INTRO

Resources

- Serially reusable (time multiplexing)
    - Multiple clients one at a time
    - Requires graceful transition (like new condition)
- Partitionable (spacial multiplexing)
    - Disjoint pieces for different clients
    - Access control between partitions
- Sharable
    - Used by multiple clients at once
    - Don't have to wait for resource

Libraries

- Reuse code in well-maintained copy
- Static: include in load module when linked
- Dynamic: choose and load at run-time
- Shared: map into address space at execution time
    - Advantage is reduced memory consumption, faster startup, simplified update
    - Disadvantage is can't have global data, always added to program memory

PROCESS

Process and Stack Frames

- Procedure call creates a new stack frame
- Local variables
    - Save registers (PC, etc.)
- CPU has stack support
- Hardware solutions for push/pop

Address Space - Stack

- Size depends on the program
- Stack grows as program calls more procedure calls
    - Can be recycled once call returns
- OS manages process's stack
    - Can be fixed sized or dynamically extended
    - Read/write and process private

Process State

- Registers (general, PC, processor status, stack/frame pointer)
- OS resources

    - Open files, cwd, locks
- Requires a data structure to hold this information
- Some are not stored in process descriptor
    - Execution state is on stack
    - Can be stored in supervisor-mode stack

Process Descriptor

- Stores state, references to resources, information about support processes
- Used for scheduling, security, allocation
- Inserted into the process table (unique key-value pairs)

Creating new process

- OS using a method to initialize
    - No initial state or resources (windows approach)
- Requested by another process
    - Clone the calling process (unix approach)
    - Notion of parent/child relationship
    -

Fork

- Creates two processes with diff IDs but mostly same
- Parent goes 'one way' and child goes 'the other'
- Child process
    - Own empty stack space
    - Shared code reference
    - Data starts out the same but may not stay the same...
- Copy on write
    - Creating an entire new copy for the child is expensive
    - Only when a process writes to data, the copy is made
    - Lazy way of creating data segments
    - Done at fine granularity (by pages, not by copying the entire data segment)

Exec

- For making an entirely new process
- Used in conjunction with Fork call (can't be run by itself)
- Changes the code section of a process and resets state

Destroying a process

- Can be killed by the OS
- Needs to reclaim memory, locks, and other resources
- Inform other processes that this process is over
- Remove from process table

Running a process

- Ran by CPU (hardware)
- num processes >> num cores
- Scheduler regulates when and where processes are run
- Limited Direct Execution
    - Without OS intervention...
    - Unless the program makes a system call (hits a trap) and transfers control to the OS
    - To optimize performance, enter the OS as seldom as possible

Loading a processes

- Initialize hardware to clean state (process must get CPU in like-new condition)
- Load registers
- Init stack and stack pointer
- Set up memory structures
- Set PC

Exceptions

- Sync exceptions
    - Can be handled by the code or the OS (may ekill program)
- Async exceptions (seg fault, abort, power failure)
    - Unpredictable so the code can't check for them
    - Try/catch blocks
    - Sometimes they are used for system calls
    - Hardware and OS catch exceptions and give control to OS

Using Traps for system calls

- Priviledged instructions for system calls
- Prepare args for the sys call
- Linker will replace the original system call instruction with a trap
- Send the particular system call code to the OS
- Return back to instruction after the sys call

System Call Trap Gates

- Trap goes to trap vector table, where PS/PC are pushed onto the stack
- Trap handler then redirects to the system call dispatch table
- Dispatch table then goes to the 2nd level handler where the system call is impelemented
- When 2nd level handler returns, program returns to user mode, registers are restored

Stacking and Unstacking System calls

- Two stacks: one for user mode, one for kernel mode
- System calls use the kernel mode stack (contain return address to user mode, etc)

Blocked Process

- OS maintains which processes are Blocked

- Could be waiting for I/O
- Once resource is available, scheduler/resource manager can mark the process as unblocked
- Blocking is needed for the schedule to know to wait

Process Queue

- Data structure created by scheduler to determine the order to run processes
- All processes in queue are in 'ready' state

SCHEDULER

Scheduling Goals (relative priority varies depending on use case)

- Throughput - as much work as possible (for servers)
- Average wait time - interactiveness (for smartphones)
- Fairness - minimize worst case time (for multi-users)
- Priority goals - certain processes are more important (for different groups)
- Real time - items have deadlines to be met (niche case ie missile defense)

Scheduling: Policy and Mechanism

- Policy is the ideas of how the OS should act
- Mechanism is how the OS accomplishes the desired policy
- Separation of policy and mechanism makes it easier to change just one

Why don't we get ideal Throughput

- Overhead to switch (ie save registers, switch)
- Scheduler takes time to dispatch (super linear??)
- Response time exploding
    - Systems have finite limits (queue size)
    - Graceful degradation
        - When system overloads, should continue to work but slightly worse
-

Real time schedule

- Certain taks need to happen at particular times
- Hard real time schedulers
    - System fails if the deadline is not met
    - Requires very careful analysis, cannot be dynamic
- Soft deadlines
    - Okay missing deadline, but goal is to meet
    - Different classes of deadlines (some more important than others)
    - Can be dynamic
- If deadlines are missed...
    - Drop the job
    - System may fall behind

-

Preemptive scheduling

Clock scheduling

- Requires a clock if programs don't relinquish control
- Clock generates an interrupt at a fixed time interval

Costs of Context switch

- Entering the OS - Interrupt, save register, call scheduler
- Time for scheduler to decide which work to run
- Switch stack and address spaces to new process
- Lose instruction and data caches

Multi-Level Feedback Queue

- Create multiple ready queues
  - Short time tasks finish quickly (small time slices) -> improve interaction
  - Long time tasks take longer (large time slices) -> becomes more like non-preemptive system, more efficient
- Deciding which processes go in which queues
  - Start all processes in short time queue
  - Move to longer queue if too many time slice ends
  - Move to back to short time queue if end before time slice
- Real time queue doesn't use preemptive scheduling
- Dynamic and automatic adjustment based on job behavior

Priority scheduling Linux

- "nice" value
- Cannot raise priority by normal user, need to be sudo

MEMORY

-Physical and Virtual Address

- Each process appears to have infinite memory
- Layer of abstraction between process and the hard disk

Memory Management strategies

- Protection
  - Enforced partition boundaries
  - Implemented using special hardware registers
- Fixed Partition allocation

    - Preallocate partition for each processes
    - Assume set sized chunks
    - Simple, easy to implement

- Dynamic Partition Allocation

    - Variable sizes
    - Can't relocate memory after being given to a process
    - Still subject to internal fragmentation since process asks for more than used
    - Also subject to external fragmentation

Fragmentation

- Internal: memory given to a process that is not being used
    - Occurs as a result of fixed size partition
    - Average waste is 50% of a block
- External: each allocation creates extra chunks between Partitions
    - Leftover chunks become smaller and smaller
    - Requires a daemon process to consolidate small pieces back into consolidated pieces

Relocation

- Requires copying an entire segment of memory to a new space
- Very difficult and expensive
-

Free Lists for Dynamic Partition

- Store metadata - size of chunk, status, pointer to next
- Algorithm:
    - Start with single "heap"
    - Maintain a 'free list' to keep track of unallocated memory
    - When process asks for memory
        - Find a large chunk
        - Carve out piece of requested size, make new header for residual chunk
        - Put remainder back in list
    - When process gives up memory
        - Put memory back in free list
    - Free
- Run 1 layer in the kernel

Deciding how much to allocate per request

- "smart" choices are computationally expensive -> takes time
- Best fit
    - Smallest size greater than or equal to requested
    - Advantage - may find perfect fit (minimize internal fragmentation)
    - Disadvantage - requires searching the entire free list; creates very small fragments
- Worst fit

    - Largest size greater than or equal to requested
    - Advantage - Creates very large pieces (minimize external fragmentation)
    - Disadvantage - requires searching the entire free list
- First fit
    - Finds first piece large enough for request
    - Advantage - Short search time; creates randomly sized partitions
    - Disadvantage - Search time approaches O(n); in the long term small fragments
- Next fit
    - Use first fit starting from the ending of previous search (uses a guess ptr)
    - If guess is right, saves a lot of time
    - If guess is wrong, algo still works
    - Advantage - Shorter search time, spreads out searches
    - Most modern systems use this

Coalescing Partitions

- Reassemble fragments that are given back the OS
- Algorithm:
    - Check neighbors when a chunk is freed
    - Recombine if possible (using free list)

A Special Case for Fixed Allocations

-

Distribution of memory requests

- Certain sizes are much more popular (often in buffer sizes)
- Buffer Pools: create independent list of x sized chunks -> no fragmentation, perfect match
    - Eliminates searching, carving, coalescing
    - Once buffer has been used, give back to OS and can easily redistribute
    - Per-process data structure (buffer size is more constant within a process, use profiling from history data)

Dynamically Sizing Buffer Pools

- Creates a load-adaptive system
- sIf low on fixed size buffers
    - Get more memory from the free list
    - Carve it up into more fixed sized buffers
- If our fixed buffer list gets too large
    - Return some buffers to the free list
- If the free list gets dangerously low
    - Ask each major service with a buffer pool to return space
- Requires tuned parameters: Low space (need more) threshold, High space (have too much) threshold, Nominal allocation (what we free down to)

Memory Leaks

- Process done with memory but doesn't call free once done, long running processes can waste lots of memory
- Garbage Collection can be a solutions
    - Automatically deallocate if no more references to object
    - Runs when memory is low

Finding accessible memory

- References are tagged with size information
- Object oriented languages often enable this

General Garbage Collection

- Algorithm
    - Find all the pointers in allocated memory
    - Determine “how much” each points to
    - Determine what is and is not still pointed to
    - Free what isn’t pointed to
- Problems With General Garbage Collection
    - Locations may look like addresses but could be data or bad pointer
    - Addresses don't indicate size

Compaction and Relocation

- Ongoing processes can starve coalescing
- Chunks reallocated before neighbors become free
- Need a way to rearrange active memory into a large chunk
-

The Relocation Problem

- All addresses, pointers, code segments in the program will be wrong
- Too many address references in a process -> can't resolve all after a relocation
- Virtualization solves this and makes processes location independent
    - Does not solve the issue of having to copy data

Virtual Address Spaces

- Process sees a different address space than the physical address
- Address translation unit converts from one to another

Memory Segment Relocation

- Process address space is made up of multiple segments
- Computer has special relocation registers
    - Segment base registers point to start of each segment
    - CPU automatically adds base register to every address
- OS uses these to perform virtual address translation
    - Set base register to start of region where program is loaded

    - If program is moved, reset base registers to new location
    - physical = virtual + baseseg

Relocation and Safety

- Need protection to prevent process from reaching outside its allocated memory
- Segments also need a length (or limit) register
    - Specifies maximum legal offset (from start of segment)
    - Any address greater than this is illegal (in the hole)
    - CPU should report it via a segmentation exception (trap)

How Much of Our Problem Does Relocation Solve?

- Use variable sized partitions -> less internal fragmentation
- Move partitions around -> helps coalescing be more effective, not solving external fragmentation problem
- Still requires contiguous chunks of data for segments

PAGING

Swapping

- Process can want more storage than in RAM
- Can store memory using the disk

Swapping To Disk

- When a process yields, copy its memory to disk
- When it is scheduled, copy it back
- If we have relocation hardware, we can put the memory in different RAM locations
- Each process could see a memory space as big as the total amount of RAM
- Downsides To Simple Swapping - Costs of context switch
    - Copy old process to disk
    - Copy new process to RAMA
    - Still limiting processes to the amount of RAM we actually have

The Paging Approach

- Divide physical/virtual memory into units of a single fixed size called page frame
- For each virtual address space page, store its data in one physical address page frame
- Use some magic per-page translation mechanism to convert virtual to physical pages
-

Paging and Fragmentation

- Segment is implemented as a set of virtual pages
- Internal fragmentation average 1/2 page (last page may be underused)
- No external fragmentation

Paging and MMUs

- On per page basis, we need to change a virtual address to a physical address for each reference
- Needs to be fast -> use hardware Memory Management Unit (MMU)
- The MMU Hardware
    - MMUs used to sit between the CPU and bus (now in CPU)
    - Page tables originally implemented in special fast registers
        - Would require too many fast registers as memory grows

Handling Big Page Tables

- Stored in normal memory
- Use a fast set of MMU registers used as a cache
    - Need to worry about hit ratios, cache invalidation, and other nasty issues

The MMU and Multiple Processes

- Each process needs pages
- Can run into issue of more pages wanted than available

Ongoing MMU Operations

- What if the current process adds or removes pages?
    - Directly update active page table in memory
    - Privileged instruction to flush (stale) cached entries
- What if we switch 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
- How to share pages between multiple processes?
    - Make each page table point to same physical page
    - Can be read-only or read/write sharing

Demand Paging

- Process may not use all pages to run, only the ones referenced
- Move pages onto and off disk "on demand"
- How To Make Demand Paging Work
    - MMU generates a fault/trap when "not present" pages are referenced
        - OS can bring in page and retry the faulted reference
    - Entire process needn’t be in memory to start running
        - Start each process with a subset of its pages
        - Load additional pages as program demands them

Achieving Good Performance for Demand Paging

- Demand paging will perform poorly if most memory references require disk access
- Locality of Reference
    - Next address you ask for is likely to be close to the last address you asked for

    - Code runs consecutive instructions
    - Access same stack frame
    - Exists in all 3 types of memory

Page Faults

- In some cases, the page table has an entry to disk, not RAM
- Generate a page fault so OS fetches data
- Handling a Page Fault
    - Initialize page table entries to “not present”, cause CPU to fault and enter kernel
    - Page fault handler determine which page is required
    - Schedule I/O to fetch it, then block the process
    - Change page table pointer at newly read-in page
    - Back up user-mode PC to retry failed instruction
    - Meanwhile, other processes can run
- Effect of page fault
    - Page faults only slow a process down
    - Desired page is in RAM

Pages and Secondary Storage

- Pages on disk are in "swap space"
- How do we manage 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

Demand Paging Performance

- Overhead (fault handling, paging in and out)
    - Process is blocked while we are reading in pages
    - Delaying execution and consuming cycles
    - Directly proportional to the number of page faults
- Key is having the “right” pages in memory
    - Right pages -> few faults, little paging activity
    - Wrong pages -> many faults, much paging
- We can’t control which pages we read in
- Key to performance is choosing which to kick out

VIRTUAL MEMORY

Virtual Memory

- Generalization of what demand paging allows
- System gives abstraction of large quantity of memory addressible at the speed of RAM
- The Basic Concept
    - Give each process an address space of immense size
    - Allow processes to request segments within that space
    - Use dynamic paging and swapping to support the abstraction
    - The key issue is how to create the abstraction when you don’t have that much real memory

The Key VM Technology: Replacement Algorithms

- The goal is to have each page already in memory when a process accesses it
- We can’t know ahead of time what pages will be accessed -> use locality to decide what to move out of memory
- If we make wise choices, the pages we need in memory will be there
- The Basics of Page Replacement
    - Keep as many pages as possible in memory
    - When there is a page fault or other reason, replace a paage with one on disk
- The Optimal Replacement Algorithm
    - Replace the page that will be next referenced furthest in the future
    - Why is this the right page?
        - It delays the next page fault as long as possible
        - Fewer page faults per unit time = lower overhead
    - Requires being able to predict the future (theoretical)
- Approximating the Optimal
    - Rely on locality of reference
    - Note which pages have recently been used
    - Perhaps with extra bits in the page tables
    - Updated when the page is accessed
    - Use this data to predict future behavior
- Candidate Replacement Algorithms
    - Random, FIFO
        - Very bad
    - Least Frequently Used
        - Sounds better, but it really isn’t
    - Least Recently Used
        - Note time of page access
        - Choose oldest timestamp in memory to kick out
        - Can't use MMU for timestamp calculation so need to use software
- Clock Algorithms
    - A surrogate for LRU
    - Organize all pages in a circular list
        - MMU sets a reference bit for the page on access
        - Scan whenever we need another page
        - For each page, ask MMU if page has been referenced
            - If so, reset the reference bit in the MMU & skip this page
            - If not, consider this page to be the least recently used
        - Next search starts from this position, not head of list

    - Position in scan represents age
    - No extra page faults, usually scan only a few pages
    - Clock runs 98% as good as LRU but with 1% of cost

Page Replacement and Multiprogramming

- We don’t want to clear out all the page frames on each context switch
- How do we deal with sharing page frames?
- Single Global Page Frame Pool
    - Treat the entire set of page frames as a shared resource
    - Replace whichever process’ page is LRU
    - Bad with round robin scheduling
        - Last in queue will have lots of page faults
- Per-Process Page Frame Pools
    - Set aside some number of page frames for each running process
    - Use an LRU approximation separately for each
    - Fixed number of pages per process is bad
        - Different processes exhibit different locality
        - Number of pages needed changes over time
- Working Sets
    - Give each running process an allocation of page frames matched to its needs
        - Set of pages used by a process in a fixed length sampling window in the immediate past
        - Allocate enough page frames to hold each process’ working set
    - Each process runs replacement within its own set
-

Optimal Working Sets

- Optimal is the number of pages needed during next time slice
- If less pages, lots of replacement

Implementing Working Sets

- Manage the working set size
    - Assign page frames to each in-memory process
    - Observe paging behavior (faults per unit time) and adjust number of assigned page frames
- Page stealing algorithms/Working Set-Clock
    - Track last use time for each page, for owning process
    - Find page (approximately) least recently used (by its owner)
    - Processes that need more pages tend to get more
    - Processes that don't use their pages tend to lose them

Thrashing

- Working set size characterizes each process
- What if we don’t have enough memory?
    - Sum of working sets exceeds available memory
    - No one will have enough pages in memory

    - Whenever anything runs, it will grab a page from someone else (infinite loop of page faults)
- When systems thrash, all processes run slow
- Generally continues till system takes action

Preventing Thrashing

- Can't add memory or change working set sizes
- Reduce number of competing processes
    - Swap some of the ready processes out to let other processes run
    - Swapped-out processes won’t run for quite a while (round robin who is swapped)

Unswapping a Process

-What happens when a swapped process comes in from disk? -Pure swapping? -Bring in all pages before process is run, no page faults -Pure demand paging? -Pages are only brought in as needed -Fewer pages per process, more processes in memory -What if we pre-loaded the last working set? -Far fewer pages to be read in than swapping -Probably the same disk reads as pure demand paging -Far fewer initial page faults than pure demand paging

Clean Vs. Dirty Pages

-Consider a page, recently paged in from disk -There are two copies, one on disk, one in memory -If the in-memory copy has not been modified, there is still an identical valid copy on disk -The in-memory copy is said to be “clean” -Clean pages can be replaced without writing them back to disk -If the in-memory copy has been modified, the copy on disk is no longer up-to-date -The in-memory copy is said to be “dirty” -If paged out of memory, must be written to disk

Dirty Pages and Page Replacement

-Clean pages can be replaced at any time -The copy on disk is already up to date -Dirty pages must be written to disk before the frame can be reused -A slow operation we don’t want to wait for -Could only kick out clean pages -But that would limit flexibility -How to avoid being hamstrung by too many dirty page frames in memory?

Pre-Emptive Page Laundering

-Clean pages give memory manager flexibility -Many pages that can, if necessary, be replaced -We can increase flexibility by converting dirty pages to clean ones -Ongoing background write-out of dirty pages -Find and write out all dirty, non-running pages -No point in writing out a page that is actively in use -On assumption we will eventually have to page out -Make them clean again, available for replacement -An outgoing equivalent of pre-loading

Paging and Shared Segments

-Some memory segments will be shared -Shared memory, executables, DLLs -Created/managed as mappable segments -One copy mapped into multiple processes -Demand paging same as with any other pages -Secondary home may be in a file system -Shared pages don't fit working set model -May not be associated with just one process -Global LRU may be more appropriate -Shared pages often need/get special handling