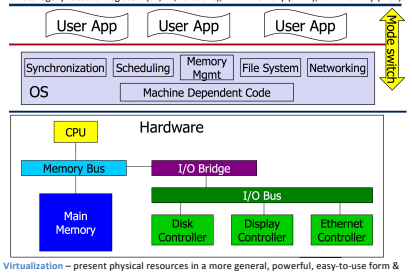
INTRODUCTION

- OS goals = **1. Convenience** for user; **2. Efficient** operation of computer system
  - May contradict, tradeoffs made (windows = convenience, supercomputer = efficiency)
- OS Roles: **virtual machine** – simplified interface to HW; **resource allocator** – shares resources; **control program** – control execution of user program, prevent errors
- Overview of Computer System
  - Red line separates user software and OS software (kernel)
  - Storage: processor registers (L1, L2, L3 cache), main memory (RAM), aux memory (Disk)



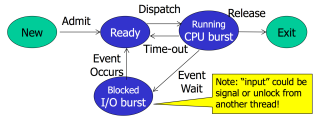
- **Virtualization** – present physical resources in a more general, powerful, easy-to-use form & present illusion of multiple/unlimited resources (goal #1 + 2)
  - **Limited Direct Execution** – set up CPU so that next instruction is fetched from code of a process to be executed (no overhead b/c CPU in fetch-decode-execute loop)
    - **Direct** = process given direct CPU, (some) memory access, other resource access
    - **Limited** = need to restrict operations a process can perform → trap to kernel after certain time intervals, or b/c illegal operations, system calls, etc.
- **Hardware Support for OS**
  - CPU dual-mode operation: **user mode** & **system mode** (supervisor/privileged/monitor)
    - **Mode bit** in HW designate some instructions as **privileged instructions**
      - Privileged instruction in user mode → protection fault, trap to OS (handle safely)
      - Ex. instructions: *setting mode bit; disabling interrupts; halting the CPU*
  - **Interrupts** – HW signal, causes CPU to jump to pre-defined instruction (interrupt handler)
    1. A boot OS fills in Interrupt Descriptor Table; physical addr stored in IDT Register
    2. CPU execution (infinite) loop: fetch instr at PC → decode instr → execute instr
       - checks for interrupts at the same time
    3. Interrupt occurs, signal from the hardware
    4. CPU switch to sys mode, disables interrupts
    5. Interrupted PC value stored by OldPC register
    6. IDTR + interrupt number used to set PC to start of interrupt handler in the IDT
    7. Execution continues
    - **HW interrupts** = asynchronous events
      - ex. Device control signal some event (I/O complete), periodic timer interrupt
    - **SW interrupts** = synchronous; signal errors/request OS service from user program
      - **Exceptions** = raised when abnormal conditions occurred during execution
      - **Traps** = an exception by user process –ex. syscalls
    - Interrupts support OS goal #2
      1. Illegal instructions causes software generated interrupts (**exceptions**)
      2. Periodic HW timer interrupts ensure OS gets control at regular intervals
  - Bootstrapping
    - HW stores BIOS (basic input-output system) in non-volatile memory (NVRM)
    - On computer startup BIOS executes, loads boot sector of disk into memory
      - Boot loader loads OS code into RAM
  - OS startup in system mode
    1. Initialized internal data structures, create first process (`init`)
    2. Switch to user mode, run first process, wait for something to happen
- **System Call** – a function call that invokes the OS; for resource/service managed by OS
  - 1. User program calls C Library function w/ args; C library pass syscall ID + args to OS
  - 2. Interrupt (trap) signals CPU → switch to kernel mode in OS code, invoke handler
  - 3. Syscall returns, switch to user mode; jumps to next instruction in the application
  - Extra level of indirection → enforce separation of user-space & kernel-space

THREADS & PROCESSES

- **Processes** – A program in execution, active entity; OS abstraction for execution, aka job, task
  - OS manages processes by keeping track of their state **(Process State)**
  - Only 1 in the **Running** state; others in **New**, **Ready**, **Blocked**, or **Exit** state
    - **Time-out** = timer interrupt that shifts control back to the OS
  - OS data about process in **Process Control Block** (PCD)
    - **Process state**: ready, running, block, etc.
    - **CPU registers**: saved at an interrupt
    - **Memory Management info**: page tables
    - **I/O status info**: list of open files
    - **Program counter**: addr of the next instr
    - **CPU scheduling info**: process priority
    - **Accounting info**: resource use de;
  - **Context Switch** – switch CPU to another process, pure overhead; by
    1. Saving the state of old process;
    2. Load saved state for the new process
  - Lifecycle of Process:
    - Program to Process (creation)
      1. Source file → object file;
      2. Object file → executable;
      3. Create new process (incl. AS, PCB);
      4. Load executable
      5. Initialize process' start state (ready)
      6. Dispatch process (running)
    - Process Destruction
      - Process voluntarily releases all resources on `exit()` → addr space freed, files closed, **PID** & **exit state** retained (**zombie** till parent cleans up → context switch)
- **Threads** – a single control flow through a program
  - **Control Flow** = an order in which individual instructions are executed and evaluated → weaves its way through the program
    - Program w/ multiple control flows = multithreaded –ex. OS, interacts w/ programs
  - Each thread has its own stack, but OS doesn't protect threads stacks in same process from each other → Fix: add an inaccessible guard region (page of memory) btwn stacks
    - Global variables & heap allocated variables of process still shared btwn threads

  - Multithread better vs. Fork = inefficient, need new PCB, page tables, etc. (space & time)
    - Inter-process communication need extra work b/c diff processes = isolated
  - **Kernel-level Threads** (aka. Lightweight Processes)
    - Originally a process encapsulate **ownership of resources** + **execution state**
      - Modern OS **thread abstraction** now encapsulates execution state to make concurrency cheaper → b/c less state to allocate and init.
    - OS manages kernel thread: schedules threads, thread operations in kernel
      - Limitation: too much overhead (thread operations still req. syscalls)
  - **User-Level Threads**
    - User-level threads managed by run-time system (user library)
      - Small & fast: represented by PC, registers, stack, small thread control block (TCB)
      - Creating threads, switching btwn threads, synchronizing threads done via procedure calls → no kernel involvement
    - Limitation: User-level threads are invisible to the OS → OS can make poor decision
      - Schedule process w/ only a few threads; de-schedule thread holding a lock
  - **Hybrid Kernel & User Threads** – use both kernel & user-level threads
    - Associate user thread w/ kernel thread; multiplex user thread on top kernel threads

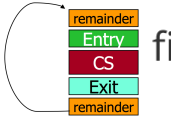
- **Thread/Process Life Cycle**, dependent on scheduler/physical CPU
  - There are queues for each state (ready, block, etc.) b/c easier to search for a process
  - Process changes states = PCB unlinked from old queue, linked to the new queue
  - **CPU bursts** – points in threads execution where it needs the CPU to execute instructions
    - Last CPU burst ends w/ termination call `thread_exit()`
  - **I/O bursts** – points in the threads execution where it is blocked, waiting on I/O requests to be complete; (incl. threads waiting on acquiring a lock)
    - CPU not needed during I/O → can schedule other processes
  - **Ready State** (part of CPU bursts) - time spent in here depends on schedule & CPU load



- **Processor Scheduling** – Allocation of threads to processors over time; goals:
  - **Fairness** – each thread receives fair share of CPU; equal or proportional shares?
  - Avoid **starvation**
  - **Policy Enforcement** – Meet usage policies; ex. avoid monopolization, or have biases
  - **Balance** – All parts of the system should be busy
  - **Throughout** (batch systems) – maximize jobs completed per hour
  - **Turnaround Time** (batch systems) – minimize time btwn submission and completion
  - **CPU Utilization** (batch systems) – keep the CPU busy all the time
  - **Response Time** (PCs) – min time btwn receiving requests & starting to produce output
  - **Proportionality** (PCs) – "simple" tasks complete quickly
  - Meet **Deadlines** (Real-time systems) – work has to be completed by deadline
    - Soft vs. hard real-time systems based upon the consequence of missing a deadline
    - Ex. soft = video decoding; hard = nuclear power generator cooler
  - **Predictability** (Real-time systems) – worst case running time for an operation is known
  - On dispatch:
    1. Select next thread from ready queue
    2. Save current thread running state
    3. Restore state of next thread
       - a. Restore registers
       - b. restore OS control structures
       - c. Switch to user mode (if needed)
       - d. set PC to next instr in this thread
  - When to schedule:
    1. At fixed intervals on regular timer interrupts
       - Problem: If this is the only situation where the CPU schedules then the CPU is idle btwn when last thread ends & next interval occurs
    2. When running thread blocks/exits (`SIGSTOP`) and yields the CPU
    3. When a thread enters **Ready** state, reasons:
       - I/O interrupts might make this thread more preferable;
       - Signals (`SIGCONT`); on thread creation
- **Scheduling Algorithms**, 2 types
  - **Non-preemptive** scheduling – Once the CPU been allocated to a thread, the threads keeps the CPU until it terminates or blocks
  - **Preemptive scheduling** – has involuntary context switch
  - Algorithm 1: **First Come First Serve** (FCFS) is Non-Preemptive
    - Choose the thread at the head of the ready queue (in **First In First Out** (FIFO) order)
    - **Convoy effect**: all other threads wait for a big thread to finish → avg wait time long
  - Algorithm 2: **Shortest-Job-First** (SJF) / **Shortest Process Next** (SPN) is Non-Preemptive
    - Choose the thread w/ the shortest expected processing time
      - based on: *programmer estimate; history statistics; shortest-next-CPU-burst*
    - Problem: **starvation** possible if short jobs keep arriving w/ a long job in queue
    - Pre-emptive variant: **Shortest Remaining Time** scheduling
  - Algorithm 3: **Round Robin** (RR) is Preemptive
    - Read queue is circular → each thread allowed to run for time **quantum** (time slice) before being preempted and put back on the ready queue
      - quantum → infinity, RR → FCFS; quantum → 0, RR → processor sharing
      - Want quantum large w.r.t. the context switch time
  - Algorithm 4: **Priority Scheduling** can be Preemptive & Non-Preemptive
    - A priority is associated w/ each thread, highest priority selected from Ready queue
    - Problem: **Priority inversion** – a low priority task may prevent a high priority task from making progress by holding on a resource
      - Ex. Mars Rover low priority "data gathering" task locks "information bus" and is preempted by higher priority "bus management" task, which blocks on the lock; whenever medium priority "communications" tasks is runnable, its always scheduled before the "data gathering" task so the "bus management" task stays blocked b/c the lock never gets released by the "data gathering" task
    - Problem: **Starvation** possible
  - Algorithm 5: **Multi-level-Queue** (MLQ)
    - Have multiple ready queue, each runnable thread is on only one queue
      - Threads are permanently assigned to a queue, based on: *jobs class, priority*, etc.
    - Each queue has own scheduling algorithm → usually RR
    - Another level of scheduling decides which queue to choose next → usually priority
  - Algorithm 6: **Feedback Scheduling**
    - Adjust criteria for choosing a particular thread based on past history
      - Can boost priority of threads that have waited a long time (aging)
      - Can prefer threads that do not use full quantum
      - Can boost priority following a user-input event
      - Can adjust expected next-CPU-burst
    - Combined with MLQ to move threads btwn queues → **Multi-Level Feedback Queue**
  - Algorithm 7: **Fair Share Scheduling**
    - Group threads by user/department → each group receives proportional share of CPU
    - Priority of a thread depends on its own priority + past history of entire group
- Ex. Schedulers
  - **UNIX** uses MLFQ with RR within each priority queue
    - Priority based on process type + execution history
  - **Linux 2.4** has 2 algorithms = Time Sharing + Real-time tasks
    - Team-sharing = preemptive, prioritized, credit-based algorithm
      - Scheduler choose task w/ most credit
      - At every timer interrupt, current task loses one credit; credit = 0 → suspended
      - **Re-credit** if no runnable task have any credit → adding credits to every task, favors non-suspended tasks
    - Problem: re-credit step proportional to # of process = O(n) time
  - **Linux 2.6** MLFQ w/ 140 queues = 140 priority levels
    - Higher priority = longer time slice
    - Thread run until it uses up its time slice → after running, priority + time slice are recalculated & thread is moved to the expired queues
      - When all active queues are empty, swap w/ expired queue → O(1) time
    - Problem: `nice` = length of time slice (absolute niceness)
      - Ex. niceness 1→2 = time-slice 100→95ms; niceness 18→19 = 10→5ms
  - **Linux Completely Fair Scheduling** (CFS) Scheduler
    - Each thread accumulates "virtual runtime" weighted by niceness
    - Scheduler picks next thread as one with the smallest v-runtime
    - Threads organized in red-black trees → O(logN) time
  - **Windows CPU Scheduling** is 32-level MLFQ priority scheme, Preemptive
    - **Real-time** class = 16 – 30 priorities, RR queue, fixed thread priorities
    - **Variable** class = 0 – 15 priorities, FIFO queue, variable thread priorities
      - used up quantum → priority lowered; waiting on I/O event → priority increased
    - Dispatcher traverses the queues from highest to lowest until it finds a ready thread

CONCURRENCY

- **Synchronization** – mechanism that restrict the possible interleaving of executing threads/processes → manages shared resources ; purpose
  - 1. Enforce single use of a shared resource **(critical section problem)**
  - 2. Control order of thread execution –ex. a parent can wait for child to finish
  - Note: **Local Variables** are not shared (in a thread's own stack); **Global Variables** are shared (stored in static data segments); **Dynamic/Heap** objects shared (`malloc`)
- **Critical Section problem**: concurrent threads manipulate a **shared resource**
  - Given: A set of n threads: $T_0, T_1, ..., T_n$ • A set of resources shared between threads
    - A segment of code which accesses the shared resources → **critical section** (CS)
  - **Critical Section requirements**
    1. **Mutual Exclusion** – If one thread is in the CS, then other is not
    2. **Progress** – If no thread is in the CS, and some threads want to enter CS, only threads not in the "remainder" section can influence the choice of which thread enters next, and choice cannot be postponed indefinitely
    3. **Bounded Waiting** (no starvation) – If some thread T is waiting on the CS, then there is a limit # of times other threads can enter CS before this thread
    4. **Performance** – the overhead of entering and exiting the CS is small with respect to the work being done within it



  - w/o synchronization outcome depends on the order in which the accesses btwn all threads take place inbtwn context switches → a **race condition**
- Ex. On Uniprocessor thread T1 and T2 share a variable `X`
  - T1 increments `X` →i.e. `X=X+1` T2 decrements `X` →i.e. `X=X-1`
- Design a protocol that threads can use to cooperate w/ synchronization, that:
  - 1. Each thread must request permission to enter its CS, in its **entry** section
  - 2. CS may be followed by an **exit** section
  - 3. Remaining code is the **remainder** section
  - Assume no special hardware instructions, no restrictions on the # of processors
  - Assume that basic machine language instructions (`LOAD`, `STORE`, etc.) are **atomic** if two such instructions are executed concurrently, the result is equivalent to their sequential execution in some unknown order
- **Lock algorithms**
  - **Peterson's Algorithm** (can be extended to N threads) uses shared `turn` and `flag` vars
    - 1. Set own flag (indicate interest) and set `turn` to self
    - 2. Spin wait while `turn == self` AND other has flag set
      - If one of these conditions break → the other thread has exited CS
      - If both threads enter CS same time, `turn` will be set one or other not both

```
1.  my_work(id_t id) {          // id_t can be 0 or 1
2.      ...                     // remainder section
3.      flag[id] = true;        // entry section
4.      turn = id;
5.      while (flag[i-id] && turn == id) {
6.          // busy waiting
7.      }
8.      ...                     // critical section
9.      flag[id]= false;        // exit section
10.     ...                     // remainder section
11. }
```

  - **Lamport's Bakery Algorithm** - Upon entering each thread gets a #; lower # served next
    - In case of a tie, thread with the lowest id (unique & totally ordered) is served first
- Synchronization Hardware
  - On uniprocessor → can disable interrupts before entering CS (prevents context switches)
  - Disabling interrupts is insufficient on a multiprocessor → need special atomic instructions
- **Spinlock** – uses Atomic instructions: `test-and-set`
  - 1. Record the old value of variable
  - 2. Set the variable to some non-zero value
  - 3. Return old value (hardware executes this atomically)

```
1.  boolean test_and_set_v1(boolean *lock) {
2.      boolean old = *lock;
3.      *lock = True;
4.      return old;
5.  }
```

  - Notice: Lock is always `True` on exit
    - `True` (locked) already → nothing changed; `False` (available), the caller now holds it
    - Return value is `True` if it was already locked; `False` if it was previously available
  - Lock Implementation using `test-and-set`
    - `acquire()` blocks until the shared resource is free; `release()` releases the lock

```
1.  boolean lock;
2.  void acquire(boolean *lock)    {
3.      while(test_and_set(lock));}
4.  void release(boolean *lock) {
5.      *lock = false;          }
```

  - Uses **busy waiting** → thread continually executes while-loop in `acquire`, consumes CPU
  - **Starvation** possible → when thread leaves CS, frees locks, scheduler might schedule a thread that's not waiting on the lock
  - **Deadlock** possible → through **priority inversion**
- **Sleep Locks** – put thread into "blocked" state while waiting to acquire lock, don't let scheduler run a thread waiting on a lock until the lock becomes available
  - **Wait Channel** – queue for waiting threads; need spinlock to protect thread list

```
1.  struct wchan {
2.      const char *wc_name;            // channel name
3.      struct threadlist wc_threads;   // queue of waiting
4.      struct spinlock wc_lock;        // threadlist lock
5.  }
6.  void lock_acquire(struct lock *lock) {
7.      spinlock_acquire(&lock->lk_lock);
8.      while (lock->lk_holder != NULL) {
9.          wchan_lock(lock->lk_wchan);
10.         spinlock_release(&lock->lk_lock);
11.         wchan_sleep(lock->lk_wchan);
12.         spinlock_acquire(&lock->lk_lock);
13.     }
14.     lock->lk_holder = curthread;
15.     spinlock_release(&lock->lk_lock);
16. }
```

  - Initially acquire the spinlock for the sleep lock `lk_lock`
    - b/c lock is a shared resource → checking the lock state = CS
  - Wait for `lk_holder==NULL`; `lk_holder!=NULL` → another thread holds the lock
    - `wchan_lock` locks the wait channel, adds the current thread to queue
    - Release the spinlock `lk_lock` on the sleeplock b/c can't go to sleep while holding the spinlock → else no other threads can acquire the spinlock `lk_lock`
    - Don't release the spinlock `lk_lock` before locking the wait channel lock `lk_wchan` b/c when the spinlock is release then an interrupt can occur and the `lk_holder` might become `NULL` however the current thread still proceeds to go to sleep waiting on a wakeup call (has already occurred) ← blocks indefinitely
    - `wchan_sleep` → puts the thread to sleep (block it) and releases the hold on the `wchan_lock` the current thread has
    - return from `wchan_sleep` → some other thread has done a wakeup, reacquire the spinlock `lk_lock` and check the lock holder state
  - Once exited while loop its guaranteed `lk_holder==NULL` and that cannot be changed by another thread b/c the current thread holds the spinlock on the sleeplock
    - Record the current thread as the lock holder
- Note: threads still do not acquire the lock in the order its requested
- **Semaphores** – an abstract types that provide synchronization
  - **wait channel** = a queue of waiting threads
  - An integer variable, accessed only through 2 atomic operations:
    - **wait** (**P** or **decrement**) – if current semaphore value = 0 → block until semaphore is free then decrement; if value > 0 → it doesn't wait, just decrements
      - Internal spinlock `sem_lock`; once out of while-loop `sem_count>0`
    - **signal** (**V** or **increment**) – Increment variable, unblock a waiting thread if any
      - `wchan_wakeone` to release one waiting thread on the queue

```
1.  void P(struct semaphore *sem) {
2.      spinlock_acquire(&sem->sem_lock);
3.      while (sem->sem_count==0) {
4.          wchan_lock(sem->sem_wchan);
5.          spinlock_release(&sem->sem_lock);
6.          wchan_sleep(sem->sem_wchan);
7.          spinlock_acquire(&sem->sem_lock);
8.      }
9.      sem->sem_count--;
10.     spinlock_release(&sem->sem_lock);
11. }
12. void V(struct semaphore *sem) {
13.     spinlock_acquire(&sem->sem_lock);
14.     sem->sem_count++;
15.     wchan_wakeone(sem->sem_wchan);
16.     spinlock_release(&sem->sem_lock);
17. }
```

  - **Mutex** (Binary) **Semaphores** – represents access to a resource (lock like); `count = 1`
  - **Count Semaphores** – represents a resource w/ many units available, or allow unsynchronized concurrent access –ex. reading; `count = N`
    - Ex. **Reader/Writer** problem: shared object btwn threads (light switch design pattern)
      - Allow multiple concurrent readers; only 1 writer w/ no reader concurrent reader

```
1.  int readcount = 0;    // # threads reading object atm
2.  Semaphore mutex = 1;  // control/protect readcount
3.  Semaphore w_or_r = 1; // is currently read or write
4.  Writer {
5.      P(w_or_r);            // lock out others
6.      ...                   // write CS
7.      V(w_or_r);
8.  }
9.  Reader {
10.     P(mutex);
11.     readcount += 1;
12.     if (readcount == 1) {   // the first reader
13.         P(w_or_r);          // signal currently reading
14.     }
15.     V(mutex);
16.     ...                     // read CS
17.     P(mutex);
18.     readcount -= 1;
19.     if (readcount == 0) {   // the last reader
20.         V(w_or_r);          // signal obj now free
21.     }
22.     V(mutex);
23. }
```

      - **Starvation** possible → once a reader arrives, can keep accepting new readers but block any writers till all readers are done blocks
    - Ex. Complex wait condition: `if x == 0 && (y > 0 || z > 0); → wait`
      - Only care if `count>0` or `count==0`

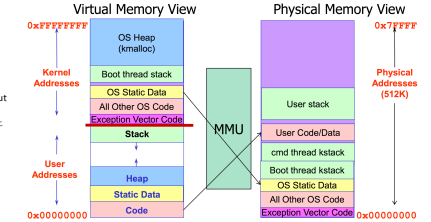
```
1.  int x, y, z;          // some initial values
2.  Semaphore mutex = 1; // mutual excl. for shared vars
3.  Semaphore no_go = 0; // semaphore to wait if cond.
4.  compute_a_thing {                   // is false
5.      P(mutex);                   // lock out others
6.      x, y, z = f(x, y, z);   // compute new state
7.      if (x != 0 || (y <= 0 && z <= 0)) { // not cond.
8.          V(no_go);
9.      }
10.     V(mutex);   // up for grabs
11. }
12. use_a_thing {
13.     P(mutex);     // lock out others
14.     while (x == 0 && (y > 0 || z > 0)) {
15.         V(mutex);  // allow others mutex, no deadlock
16.         P(no_go);
17.         P(mutex);
18.     } // now either x is non-zero or y & z are non-pos
19.     process(x, y, z);
20.     V(mutex);
21. }
```

- **Condition Variables** – an abstract data type that encapsulates pattern of: (release mutex → sleep → re-acquire mutex); internally just queue of waiting threads & no counts
  - The operations are atomic
    - **cv_wait**(`struct cv *cv, struct lock *lk`)
      - Releases lock, waits, re-acquires lock before return
    - **cv_signal**(`struct cv *cv, struct lock *lk`)– wakes one thread
      - Doesn't use the lock, just checks it's held & holds the lock before sending the signal
    - **cv_broadcast** (`struct cv *cv, struct lock *lk`)– wakes all enqueue
    - If no thread is waiting, `cv_signal`/`cv_broadcast` has no effect, unlike `V()`
  - Always used together with locks
- **Monitors** an abstract data type w/ restriction that 1 process at a time be active w/l monitor
  - Local data accessible via monitor procures (monitor enforces mutual exclusion)
  - Condition variables can be used to allow another process to use monitor while 1 waits
  - Operations: `wait` (suspend invoking process), `signal` (resumes 1 suspended process)
  - Semantics: **Local data** = what monitor protects; **Entry queue** = queue of waiting threads
    - Each condition variable has a queue of waiting threads
  - **Hoare monitors** – signal immediately switches from caller to a waiting threads
    - Upon exiting wait queue, thread knows the condition it was blocked on must be true when it resumes b/c singular send the signal when it was true and nothing ahs been allowed to occur in-between to affect the CV
    - Semantics: `if (empty) ← condition guaranteed to hold after signal wait(condition);`
  - Mesa Monitors – signal places waiting thread on ready queue but the signaler thread continues inside the monitor
    - Upon exiting wait queue, the condition may not necessarily be true → must check condition again for previously waiting thread
    - Semantics: `while (empty) ← condition not guaranteed to hold after signal wait(condition);`
  - Ex. Bounded buffer implemented in C
    - Want monitor to control access to buffer of limited size = N
      - Producers add to buffer if not full; consumers remove from buff if not empty
    - Need two functions: `add_to_buffer()` `remove_from_buffer()`
    - Need one lock

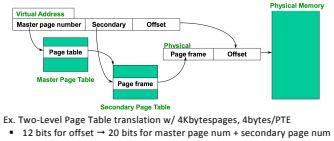
```
1.  #define N 100
2.  typedef struct buf_s {
3.      int data[N];        // buffer storage
4.      int inpos;          // producer insert here
5.      int outpos;         // consumer removes from here
6.      int n_elements;     // #items in buffer
7.      struct lock *buflock;   // access to monitor
8.      struct cv *notFull;     // for producers to wait
9.      struct cv *notEmpty;    // for consumers to wait
10. } buf_t buffer;
11.
12. void add_to_buff(buf_t *b, int value) {
13.     lock_acquire(b->buflock);
14.     while (b->n_elements == N) {   // buf b is full
15.         cv_wait(b->notFull, b->buflock);
16.     }
17.     b->data[b->inpos] = value;
18.     b->inpos = (b->inpos + 1)%N;
19.     b->n_elements++;
20.     cv_signal(b->noEmpty, b->buflock); // just added
21.         // something into buffer, cannot be empty
22.     lock_release(b->buflock);
23. }
24. int remove_from_buff(buf_t *b) {
25.     lock_acquire(b->buflock);
26.     while (b->n_elements == 0) {   // buf b is full
27.         cv_wait(b->notEmpty, b->buflock);
28.     }
29.     int val = b->data[b->outpos];
30.     b->outpos = (b->outpos + 1)%N;
31.     b->n_elements--;
32.     cv_signal(b->notFull, b->buflock);  // just rm
33.         // something from buffer, cannot be full
34.     lock_release(b->buflock);
35.     return val;
36. }
```

VIRTUAL MEMORY

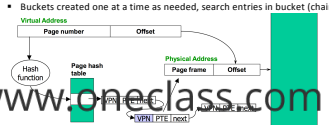
- **Virtual Memory**, 5 basics requirements
  - 1. **Relocation** – some type of address translation; programmers don't know about what physical memory they're available when they're programs run
  - 2. **Protection** – A process's memory should be protected from unwanted access by other processes, both intentional and accidental; requires hardware support
  - 3. **Sharing** – Processes may need to be able to access the same memory
  - 4. **Logical Organization** – Machine accesses/addresses memory as a one-dimensional array of bytes needs to map to code in modules
  - 5. **Physical Organization** – Memory + Disk = two-level hierarchy; flow of information btwn two levels managed → CPU only access data in registers/memory, not disk
  - **Address Binding** – translating symbolic addresses (variable names, pointer to memory location) to physical memory in a program; at...
    - **Compile Time** – must know what memory process will use during compilation; binary contains real addresses; no relation possible
    - **Load Time (static relocation)** – Compiler translates (binds) symbolic addresses to **logical** (relocatable) **addresses** within compilation unit (source file); Loader translates logical absolute addresses to **physical addresses** when program loaded into memory
      - Programs can be loaded to different addresses when they start
      - Cannot be relocated later once programs start running
    - **Execution (dynamic relocation)** – executable object file contains logical addresses for entire program, translated to real physical address during execution; req. special HW
- Logical (virtual) & Physical Addresses
  - **Virtual addresses** generated by CPU in process' virtual/logical address space in execution
  - Memory Management Unit (MMU) translate virtual → physical (real) addresses



  - **Fixed Partitioning** of Physical Memory – divide memory into regions w/ fixed boundaries
    - Single processes can be loaded into each free partition
      - **Internal Fragmentation** – Memory is wasted if process is smaller than partition
      - **Overlays** – Programmer must deal with programs that are larger than partition
    - Placement algorithms: # partitions determine # of active processes
      - **Equal-Sized Partition** – process can be loaded into any available partition
      - **Unequal-Sized Partitions w/ Smallest-Partition Policy** – assign process to smallest partition in which it will fit, queue for each partition size
  - **Dynamic Partitioning** of Physical Memory – Partitions vary in length & number over time
    - When process enters into memory, a partition of exactly its size is created to hold it
      - **External Fragmentation** – over time "holes" created, too small for any processes
      - **Compaction** – if processes relocatable, OS move it around to create large chucks
    - Placement algorithms: want to reduce need for compaction (time consuming)
      - **First-Fit** – choose 1st large enough block from beginning or where prev search ended
        - Simplest/fastest & most efficient algorithm;
        - Problem: leave small fragments near start of memory
      - **Best-Fit** – choose the block that is closest in size to the request; worst than First
        - Problem: left-over fragments tend to be small & unusable
      - **Worst-Fit** – choose the largest block; worst than First & Best
      - **Quick-Fit** – keep multiple free lists for common block sizes → fast allocation
  - Relocation (dynamic) requires execution-time binding of addresses
    - Processes use relative addresses→ HW translate to physical addresses in execution
  - Basic Idea: add relative address to process starting at "base" address → physical address
    - 2 registers "base" & "limit", when process is assigned to CPU load "base" register with starting address of process, load "limit" register with last address of process
    - On (`load`, `store`) instr add "base" to address, compare w/ "limit" → trap if invalid.
  - **Paging** – logically partition physical memory into equal fixed size chucks (**pages frames / frames**) & divide processes' memory into chunks of the same size (**virtual pages / pages**)
    - Any page can be assigned to any free frame → eliminate external fragmentation
      - Internal fragmentation limited to at most size of one page per process
    - OS maintains a **page table** for each process (SW data structure, stored in OS memory)
      - Page table records which physical frame holds each page ← PTBR to quickly locate
        - CPU loads **Page Table Base Register (PTBR)** for new process on context switch
      - **Virtual address**es: `page_num + page_off` in hex
        - `page_num=vaddr/page_size  page_off=vaddr%page_size`
      - **Logical address** = virtual address in bits; **Physical address** = `frame_num + off`
      - # bits in address for offset = `log2(page_size)`; rest = **Virtual Page Num** (VPN)
    - **Page Table Entries** (PTE) control mappings, 4 bytes = 32 bits
      - **Modify Bit** (M) = if page been written; **Reference Bit** (R) = if page been accessed;
      - **Valid Bit** (V) = whether PTE can be used; **Protection Bits** (Prot) = what ops (R/W/X)
      - **Page Frame Number** (PFN) = determines physical pages
        `PTE = [M-bit | R-bit | V-bit | 3 Prot-bits | 26 PFN bits]`
    - **Page Table** (simple) – linear array of page table entries, 1 per virtual page VPN = index
      - Problem: **space overhead** b/c need 1 PTE per Page
        - Ex. Min 2^20 PTEs; 4bytes/PTE = 4MBytes/page table; 25 processes = 100MBytes
      - Problem: **time overhead** b/c at least 2 memory reads (~100 CPU cycles each)
    - Ex. Simple Page Table translation
      - 32-bit virtual addrs, 4Kbyte (4096 bytes) pages ← set by MMU hardware
        - offset = 12-bits (2^{12}=4096); VPN = 20-bits → 2^{20}=~100k pages
      - 1. Program generates virtual address `0x00007469 = 0x7468`
      - 2. VPN = `0x7`, offset = `0x468`
      - 3. Page table entry `0x7 - 0x42` → page frame number = `0x42`
      - 4. Physical address = `(0x42 << 12) + 0x468 = 0x42468`
    - **Two-Level (Hierarchical) Page Tables** → reduce space overhead;
      - Virtual addresses: Master Page # + Secondary Page # + Offset
        - Mapping: master → secondary; secondary → physical
      - On startup **Master Page Table** created, **Secondary Page Tables** created one frame at a time as needed → both pages tables are a frame of memory = 4Kbytes



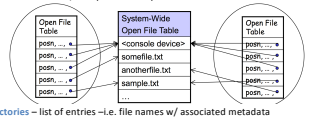
    - Ex. Two-Level Page Table translation w/ 4Kbytespages, 4bytes/PTE
      - 12 bits for offset → 20 bits for master page num + secondary page num
      - page table = 1 frame → 4Kbytes/4bytes = 1024 entries in same page table
        - Master/Secondary 1024 entries each = 10 bits, 12 bits offset = 32 bits addr
      - Need 3 memory reads to get to data (1 master, 1 secondary, 1 for physical)
    - **Hashed Page Tables** – hash function maps VPN to bucket in fixed-sized hash table
      - Buckets created one at a time as needed, search entries in bucket (chain) by VPN



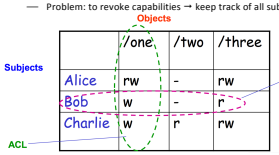
- **Translation Lookaside Buffer** (TLB) caches recent VPN → PTE, lookup by VPN (16 – 64 entries)
  - Part of MMU; >1 port rate; loop up done in parallel = 1 CPU cycle
  - TLBs exploit **locality** → tendency for processes to access same files over time
  - Managing TLBs can be **hardware-loaded/software-loaded** (add entries)
  - OS needs to ensures TLB & page table are consistent
    - Ex. change protection bit in PTE, need to update copy in TLB
  - Context switch → new address space → need to invalidate/flush all TLB
  - When TLBC misses, new PTE has to be loaded, a cached PTE must be evicted → use TLB **replacement policy** (implement HW, often simple)
- Address Translation Possibilities
  - Common case: process is executing on the CPU, and it issues a read to a virtual address
    - Each read address goes to the TLB in the MMU (all done by hardware)
    - 1. TLB does loopup using the **page number** of the address
    - 2. Common case is that the page number matches, returning a **Page Table Entry (PTE)** for the mapping for this address (Minor Faults here)
    - 3. TLB validates that the **PTE protection** allows for reads (Major Faults here)
    - PTE marked valid & read permission is on
    - 4. PTE specifies which **physical frame** holds the page
    - 5. MMU combines physical frame & offset into a **physical address**
    - 6 MMU reads from that physical address, returns value to CPU
- **Page Faults** – what happens address translate is not immediate
  - **Minor Fault** – TLB does not have a PTE mapping this virtual address; 2 possible cases:
    - 1. Hardware TLB → MMU loads PTE from page table in memory
      - OS has already set up the page tables so that the hardware can access it directly
    - 2. Software TLB → Trap to OS, lookup & load PTE from page table into TLB
      - Returns from exception, retry memory access
    - Now there's a PTE for the requested address; minor b/c no I/O needed
  - **Major Fault** – PTE exists, but memory access violates PTE protection bits, trap to OS:
    - 1. **R/W/X** – op not permitted on page, may send fault back up to process
    - 2. **Invalid** – virtual page not allowed/not in physical memory
      - Virtual page not allowed in address space → OS sends segmentation fault back
      - Page not in memory (**Major Fault here**) OS allocates frame, swap it in from disk
    - Minor Fault must proceed a major fault
- **Temporal Locality** – local ref recently likely be ref again –ex. i var in loop structure;
- **Spatial Locality** – local near recently ref local are likely to be ref soon –ex. array being iterated
- Memory Policies
  - **Fetch Policy** – when to fetch a page ← **somewhat important** w.r.t saving time
  - **Placement Policy** – where to put the page in memory ← **least important**
  - **Replacement Policy** – what page to evict to make room? ← **most important**
- Fetch Policy: **Demand Paging** – pages loaded from disk when referenced; evicted to disk when memory is full and there's more demand ← cause Major Page Fault miss
  - OS allocates a page frame, reads page from disk
  - **Dirty pages** need to be written to disk b/c page in memory is diff from the page in disk
    - **Write Eviction** → indicated by Modify/Dirty bit in PTE
  - **Clean pages** don't but, but still need to know where on disk to read them from again
  - **Timing** disk read is initiated when process needs page; but only fault on 1 page at a time
- Fetch Policy: **Pre-paging** (**Prefetching**) – predict future page use at time of current fault
  - Based on: Spatial & Temporal locality or history
  - Good b/c disk good at large transfers, bad at individual reads
  - If guess wrong → loaded unneeded things, evicting something important (tradeoff)
- **Placement Policy** – in paging systems, memory management hardware can translate any virtual-to-physical mapping equally well; why preference for some mapping?
  - **Non-Uniform Memory Access** (NUMA) multiprocessors → any processor can access entire memory, but local memory is faster for some CPUs dependent on location
- **Replacement Policy** – how to pick a victim page to evict if memory is full
  - Goal of replacement algorithm reduce the fault rate by selecting the best victim
  - **Best page** to evict = one never used again; **next best** = one won't be used for longest time
- Replacement: **Belady's Algorithm** (OPT/MIN algorithm) –lowest fault rate or any ref stream
  - **IDEA**: replace the page that will not be used for the longest period of time
  - Problem: have to know the future perfectly
    - Used as a yardstick to compare w/ other page replacement algorithms
  - Random replacement = lower bound for replacement algorithms
- Replacement: **First-In First-Out (FIFO)**
  - **IDEA**: Maintain a list of pages in order which they were paged in; evict the oldest page
  - Problem: uncertain whether 1st page might not be used again b/c no additional info.
  - **Belady's Anomaly** – fault rate might increase when algorithm is given more memory
    - Condition: 2 memory sizes: large set L, small set S; 1. At least one page in L contained within S; 2. S contains at least one page not in L
  - Used by Windows XP
- Replacement: **Least Recently Used (LRU)**
  - **IDEA**: evict page that hasn't been used for longest time in past
  - Problem: looping pattern that's just larger than the # frames
  - Exact implementation is costly, need to approximate
    - Option 1: timestamp every reference, evict oldest page
      - Problem: need to make PTE large enough to hold meaningful time stamp
      - Problem: need to examine every page on eviction; find one w/ oldest time stamp
    - Option 2: keep pages in a stack;
      - On reference move the page to the top of the stack
      - On eviction, replace page at bottom
      - Problem: need costly to manipulate stack on EVERY memory reference
- Replacement: **Second Chance Algorithm**
  - **IDEA:** FIFO, but inspect reference bit when page is selected
    - If Ref bit = 0, replace the page; if Ref bit = 1, clear ref bit;
    - Combine w/ Modify bit to create 4 classes of pages:
      - clean + not recently used; dirty + not recently;
      - clean + recently used; dirty + recently used
  - Implementation: arrange all physical page frames in big circle (clock)
  - Linux uses second-chance clock algorithm
- Replacement: **Counting-based replacement**
  - **IDEA:** count number of uses of a page
  - **Least-Frequently-Used** (LFU) – replace the page used least often
    - Problem: pages that are heavily used at one time sticks around; even if not needed
  - **Most-Frequently- Used** (MFU) – favors new pages
- **Page Buffering** → Replacement Policies assumed replacement algorithm is ran and a victim page selected when a new page needs to be brought in → too costly on every page fault
  - Maintain pool of free pages, Run algorithm when pool too low "**low water mark**"
  - Free enough pages to at once replenish pool "**high water mark**"; on fault grab from list
- **OS161 Implementations**
  - **Coremap** – used to track the use of each physical page frame of memory
    - An array of coremap_entry structs, 1 per page frame
  - **Software-Loaded TLB**, uses a sparse & linear page table
    - When process loads, define regions in virtual addr space via `as_define_region`
    - Each region has array of `lpages` (logical pages) ← page table entries
- **Fixed vs. Variable Space Allocation** – determine how much memory to allocate to processes
  - **Fixed space algorithms** – each process is given a limit of pages it can use
    - **Local Replacement** – When reaches the limit, it replaces from its own pages
  - **Variable space algorithms** – process' set of pages grows/shrinks dynamically
    - **Global replacement** – one process can ruin it for the rest, take up all of memory
    - **Local Replacement** – replacement and set size are separate
- **Working Set Model** – used to model dynamic locality of a process' memory usage
  - **IDEA**: get a sense of how many pages this process needs
  - Working Set WC(t, delta) = { pages P | P was ref'd in the time interval (t, t-delta) }
    - t = time; delta = working set window (measured in page refs)
  - Problem: how to determine delta, how do we know when working set changes ← hard
  - Working Set used as an abstraction, intuition still valid
- **Page Fault Frequency** (PFF) – variable space algorithm w/ ad-hoc approach, monitor fault rate
  - If fault rate above high threshold → more memory; If below a threshold, reduce memory
- **Thrashing**: OS time paging data (I/O) > time executing user programs; no useful work done
  - When thrashing → OS **overcommitted;** Page replacement algorithms should avoid this
    - Can't determine which page should be in memory to reduce faults
  - Solution: by more RAM
- **Sharing** – shared memory allows processes to share data using direct memory references
  - Have PTEs in both tasks map to the same physical frame, can have diff protection values
  - Must update both PTEs when page becomes evicted/invalid
  - Can map share memory at same/different virtual addresses in each address space
- Problem: shared pointers inside shared memory segments are invalid
- Copy-On-Write

FILE SYSTEMS

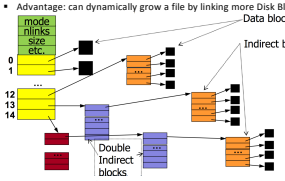
- **File (management) systems** – long-term (persistent) information storage, requires
  - 1. Store very large amounts of information
  - 2. Information must survive the termination of process using it
  - 3. Multiple processes must be able to access info concurrently
  - Need: Implement an abstraction for secondary storage (**files**); Organize files logically (**directories**); Permit sharing of data between processes, people, and machines; Protect data from unwanted access (**security**)
- **File** – a named collection of data w/ some properties (attributes)
  - **File Types** – encoded in the file name or contents (windows = extensions, UNIX = char)
  - **File Access Methods** → general-purpose FS = simple methods; DB more sophisticated
    - **Sequential access** – read bytes one at a time, in order
    - **Direct access** – random access given block/byte number
    - **Record access** (DB) – fixed or variable length; **Indexed access** (DB)
  - **File Attributes** – part of FS metadata (the "other" info beside filename, file data)
    - Protection (R/W/X) | Creator/Owner | Current Size
    - Creation Time | Last Access Time | Last Change Time
    - **Metadata** is the information that describes properties of the data; not the data itself
  - **File Operations**
    - Creating (find space in FS, add entry in dir mapping filename & attributes to location)
    - Writing | Reading | Repositioning
    - Deleting a file | Truncating (keep attributes, erase (part of) file contents)
  - Handling file operations → search dir for entry associate w/ the given filename
    - Index into this table used on subsequent operations → no need to search twice
  - When files first used actively, information stored in 2 levels
    - 1. **A System-Wide Open-File Table** stores file's attributes info.
    - 2. **A Per-Process Table** records all files that each process has opened → holds current file position for each process
      - Entries in the Per-Process Table points to the entry in the system-wide Open File Table b/c for process independent info



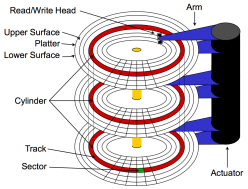
- **Directories** – list of entries –i.e. file names w/ associated metadata
  - Multi-level directories (`/`, `/usr/`, `/usr/local/`, `/home`,..)
  - **Current Working Directory** – the directory which filenames or paths are specified w.r.t.
    - Absolute path/filename start from root of the directory tree
    - Special names: "." == current directory; ".." = parent directory
  - Directory's entries list unordered/random ← sorted by algorithm that reads the directory
  - **Directory Operations**
    - Search – find a particular file within dir | Create file – add a new entry to dir
    - Delete file – remove an entry from dir
    - List directory – return filenames and requested attributes of entries
    - Update directory – record a change to some file's attributes
    - UNIX → C runtime library provides high-level abstraction for reading directories
  - **Directory Implementation** – usually a **tree-structure**
    - **Acyclic graph** allows for shared directory – same file/dir in 2 different locations
- **File Links** – file sharing implemented w/ dir entry called a **link** (pointer to another file/dir)
  - **Hard Links** – secondary dir entry is identical to the first
    - Points to same iNode → if file moves or filename changes, Hard Link follows
  - **Symbolic (soft) link** – directory entry refers to file that holds the true path to the linked file ← level of indirection
    - Is a path resolve upon access → if file moves/filename changes, Soft Link don't follow
  - Issues w/ Acyclic Graphs – a file may have multiple absolute names w/ Hard Links
    - Deletion: when can the space allocated to a shared file be de-allocated, reused
      - Symbolic links: deletion of a link = OK; deletion of the file entry itself de-allocates space and leaves the link pointers dangling
      - Hard links: keep a reference count
    - Sharing: permissions of a directory w/ hard link might update
- **Path Name Translation**; ex. open `/one/two/three` in FS
  - 1. Open directory `/` (the root directory, well known, can always find)
  - 2. Search for the entry `one` → get location of `one` in the directory entry
  - 3. Open directory `one`
  - 4. Search for `two`, get location of `two`
  - 5. Open directory `two`
  - 6. Search for `three`, get location of `three`
  - 7. Open file `three`
  - System spends a lot time walking dir paths → this is the reason open call is separate from read/write call; OS will cache prefix lookups for performance
- **File System Mounting** – FS "namespace" might be built by gluing together sub-trees from multiple physical partitions w/ each device (partition) stores a single file system
  - Mount point = empty directory in the namespace
- **File Protection/Permission** – protection system dictates whether a given **action** ("how") performed by a given **subject** ("who") on a given **object** ("what") should be allowed
  - Types of Access
    - None | Reading | Changing Protection
    - Knowledge | Appending | Deletion
    - Execution | Updating | UNIX: R/W/X only
  - 2 Approaches to protection
    - **Access Control List (ACL)** – for each object maintain a list of subjects and their permitted actions → object-centric, easy to grant/revoke/manage
      - Problem: when objects are heavily shared, ACLs become large (UNIX use groups)
    - **Capabilities** – for each subject, maintain a list of objects and their permitted actions
      - Easier to transfer (like keys) can handoff; does not depend on object
      - Problem: to revoke capabilities → keep track of all subjects that have capabilities



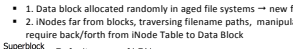
- **File System Implementation** Overview
  - **Master Block** – determines location of root directory, location on disk always known
  - **Free Space Bitmap** – determines which blocks are free/allocated; on disk/cached in RAM
  - Remaining space = disk blocks used to store files/directories
    - Block size (usually 4KB, defined by a FS); disk space allocated w.r.t num of blocks
- **Directory Implementation**
  - Option 1: **Linear List** of filenames + pointers to their data blocks
    - Requires linear search to find entries ← easy to implement, slow to execute
  - Option 2: **Hash Table**, hash a filename to get pointer to entry in the linear list
- **Disk Layout Strategies** – how to find all the blocks of a file? 3 options:
  - 1. **Contiguous allocation** – like memory; fast, simplifies direct access
    - Problem: inflexible, causes fragmentation, need compaction ← slow
  - 2. **Linked/chained structure** – each block points to the next, directories points to 1st
    - Problem: good for sequential access, bad for all others
  - 3. **Indexed structure** – indirection, hierarchy;
    - need **Index Block** to points to data blocks; may need multiple index blocks
    - Handles random access better, still good for sequential access
  -
- **UNIX iNode** – indexed structure for files
  - All file **metadata** is stored in an iNode
    - mode, file size, owner, permissions, nlink = number of dirs w/ this iNode # as a link
  - iNode smaller than data blocks: UNIX = 64bytes, Linux Ext = 72bytes
  - Each iNode contains 15 block pointers
    - 12 are Direct lock pointers = disk address of 1st 12 Disk Blocks in the file
    - **Single Indirect** block ptr = addr of Disk Block w/ addrs of Disk Blocks
      - Suppose Disk Block address = 4btes; Disk Block Size = 4096Bytes → a single direct block ptr gives 4096/4 more disk blocks = 4 Mbytes
    - **Double Indirect** block ptr = addr of Disk Block w/ addrs of single indirect blocks
    - **Triple Indirect** block pointer
    - Advantage: can dynamically grow a file by linking more Disk Blocks to iNode



  - Directory entries map filenames to iNodes
    - iNodes are not directories → they describe where on the disk blocks for a file are
    - Directories are files in UNIX, iNodes are also used to describe where the blocks for directories are place on the disk
      - Each **entry** in a directory's entries stored as <iNode number, filename>
    - ex. to open `/one`
      - 1. Use Master Block to find iNode for `/` on disk, read that iNode into memory
      - 2. Read the Data Block for `/`, look for the entry `one` in its directory entries
      - 3. From the entry, read the iNode for `one` into memory
      - 4. From the iNode for `one`, find & read its first Data Block → accessed file
- **Secondary Storage** – anything outside primary memory (RAM); does not permit direct instruction execution/data fetch via machine LOAD/STORE instructions
  - Characteristics: large, cheap, persistent, slow
- **Disk Components** (for magnetic spinning disk)
  - **Actuator** rotates, swings the **Arm** across the Platter
    - **Read/Write Head** at the end of the Arm, reads the magnetic encoded information from the surfaces of the platter → multiple heads, but move in sync (by 1 actuator)
  - **Platter** 2 surfaces (**Upper Surface**, **Lower Surface**); stores magnetically encoded data
  - **Cylinder** – slice through the platter where all the heads are over the same Track on all the platters at the same time
    - **Track** = concentric rings making up the platter
    - **Sector** = division of a track, **sector size** = 512 bytes at the very least
      - Can't read/write single bytes, must be at least 1 sector

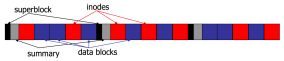


- **Disk Request Performance** – Depends on a number of steps
  - **Seek** – moving the disk arm to the correct cylinder; latency = arm moving speed
    - Typically = 1 – 15 ms; avg = 5 – 6 ms; improving slowly (7 – 10% per year)
  - **Rotation** – waiting for sector to rotate under the head; latency = disk rotation rate (RPM)
    - Typically 5400/7200 RMP; avg latency = ½ disk rotation = 4ms (7200 RPM); no change
  - **Transfer** – transferring data surface → disk controller electronics + sending to the host
    - Latency depends on data density = more bytes per platter
    - Typically ~100 Mbytes/s, average sector transfer time = 5 us; improving rapidly
  - Most disk latency comes from positional delays = Seek + Rotation ← want to minimize
- **Disk & OS** – OS hides disk messes details from higher level software –ex retry failed reads
  - OS provide different levels of disk access to different clients
    - Each layer abstracts detail below it for layers above it
  - **Program** → <File Name, Offset> (logical file)
  - <File Name, Offset> → **File System/DB** → <Partition, Block #> (logical disk)
  - <Partition, Block #> → **Device Driver** → <Disk #, Sector #> (physical disk)
  - <Disk #, Sector #> → **I/O Controller** → <Cylinder, Track, Sector> (physical disk)
  - <Cylinder, Track, Sector> → **Disk Media**
- **Disk Interaction** – exact disk request = Cylinder #, Surface #, Track #, Sector #, Transfer Size
  - Not needed for modern disk → provide a higher level interface (SCSI/SATA)
    - Disk expose its data as logical array of block [0, ..., N] which maps to cylinder/surface/track/sector via disk electronics
  - Common Storage Interface for Modern Disks:
    - **Linear array of block** → typical block size = sector size = 512 bytes
    - **Number of blocks** = device capacity / block size
  - Increase disk size effects free space managements
    - Ex. 1GB disk, block size = 512 bytes
      - 2^{21} blocks = 2^{21} bits = 2^{18} bytes = 256 Kbyte for bitmap
    - Ex. 80 GB disk, same everything else → 20 Mbyte bitmap
    - Alternatives:
      - **Linked List** of free blocks = inefficient
      - **Grouping** = first free block contains addresses of N other free blocks
      - **Counting** = record first free, number of consecutive frees
- **Optimize disk performance**; high-level disk characteristics →2 goals:
  - **Closeness** – reduce seek times by putting related things close –i.e. same cylinder
  - **Amortization** – amortize rotation delay by grabbing lots of useful data (read ahead)
    - Assumes closeness → adjacent blocks to the requested block contains related data
    - Ex. **Track Buffer** → lets disk read a whole track, after seek completes the disk reads entire track into memory w/o waiting for rotation; the right sector is read, send a reply and continues to finish reading the track → cached an entire cylinder.
  - Disks perform w/ **seeks reduced** + **large transfers used**; through
    - 1. **Disk Scheduling** → order access requests
    - 2. **Allocation Algorithms**, put related data close together ← most important
- **Original UNIX FS** – sees storage as linear array of blocks w/ **Logical Block Number** (LBN)
  - **Superblock** – first block, read by OS at mount; stores FS meta information:
    - FS size, iNode table size, iNode table start, Data Block start
    - First 2 bits of FS = **magic number**, uniquely identify the FS type
  - **Free Space Bitmap** – bit for every Data Block indicating if its free
  - **iNode Table** – contains all the iNodes of the FS
  - LBN space allocated in logical fashion: Superblock → Bitmap → iNodes → Data Blocks
  - **Pro**: simple, straight forward implementation; easy to recover from errors
    - Scan entire iNode table for a lost entry since location of all iNodes are known
  - **Con**: Poor utilization of disk bandwidth
    - 1. Data block allocated randomly in aged file systems → new files scattered iNodes
    - 2. iNodes far from blocks, traversing filename paths, manipulating files/directories require back/forth from iNode Table to Data Block



- **Fast File Systems** (FFS) – improved disk utilization, device-aware for performance
  - **Cylinder (allocation) Groups** – partitioned data into groups of cylinders
    - Data blocks of same file allocated in same cylinder group
    - Files in same/parent directory allocated in same cylinder group
      - Except for Root b/c need to purposely spread out to other cylinder groups
    - iNodes for files allocated in same cylinder group as file data blocks
    - Each Cylinder Group includes Free-space bitmap & Summary block (inodes)
  - Allocation in cylinder group provides **closeness** → **reduce number of long seeks**

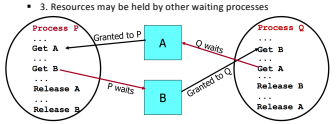
  - Disk must have free space in each cylinder group so for new files
    - 10% each cylinder reserved for root access only
    - If preferred cylinder group is full, allocate to nearby group
  - Other improvements
    - 1. Increased block size 1Kbyte → 4Kbyte
      - Problem: **Internal Fragmentation** b/c small files only need a portion of the block size, but still require a full block to be allocated
      - Solution: **Fragments** → single block used to store 1Kbyte pieces of multiple files
    - 2. FFS solves media failures w/ backing up superblock
  - FFS Consistency
    - File systems operations affect multiple metadata blocks
    - Metadata updates are synchronous operations even w/ Write-Behind to prevent dangling pointers in the FS incase a crash in middle of syscall
      - Write newly allocated iNode to disk before its name is entered in a directory; reverse when removing a file
  - FFS Observation 1: If the server crashes in between any of these synchronous operation, then the file system is in an inconsistent state
    - Solution 1: **f**sck (fisk check) post-crash recovery process to scan file system structure and restore consistency by applying consistency rules, recurs from root;
      - 1. All blocks pointed by iNode/indirect block → marked allocated in bitmaps
      - 2. All allocated iNodes must be in some directory entry
      - 3. iNode link count must match
    - Solution 2: log all updates to disk, enable roll-back or roll-forward
  - FFS Observation 2:
    - The performance of FFS is optimized for disk block clustering, improve reads
    - Writes are not well-clustered → have worse locality than reads b/c they include updating iNodes, Data Blocks, and other metadata info.
- **Disk Scheduling** – order to handle read/write requests
  - **First Come First Serve** (FCFS) –i.e. do nothing
    - Pro: Reasonable when load is low
    - Con: Long waiting times for long request queues
  - **Shortest Seek Time First** (SSTF)
    - **IDEA**: take the next item in the queue closest to the current position
    - Favors middle blocks b/c given a stream of requests arm tends to be in the middle
      - forcing arm to seek outer/inner track difficult → **starvation** possible
  - **SCAN/C-SCAN** (elevator)
    - **IDEA**: Service requests in one direction until done, then reverse
      - Move arm all the way to one end, even if there's no requests there
    - Variant: Typewriter → only go in 1 direction
  - **LOOK/C-LOOK**
    - **IDEA**: Like SCAN but only go as far as the last request in each direction, not the full width of the disk
  - If requests coming slow → all scheduling degrades to FCFS
- **File Caching**
  - **File Buffer Cache** – in memory copy of persistent data, system wide
    - Tradeoff: competes w/ VM for physical RAM space ← replacement algorithm (LRU)
  - **Caching Writes (Write-Behind)** – maintain a queue of uncommitted `write`, flush to disk periodically or force a flush w/ `sync`
  - **Read Ahead** – FS predicts next block requested by process → requests it from disk
    - Overlaps I/O with computation time; when process requests next block ← cached
    - Big win w/ sequentially accessing files
- **Log Structure File System** – **IDEA**: Write all file data in a continuous log including current system data & metadata appending to a continuous log
  - Continuous log = large sequential writes → Amortize write positioning delays
  - **Cleaner** process runs in the background to reclaims space from overwritten/deleted blocks
  - Needs an **iNode Map** b/c an iNode # is no longer a simple index into a fixed location
  - **LFS Read** – to read a file from disk, lower than traditional FFS but expect TLB most times
    - 1. Read the superblock to find the index file
    - 2. Read the index file (linear search on block of iNodes)
    - 3. Use the disk address in iNode to read the block of index file containing iNode-Map
    - 4. Get the file's iNode
    - 5. Use the iNode as usual to find the file's data block



- Linux Ext3
- **New Technology File System** (NTFS)
  - Each volume (**partition**) is linear blocks = 4 Kbytes; ref block **Logical Block Number** (LBN)
  - Each volume has a **Master File Table** (MFT) = seq of 1 Kbyte records
  - One/more record per file/directory; similar to iNodes, more flexible
  - Each MFT Record is sequence of variable length (attribute header, attribute value) pairs

DEADLOCK

- **Deadlock** – permanent blocking of a set of processes that either:
  - 1. **Compete** for system resources; or 2. **Communicate** with each other
  - Each process is set blocked, waiting for an event can be cause by another process in set
- **Resource** – any object that might be needed by a process to do its work
  - **Reusable resources** – used by one process at a time, then released
  - **Resource deadlocks,** 3 reasons:
    - 1. Resources are finite (Mutual Exclusion)
    - 2. Processes wait if a resource they need is unavailable
    - 3. Resources may be held by other waiting processes



- **Necessary & Sufficient** Conditions for Deadlock
  - **Necessary Conditions** – remove any = no deadlock
    - 1. **Mutual Exclusion** – only one process may use a resource at a time
    - 2. **Hold & Wait** – process may hold allocated resources while awaiting on resources
    - 3. **No Preemption** – no resource can be forcibly removed from a process holding it; owning process must volunteer release
  - **Sufficient Condition** – sufficient for deadlock b/c necessarily conditions already in-place
    - 4. **Circular Wait** – a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain; implies Hold & Wait
      - Circular Wait arises from program execution sequence
      - Hold & Wait is a policy decision w/ OS design
- **Deadlock Prevention** → ensure one of the four conditions doesn't occur
  - **Prevent Mutual Exclusion**, but also breaks correctness
  - **Prevent "Hold & Wait"** – must request all resources at once, will block until entire request can be granted simultaneously → may wait along time (**starvation**)
    - Processes need to know all resources requirements in advance
    - Alt: release all currently held resources when a new one is needed
      - make a new request for entire set of resources
  - **Prevent No-Preemption**, forcibly remove a resource from a process, assign to another
    - Need to save the state of the process losing the source so it can recover later
    - Problem: when process granted a lock on CV, it assumes that it's the only entity accessing that resource → **incorrectness**
    - Impossible for consumable resources
  - **Prevent Circular Wait**, assign a linear ordering to resource types & require that a process holding a resource of one type R can only request resources that follow R in the ordering
    - Hard to come up w/ total order of resources; May hold unwanted locks
- **Deadlock Avoidance** – knows list of conditions, but orders events to ensure no Circular Wait
  - 1. Don't start a process if its maximum resource requirements, together w/ the maximum needs of all processes already running, exceed the total system resources
    - Pessimistic b/c assumes all processes will need all their resources at the same time
  - 2. Do not grant an individual resource request if this request combined with everything already granted might lead to deadlock
  - **Safe States** = state where there is **A** sequence of process executions does cause deadlock;
    - **IDEA:** Even if every process requests their maximum allocation immediately → system doesn't have to grant requests immediately
  - **Unsafe States** = system no longer can enforce execution to ensure no deadlock
    - In the worst case a deadlock might occur; but does not imply already deadlocked
  - **Deadlock Avoidance Algorithm** (Banker's algorithm) – for every resource request
    - Update state assuming request is granted, check state == safe
    - If so continue; if not, restore the old state, block process till safe to grant the request
  - Restrictions on Avoidance
    - 1. Maximum resource requirements for each process must be known in advance before their execution
    - 2. Processes must be independent
    - 3. There must be a fixed number of resources to allocate (can't change)
- **Deadlock Detection & Recovery** → let deadlocks occur, detect it happens, try to break it
  - Check for circular wait condition periodically
  - **Deadlock Detection** → finding Circular Waits == a cycle in the **Resource Allocation Graph**
    - **Nodes** are processes (circles) and resources (squares)
    - **Arcs** (resource → process) = allocations; (process → resource) = un-granted requests;
    - Note: w/ multiple instances of a resource type, cycles may exist w/o deadlock
  - **Deadlock Recovery** → break the cycle
    - **Drastic** – kill all deadlocked processes
    - **Painful** – backup and restart deadlocked processes (hopefully non-determinism)
    - **Better** – selectively kill deadlocked processes till cycle is broken (check after each kill)
    - **Tricky** – selectively preempt resources until cycle is broken (process rollback)
- **Ostrich Algorithm** – ignore the problem; hope it doesn't happen often;
  - Recall 3 reason of resource deadlock
    - Prevention/Avoidance/Detection deal w/ last 2 reasons
    - Modern OS virtualize most physical resource → eliminate first reasons
  - Just expect system programmer to follow lock ordering
  -

SECURITY

- **Computer Security** – techniques for computing w/ **adversaries** presence; 4 requirements
  - **Confidentiality** – preventing unauthorized release of info
  - **Integrity** – preventing unauthorized modification of info
  - **Availability** – ensuring access to legitimate users
  - **Authenticity** – verifying the identify of a user
- **Cryptography** – techniques for communicating w/ **adversaries** presence
- **Types of Threats**, 4 major ones
  - **Interception** (eavesdropping) – attacker gains knowledge they should not have access to
    - Loss of **Confidentiality**; ex. intercepting network packets
  - **Modification** – attacker alters existing files, programs, network packets, etc.
    - Attack on **Integrity**
    - Special Case: **Destruction of Data** = Attack on **Availability**
  - **Theft of service** – attacker installs daemon; Attack on **Availability**
  - **Fabrication** – attacker creates fake objects (files, messages)
    - Appears to be from trusted source; Attack on **Authenticity**
- **Vulnerabilities in the System**
  - **Physical Access** – Unauthorized physical access enables unauthorized digital access
  - **Humans** – who to trust, to what degree?
  - **Operating System** – exploit flaws in OS, circumvent security protocols
  - **Networks** – Data travels over unsecured communication lines
- **Authentication**, based on 4 principles → needs to identity
  - Something the **user knows:**
    - **Password Authentication** → storing passwords 3 options:
      - Option 1: on disk, protected by file access restrictions
      - Option 2: Store function of password on disk
      - Option 3: Add random "salt" to password
    - **Challenge-Response Authentication** → user provide right answer
  - Something the **user has:**
    - Authentication with **Physical Object** –ex. Keys, Swipe Cards, Pins, Smart Cards
  - Something the **user is:**
    - **Biometrics** → unique & hard to forge
      - Requires user enrollment + acceptable to human users
  - Something the **user does**:
    - **Behavioral Biometrics** –ex. pattern of password typing
- **Insider Attacks** –ex. Login Spoofing
- **Malicious Software**
  - **Trap Doors** – program contains secret entry point that allows attacker to bypass security
  - **Logic Bombs** – destructive code in legitimate program triggered by some event
  - **Trojan Horses** – apparently useful program, tricks users to run it, vehicle for virus/bombs
  - **Viruses** – A program that can "infect" other programs by copying itself into them
  - **Worms** – Program that spreads via network connections
- **Stack & Buffer Overflow Attacks** – most common means to gain unauthorized access to sys
  - 1. Overflow some stack-allocated input buffer past the space allocated for the input
  - 2. Overwrite the return address with the address of exploit code
  - 3. Overwrite next space in stack with the exploit code itself
- **Exploiting Code Bugs** –ex. Format String Bugs
  - C library formatted I/O string allow almost arbitrary inspection & modification of program
  - `printf` assume xtra args (%d) been pushed onto stack, dumps stack contents to output
- **Intrusion Detection** – know when system has been compromised; 2 approaches
  - 1. **Signature-based detection** – examine system activity/network traffic for patterns that match known attacks
  - 2. **Anomaly detection** – identify patterns of "normal" behavior over time
    - Detect deviations from those patterns ← needs un-tampered user activity records
- **Trusted Computing Base** (TCB) – set of components (HW, SW, people) that you trus
- **Network Communications**, 2 main categories of attacks
  - **Passive** – eavesdropping or monitoring communications
    - **Eavesdropping** – read network packets that were not intended for attacker
    - **Encryption basics**: Person A w/ (Key K_A) → Person B w/ (Key K_{PD})
      - 1. Person A sends Plaintext Message **P**
      - 2. Cipher-text **C** = Encryption Function **E**(**P**, Key K_E) ← send over network
      - 3. **P** = Decryption Function **D**(C, Key K_D)
    - Encryption & Decryption Keys can be same (symmetric/secret-key) or diff (public key)
      - **Secret-Key cryptography** – fast encryption/decryption algorithm, can't distribute
      - **Public-key cryptography** – pair of keys (encrypt & decrypt); Encryption key can be published, deception key is kept secret → algorithm slow
    - **Traffic Analysis** – infer information based on sender/recipient of messages, size of messages, frequency of communication
      - Defense: **obfuscate** communication pattern (ex. all messages same length)
  - **Active** – modification or tampering with the communication stream
    - **Replay** – capture a legitimate message, re-send it later, produce unauthorized effect
    - **Modification of messages** – alter contents of message ()
      - Defense: **message digest** use 1 way function, computer a fixed-length output, compare digest of received vs. digest of sent message
    - **Masquerade** – attacker pretends to be another entity or user
    - Denial of Service - system asset becomes unavailable to legitimate users
      - Attack on **Availability** –ex. Overload server w/ bogus requests
  - **Digital Signatures** – verify origin/integrity of message
    - Sender incl. something w/ message that could only be generated by sender
    - Public-key cryptography
      - **Confidentiality** – encrypt message with recipients **public key**; only intended recipient can decrypt using their **private key**
      - **Signatures** – encrypt message with sender's **private key**, anyone can decrypt using sender's **public key** → verifying sender created the message