211: Operating Systems

Intro

1. Managing resources
   {effective use of time and space}
   {share among multiple users fairly and without interference}
   - Processor (cores, time)
   - Memory (cache and RAM)
   - I/O devices
   - Internal devices (clock, timer, interrupt controllers...)
   - Persistent storage (HDD, SSD...)
2. Provide clean interface
   - Hide complexity of hardware (lower levels) from programs
3. What is the kernel of an operating system?
   - The part of the operating systems that is always in memory and implements the most commonly executed functions of the OS. The OS kernel executes in kernel or privileged mode and therefore has complete access to all hardware (in contrast to user-mode processes).
4. Kernels
   {User mode, kernel mode}
   1. Monolithic (single black box)
      - Advantages
        - Efficient calls within kernel
        - Easier to write kernel components due to shared memory
      - Disadvantages
        - Complex design with lots of interactions
        - No protection between kernel components
   2. Microkernels (as little as possible in kernel)
      - Advantages
        - Kernel itself not complex means its less error-prone
        - Servers have clean interfaces
        - Servers can crash & restart without bringing kernel down
      - Disadvantages
        - Overhead of IPC within kernel high
   3. Hybrid kernels (mix of both)

Processes

What is it

- An instance of a program being executed
- Provide the illusion of concurrency
- Provide isolation between programs

- Allows better utilization of resources

Types

- CPU-bound
  - Spend most of the time using cpu
- IO bound
  - Spend most of the time waiting for IO

Concurrency

- Pseudo concurrency
  - Single physical processor switching between processes interleaving
  - Gives the illusion of concurrency
- Real concurrency
  - Multiple Physical cores or cpus

- **Do not utilise cpu when waiting for IO**

CPU Utilisation = $1 - P^N$

- N = total number of processes
- P = fraction of time a process is waiting for I/O

Context Switch

- On a context switch the processor switched from executing process A to process B.
- OS may switch in response to an interrupt
- All information concerning process A is stored in order to be able to restart safely later.
- All information is stored in a PCB (process control block) stored in the process table.
- This is expensive
  - Direct cost: save/restore process state
  - Indirect Cost:
    - Memory cashes are usually flashed on context switch
    - (more in memory management)
- **Avoid unnecessary context switches**

PCB (process control block)

- Each process has its own **virtual CPU**, **address space (stack, heap, data)**, **open file descriptors**
- What should be stored?
  - PC, page table register, stack pointer...
  - PID (process ID), parent process, process group, priority, CPU used...
  - Root directory, working directory, open file descriptors.

Process Creation

- From:
  - System init

    - User request
    - System call from process
- Type:
    - Foreground processes: interact with user
    - Background processes: daemons

Process Termination

- Normal completion: done
- System call: `exit()`
- Abnormal exit: error / unhandled exception
- Aborted: killed by another process
- Never: will never end

Hierarchy

- In linux everything starts from `init`. There are child and parent processes and process groups.
    - `int fork(void)`
        - Created a child copy of the parent
        - In parent process: fork() returns process ID of child
        - In child process: fork() returns 0
        - -1 if fail
    - `int execve(...)`
        - Execute the program passed to this function
    - `int waitpid(int pid...)`
        - Suspend execution of calling process until process with PID pid terminates

Communication

Unix signals

Process can send signal to another process if it has permission to do so

SIGINT	Interrupt from keyboard
SIGABRT	Abort signal from abort
SIGFPE	Floating point exception
SIGKILL	Kill signal
SIGSEGV	Invalid memory reference
SIGPIPE	Broken pipe: write to pipe with no readers
SIGALRM	Timer signal from alarm
SIGTERM	Termination signal

- By default most signals will terminate the process but the process can choose to handle the signal however it wants.

- 2 signals (SIGKILL and SIGSTOP) cannot be handled

Unix pipes

A pipe is a method of connecting the standard output of one process to the standard input of another eg. `|`

- `int pipe(int fd[2])`
    - Returns two file descriptors in fd:
      `fd[0]` – the read end of the pipe
      `fd[1]` – the write end of the pipe
    - Sender should close the read end
    - Receiver should close the write end
    - If receiver reads from empty pipe, it blocks until data is written at the other end
    - If sender attempts to write to full pipe, it blocks until data is read at the other end

Persistent pipes that outlive process which created them eg. `>`

Threads

When multithreading is used, each process can contain one or more threads

Process items:

- Address Space
- Global Variables
- Open files
- Child processes
- Signals

Per Thread items:

- PC
- Registers
- Stack

Why use thread instead of processes?

- When execution in parallel in needed
- That has access to the same data
- Some might need to block
- Processes are hard to communicate between address spaces
- Processes that block may need to switch out entire application
- Processes are more expensive because of context switch
- Processes are more expensive to create/destroy
- But!
    - Shared address space(they may write on each others stack)
    - Concurrency bugs
    - Confusion with fork and signals

PThreads

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void * (*start_routine)(void*), void *arg)`
    - Takes thread pointer, parameters (NULL def), function to run, args of function (to pass more items, make a struct and pass pointer)
    - `0` is successfully created, error code otherwise
    - Will start execution once created
- `void pthread_exit(void *value_ptr)`
    - If called, will wait for all treads to terminate.
- `int pthread_yield(void)`
    - Release cpu to let another thread run
    - `0` on success
    - Always succeeds in linux
- `int pthread_join(pthread_t thread, void **value_ptr);`
    - Block until `thread` terminates
    - `value_ptr` stores return value

How to implement

- User-level threads
    - The kernel is not aware of threads
    - Each process manages its own threads
    - Threads implemented by software library
    - Process maintains **thread table** for **thread scheduling**
    - Advantages:
        - Thread creation and termination are fast
        - Thread switching is fast
        - Thread synchronisation (e.g. Joining other threads) is fast
        - All these operations do not require any kernel involvement
    - Disadvantages:
        - Block sys call stops all threads in process (non blocking IO can be used but harder to understand)
        - During page fault OS blocks whole process
- Kernel-level threads
    - Managed by the kernel
    - Advantages:
        - Blocking calls or page faults dont stop the whole process and kernel can schedule another thread
    - Disadvantages:
        - Creation/termination of threads is a bit more expensive (still better than processes) (Thread pools can make this even faster)
        - Thread sync is harder (requires sys call)
        - Thread switch requires sys call (still better than process switching though)
        - No application specific scheduler
- Hybrid Approaches
    - Use both

Scheduling

- New: the process is being created
- Ready: runnable & waiting for processor
- Running: executing on a processor
- Waiting/Blocked: waiting for an event
- Terminated: process is being deleted

Goals

- Ensure fairness
- Avoid indefinite postponement
- Enforce policy (priorities)
- Maximize resource utilisation (CPU, IO)
- Minimize overhead (context switches, scheduling decisions)

Depending on your system you might want different things. Ie jobs per minute (throughput) or minimize time per job (turnaround time) or time between request and response (response time crucial)

Types

Non-preemtive

{Let process run until it blocks or releases CPU}

- First come first served (FCFS)
  - Runnable processes added to the end of ready queue
  - Advantages:
    - Easy to implement
    - All processes are equally scheduled
  - Disadvantages
    - Everything depends on order of jobs. Can lead to unfortunate waiting times of large job is ahead of small jobs
- Shortest Job first (SJF)
  - Works best when we know all jobs in advance

Preemtive

{Stop it after max amount of time}
{Requires clock interrupt}

- Round Robin (RR)
  - Process runs until blocks or time quantum exceeded
  - Fair (equal cpu time for all)
  - Response time: Good for small number of jobs
  - Turnaround time:
    - Good when runtimes differ
    - Poor for similar runtimes

    - Quantum (time slice) (usually 10ms - 200ms)
        - Large quantum $\Rightarrow$ smaller overhead, worse response time
        - Small quantum $\Rightarrow$ larger overhead, better response time
- Shortest Remaining Time (SRT)
    - Like SJF but when new small jobs join they are pushed to front and current running is stopped

But how do you know estimated runtime from before?

- Based on history
    - Unreliable
- Based on user supplied estimate
    - What if it is wrong. Need to penalise process or else can be used to gain priority

Fair share.

Multiple users each have same cpu time

Priority Scheduling (can be both types)

- Always run job with higher priority
- Multy Feedback queues
    - Priority queue for each level and round robin on each level
- Increase job priority as it waits
- Change priority according to cpu usage etc.

Lottery

- Chance to run

Synchronisation (for both processes and threads)

- Critical sections
    - Section of code in which processes access a shared resource
- Mutual exclusion (mutex)
    - Ensures that if a process is executing its critical section, no other process can be executing it
    - Must request permission to enter critical sections
    - If no other process is in the critical section, then permission to a request must be allowed immediately
    - No request can be delayed forever
- Race conditions
    - These occurs when shared data is accessed by multiple processes and the final results depends on timing during execution
- Starvation
- Happens-Before Relationship
    - Consider a, b with a occurring before b in the trace
        - If a, b are in the same thread, then $a \Rightarrow b$
        - Irreflexive: $\forall a, a \not\Rightarrow a$

        - Antisymmetric: $\forall a, b: a \Rightarrow b\ then\ b \Rightarrow a$
        - Transitive: $\forall a, b, c: a \Rightarrow b \land b \Rightarrow c\ then\ a \Rightarrow c$

Synchronisation mechanisms

{required at the entry and exit of the critical section}

Disabling interrupts (bad)

- Only works in single processor systems, and in kernel mode
- Should be avoided if possible

Busy waiting (bad)

- Caused when CPU is constantly checking a value of a variable
- Wasted CPU time
- Should only be used when wait is expected to be short (or just not used)

Atomic operations

- Sequence of statements that are indivisible

Locks

Granularity

- Try to lock as close to and as small regions as required
    - Think about **overhead**, **contention** and **complexity** though
- Coarse $\Rightarrow$ one lock for everything (low granularity)
- Fine $\Rightarrow$ lock a very small region (high granularity)

Overhead

Memory, initialization and time required to lock/release all have a cost that should be taken into consideration

Contention

- Number of processes waiting for a lock
- More $\Rightarrow$ less parallelism
- To minimize, use fine granularity and release lock as soon as it's not needed

Examples

- TSL(LOCK) (test ans set lock) aka spin locks
    - Provided by most CPUs
    - Lock by busy waiting
    - Should only be used when the wait is expected to be short
    - **Priority inversion problem**
        - Problem when low priority process holds lock but high priority process is now running

  -
    ```
    void lock(int L) {
      while (TSL(L) != 0) /* wait */ ;
    }
    ```

- Read/Write Locks
  - Lock_RD(L) $\Rightarrow$ acquire L in read mode
  - Lock_WR(L) $\Rightarrow$ acquire L in write mode
  - In write mode, the thread has exclusive access
  - Multiple threads can acquire the lock in read mode

Semaphores

They are special variables accessible by the following atomic operations:

- `down(s)` or `P(s)`: receive a signal via semaphore `s`
  - A process will stop, waiting for a specific signal
  - A process will continue if it has received a specific signal
- `up(s)` or `V(s)`: transmit a signal via semaphore `s`
  - Send a signal 'done' so another process can gain access
- `init(s, i)`: initialise semaphore s with value `i`
  - `i` indicates how many processes can access at the same time

Semaphores have two private components:

- A counter (non-negative integer)
- A queue of processes currently waiting for that semaphore

```
void init(s, i) {
  counter(s) = i;
  queue(s) = {};
}
void down(s) {
  if (counter(s) > 0) {
    counter(s) = counter(s) - 1
  } else {
    add P to queue(s)
    suspend current process P
  }
}
void up(s) {
  if (queue(s) not empty) {
    resume one process in queue(s)
  } else {
    counter(s) = counter(s) + 1
  }
}
```

Monitors

Keep critical region within monitor and use semaphores and locks to keep it safe as well as conditional variables to signal between processes. Only one process can access monitor at a time

Deadlocks

Set of processes is **deadlocked** if each process is **waiting for an event** that only **another process** can cause.

Deadlock Conditions

- **Mutual exclusion**: only one process can access each resource at a time
- **Hold and wait**: process can request a resource while it holds other resources
- **No preemption**: resources can't be forcibly revoked
- **Circular wait**: two or more processes are in a circular chain waiting for a resource held by the next process

Prevent Deadlocks

1. Ignore it (no)
2. Detection & recovery (after deadlock detect and fix)
    1. Dynamically build a resource ownership graph **(Resource allocation graphs)** and look for cycles
    2. If deadlock detected:
        - **Preemption**: Temporarily take resource from owner and give it to another
        - **Rollback**: processes are periodically checkpointed (memory images). On deadlock roll back (but how far?)
        - **Killing process**: Select random process and kill it. (but what if job is important)
3. Dynamic avoidance (consider every request and decide if it is safe to grant it)
    - Banker’s Algorithm: Safe vs. Unsafe States
        - **Safe state**: there are there enough resources to satisfy any (maximum) request from some customer
        - Unsafe state dose not guaranty deadlock but safe does guaranty NO deadlock
4. Prevention (ensure at least one if the conditions never holds)
    - **Mutual exclusion**: share the resource
    - **Hold and wait**: Require to request all resources in advance. (what if you dont know what you need)
    - **No preemption**: (not good)
    - **Circular wait**: one resource pre process OR ask for resources in the same order.
5. Communication deadlocks: Need to consider miscommunication
6. Licelock: overall system makes not progress but no 'deadlock' possibly because of try to lock, retry on fail

Memory management

- **Register** access is 1 CPU clock cycle or less
- **Main memory** can take many cycles
- **Cashes** are between main memory and CPU registers

Basic concepts

Memory management binds **logical** address space to **physical** address space

- Logical address
  - Generated by the CPU
  - Address space seen by process
- Physical address
  - Address seen by the memory unit
  - Refers to physical system memory

Memory Management Unit (MMU): Hardware device (fast) for mapping logical to physical addresses

Memory allocation

Main memory is split into 2 partitions:

- Kernel memory (OS) (low memory)
- User processes (high memory)

Contiguous Memory Allocation:

Relocation Registers:

- Base: Contains the smallest physical addedd of the process
- Limit: Contains the range of logical addresses
- $Base \Rightarrow lower\ limit$
- $Base + Limit \Rightarrow upper\ limit$
- Will be checked by MMU for **memory security**

Multiple partition allocation:

- OS keeps track of:
  - Allocated partitions
  - Free partitions (Holes)
    - When new process arrive, allocated memory will be selected from holes that are large enough

Dynamic Storage Allocation

- First-fit: Allocate first hole that is big enough
- Best-fit: Allocate smallest hole that is big enough
  - Must search entire list, unless ordered by size
  - Produces smallest leftover hole
- Worst-fit: Allocate largest hole (not as good)

    - Must also search entire list
    - Produces largest leftover hole

- **External Fragmentation**: Total memory exists to satisfy request but not contiguous.
- **Internal Fragmentation**: allocated memory larger than requested memory. Unused space within partitions
- Fix by **Compaction** (put all partitions in one big block next to each other) but leads to I/O bottlenecks (slow)

Swapping

- When memory full, Swap out not running process memory to disk.
- Swap back in when that process is ready to run.
- This requires swap space. Can be a file or a dedicated partition.
- **Warning!** Large transfer time.

Virtual Memory

Paging

Allows for logical address space of process to be non continuous

Frames:

- Fixed size block of **physical memory**
- (Need to) keep track of free frames **Free Frame List**

Pages:

- Fixes size block of **Logical memory**

Page table: maps pages to frames

Smaller page sizes $\Rightarrow$ **less internal fragmentation** and therefore **more efficient memory use**.
Larger page sizes $\Rightarrow$ **less overhead** for address translation and therefore **faster memory access**.

Address Translation:

Page Number (p): is page index in page table
Page Offset (d): is the offset within the page you want

```
logical address = p + d
f = pageTable(p)
physical address = f + d
```

(remember big endian is normal, little endian is reverse)

Memory protection:

Each page table entry has a valid/invalid bit.
If invalid, page might be swapped out or some other error

Page Table Implementation:

- Page-table base register (PTBR) points to page table
- Page-table length register (PRLR) indicates size

This is slow though so use **Cache for page table**
Associative Memory:
If p is in associative register get frame from there,
otherwise fall back to page table in memory
Translation Look-aside Buffers (TLBs):

- Store address space ids (ASIDs) in entries
- Need to be cleared in context switch
- Performance: Effective Access Time
    - Associative Lookup $= \epsilon$
    - Assume memory cycle time $= m$
    - Hit ratio $= \alpha$
    - Depth $= d$
    - $EAT = (\epsilon + m)\ \alpha + (\epsilon + (d+1)m)(1 - \alpha)$

Page Table Types

- Hierarchical
    - Page number divided in 2
    - Eg. If p -> 22 bits, it can be split up into
        - P1 -> 12 bits to index outer page table
        - P2 -> 10 bits to index inner page table
        - D -> 10
    - Or 3 level paging with:
        - P1 -> 32
        - P2 -> 10
        - P3 -> 10
        - D -> 12
    - But this page table is huge for 64 bit machine
- Hashed
    - Use hash function to hash p. Access hash table to find list of possible entries and search them to find f
    - This decreases search time
- Inverted
    - Store pid and p in page table only. Use the **index** of the entry as f
    - This decreases memory needed to store page table but increases search time
- Hash and Inverted together gives best outcome!

Sharing memory

After shared memory is established, no need for kernel involvement

Compared to pipes:

- Good: Better performance because no kernel and better bidirectional communication

- Bad: no synchronisation provided, so not as good for unidirectional communication

Demand Paging

- Bring page into memory only when needed
- Use valid bit to see if page is in memory or not
- When invalid will cause page fault
- Then kernel will handle loading required page
    - Get empty frame
    - Swap page into frame
    - Reset tables valid bit to 1
    - Restart last instruction
- Page fault rate
    - $0 \leq p \leq 1.0$
    - $p = 0$, no page faults
    - $p = 1$, every reference is a page fault
    - $EAT = (1 - p)\times EAT\ of\ memory\ access + p \times (page\ fault\ overhead + [swap\ page\ out] + swap\ page\ in + restart\ overhead)$
- Some Tricks
    - Copy-on-Write (COW)
        - For processes with children, use same page until one of them modifies it, and only then copy to new page
    - Memory Mapped files

Page replacement

- Find unused page in memory to swap out

Algorithms:

- First-In-First-Out (FIFO)
    - Replace oldest page
    - May replace heavily used page
- Least Recently Used (LRU) (optimal)
    - When paged referenced, copy clock into page counter
    - When page needs to be replaces, choose lowest counter
- LRU is expensive, so we use approximation
    - Second Chance
        - Go round in circles
        - Use Reference bit r
            - When page is referenced set to 1

        - Periodically set all to 0
    - When try to replace if r = 1 set to 0, else replace it it
- LFU (least frequently used)
    - Replace page with smallest count
    - May replace page just brought into memory
    - Uses reset counter or ageing to forget
- MFU (most frequently used)
    - Replace page with largest count
- Consider page fault frequency and allow more/less pages to be allocated if faults are high/low

Device management

- Device type: disk, dvd drive...
- Device instance: which disk
-

IO Levels

- **Interrupt handler (lowest level)**
    - Process each interrupt
    - For block devices:
        - on transfer completion, signal device handler
    - For character devices
        - when character transferred, process next character
- **Device handler/driver (above ^)**
    - Handles one device type
        - but may control multiple devices of same type
    - Implements block read or write
    - Access device registers
    - Initiate operations
    - Schedule requests
    - Handle errors
- **Device Independent OS Layer (above ^)**
  Device independent layer provides device independence
    - Mapping logical to physical devices (naming and switching)
    - Request validation against device characteristics
    - Allocation of dedicated devices
    - Protection/user access validation
    - Buffering for performance and block size independence
    - Error reporting
- **User-Level IO software**

Device allocation

- Dedicated
    - Simple:

        - Open fails if already open
        - Queue open requests
    - Allocates for long periods
    - Only allocated to authorized processes
- Shared (disks...)
    - OS provides file system for disks
- Spooling
    - instead of blocking user access to devices, save to disk instead
    - eg. printers
        - printer output saved to disk file
        - file will slowly be printed by spooler daemon
    - provides sharing of non shareable devices
    - reduce IO time

Buffered IO

- used to cater for differences in transfer rate between devices

Unbuffered IO

- Data transfer directly from user space to/from device
- High process switching overhead

User interface

- `open`, `close`, `read`, `write`, `seek`
- Unix access virtual devices as files

Memory mapped IO

Remember raspberry pi. Led on/off was controlled by memory address

Blocking I/O

- I/O call returns when operation completed
- Process suspended $\Rightarrow$ I/O appears “instantaneous”
- Easy to understand but leads to multi-threaded code

Non-blocking I/O

- I/O call returns as much as available (e.g. `read` with 0 bytes)
- Turn on for file descriptor using `fcntl` system call
- Provides application-level polling for I/O

Asynchronous I/O

- Process executes in parallel with I/O operation
    - No blocking in interface procedure
- I/O subsystems notifies process upon completion

    - Callback function, process signal, ...
- Supports check/wait if I/O operation completed
- Very flexible and efficient
- Harder to use and potentially less secure

Disk management

Hard disk

- Disk Storage Devices
    - Cylinders
        - Zone
            - Sectors
                - Tracks
            - Sector gap
            - Inner track gap
        - Ensures that sectors have same physical length
        - Zones hidden using virtual geometry
        - Logical sector addressing (or logical block addressing LBA)
            - Numbering 0..n
            - Easier
- Formatting
    - Low level format
        - disk sector layout
        - cylinder skew
    - High level format
        - boot block
        - root directory
        - empty file system

Performance

- Seek time: $t_\text{seek}$
- Latency time (rotational delay): $t_\text{latency} = {1 \over 2 r}$
- Transfer time: $t_\text{transfer} = {b \over rN}$
- where
    - b - number of bytes to be transferred
    - N - number of bytes per track
    - r - rotation speed in revolutions per second
- Total access time: $t_\text{access} = t_\text{seek} + t_\text{latency} + t_\text{transfer}$

Disk Scheduling:

- First Come First Served (FCFS)
    - No ordering of requests
    - random seek pattern
- Shortest Seek TIme First (SSTF)

    - Order requests according to shortest seek distance from current head position
- SCAN (most common)
    - Only change direction when reaching outermost/innermost cylinder
    - long delays for extreme locations
- C-SCAN
    - only service requests in one directions
    - when head reaches innermost request, jump to outermost
- N-Step SCAN
    - same as scan but only serves requests waiting when sweep begins

SSD

- More Bandwidth
- smaller latencies
- More expensive
    - Less GB/$

RAID (Redundant array of inexpensive disks)

- array of physical devices appearing as single virtual device
- striping: store data distributed over array
- more disks -> lower time to failure (MTTF)
- Level 0
    - no fault tolerance
    - balance load over many discs
- Level 1
    - mirror data across disks
    - read faster as can be read by any disk
    - write slower as must be written to both disks
    - more cost
    - easy recovery
- Level 2
    - Parallel access by stripping at bit level
    - use hamming error correction code ECC
    - corrects single but errors
    - vary high throughput but no concurrency
    - Only used if high error expected
- Level 3 (Byte Level XOR)
    - parity = data1 XOR data2 XOR data3....
    - reconstruct missing data from parity
    - lower storage overhead that 2 but still no concurrency
    - cheaper
- Level 4
    - Parity like 3 but with block level
- Level 5
    - same as 4 but parity is kept in multiple disks

Category	Level	Description	Data Transfer read	Data Transfer write	Request Rate read	Request Rate write
Striping	0	Non-redundant	better	better	better	better
Mirroring	1	Mirrored	better	same	better	same
Parallel Access	2	Redundant via Hamming code	much better	much better	same	same
Parallel Access	3	Bit interleaved parity	much better	much better	same	same
Independent access	4	Block interleaved parity	better	worse	better	worse
Independent access	5	Block interleaved distributed parity	better	worse	better	worse or same

Disk caching

Improve disc access by storing some disk sectors in main memory (buffer)

- need replacement policy when buffer full
  - Least Recently used (LRU)
    - Stack of blocks (remove bottom when full, fill from top)
    - problem: doesn't care about how much block is used.
  - Least Frequently Used (LFU)
    - replace block that has experienced fewest references
    - counter associated with each block
    - block with smallest count is replaces
  - Frequency based replacement
  - Divide LRU stack into two sections: new and old
    - Block referenced $\Rightarrow$ move to top of stack
    - Only increment reference count if not already in new
    - (to decrease fast ageing, may use 3 sections (middle))

File systems

Security