  – create one big chunk of free space at other end

Memory Compaction



Primary and Secondary Storage

- primary = main (executable) memory
  – primary storage is expensive and very limited
  – only processes in primary storage can be run
- secondary = non-executable (e.g. disk/SSD)
  – blocked processes can move to secondary storage
  – swap out code, data, stack, non-resident context
  – make room in primary for other "ready" processes
- returning to primary memory
  – copy process back when it is unblocked

Why we swap

- Make the best use of limited memory
  – a process can only execute if it is in memory
  – max # of processes limited by memory size
  – if it isn't READY, it doesn't need to be in memory
- Improve CPU utilization
  – when there are no READY processes, CPU is idle
  – idle CPU time is wasted, reduced throughput
  – we need READY processes in memory
- Swapping takes time and consumes I/O
  – so we want to do it as little as possible

Swapping Out

- Process' state is in main memory
  – code and data segments
  – non-resident process descriptor
- Copy them out to secondary storage
  – if we are lucky, some may still be there
- Update resident process descriptor
  – process is no longer in memory (ready)
  – pointer to location on 2ndary storage device
- Freed memory available for other processes

Swapping Back In

- Re-Allocate memory to contain process
  - code and data segments, non-resident process descriptor
- Read that data back from secondary storage
- Change process state back to Ready
- What about the state of the computations
  - saved registers are on the stack
  - user-mode stack is in the saved data segments
  - supervisor-mode stack is in non-resident descriptor
- This involves a lot of time and I/O

un-dispatching a running process

- somehow we enter the operating system
  - e.g. via a system call or a clock interrupt
- state of the process has already been preserved
  - user mode PC, PS, registers are already saved on stack
  - supervisor mode registers are also saved on (the supervisor mode) stack
  - descriptions of address space, pointers to code, data and stack segments, and all other resources are already stored in the process descriptor
- yield CPU – call scheduler to select next process

Blocking and Unblocking Processes

- Process needs an unavailable resource
  - data that has not yet been read in from disk
  - additional memory that is not yet available
  - a message that has not yet been sent
  - a lock that has not yet been released
- Must be blocked until resource is available
  - change process state to blocked
- Un-block when resource becomes available
  - change process state to ready

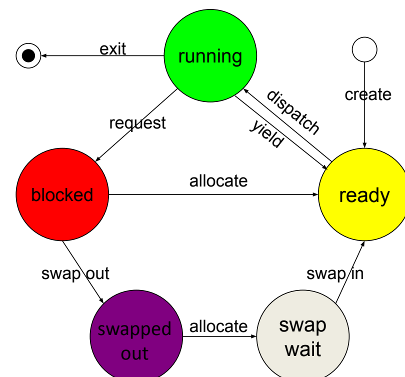
Blocking and unblocking processes

- blocked/unblocked are merely notes to scheduler
  - blocked processes are not eligible to be dispatched
- anyone can set them, anyone can change them
- this usually happens in a resource manager
  - when process needs an unavailable resource
    - change process's scheduling state to "blocked"
    - call the scheduler and yield the CPU
  - when the required resource becomes available
    - change process's scheduling state to "ready"
    - notify scheduler that a change has occurred

(re-)dispatching a process

- decision to switch is made in supv mode
  - after state of current process has been saved
  - the scheduler has been called to yield the CPU
- select the next process to be run
  - get pointer to its process descriptor(s)
- locate and restore its saved state
  - restore code, data, stack segments
  - restore saved registers, PS, and finally the PC
- and we are now executing in a new process

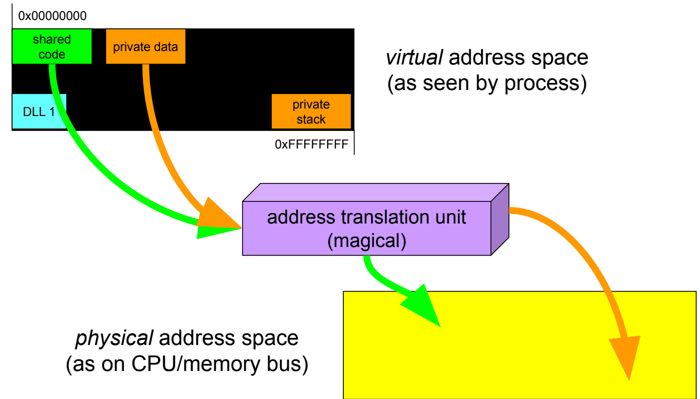
execution states with swapping



The Need for Dynamic Relocation

- there are a few reasons to move a process
  - needs a larger chunk of memory
  - swapped out, swapped back in to a new location
  - to compact fragmented free space
- all addresses in the program will be wrong
  - references in the code, pointers in the data
- it is not feasible to re-linkage edit the program
  - new pointers have been created during run-time

Virtual Address Translation



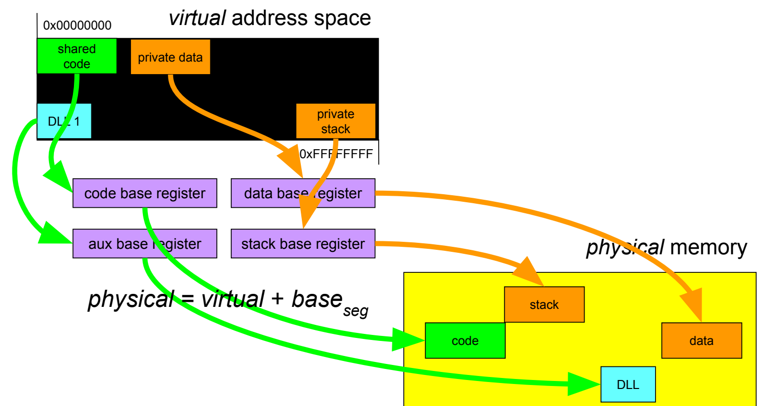
virtual address space (as seen by process)

physical address space (as on CPU/memory bus)

Segment Relocation

- a natural unit of allocation and relocation
  - process address space made up of segments
  - each segment is contiguous w/no holes
- CPU has segment base registers
  - point to (physical memory) base of each segment
  - CPU automatically relocates all references
- OS uses for virtual address translation
  - set base to region where segment is loaded
  - efficient: CPU can relocate every reference
  - transparent: any segment can move anywhere

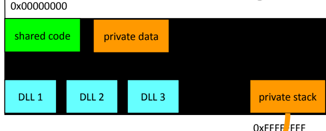
Segment Relocation

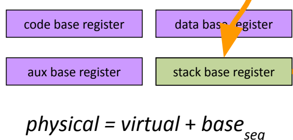


physical = virtual + base$_{seg}$

Moving a Segment

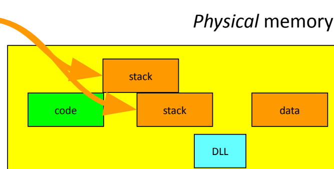
The virtual address of the stack doesn't change





Let's say we need to move the stack in physical memory

Physical memory



physical = virtual + base$_{seg}$

We just change the value in the stack base register

Privacy and Protection

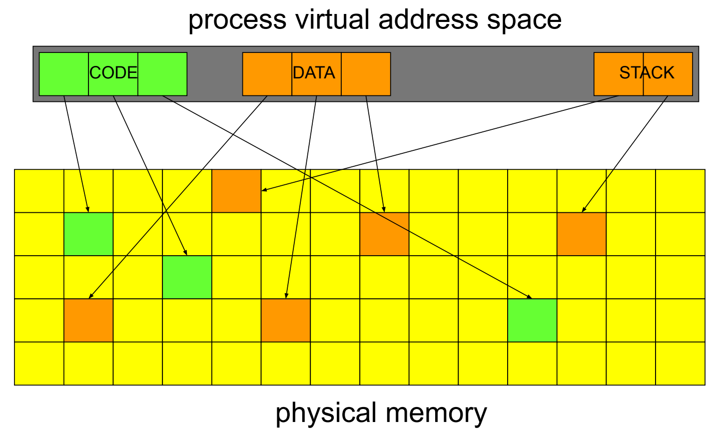
- confine process to its own address space
  - each segment also has a length/limit register
  - CPU verifies all offsets are within range
  - generates addressing exception if not
- protecting read-only segments
  - associate read/write access with each segment
  - CPU ensures integrity of read-only segments
- segmentation register update is privileged
  - only kernel-mode code can do this

Are Segments the Answer?

- a very natural unit of address space
  - variable length, contiguous data blobs
  - all-or-none with uniform r/w or r/o access
  - convenient/powerful virtual address abstraction
- but they are variable length
  - they require contiguous physical memory
  - ultimately leading to external fragmentation
  - requiring expensive swapping for compaction

… and in that moment he was enlightened …

paged address translation

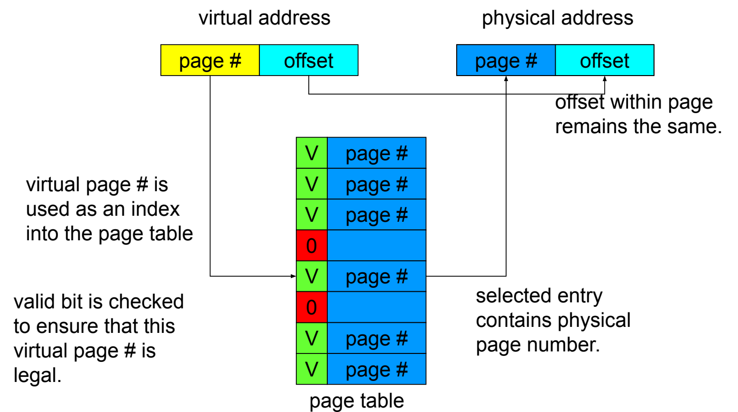


Paging and Fragmentation

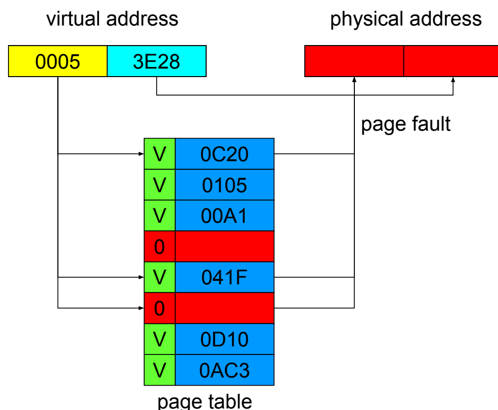
a segment is implemented as a set of virtual pages

- internal fragmentation
  - averages only ½ page (half of the last one)
- external fragmentation
  - completely non-existent (we never carve up pages)

Paging Memory Management Unit



Paging Relocation Examples



Reading and Assignments

Reading:

- Arpaci C19 … TLBs (h/w, but s/w works the same way)
- Arpaci C21 … swapping
- Arpaci C22 … swapping policy
- Kampe: Working-set Replacement

Projects:

- start looking at project 4A (set-up and bring-up)

Supplementary Slides

Pure Swapping

- each segment is contiguous
  - in memory, and on secondary storage
  - all in memory, or all on swap device
- swapping takes a great deal of time
  - transferring entire data (and text) segments
- swapping wastes a great deal of memory
  - processes seldom need the entire segment
- variable length memory/disk allocation
  - complex, expensive, external fragmentation

Managing Secondary Storage

- where do pages live when not in memory?
  - we swap them out to secondary storage (disk)
  - how do we manage our swap space?
- as a pool of variable length partitions?
  - allocate a contiguous region for each process
- as a random collection of pages?
  - just use a bit-map to keep track of which are free
- as a file system?
  - create a file per process (or segment)
  - file offsets correspond to virtual address offsets

implementing a paging MMU

- MMUs used to sit between the CPU and bus
  - now they are typically integrated into the CPU
- page tables
  - originally implemented in special fast registers
  - now, w/larger address spaces, stored in memory
  - entries cached in very fast registers as they are used
    - which makes cache invalidation an issue
- optional features
  - read/write access control, referenced/dirty bits
  - separate page tables for each processor mode

updating a paging MMU

- adding/removing pages for current process
  - directly update active page table in memory
  - privileged instruction to flush (stale) cached entries
- switching from one process to another
  - maintain separate page tables for each process
  - privileged instruction loads pointer to new page table
  - reload instruction flushes previously cached entries
- sharing pages between multiple processes
  - make each page table point to same physical page
  - can be read-only or read/write sharing