## Major themes on OS

**Virtualization**

- Making physical resources (hardware) easier to use
  - E.g. CPU, memory, storage devices
- Making threads and processes seem like they own the whole hardware

**Concurrency**

- Coordinate multiple activities to run together without issues

**Persistence**

- Some data needs to survive crashes and power failures

**Limited Direct Execution on Processes**

- Give the process the control of the CPU but limit what they're able to do

**Kernel code or data structures that must be wired/pinned**

- Kernel Page Table
- Page Fault Handler
- Storage Device Driver
- Interrupt Descriptor Table

**Process Control Block**

- PID
- Process state (Run, ready, blocked)
- Program counter
- CPU Registers
- CPU scheduling information (process priority)
- Memory management info (page tables)
- Accounting information (resource use info)
- I/O status information (open files)

**Step needs between two modes:**

User -> Kernel:

- Boot time
- Hardware interrupt (keyboard press)
- Software exception (trap or fault, e.g. zero division)
- Explicit system call (getpid() or other syscalls)
- Interrupt handlers

Kernel -> User

- Context switching back from kernel to user (Process A -> Kernel -> Process B)
- Jumps to next available instruction

**Sharing Memory between threads / processes**

- shmget(): Get a shared memory segment region
- shmat(): Map shared memory segment to local address
- mmap(): Create shared memory regions (mapping from virtual to physical)

**Kernel Thread:**

Runs entirely in the Kernel.
User don't know anything about.

**Kernel-level Thread:**

Kernel-level threads are the threads that run in user programs, making things run concurrently.
Suffer from too much overhead

**User-level Thread:**

Fast and cheap
But totally invisible to the kernel, so kernel can make bad scheduling plans. E.x. block process whose thread initialized an I/O, while process has other threads can execute.

**System call:**

time(), getpid(), exec(), fork(), write(), read().

**Not system call:**

- malloc(), free(), printf(), scanf()
- fopen(), fread(), fclose(), fwrite(), exit(), pthread_create()

## Synchronazation

**Critical Section Solution requirements:**

- Mutual Exclusion
  (only one at a time, atomicity)
- Fairness
  (no starvation, fairly entered by all)
- Performance
  (overhead is not high compared to the work done in critical section)

**Implementing locks using atomic instruction**

```
typedef struct __lock_t { int flag; } lock_t;
void init(lock_t *mutex) {
  mutex->flag = 0; // 0 -> lock is available, 1 -> held
}
void lock(lock_t *mutex) {
  while (test_and_set(&mutex->flag, 1) == 1)
    ; // spin-wait (do nothing)
}
void unlock(lock_t *mutex) {
  mutex->flag = 0;
}
```

**Problem with Spinlock**

Simple spinlocks have three problems:

- Busy waiting wasting CPU cycles
- Starvation is possible
- Deadlock is possible through priority inversion

**Common function for lock**

**Structure:**

// Is a lock
pthread_mutex_t mutex;
// Is a conditional variable
pthread_cond_t cv;
// Is a binary semaphore
Sem sem = sem_init(1);

**Operations**

pthread_mutex_lock(pthread_mutex_t *mutex);
pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mutex);

- Releases mutex, adds thread to cv's wait queue and sleeps
- Re-acquires mutex before return

pthread_cond_signal(pthread_cond_t *cv);

- Wake one enqueued thread

pthread_cond_broadcast(pthread_cond_t *cv);

- Wak up all enqueued thread

sem_wait(sem);

- Waits for a sem aphore to have a count > 0, sleeps

sem_signal(sem);

- increment semaphore count by 1, wakes up previously slept on semaphore if waiting

Readers/Writer with Semaphores

```
//number of readers
int readcount = 0;

//mutual exclusion to readcount
Sem mutex = sem_init(1);

//exclusive writer or reading
Sem w_or_r = sem_init(1);
```

```
Writer {
    // lock out others
    sem_wait(w_or_r);

    Write;

    // up for grabs
    sem_signal(w_or_r);
}
```

```
Reader {
    sem_wait(mutex);
    readcount += 1;

    if(readcount == 1) // first reader
        sem_wait(w_or_r);
    sem_signal(mutex);

    Read;

    sem_wait(mutex);
    readcount -= 1;

    if(readcount == 0) // last reader
        sem_signal(w_or_r);
    sem_signal(mutex);
}
```

**Deadlock solutions**

- Break mutual exclusion
(using atomic instructions so we can remove locks, but implementing atomic instructions for complex stuff is hard)
- Break hold and wait
(so that processes only continue after obtaining all resources needed, so processes doesn't hold and do nothing about it, e.g. trylock(), limits concurrency and sometimes doesn't know what resources are necessary but may cause live lock)
- Break no-preemption
(allow threads to forcibly stop other threads from holding a resource so the thread can take it, not safe because resources cannot rollback most of the time)
- Preventing circular wait
(so that processes can only request for resources in a certain order, one after the other, e.g. R1 then R2 then R3, no R1 means cannot request R2, but its hard to find a good order for complicated stuff)

## Scheduling

Goal:

- Fairness: each thread receives fair share of CPU
- Avoid starvation
- Policy enforement: sometimes tha twe want to prioritize some processes
- Balance: all parts of the system should be busy

Bases on different system they may have their own goals:

**Batch system**:
Group all similar jobs in to a batch. Focusing on finishing job. Need to maximize the efficiency

**Interactive system**:
Minimize time between receiving request and starting to produce output "simple" tasks complete quickly

**Convoy effect** - Processes all wait for one big process to finish, present in FCFS

**Timer interrupts** are necessary to implement preemptive scheduling.

**Strawman Implementation**

Single-level Queue Scheduling
High overhead to find highest priority
O(n) search or insertion

**Multi-Level Feedback Queue Scheduling**

One queue per priority level
Process change among queues
Always choose to run jobs in the queue with highest priority
Each queue can have its own scheduling algorithm

**Priority Inversion** - Lower priority stops higher priority things to be ran (i.e. lower priority thread holds a lock and is preempted, then higher priority can't take it)

**Priority Inheritance**

Priority of task holding the mutex inherits the priority of a higher priority task when the higher priority task requests the semaphore

**Unix CPU Schedulers:**

Add num tickets to PCB
Choose random winner, use counter incremented
If coutner > winner, pick that process.

More Unix Scheduling

- MLFQ with RR within each priority queue
- Priority is based on process type and execution history
- $P_j(i) = base_j + [CPU_j(i-1)]/2 + nice_j$
- $CPU_j(i) = U_j(i)/2 + CPU_j(i-1)/2$
  - $P_j(i)$: priority of process j at beginning of interval i; lower values equals higher priorities
  - $base_j$: base priority of process j
  - $nice_j$: user-controllable adjustment factor Prefer level of the thread given by user.
  - $U_j(i)$: processor utilization of process j at interval i
  - $CPU_j(i)$: exponentially weighted average processor utilization by process j through interval i

**Linux (2.4) CPU Scheduling**

2 Scheduling algorithms for tasks:
- time sharing - Real time tasks

**Credit based algo for time sharing**

Choose the process with most creadits
At timer interrupt running process loses one credit
Credits reach 0, it is suspended
If no runnable process have credit, do recrediting: credits /2 _ base
Block Processes get more credits

**Linux 2.5 O(1) Scheduling**

Each process gets a time quantum, based on its prority
Two arrays of runnable processes, activated and expired
Processes are chosen from active array
When quantuns expires, go to expired array with a new priority
When active empty swap two arrays.

**Linex 2.6 Scheduling**

140 queues one for each priority level, 0-99 for real time, 100-139 for time sharing
Each priority level has its own time slice
A thread stays active until it uses up time slice
After running, priority and time slice recalculated and moved to a set of expired queues
"O(1) scheduling algorithm": uses bitmap indicates non-empty queues
Instruction `find-first-bit-set` finds first non-empty queue
"Dynamic priority": Threads assigned an initial priority of 120
Whenever a thread is awakened, decrement (max -5 toward 100)
When a thread is preempted, increment(max +5, toward 139)
Bonus based on average sleep time of thread
"Time Quantum": HIgher priority threads get longer time slice, aim to ensure CPU burst can complete without being preempted.

**Linux Completely Fair Scheduler(CFS)**

"Weighted fair queueing"
Divide CPU cycles among threads in proportion to their weights
"Algorithm"
Divide CPU cycles among threads in proportion to their weights
The thread accumulates virtual runtime(vruntime) when it runs
Higher priority accumulate vruntime slower
Threads are ordered by vruntim in the run queue. Lowest vruntime is scheduled next

**Security**

Authentication - Verifying the identity of a user, process, or device.
Confidentiality - Preserving authorized restrictions on information access. (Controlling who has access to information.).
Integrity - Guarding against improper information modification of data
Exploit - A piece of software that takes advantage of a flaw or vulnerability to cause unintended or unanticipated behaviour to occur on software or hardware.
Availability - Ensuring timely and reliable access to and use of information.

**Memory Management**
**Requirement**
**Relotation**: can map from virtual address to physical
**Protection**: protect private address from unwanted access
**Sharing**: Control how processes can share a part of memory
**Logical organization:** make things organized and separated in regions
**Address Binding:**
**Compile Binding:** No relocation is possible
**Loadtime binding**: Programs can be loaded to different address when they start but cannot be relocated later.
**Execution time binding:** Dynamic relation, change to physical address when running the program
**Dynamic Partitioning**: External framentation, no space for big stuff
Internal
**Compaction**
- OS may move processes around to create larger chunks of free space:

**Fixed Partitioning**:
Number of partitions determines number of active process & Internal fragement
**Relation for partitioning**
Need 2 hardware registers “base”, “limit” for relocation
Common problem is that processes must be allocated to contiguous blocks of physical memory
**Paging**
- Each process has its own page table, page tabel is stored in OS memory
- Translation done by MMU converts VAs into PAs using page table

**Calculation**
- offset bits = log2(page_size)
- Virtual address space bits - offset bits = bits used for levels
- # PTE per table:
2^(Virtual address bits - log2(page size))
- # of pages perlevel
Page size/PTE size

**TLB:**
- Caches mapping from VPN (all the level bits together) to physical page frame number and other information(Like protection, valid bits)
- We need to reload TLB on a process context switch

**Demand paging:**
- When you use swap files to swap pages from memory to your disk and reverse
- It is also used when a process first starts up

**sbrk():**
sbrk() updates VMAs(especially for heap area) with new address space range
**Replacement Policy:**
- Cold misses: First access to a page(unavoidable)
- Capacity misses: Replacement due to limited memory

**Thrashing:**
When you do too much switching, and you can’t do actually work.
**“Algorithm**:”
- **Belady’s Algorithm** (optimal): Replace the page that will not be used for the longest period of time.
- **First-In-First-Out**: Evict the one brought in longest time ago. Suffers from Belady’s Anomaly: The fault rate might increase when the algorithm is given more memory.
- **Least Recently Used**: Evict the page that has not been used for the longest time in the past. Exactg LRU is too expensive to implement in general.
- **Clock(second chance)**: A clock hand is used to select. If the ref bit is 0, choose this evicit. If ref bit is 1,set to 0 and move to the next.
- **S2Q:** Maintain two queues, A1: limited size FIFO queue (Pages only used once) Am: LRU queue (hold most pages)

**Simplified 2Q Algorithm**

On reference to page p
(equivalently, the frame allocated for page p):

if p is on the Am queue then
  move p to MRU position of Am
else if p is on the A1 queue then
  remove p from A1
  put p on Am queue in MRU position
else // first access we know about for p
  put p on A1 queue in youngest position

To allocate a frame for page p
(when all frames are in use):

if A1’s size is above its threshold then
  evict oldest page from A1 (first-in)
  put p in the freed frame
else
  delete LRU page from Am
  put p in the freed frame

- we have looping-sequential workload, LRU and FIFO are really bad

**Page buffering**:
- We can maintain a pool of free pages, and free pages when the pool becomes too small. On page fault, grab a frame from the free list. Can be rescued if page ref before reallocation.

**Kernel Virtual Memory**
- Wire Special code and data prevent the entire OS address be swaped

**Virtual Memory Area**
- Virtual address space contains regions with different modes
- VMAs give us a way to track space that the process expects to use, and some policy associated with it

**Working Set:**
- want it to be the set of pages a process needs in memory to prevent miss

WS(t, D) = pages P that was referenced in the time intreval [t, t – D)
where t is time and D is the working set window (measured in page refs). A page is in the wroking set only if it was referenced in the last D references.
**Sharing Memory:**
Multiple processes can have the same or different addresses mapped to same physical address
- The pointers inside the shared memory segment are invalid.

**Copy on Write:**
Defer large copies for as long as possible. Shared pages are protected as read-only in parent and child
Write generate a protection fault, trap to OS, copy page, change page mapping in page table, enable write permission, restart write instruction.
**mmap()**
Maps a virtual memory address to a specific file on the hard disk. Creates a VMA and maps to a file. Pagein/out apply to the file, not swap
**When given a virtual address:**
**1.** Check in TLB, if TLB hit, then use physical page frame number + offset to get PAs
2. If a TLB miss, then loop up in kernel page Table
3. If Kernel Page Table(valid address), then use physical frame number + offset to get physical address, also add the translation into TLB
4. If Kernel Page Table miss(Invalid Address), raise exception. (e.g. 0xffffffff)

## File System
**Requirement:**
- Store very large amount of information
- Information must survive termination of process using it
- Multiple process must be able to access data concurrently

**Link Type:**
Hard Link: Create a new directory entry with inode that points to the data block on disk. Real delete it if link_count = 0
Soft Link: Create a new file that contains the path to the linked file is created (Removing a file may lead to a dangling link)

**Allocating Disk blocks:**
- Contiguous: Each file is given a continuous set of blocks. Store the starting block and length
- Linked Allocation: Store the start and end block. Each block in a file contains a pointer to next block
- Indexed Allocation: Unix inodes. All file metadata are stored in an inode. Each inode contains 15 blocks pointer. First 12 direct, 13 single, 14 doulbe 15 triple

**Data Layout Strategies**

| | Advantages | Disadvantages |
|---|---|---|
| Contiguous | • Sequential access fast<br>• Allocation fast<br>• Deallocation fast<br>• Small amount of metadata | • External fragmentation<br>• Need compaction<br>• Need to move whole files around<br>• Inflexible |
| Linked | • Sequential access easy<br>• Disk blocks can be anywhere<br>• No external fragmentation | • Direct access is expensive<br>• If a data block is corrupted, could lose the rest of the file |
| Indexed | • Handles random access well<br>• Small files: quick sequential and random access<br>• No external fragmentation | • Limits file size<br>• Cost of access to bytes near the end of large files grows |

**Superblock**
Holds metadata about overall file system, including magic number(identifier). Often replicated across disk for reliability
**Maps**
Inode bitmap- Show which inodes are avilable (1 is used, 0 available)
Data bitmap- Show which data blocks are available
**Inode table**
Allocaed when the file system is created -> predefined max # files.
**Calculation**
BLOCK_SIZE/INODE_SIEZE inodes
For 128 bytes inode, 4K blocks, we need 4096/128 = 32 inodes per block
To allocate 160 inodes, we need 160/32=5 blocks
Pointers/Block = BLOCK_SIZE/POINTER_SIZE
Max file size =
(12 + 2^(10 + 20 + 30) *4KB \approx 4TB)
**Disk Representation:**
An **inode** is represented by square brackets [] with 3 elements
type (d - directory, f - file)
adddress of first block: a:0 means the first data block os file is block 0. a = -1, empty file
number of links: r: 3 there are 3 directory entries that reference this inode
Each **Data block** represent by square brackets **[]**
Directory blocks contain one tuple for each directory entry
**[(**name, inode_number**)]**
File block contain one character
**File operation: open**:
when the file is first used(open), the OS store an in-memory copy of its inode in a system-wide open-file table. Then create an entry in the process’s private open-file table
**Disk Characteristics**
- Try to minimize disk arm movement by clever scheduling of I/O requests.
- The seek time is the time needed for the arm to move from inward to outward(we want to minimize)
- The Latency time for the disk to spin to the correct data block position
- Two allocation strategies: Closeness(Putting related things together) and Amortization (amortize each positioning delay by grabbing lots of useful data)

**Problem with Linear array of blocks**
- Inode blocks are palced far away from data blocks, but we need to check inode table everytime accessing data block
- As the FS age, data block become sparse.

**Berkely Fast File System FFS:**
First disk aware FS
- Split the disk into multiple “cylinder groups”, each corresponding to a “track” on the disk. So minimizes seek time
- Put superblock and other metadata copies in each cylinder group
- Chops big file into smaller ones across cylinder groups.

**NTFS:**
Use MFT(Master file table) records to store files.
Each MFT records has just standard information, with a bunch of headers to separate the regiosn
Small files stores in MTF directly
Larrger files requires the data region to store the data in a bunch of “contiguous memory”, start block + leagth multiple times.
**FS Reliability:**
Consistant state: Either file operation not happen or operation completed (failure atomicity)
**fsck:**Checks super block, free blocks, inode state, inode links, duplicates, bad blocks and directories checks. Cannot know if a data block did get written and TOO SLOW
**Jounaling**
Write a log on disk of the operation you are about to do, before making changes in actual file system (checkpointing).
- 1. Transaction begin“TxBegin” (contain transaction ID) 2. Blocks with the content to be written 3. Transaction end “TxEnd”

1. In between writing the data block and TxEnd (You can only start writing TxEnd when all the other information that is needed for the update is doen writing)
2. After writing TxEnd, you also need a barrier (Update FS based on this safe Journaling when TxEnd is completed.
3. Mark transaction as free in journal
- Metadata journaling,

Metadata Journalling: Write data to data block, then write the journal of updating the metadata.
**SSD**
• Dies->Planes->Blocks->Pages
• Includes volatile on-device memory, processor, and interface logic to accept host commands and send responses.
• Write to a file system block: 1. Find the block in a plane. 2. Read all active pages in the block into the mem. 3. Update the target page in-memory. 4. Earse the block. 5. Write the entire block.
• **Flash Translation Layer (FTL)** to fit and optimize SSD based on logging.
  - Goal: 1. Translate R/W to logical blocks into R/Erases/progrmas on physical pages + blocks 2. Reduce write amplification( the amount of extra copying needed to deal with block-level erases) 3. Avoid early wear-out frequently written logical blocks.
• **Wear-leveling:** Technique ensures all of the Flash cells are being used roughly similarly.
• SSD blocks wear out quickly, we need to distribute writie operations uniformly.
• **Garbage Collection:** In log based mapping. At some point, there will be useless pages in a SSD blocks, then we read in block, log-write the useful pages, erase the SSD Block

**Security**
- **Stack Attack**: type long input str to execute things in code section
- **Format String Bug**: You can type in format specifiers in the input string
- **Access Control List** - Focus on objects, each object has a list of subjects and their permitted actions
- **Capabilities**: Focus on subjects: each subject has a list of objects and their permitted actions