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 PIDs 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
- 2 phases to Limited Direct Execution 1. Boot mode in which the trap table is initialized by kernel and saved by CPU 2. Kernel/User mode: Kernel sets up a few things OS switches between user and kernel mode based on traps/return-to-trap instructions

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

Context switching

- Switching from Process A to B
    1. A is running and has a timer interrupt. It's register's get saved on the kernel stack by the HARDWARE
    2. The OS switches to kernel mode and goes to the the trap handler
    3. Calls the switch() routine, in which A's registers are saved by the SOFTWARE into the memory in the process structure of A
    4. Restores B's registers from its process structure
    5. Switches contexts by changing the stack pointer to B's kernel stack
    6. Moves back to user mode and runs process B

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 Metrics

    - Turnaround time: Time of Completion - Time of Arrival --> maximizes performance
    - Response time: Time of First Run - Time of Arrival --> maximizes fairness

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 tasks 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

Scheduling methods

- Methods of Scheduling

    - First in, First Out (FIFO)

        - As the name states, the processes are completed as they arrive in the scheduler

        - Bad if short processes are backlogged behind longer processeses that come first (Fails workload assumption 1)

        - Shortest Job First (SJF)

            - The scheduler will prioritize finishing up jobs that take the least amount of time
            - Bad if the arrival times are not the same (Fails workload assumption 2)

        - Shortest Time to Completion First (STCF)

            - aka Preemptive Shortest Job First (PSJF)
            - Essentially can preemptively switch to another shorter task in the middle of a longer task
            - **Optimal scheduling algorithm**

        - Round Robin

            - Divides each task into time slices and runs them one after the other (ie: ABCABC)
            - Need to amortize time slice- if it's too small then not worth because switching has its own time

        - ex: if the time slice is set to 10 ms, and the context-switch cost is 1 ms, roughly 10% of time is spent context switching and is thus wasted
        - instead, make the time slice 100 ms so then 1% of the time is spent context switching

Preemptive scheduling

- Modified shortest job first algorithm
- Enables a later arriving job to stop a currently running process
    - Helps for the case where a 100ms job is running but 10ms job has arrived

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
- Key part of the MLFQ is that it can change priority based on observed behavior
    - If it uses CPU for extensive amount of time, priority is reduced
    - If it relinquishes priority to CPU, then priority is high
- Algorithm:
    1. If Priority(A) > Priority(B), A runs (B doesn’t).
    2. If Priority(A) = Priority(B), A & B run in RR.
    3. When a job enters the system, it is placed at the highest priority (the topmost queue).
        - Because you don't know the time of the process, you start with the assumption that it will be short
        - However, you can then adjust this assumption based on what actually happen
    4. Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue). - This rule was modified from a "forgetful time slice" (where once the program gave permission to CPU it reset the slice) idea

because programs could be written to game the system - Game the system: Programs may arbitrarily include I/O in order to hand permission back to the CPU, and thus maintain their high priority - New rule fixes this because even if it gives up control the CPU, if its time is still long its priority is still decreased
5. After some time period S, move all the jobs in the system to the topmost queue.
    - Solves problem of starvation: lower priority programs never being run- if they are that long they will eventually be bumped up
    - Solves problem of change: if there is now a point where there is lots of I/O in program, the boost will help properly treat it

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
- If 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
- Not feasible to run compaction often because inefficient
  - Means external fragmentation will still exist

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 size 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
    - If page is visited, set reference bit to 1
    - 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 by cycling nature
- 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

Concurrency

Threads

- Similar to process, but share same address space
- Used with multiprocessor computers
  - Split up task across multiple CPUs
- Avoids blocking processes due to slow I/O
  - Another part of the code can be run in the meantime
- Context switch
  - Save registers and PC in thread control block (TCB)

Accessing shared data

- Program to add 1 to a common variable gives different and wrong result every time (indeterminate)
  - mov from address to register
  - add 1 to register
  - mov from register to address
- Critical section: piece of code that accesses a *shared* resource
- Race condition: multiple threads enter the critical section around the same time, and attempt to use the resource simultaneously
- If OS scheduler interrupts and switches, those 3 steps may not happen in order
  - Solution: make atomic (as a unit)
    - Use the hardware for synchronization/mutual exclusion primitives

Thread API

- `pthread_create()`
  - Makes new threads with a function pointer
- `pthread_join()`
  - Wait for threads to complete
- `pthread_mutex_lock`
  - Locks a variable before a critical section
  - Doesn't let other threads access that variable when lock is held
- `pthread_mutex_unlock`
  - Allows other processes to access variable

Mutual Exclusion

- Critical sections create issues when multiple threads run at the same time
  - Need to enforce "mutual exlcusion" of critical section
- Mutual exclusion allows us to create atomicity

1. Before or after atomicity
   - A enters before B starts, and B enters after A is done
2. All or none atomicity
   - Update that starts will complete, uncompleted updates have no effect

Methods for protecting critical sections

- Turn off interrupt (disable context switch), but no concurrency
- Hardware atomic CPU instructions
  - Very limited instructions (read/write of contiguous bytes, increment/decrement, etc.)
- Software locking

Locks

- If the thread obtains the lock, then it goes ahead
- If another thread is occupying the lock, thread must wait
- Thread will release the lock at the end of critical section
- Example: Concurrent counters
  - Adds a single lock at the start of routine, release at the end
- Implementation of locks
  - Locking and unlocking is itself a critical section
  - Hardware CPU instruction is atomic
    - TS, CAS
- Spin locks
  - Pros
    - Properly enforces critical sections
    - Simple to program
  - Cons
    - Wasteful (lots of CPU cycles wasted on no-ops)
    - Bugs mean infinite loops
  - Good for certain cases
    - Awaiting program can operate in paralle
    - Quickly finishing critical sections
- Asynchronous Completion Problem
  - How to add locks for high performance (without spinning)
    - Parallel code runs at different speeds
- Yield and spin
  - Check if event occurred (a few times)
    - If not, then yield and block yourself
    - Will be rescheduled soon
  - Avoids indefinite spinning and reduces waiting time
  - Problems
    - More context switches since yielding
    - Still wasted cycles if spinning when scheduled
    - Creates unfairness (rescheduling up to scheduler)

Fairness and mutual exclusion

- Multiple process/thread/machine needing access to a resource
- Locking requires scheduling algorithm to ensure fairness
- Threads don't check for locks automatically
  - Rely on the OS to let you thread know
- Conditional variables
  - Synchonization object associatied with a request
    - Requester blocks and is queued awaiting event on object
    - Posting event unblocks waiter
- Waitlist
  - Shared data structure
- Who to wake up?
  - Wake up a single thread
  - Broadcast and wake up all blocked threads
    - May be wasteful, good if there are lots of resources
- Locking + async events should yield the following benefits...
  - Effectiveness
  - Progress
  - Fairness
  - Performance

Semaphores

- *synchronization platform for multiple processing units*
- Synchronization choices
  - Use locks
    - Spin loops
    - Primitives that block resources
- Semaphores
  - Logically sound way to implement locks
    - Extra functionality for sync
- Computational semaphores (Dijkstra)
  - Use a counter rather than binary flag for locks
  - FIFO wait queue
- Operations
  - Initialized semaphore count to the number of available resources
  - P "wait"
    - Decrement count
    - if count >=0, return
    - if count <0, add to queue
  - V "post/signal"
    - increment counter
- Limitations
  - Counter update errors
    - Data races (don't crash program but give wrong results)
  - Lack practical sync features
    - Easy to deadlock
    - Can't check lock without blocking

        - No support for priority

Mutexes

- Linux/Unix system
- Locks sections of code briefly
- Protects *data*, not code
    - Don't need to protect sections that don't touch data
- Object locking
- File descriptor locking
    - `int flock(fd, operation)`
    - Locks the file trying to be accessed
    - Has shared and exclusive lock
- Ranged file locking
    - `int lockf(fd, cmd, offset, len)`
    - finer grain lock (specific bytes)

Advisory vs enforced

- Enforced
    - In implementation of object
    - May be too conservative
- Advisory (implemented in Linux)
    - Convention
    - Give users flexibility/freedom
    - Example is mutex and flock

Locking problems

- Contention
- Locking performance
    - If long critical section
        - Time to lock << Time of running critical section
    - If short critical section
        - Not always worth it to lock
    - Cost of waiting depends on conflict probability
        - C_expected = (C_block _ P_conflict) + (C_get _ (1-P_conflict))
    - Context switches are in microseconds

Priority and locking

- Locking can prevent high priority processes from executing first
- Mars rover example
    - Shared information bus (shared memory region protected by mutex)
    - Low priority threasd that own the bud are put to sleep, when threads are preempted by higher priority, cannot get the bus.
- Temporarily raise the priority of lower priority process once they get lock so that it cannot be preempted

- Only raise priority when it holds the lock

Reducing contention

- Eliminate/shorten critical section
- Eliminate preemption when in critical section
- Use private resources instead of shared
- Batch operations
  - Use "sloppy counter"
  - Eventually transfer updates to global counter
  - Global counter is not always up to date
- Remove requirement for full exclusivity
  - Allow unprotected reads
  - Lock parts of FDs instead of entire FDs

Deadlock

- P1 and P2 both need resource A and B
  - P1 has A
  - P2 has B
  - Both processes are stalled because they cannot get the other resource
- Resource types
  - Commodity -> can ask for an amount
  - General
- 4 Conditions
  1. Mutual exclusion
     - Only one process can use a resource at a time
     - Solution: don't use shared resources
  2. Incremental allocation
     - Processes/threads have to be able to ask for resources as needed
     - Solution: pre-allocation, requires predicting resources needed
  3. No pre-emption
     - If an entity has the resource, you can't take it away
     - Solution: turn off interrupts
  4. Circular waiting
     - A waits for B, B waits for A
     - Cycle in dependency/wait-for graph
     - Solution: Reservations in advance (facilitated by resource manager) for commodity resources.

Resource Management

- Commodity resource management
  - Advanced reservation mechanisms much easier commodities
  - System must guarantee reservations if granted
  - Must deal with reservation failures
    - Application must have a way of reporting and continue running
- Rejecting resources

    - Not great, better than failure

Breaking circular dependencies

- Total resource ordering
    - All requesters allocate resources in the same order
- *Lock dance*
    - Release R2, allocate R1, reacquire R2

Deadlock detection

- Allow deadlock, but detect when it happens
- Not practical

Watchdog threads

- Demon process

Devices

Devices and Interrupts

- Drivers rely on interrupts
- Devices much slower than CPU

Busses

- CPU and devices connected by the bus
- Control and data information
- Send/recieve interrupts
    - Devices give signal when they are done/ready
    - Controller gives interrupt on bus
    - Bus transfers interrupt to CPU
    - Leads to data movement

CPU interrupts

- Similar to traps, but caused externally from CPU
- Can be disabled by CPU
    - Interrupt can be *pending* until the CPU is ready

Device performance

- System devices limit performance
- If device is idle, throughput drops
- Delays disrupt real time data flows
- Start *n+1* once *n* is done (popline?)

Improving performance

- Parallelization
  - Devices and CPU work separately
- Device needs to use RAM
  - Modern CPU avoids RAM, uses CPU caches instead
- Let device use device bus instead of CPU
- Direct memory access (DMA)
  - Any two devices on the bus can pass data (without using CPU)
  - CPU rarely needs DMA
  - Facilitates parallelism of devices and CPU
- Bigger transfers are better
  - Small blocks mean disk have to spin a lot
  - Per-operation overhead is amortized
    - Instructions to setup

I/O and buffering

- OS consolidates requests
  - Cache recently used disk blocks
  - Accumulate small writes and do at once
- Read-ahead
  - Cache blocks before they are requested
- Deep request queue
  - Minimizes overhead
  - How to implement
    - Many processes making requests
    - Read-ahead
- Double buffered output
  - Application and device I/O go in parallel
    - Application queues up successive writes
    - Devices picks up next buffer as soon as it's ready
  - CPU bound programs
    - Application speeds up since it doesn't wait for I/O
  - I/O bound programs
    - Device is busy, improving throughput

Threading and buffers

- Buffer requires storing a current (head/tail) pointer
- Multiple threads writing to a shared buffer creates contention for pointer
  - Requires locking to enforce; overhead

Scatter/Gather I/O

- Entire transfer in DMA must be contiguous in physical memory
- Gather - writes from the page to device
  - Copy information from pages into contiguous buffer in physical memory level
  - Send buffer out to device
- Scatter - reads into paged memory

    - Data from DMA stream scattered in physical memory
- DMA + Gather/Scatter implemented in the hardware

Memory mapped I/O

- DMA not good for small transfers
- Treat registers in device as part of regular memory space
- Map I/O device into process address space
    - Use as if it were memory
    - Example of bit mapped display adapter, can just change a single pixel as memory

Memory mapped I/O vs DMA

- DMA better for large transfers
    - Better utilization of device and CPU
    - Faster for occaisonal large transfers
- Memory mapped I/O has no per-op overhead
    - Good for frequent small transfers
- **batching is good for throughput, bad for latency**

Generalizing abstractions for device drivers

- Many commonalities
- Device driver interface (DDI)
    - Standard device driver entry points
    - Entry points corresponding to system calls (open, read, write)

File systems

Persistent data storage

- Raw storage blocks
    - Hard drive, SSD
- Database
    - Extra overhead and structure
- **File system**
    - Organized method that is logical to developers
    - Inspired by physical file cabinets
    - Every unit is a file
    - Goals: persistence, easy access, performance (as fast as CPU), reliability (survives crashes), security (using access control lists)

File system and hardware

- HDD much slower than SSD
    - 50-70x slower
    - SSD has no penalty for random access
    - HDD must spin far to get to new location

- Data and metadata
  - Data - actual information in the file
  - Metadata - information about the file; permissions, size, timestamp
  - Need to be stored persistently
- File system needs to be agnostic to hardware
  - RAID, SSD, HDD, etc.

File system API

- File container operations
  - Changing the information about a file, not editing the actual file
  - Ownership, protection, create/destroy, links
- Directory applications
  - Create/update directories
  - Find files by name
  - List files
- File I/O
  - Read and write to file
  - Seek
  - Map into address space (MMIO)

Layered abstractions

- At the top, apps think they are accessing files
- At the bottom, various block devices are reading and writing blocks
- Virtual file system (VFS) layer
  - Interface for different file system implementations
  - File system layer implements interface (with different goals)

File systems and block I/O

- Implements async read/write
- Unified LRU buffer cache to optimize locality
  - Users read/write contiguous blocks or same files

File system control structure

- File consists of multiple data blocks
- Finding information needs to be fast
- Files can be sparesly populated
- On disk and in memory version
- On disk version
  - Contains pointers to blocks
- In-memory version
  - Open files
  - All processes must share this version
  - Contains pointers to memory
  - Maintains dirty bits to update disk copy

File system structure

- Live on block-oriented devices, use blocks to store user data
- Need to have pointers
- Boot block
  - 0th block is use for code allowing to boot the OS
  - File systems start at block 1
- Managing alllocated space
  - Internal/external fragmentation, paging
- Linked extents (DOS method)
  - File control block has a pointer to the next chunk
  - Use a pointer table to speed up searches
  - File allocation table (FAT) holds next cluster number for each cluster
  - Capacity of file system is determined by "width" of FAT table
    - Can accomodate larger chunks
- System V (Unix method)
  - Inode - in memory file descriptors
  - Dinode - on disk file desciptors
  - Open file references stored in process descriptor
  - Open file instance descriptors (unique to each process)
  - Processes can share an open fd
  - Layout
    - Super block (block 1) saves block size and num inodes (filesystem metadata)
    - Inodes have metadata for individual files
    - Data blocks start after inodes
  - Scaling while keeping performance
    - First 10 blocks point directly -> 40K bytes
    - Block 11 points to indirect (stored in the data blocks) -> 4M bytes
      - Must read indirect blocks
    - Block 12 points to double indirect -> 4G bytes
    - Block 13 points to triple indirect -> 4T bytes
    - **Any block can be found in 3 or less reads**

Free space and allocation

- File system are not static; users create, update, destroy files
- Creating files
  - Unix - search super block inode list and find first free
  - DOS - search parent directory for unused directory entry
  - Initialize properties and name file
- Extending files
  - Unix - Find free chunk from free list (and remove), link new space with the address in the file
  - DOS - update FAT table
- Deleting files
  - Unix - return to free list, zero inode
  - DOS - use garbage collector (will eventually be freed), zero first byte of name in parent directory

Flash storage

-

Caching

- Read cache
  - Disk I/O is slow
  - Repeated reads to same parts of disk
  - Read-ahead to predict sequential read
- Write cache
  - Aggregate small writes to cache, flush out to disk later
  - Save on moot writes (rewriting same data, temp files)
- Special caching for directories/inodes

Naming

- File system handles name-to-file mapping
- Within directory, must be unique names
- Hierarchical namespace; leaf nodes are files, other nodes are directories
- Directories
  - Read only for user, updated by FS
  - DOS - have true file names
  - Unix - names separated by slashes,

Links

- Links provide same access to the file
  - Need read access to create a link
  - All links are equal
- Deallocate file once there are no hard links to file
  - Maintain reference count in inode
- Symbolic links are a new file with pointer to the original file
  - Does not add an edge to the graph

Reliability

- Data loss
- Corruption
  - Invalid references
  - Corrupted free list, directories, inodes
- Queued writes not completed
- Power failures
  - Solution: non-volatile RAM, uninterrupted power supply
- Ordered writes
  - Write out data before writing to it
  - Write out deallocations before allocations
- Audit and repair

- Redundancy allows for audit for correctness