Lecture 01

1. Systems / Operating Systems
   - mechanic of the computer science/engineering world
   - the under-the-hood domain
   - fix a printer, design an engine, fix an engine/axle... build better cars/tyres!

2. What is an operating system?
   - show to manage computer hardware
   - provide an environment for execution of programs
   - show layers to help users build applications
   - s/w to provide an usable computing system
3. Why OS?
- the interface to communicate/program the hardware is not scalable, especially for general purpose programs.

- still we keep programs (easily) if — how?
- ans: abstractions, concepts of
- An abstraction is a functionality implemented by a layer, other "layers"
  assume functionality and build on top of it.

- OS abstractions:
  - files (tracks, sectors, storage IO)
  - Network interface (wired, wireless, DMA)
  - (virtual) address space
  - CPU ownership

- the OS exposes/hosts several common functionalities related to IO & resources
  via abstractions & accompanying interfaces.

- required to control the system
  two main tasks,
  (i) resource allocation/multiplexing
  (ii) isolation & control.

- OS owns all resources.

- the Software (basic) Stack.

```
+--------------------------+
| Applications             |
| API                     |
| Libraries               |
| OS                      |
| ISA                     |
| CPU - MEM - IO          |
```

users
(i) **Monolithic vs. Microkernel vs. Unikernel**
- **(all-in-one)** vs. **(modular)** (app-to-OS vs. rolled up)

(ii) **Server** vs. **Desktop**
- Multi-user domain vs. Single-user ease of use

- **Handhelds** vs. **OS w/ interventions usability**

- **Vehicles/aeroplanes** vs. **real-time requirements**

---

**Two Key Building Blocks of OS**
- **Computer System Organization**
  - **CPU**
  - **Memory**
  - **Disks**
    - **Controller**
    - **USB Controller**
    - **NAT adapter**

- **Parallel execution**
- **Shared memory**
  - **Computing for cycles**
  - Setup via CPU may be required
  - Device driver per device controller
  - **Load OS & programs stored on disk**
  - CPU fetches instructions from memory (pointed by IP) to an (instruction) register for decode + execution
* Boot / Startup process (details later):

- Initial / boot program is stored within the flash ROM or EEPROM based.
- Inspects, initializes (resets) all devices / registers.
- Loads OS (first few regions) in memory.
- Jumps to location & starts execution & OS.

# Two key building blocks for OSes
(i) interrupts (interrupt-driven operations).
(ii) privileged modes of execution (not supported)

(i) Interrupts

- All device IO is interrupt-driven.
- Trap, exceptions.
- Pkt. arrival from disk.
- Kbd. press.

- OS pauses current execution thread / process and services interrupt (ISR)
privileged instructions

- instructions that require kernel mode of operation for correct effect.
- if insufficient privileges:
  - noop
  - or exception
  - e.g., timer duration for interrupt instruction

- setup of memory per process
- memory read permissions
- scheduler for 10 requests

privileged levels for instruction are now supported,
each instruction has a minimum privilege level for execution.

(i) interrupts (HW).
(ii) exceptions (SW) — page fault, divide by zero.
(iii) explicit calls (system calls), in SW interrupt.
also the OS interface for functionality.

e.g., on timer interrupt, OS decides — which process to schedule next.
on page fault decides which memory page to access!
Lecture 02:

Summary:
- Isolation & resource control are key OS functions.
- Privileged modes & interrupt-driven execution.
- Abstractions & interfaces.

OS components:

User interface

System calls
- System interface
- User interface
- Program execution
- I/O operations
- File I/O / disk I/O
- Communication (IPC)
- Error detection
- Resource allocation
- User mgmt. & accounting
- Protection & security

System call categories:
1. Process control
2. File management
3. Device management
4. Information / state maintenance
5. Protection & security
6. Communications
Process management:
- A process is a program in execution.
- process, job, thread, task
- what is a process?
  A process is an instance of a program.
- Can have several instance of same program.
- An instance implies a program loaded/copied in memory & setup for execution.

What else is required for program execution other than program code/ binary?
- text section — the executable
- program counter — process activity
- processors regs.
- stack for FV calling
- heap for dynamic memory allocation
- data section for static allocation

All of this is in memory?

What would be layout for processes based on the same program?

Process state:
- Are all processes always waiting, executing?
  NO!
- Why?
  - waiting for their turn
  - cannot execute till condition
    (waiting for disk read)
  - job done, waiting for cleanup
  - process being setup, not ready to execute yet

Create
Waiting
Emit/Kill/Error
Ready
Running
Terminated
New

Event
Done

Ready
scheduled

descheduled
- how does OS figure in this? (state).
- Each process needs a representation in the OS.
  This state is used by OS to store state.
  Schedule processes, allocate memory (per process).
  Set/make map/manager.

- PCB: process control block, task_struct (in Linux).
  This is the per-process state representation of the OS.

- Typical sub-fields of the PCB:
  - state
  - program counter, CPU regs
  - scheduling information
  - memory mgmt. information
  - file id, status info
  - accounting info
  - locks, lookup task-struct.

- Note that each PCB (typically a C's struct) consumes memory. This is not process-demanded memory on its local variables, it is memory required by the OS to maintain its own state to run the show.

- Mgmt. Of kernel state / memory will be covered later.

- Why does the executable program of one OS not work on the other?
- Labs in C.
  - shell, related, exit, kill
  - dup & pipes, tee
  - fd, 0, 1, 2

# why two calls fork and exec?
- duplicates a process
- replaces a process
- process state
- can be customized.

fd, memory, pwd, signals, areas, process groups, resource limits.

- state/action: process mgmt.
  (i) what is a process? - PC, registers, memory region, open files
  (ii) PCB - process control block

C meta-data state of a process
- PC, registers, state
- memory description
- file usage description /10 info.
- signals/event info.
- accounting & usage

⇒ whatever is the current PCB for
  execution is usually the process context.

⇒ Do both interrupts and system calls have a process context?

⇒ context is process-centric information!
(iii) fork + exec.  
- duplicate 
- replace. 

- copy pcb 
- give new id, set parent info. 
- set share memory region and mark it copy-on-write.

Two modes of possible operations:

(i) child is same as parent.
(ii) child is different from parent.

Why copy-on-write required?

Q on fork guest & child execute before parent, good strategy?

Q process states:
- running, blocked/waiting
- ready
- terminated

# schedule only considers 'ready' processes.
- w/ multiple processors, one strategy is
to provide one ready processes queue per cpu.

Q is everything scheduled a user-level process?

No! kernel thread/processes
- non-user, non-10 initiated
- kernel work
- ps -ef

40
system calls implement process context

- why process context matters?

- Linux electrons task struct

- fork + exec: summary + memory state + memory context update

- zombie processes

- bash process context

- signals

how does OS s/w process to handle signal?

- s/w from kernel to user mode

check it signal pending, handle signal first

"highest" priority work of a process?

software mechanism for urgent work/events to be handled

SIGINT - interrupt process CRTL+C

SIGKILL - kill -9

SIGTSTP - CRTL + 2 - suspend

SIGCHLD - child terminals

SIGALRM - time out/ alarm goes off

SIGFPE

SIGSEGV

load balancing memory limit be beyond swapping

process reaping cpu
**signals**

- A signal is used to notify a process that an event has occurred.

  Mostly always related to execution, divide by zero, memory overflow, access etc.

  Kill process.

**basic loop**

(i) Signal generated based on event. **Kernel**

(ii) Signal delivered to desired process

(iii) Signal is handled. **User + Kernel**


Synchronous vs asynchronous

Delivered to process that caused event. Next step is to handle signal.

- 
  - aysch. recipt of signals generated by a process, delivered to another process (IPC!)
  - SIGKILL, SIGINT, SIGTERM

**Recap:**
- signals, files, stdout, stdin
- redirection
- scheduling & sign.

**2.6.17: mmm... memory!**

- CPU uses memory to fetch and execute instructions, and to load from & store to memory.

  The basic fetch-decode-execute cycle.

- CPU exec rate much faster than memory bus access and fetch of instruction.
  - CPU-based caching (L1, L2, L3 caches).
  - Pipelining

  \[ \text{Hw sub-component of cpu deals with memory. } \Rightarrow \text{ no os intervention. } \text{the MMU.} \]
Design 0
- Similar to PCB, maintain and enforce all memory partitioning and access permissions.
- Why not feasible?
  - Disastrous on the performance trajectory.
  - On each memory access (by CPU) OS will have to intervene.
    - CPU \rightarrow Memory bus \rightarrow OS processing based fetch check.
    - Many many memory accesses.

Design 1:
- Use hardware registers that are used by MMU to perform checks.
- Base registers: stores per process start of memory region
  - Limit register: stores limit/endpoint of region.
- OS performs store-and-restore of values in registers as part of process context switch. Stores in PCB.
MMU uses LR & BR values to check validity of each address in hardware. No OS intervention required after process is setup for execution.

(cpu) address → base? → base + limit? (mem) y

Limitations?
- related to address binding & fragmentation

What is address binding?
- how does CPU generate an address for access (data)?

Example: int main ()
{
    int a;
    what is the address of variable 'a' for the CPU?
}

Compile-time binding:
- compiler knows where programs are loaded.
- can calculate actual address.

Load-time binding:
- relative addresses in compiled code.
- during load initialize actual addresses.
- compile time: module start | load time + x | start = addr.

Run-time / execution time
- segment regions of program move around in memory.
- need binding at runtime!
**Design 2**

(1) Introduce the notion of **logical/virtual addresses**, different from **physical addresses**.

- **Logical/virtual address space**
  - Space of **logical addresses**
  - Issued by CPU
  - Consistent compilation/linking process, no address binding headache!

---

**Diagram**

- CPU -> MMU
- MMU: the Hyper Unit, use the V2P
- MMU: per process information (relocation register)
- Logical/virtual address to generate
  - the physical address

- Physical address

---

**Diagram**

- Logical address
  - Limit
  - Realloc
  - Physical address

- MMU

---

**Question**

- How helps execution time binding of addresses?

- Memory info PCB holds what? RL value!
- LR value!
why does granularity at allocation matter?
- fragmentation
  - external
  - internal
- memory allocated to process never used.
  - e.g.: allocate memory in multiples of 1 MB.
- not continuous hence cannot load program.
- solution: compaction
  - kernel work swap
  - swamping
  - process request

Design #3 Segmentation.

Similar to design #2, but breaks up memory regions similar to programmer's view.
- code
- stack
- data
- extended

Logical address: <segment, offset>
number.

Segment table

Segment

choice of segment is decided by CPU
based on function/instruction being executed.

Compile-time binding and
load-time binding
logical address = physical address.

Execution-time/ run-time binding,
logical address ≠ physical address.
Design 2/3

Also solves contiguous allocation issue.

Pros:
- Ensures loader/linker orthogonality.
- Consistent address ranges for programmers.
- Fragmentation ≤ contiguous physical allocation.
  - Dynamic multiplexing/granularity
  - Dynamic loading/allocations.
- Memory efficiency (sharing of libraries).

# Paging: Design 4

Change granularity & allocation.

Program → Segment → Page/Frame.

Logical address → Physical address
- Logical page
  - Physical frame.

Logical address

Physical address

Extent of fragmentation goes down.
Enables lazy/on-demand physical allocation efficient.
The paging mechanisms

VA: \[ \text{page no.}, \text{page offset.} \]

\[ p \]

at \[ \text{offset} \]

\[ \text{page table} \]

(set of page frames allocated)

\[ \text{page frame} \]

(physical)

Q: Who does the VA to PA translation?

-MMU- in hw.
Design 0:
- s/w enforced memory mgmt
- on each memory access, invoke OS to perform checks & address mapping
- s/w way slower than h/w.

Design 1:
- h/w assisted mapping
- each process has a contiguous region of allocated memory & uses base & limit regs for mapping
- CP11 addr. & mapped addr. both physical
- faster than & design 0
- but
- relocation & swapping is difficult
- needs change to program code
- fragmentation
Design 2:
- separate programmer view of address and as view of addresses.
- program need not worry about physical address mgmt. - decoupling!

Design 3: Segmentation
- programmer's view of logical address can be broken into logical segments.
  - e.g.: code, data, stack, heap.
  - non-contiguous alloc.
  - each segment is relocatable.
  - fine granularity for swapping.

- each VA:

<table>
<thead>
<tr>
<th>seg id</th>
<th>offset</th>
</tr>
</thead>
</table>

- relocate memory regions by updating base register
- VA's remain same
- swapping support better
- segmentation still on!
- granularity still a problem
Example: 4 sections & 8 bit address space.

- relocatable segments = non-contiguous alloc.
- segments can still be large!

Design 4: Paging.
- change granularity radically.
- each allocation is in terms of fixed-size memory region page/frame.
- use multiple & these for memory mgmt.

- fine grained control for sharing is easy vs.
- less manageable.

On swap updated segment table.
- Allocate page table pages only when needed!
- Do not want contiguous allocation of page table in memory!

1A-32 e.g.

- VA: page number
- Offset:

Page size = 4 bytes
32-byte mem.

VA

frame

0 1 2 3
4 5 6 7

pg#0

pg#1

pg table

32-bits

4KB

4KB

4KB

12

10

10

32-bits

byte

minest

32-bits

2^{12} 

1024

2^{10} 

1024

2^{12} 

4096

2^{12} 

1024 

1024 

4,096

4GB VA space of each process can be mapped!
1. Test
   - 5 sec
   - 1 syscall

Disadvantages:
- Segmentation / single-level paging
  - Build up the page table
  - Segment/page tables are linearly-indexed and will need large allocations (per process).
  - One page can hold 1024 (4by2) PTE entries covering 4MB of addressable space for 4G bytes, will need 1024 pages ~ 4MB per process!
  - Processes may be sparse/scattered memory
- Linear indexing not required!
**Hardware support for v2p:**

1. **MMU** - memory management unit for hardware-assisted translations.
2. **TLB** - translation lookaside buffer.
   - Virtual address (page) to physical address (frame) caching.
   - 32-1024 entries
3. **pte flags** - protection bits.

**Swapping:**

- Temporarily move/swap-out content of memory of a process to a disk.
  - 'backing store'
  - Swap-in contents from disk to memory on access, process activity.

**Questions:**

- Granularity of swapping?
- What to swap/exist?
- When to swap-out?

**Note:** pte # process is valid!
# Paging example

VA1: 0000 0000 1000 0000
VA2: 0000 0000 1000 0000

VA1: 0080 2024 h

VA2: C0 2024 h

Offset

(pae-p + Offset) 8 page

pgdir

Bits = 3

0000 0000 1000 0000 0000 0000 0010 0010 0010 0010

Bits = 3

0000 0000 1000 0000 0010 0010 0010 0010 0010 0010
# Inverted page tables.

- V2P tables single level or multi levels are per-process and can be memory consuming.

- `p2v`: inversion!
- search P space to look for matching

- Linear IPT, hashed IPT.

# Single-level table will need

$$\frac{252}{2^{12}} \approx 20 \text{ pages } \approx 2^{12} \text{ bytes}.$$ 

52-bit index acts as an offset!

# Multi-level tables.

- Each page stores 1024 entries or 10 bits
- VA space

$$\left[ \frac{52}{10} \right] = 6$$

6-level page table will get us one page per page table entry!
What every user programmer would like about memory (the resource)?
- Private (isolated), infinitely large, fast, non-volatile, inexpensive.

Not all requirements met simultaneously.

Memory hierarchy:

CPU \[\rightarrow\] Memory \[\rightarrow\] RAM

Disks

OS memory managers manage the memory hierarchy, which are in use allocate de-allocation etc.

No memory abstraction
- Each process address directly refers and accesses physical address — real mode!

Yes:
- Multiplexing processes not easy.
- Relocation needs loaders to reset addresses.
- Process-level swapping (frequent).
- Performance & loader needs compiler & link support — slow &

Virtual memory abstraction:
- Decouple processes view of memory from OS's hand of memory (physical-memory).
- Process-level operations are in VA space.
  - No loader over loading. Relocation easy.
  - Still swapping req'd.
  - Granularity of allocation.
  - V2P req'd of actual access. Protected mode.
- design 0: no abstraction
- design 0: no abstraction + relocation
- design 1: no abstraction + base + limit + registers
- design 2: no process-level swapping
- design 3: VA abstraction + base+limit
- design 3: VA abstraction + segmentation
- design 4: VA abstraction + paging

# advantages & paging
- memory granularity is one page for management decisions (allocate/deallocate)
- page allocation/demand paging
- page-level protection, sharing

# VA

\[
\begin{align*}
VA_{11} & \rightarrow P_1 \leftarrow VA_{31} \\
VA_{12} & \rightarrow P_2 \leftarrow VA_{22} \\
VA_{13} & \rightarrow P_3
\end{align*}
\]

VA on virtual address space is occupied, but not PA/page is mapped.

\[
\text{page fault handler sets up mapping & re-executes the memory access instruction.}
\]
32-bit address space

12-bit offset

1024 entries

4-byte lines each

12

2 x 2 x 2

= \(2^{12} = 4096\)

Note: page table structure will change

- page size, page table page offsets etc.

HW

1. 32-bit address space with 4 MB pages.

2. 64-bit address space w/2 MB pages.

Quiz questions:

- # pages of page table on single mem. access.

- 2 pages

- # pages of page table on entire VA accessed.

\((2^{10} + 1)\) pages.

\((4 \text{ MB} + 4 \text{ KB})\)

64-bit address space, actually on 48-bits used.

512 entries

8 64-bit each

9 x 9 x 9 x 2

= \(2^{48}\) addressable by teas!
# TLB: Translation Lookaside Buffer
- CPU component (no bus access reqd).
- Goal is to cache v2p mappings.
- Avoid page table walk overheads!
- Limited size: 256 to 1000s entries.
- Are all TLB entries always valid?
- When not?
- Mechanism for correctness:
  - Invalidate, flush entries.

# Page replacement policies
- Evict pages on memory pressure.
  - FIFO
  - LRU
  - LFU
  - MRU
  - LRU(2)
  - Clock

123 123 123 123 123
FIFO
LRU
LFU
MRU
LRU(2)

Optimal:

Cache size = 1
FIFO
LRU
LFU
MRU
LRU(2)

Miss rate: 100%
Miss rate: 66%

Cache size = 2
FIFO
LRU
LFU

2/12
6/12
6/12
1/12
**Segment registers**

<table>
<thead>
<tr>
<th>Table index</th>
<th>Global</th>
<th>Local</th>
<th>Ring</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 bits</td>
<td>1 bit</td>
<td>2 bits</td>
<td></td>
</tr>
</tbody>
</table>

* Real mode:
- During boot up, bootloaders load the kernel in PA starting from $0x00000000$.
- But kernel compiled to run with memory addresses in VA from KERN BASE.

So:
- Set segment start / base address to $-KERN BASE$.

* Code: $SEG(STA_X, STA_R, -KERN BASE, 0x00000000)$.
* Data: $SEG(STA_X, STA_W, -KERN BASE, 00000000)$.  
  - jmp $0x0f000000$.
  - jmp $0x00100000$.
in protected mode?

- set base to 0x0000 0000

limit to on/off on/off.

flat mode.

segment-based translation is a no-op.

flat mode.

but still can use to set privilege levels & read levels & permissions for segments, to be checked during execution.

* with x86 segmentation cannot be turned x86 64-bit in flat.

paging can be:

bit 0

2/31
VA: 24 bits
PA = 16 MB

12
12
2
2

entries per page

2
2

$\frac{12}{2} = 2^{10}$

$2^{12} \sim 2^{10}$

$\frac{1024}{2^7} = \frac{4 \times 1024}{3} \sim 1365$

$\frac{1024}{2^7} = 4 	imes 1024 \approx \frac{3}{1365}$

$2^{12}$

$2^{12} / 3$

$2^{12} / 2^2 = 2^{10}$

$2^{12} / 2^2 = 2^{10}$

$2^{12} / 2^2 = 2^{10}$

# PTE in 1st level = $2^{12}$

Main entry in a PT = $2^{12} / 3$

but we saw only 2^10 entries.
Belady's anomaly: \text{Thrashing.}

Memory mgmt:
- \( v \& p \) decoupling
  - \( v2p \) mapping
- swapping
- page fault handling + demand paging
  - lazy allocation
- page replacement eviction strategies

\( \Rightarrow \) eviction of which pages?
- disk-backed pages
  - less useful pages
- anonymous
  - stack
  - page table pages
  - kernel state (heap!)
- file-backed memory
  - code section
  - disk/page cache.

Hit ratio

Page faults

Replacement & swapping

\[
\text{avg. latency} = \text{hit. rate} \times L_h + (1 - \text{hit. rate}) \times L_m
\]
# frames = 3.
Pts. on access: 1, 2, 3, 4, 5, 1, 2, 5.

\[ * \text{frames} = 4. \]

- (ii) Optimal: Reference string known.
  Can calculate what to add to page & not approx.

- (iii) LRU: Least recently used.
  - Select page that is "oldest".
  - Needs vast support.

- Intercept each memory access, slow!
(iv) Second chance algorithm / clock.

- check reference bit of a page.
  - if '0' - page is victim.
  - if '1' - set to '0';
  - consider next page.

Can consider ref. & modify into as well.

R, M:
0, 0 — best
0, 1 — not recently accessed but modified
1, 0 — accessed, page is clean
1, 1 — accessed and written, hot!

(v) LFU — least frequently used.

MFU — most frequently used.

Working set size

Thrashing:

not enough frames

swapping:

all pages "active" (in WS)

CPU

Load

how much memory should a process get?

No useful work!
1. Kernel synchronization goals.
3. When in locking need?
   - Seams & execution threads and examples.
4. Synchronization terminology:
   - Mutual exclusion, critical sections.
5. Techniques:
   - Locks, mutexes, semaphores.
6. Spinlocks:
   - Impl #1 & drawbacks.
   - Requirements.
   - Impl #2 — TSL
   - Impl #3 — lock copies

* Mutexes/semaphores:
  - Impl details.

* Futures:

* Multithreads:
  - Synchronization / co-ordination:
    - Basic problem: more than one execution entity accesses and uses a shared memory region/object.
    - Examples:
      - Two processes call fork. / via back.
      - Kernel invokes schedule on two cores.
      - Processes on two cores are moved to ready/waiting.
      - Producer-consumer examples.
      - All kernel data-structure is multi-processor setups.
*examples*

1. int count = 0;  // number written to disk
write to device (buf);

2. insert(int data);

```c
int * L = new list;
L->data = data;
L->next = L;
```
```c
list = L;
```

```c
void getlock (lock * lock);
while (1) {  
if (lock->locked == 0) { // check
lock->locked = 1;
break;
}
}
```

```c
void releaselock (lock * lock);
lock->locked = 0;
```

\[ \Rightarrow C \& D \text{ are race situations!} \]

\[ \Rightarrow \text{even architecture-level exclusion is not guaranteed; } \text{e.g., inc #mul#rew } \rightarrow \text{not safe.} \]

```c
read - modify - update!
```
CPU \[\text{bus} \rightarrow \text{memory}\]

- Instruction parallelism allows access to bus & hence access to unsafe shared data.

- \text{X86} the lock primitive / prefix.

- \text{Lock: inc (var)}

- locks the memory bus / cache line for duration of instruction.

- \text{Requirements:}
  - Atomic: all or nothing
  - Test and set instruction.

- Retry: load ro, # mem \leftarrow load value \& lock.
  - cmp ro, 0
  - jnz retry
  - load # mem, 1. \leftarrow set.

- Atomic test & set / test & lock instructions.

- \text{Example: cmpxchgl eax, &flags}.

- if \(\text{eax} = \text{val} \) (input) then
  - dest = set
- else
  - dest = source

- \text{mov ecx, l}

- \text{lock:}

- \text{loop d: lock cmpxchg lock, ecx var}

- unlock: mov var, 0.
basic test idea:

1. lock:
   - critical section
   - unlock

note: we are discussing synchronization in the kernel.
examples:
1. pt. copy by interrupt
2. adding new PCB to set of process on
3. adding RCP processes to wait queues.

22.9.17
(kernel) synchronization / coordination.

- terminology
  - race condition
  - synchronization
  - mutual exclusion
  - serialization

- critical section
  - locks
    - lock-less
  - spin locks
  - mutex
  - semaphores
  - condition variables
- deadlock

scenarios:
1. system call + "interrupt"
2. interrupt + interrupt
3. system + system call
   - (kernel) (kernel thread)

- primitives:
  - disable interrupts
  - disable pre-emption
  - synchronization
    - locking
get lock:

```
TSL YD, LOCK
LOCK & RD
CMP YD, 0
SET LOCK to 1
jne getlock:
RET
```

unlock:

```
MOV LOCK, 0.
RET
```

*get lock: mov ecx, 1

```
xchg ecx, lock.
inc lock.
```

TSL instruction (test & set lock)

+ Atomic instruction

- Copy value of

- Instrucution is atomic.
- Exchanges contents of 2 locs. atomically.

- Need to test & set in one-go.

- Atomic assistance from hw.

- Atomic instruction

- CPU locks memory bus access & not available for another instruction on diff. cpu.

- xchg disable interrupts

- Before spin (or lock)?
Do we need to disable interrupts? Use process locking required?

Lab 6 description:

26.9.19: (multiple) threads, (multiple) execution entities.

# process

in execution entity (thread), with an identifier.

(in virtual) address space to store program & data.

# several situations when multiple execution threads (in shared address space) is desirable.

Reasons:

4. multiple 10 & CPU activity, 8 appl.

Examples:

1. webserver.
   + dispatch thread & worker threads.

2. word processor.
   + main UI & input/output thread.
   + backup + spellcheck, etc.

3. CPU-bound apps.
   "benefit: no cost 10 CPU-bound apps.

3. read-process
   write.

4. can exploit multiple processors for single appl. / process.
- need for synchronization via mutual exclusion

1. spin locks
   - via rolling
   - WQ-assisted mechanisms
   
   acquire_lock: spin on CPU if lock not available
   
   release_lock: move process to wait block state

2. blocking locks
   - mutex - O
   - lock access
   
   wake_queue
   
   mutex unlock:

3. condition variables

   - generalization of binary counts to
   prevent unsafe access & sleep wait

   examples of conditions:
   
   if lock variables 0/1, q = 0
   
   if queue empty, full, q = max
   
   if count > threshold

   OK: RET
(i) produce()  
lock  
buffer++  
unlock

(ii) produce()  
lock  
if (buffer <= 0)  
lock  
wait (bufsize, lock)  
unlock  
buffer++  
wakeup (bufsize)  
unlock

consumer()  
lock  
buffer--  
unlock

while (buffer > 0)  
sleep (bufsize, lock)  
buffer--  
unlock.

buffer:

1. buffer not empty  
2. sleep & wakeup

on wakeup both cannot decrement  
so need check buffer state again  
& sleep if regd.

# enqueue  
on buf=MAX cond.  
- buffer has a max size  
- need 2 CVs  
- solve in class

buffersize: cv  
buffer <= 0 - cond.
4. Semaphores:

- an integer value variable for sync.

  up (s)
  lock
  s++;
  signal wake up (s);
  unlock;

  down (s)
  lock
  s--;
  if (s > 0)
  signal wake up (s);
  unlock:

  wait (cs, lock)
  unlock
  sleep

(i) C:
- parallel threads wait for

  lock:
  while (buffer <= 0)
  sleep (buf size, lock);

  lock:
  while (buffer >= 0)
  sleep (buf size, lock);

  buffer--;  || not seen
  wakeup (buf max);

  unlock;

(ii) use it as a cv:

  producers:
  p = max
  while (p > 0)
  lock
  produce [buf +]
  unlock
  down (ps)

  consumers:
  down (cs)
  lock
  consume [buf -]
  unlock
  up (cs)
- monitor.
- semaphores.
- threads

0.9.17

(struct sleep lock * lk); spin lock * lk;
uint locked;

(struct sleep lock * lk);
uint sleep lock (lk).

acquire sleep lock (lk) &
acquire (lk -> lk)
while (lk -> locked) &
sleep (lk, lk -> lk)
lk -> locked = 1;
release (lk -> lk);

releasesleep (lk) &
acquire (lk -> lk);
lk -> locked = 0;
wakeup (lk);
release (lk -> lk);

4/10.
producer - consumer example.

- maximum production buffer
  limited.
- multiple consumers.

producer

lock

consumer

lock

buffer = MAX
while conwait (pwait, &lock);
buffer ++;
condsignal (cwait);
unlock;

buffer --;
condsignal (pwait);
unlock;
with semaphores

produce

p. down()
produce()
locked

consume

a. up()

2. semaphore implementation

* semaphore implementation as mutex

sem s(0)
(binary) semaphore

up(s)
lock()
++s;
if(s>0)
signal(sem)
unlock()
down(s)
lock;
--s;
while(s>0)
wait(sem,lock);
unlock();

1. modifica for binary semaphore

p. down()
consumer

2. producer consumer w/ bounded buffer

cond. variables & semaphores

5. reader writer w/ condition variables

and semaphores

2. Lab 6 & 7 w/ cond. variables & semaphores

ch. 6 exercises, prog. & projects

Semaphores:

< lock

count down counter

notification:

4. the dining philosophers problem

philosophers dine on a circular table

philosophers need two spoons/forks to eat

philosophers think & eat
Solution 2: Regulate the table.

Philosopher (i)

Sema 5 \([5]\) : 5 semaphores, one for each spoon.

Philosopher (i)

Sema 5 \([5]\) : 5 semaphores, one for each spoon.

Attempt pickup \(0, N-1\). Parallel attempts.

Generalization:

- Deadlock if all philosophers decide to eat simultaneously.

Deadlock if all philosophers are in the same state.

Two of them, \((i)\) and \((j)\)
state[i] = thinking, hungry, eating.
sem s[i]; 5 semaphores to signal eating.

philosopher(i) while (1) &

state[i] = thinking;
think(1); print("I am thinking.");
pick spoons(i);
run wait until (s[i]=empty);

3.

pick spoons(i) &
lock
state[i] = hungry;
test(i); if spoons available;
unlock.
down (s[i]); wait to eat;
putspool(1) {
lock();
state[i] = thinking;
test(i-1);
test(i+1);
unlock();
}

3.

solutions:
1. Do not allow all philosophers to sit and eat/think at once.
2. Change order of first chopstick.
3. Pick & chopsticks in critical section.
10/10/17

* Recall Processes

- An execution entity with private memory for data (stack, heap), code, stack.
- With isolated memory boundaries.
- Embody programs/applications.

- OS responsibilities:
  - Allocation of physical memory to processes
    - mgmt.
  - Configure/Setup wise for process execution
    - use
  - Switch between processes
    - schedule

- HW support
  - Address translation
  - Protection
  - Transfer data between users & kernel

* Process context, state and the Pcb

1. Process context, state and the Pcb
   - Pcb

2. The first process - init
   - fork

3. Context switching (in xv6)

4. The xv6 scheduler

5. Scheduling algorithms/techniques

* Struct Proc

- trapframe
- context
  - stack for kernel
  - stack

(i) Trapframe: stores state & user-level process when swining

(ii) Context: for swining execution context in the kernel (push)
# syscall in action

- `fork()` → int 80h
  - 80h regs. have arguments:
    - `alltraps`
      - pushes all regs. on stack
      - sets trap ptr.
    - calls `trap`
      - `trapframe`
        - cleans up `trapframe`
        - to return to user space

- `trap (trapframe x + tf)`
  - `syscalls`
    - service call & return
  - `interrupts`
    - service interrupt & yield

- timer interrupt ~ 100 times/sec 10ms.

# fork

- find proc
  - assign pid
  - alloc kstack
  - set state
    - copy page table
    - copy proc ADDR eax=0
      - `p -> kstack = Kalloc`
      - `sp -> p -> stack`;
      - `tf`
      - ready to schedule
        - `so that child`
          - returns in `forkret`.

# context switch

- `p1` to `p2`
  - `alltraps`
    - `ks` to `ks` switch
      - `ks` to `ks` switch
        - `ks`
          - `ks`
            - `ks`
              - `ks`
                - `trapret`
switch (struct **context, struct **content)

saves the essential registers

to the old context

& loads regs from new context

These are kernel regs of process kernel

context of processes & schedules

e.g. 1: timer interrupt

to alltraps -> trap

yield

switch -> sched

[scheduler] (by popping at sched)

switch (scheduler, new proc)

sched / forkret

yield -> trapret -> user space,

allocproc -> setup measured

grow proc

setup vm

setup um

swtch vm

initum

e.g. 2: wait

sleep

sched

exit

first process:

useful

the thread is created manually.

first page is created & binary copied

at VA = 0

$p 

state = Runnable

return

_sched

switch

forkret
process scheduling & context switching

**Summary of Keywords:**
- alltraps
- trap
- trapret
- forkret
- sched
- switch
- scheduler
- kstack
- trap-frame
- context

**fork**

parent child

setup to

return w/o

syscall

when is scheduler invoked?

- whenever in kernel mode
  - system call, yield, exit, sleep, interrupt handler
  
additional issue, no trap-frame for first process.

- sequence:
  - userinit() called in main
    - allocproc
    - pid, proc, pgdir, kstack etc.
  - setupkvm
    - map kernel VA in page table
  - initvm
    - copy code starting from VA = 0

create trapframe manually

set process to runnable (ready to run!)

# efficiency

scheduling algorithms

- two main types
  - (i) non-preemptive
    - only when current process blocks or terminates
  - (ii) preemptive
    - whenever in kernel mode, even when process not blocked/done.
other areas:
- work conserving vs. non-work conserving
- real time vs. proportionate vs. fair
  etc.

preemptive vs. non-preemptive

kernel execution
- kernel disables pre-emption (by interrupt) or
- inefficient design

kernel execution
- kernel disables pre-emption

note: kernel disables preemption
when holding lock

When scheduler is invoked?
1. pick one of the runnable processes
   the scheduling policy
   - one queue over all CPUs or
   - one queue per CPU

3. picking a process is followed by
   context switching
   context of process: CPU registers, program counter,
   virtual memory
   e.g., switch to user
   information etc.
   - context of user process stored on
     the kernel stack of each process
   - context switch happens in kernel mode

regards to next question
in which context does the scheduler run?

- process context vs. kernel context.
  - e.g., Linux: x86

### Scheduling policies

- Metrics: waiting time, finish time, throughput, fairness, response time.

#### Example policies

- **FCFS**: first come, first serve.
  - every process that arrives/ready is placed at end of queue.
  - avg. waiting time can be longer!

- **RR**: round robin.
  - response time can be high.

  - Every process for a time quantum/slice and moves to the next process.
  - time slice is crucial parameter.
  - $t_0 \gg 0$ → tends to FCFS.
  - $t_0 \rightarrow$ fair; sharing out; shoots up.

- **SR** & **DK**
  - proportional weight round robin scheduler.
  - deficit round robin.

- **Lottery scheduling**: probabilistic & proportional.
  - # tokens/tickets issued per process.
  - random number gen. to select process.
  - deterministic expected metrics.
  - # tickets $> 0$ → no starvation.
Priority scheduling
- priority is stricter than proportionate scheduling
- diff. priorities for background vs. foreground vs. interactive processes
- multi-queue scheduling
  idea: one queue per priority
  always schedule jobs/processes in higher priority queue first.
  starvation!
  ⇒ change priority based on wait time.
  ⇒ assign quantum, if quantum
      finished/
      wait till others finish
  ⇒ yester year Linux.

Linux — 3 scheduling classes/priority
- each uses
diff. scheduling policies.
  e.g.: SCHED-RT: SJF & EDF.
        need to know finish times!
  # CFS: Completely Fair Scheduler.
  for N processes give \( \frac{1}{N} \) the CPU each.
  approx. over a period of time.
  'nice' value used as weight.
  \[ \frac{w_i}{2w_i} \] for each process.
  \[ \text{vtime} = \text{calc} \text{-delta} \text{-fair} (\text{delta}_{\text{exec}}, \text{cur}) \]
Every process is associated with one scheduling class. Each class uses its own scheduling policy.

1. **SCHED_FIFO**: used for real-time processes.
   - No pre-emption. Schedule another process only on exit, block or yield.

2. **SCHED_RR**: similar to FIFO, reserved for RT processes.
   - Each process associated with a time slice.
   - Quantum leaves CPU on time quantum.

Process priority needs to be between 1 to 100.

- \[ PR = 20 + NI \]
- \[ NI \in (-20 \text{ to } 119) \]

**PR = (0 \text{ to } 39) \text{ for Normal tasks}**

- Lower priority -> higher weight.

3. **SCHED_NORMAL**
   - All processes that are not real-time are belong to this class.
   - The Linux scheduler is CFS, completely fair scheduler.

CFS idea: if \( n \) processes runnable, each should get \( 1/n \text{th of CPU} \).

**(i)** priority -> weight [40] \( \in \) array of 40 nos.

Signifying weight to each priority level.

\[
\begin{align*}
-20 & \Rightarrow 88761 \\
-19 & \Rightarrow 77215 \\
-18 & \Rightarrow 65669 \\
-17 & \Rightarrow 54123 \\
-16 & \Rightarrow 42577 \\
-15 & \Rightarrow 31031 \\
-14 & \Rightarrow 19485 \\
-13 & \Rightarrow 8407 \\
-12 & \Rightarrow 7281 \\
-11 & \Rightarrow 6155 \\
-10 & \Rightarrow 5029 \\
-9 & \Rightarrow 3903 \\
-8 & \Rightarrow 2777 \\
-7 & \Rightarrow 1651 \\
-6 & \Rightarrow 535 \\
-5 & \Rightarrow 419 \\
-4 & \Rightarrow 303 \\
-3 & \Rightarrow 187 \\
-2 & \Rightarrow 71 \\
-1 & \Rightarrow 55 \\
0 & \Rightarrow 39 \\
+1 & \Rightarrow 23 \\
+2 & \Rightarrow 7 \\
+3 & \Rightarrow 1 \\
+4 & \Rightarrow 0
\end{align*}
\]

\[ \text{priority} = \frac{100}{1 + 40} \]

\[ \text{priority} = 100. \]
(ii) **vruntime**

- Ideally, with 2 tasks running, each get 50% of the CPU. Not physically possible.

**vruntime**: the next time slice (slot) when the task would start execution in an ideal multi-tasking setup. In practice, it is the runtime normalized by the number of running processes & process priority.

**Example**: ① A = 1 millisecond every 1 second. After 1 second, Vruntime are A: 1, B: 10, C: 20. A has lowest vruntime and can be scheduled next. Till A = 10.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>10ms/1sec</td>
<td>10ms/1sec</td>
</tr>
<tr>
<td><strong>V_a</strong> = 5</td>
<td><strong>V_b</strong> = 10</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>10ms/1sec</td>
<td>10ms/1sec</td>
</tr>
<tr>
<td><strong>V_i</strong> = 1</td>
<td><strong>V_o</strong> = 1</td>
</tr>
</tbody>
</table>

**vruntime calc.**

\[ \text{delta_exec} = \text{now} - \text{curr->exec_start}; \]

\[ \text{curr->sum_exec_time} + = \text{delta_exec}; \]

\[ \text{curr->vrun_time} + = \text{delta_exec\_weighted}; \]

\[ \text{delta_exec\_weighted} = \text{delta_exec} \times \frac{\text{nice-o-load}}{\text{curr->load\_weight}} \]

**Min_vrun_time** used to pivot less than min_vrun_time a red-black tree to store processes.
* scheduling latency

- time interval in which all processes should be run at least once.

\[ \text{sched. slice} = \frac{\text{interval}}{\text{nr. processes}} \]

used to preempt processes if reqd.

\( \text{if } (\text{delta-exec} > \text{sched.slice}) \)

resched-task()

# Multi-processor scheduling

(i) Asymmetric multiprocessor (ASMP)

- execution of single CPU (subset of CPUs)
  determines what executes on other CPUs.

(ii) SMP — symmetric multiprocessing.

- all CPUs execute OS code to schedule work.

* shared memory
* local caches
* local TLB
* same access times to memory.

* What is the OS state?

* of interest for scheduling?

- process table / PCB store.

Sln 1.

PCB's

Adv:
- single doc, easier right
- by all processes

Disadv:
- cache locality can take a hit.
- global lock contention
# Storage & File Systems

- A file system is the interface to an OS for most/ several users. It organizes and accesses files.
- Responsible for storing data & programs.
- The file system's user interface is via files and directories.

- Files & "directories" reside on storage devices.

---

# Devices

1. HDD/magnetic disk.
   - Platters that are magnetically coated on each side.

2. 10" platters.
   - Rotational speed: 5000 - 2 x 10^5 tracks/radial inch.
   - Seek time: 2 - 10 ms.
   - GB - TB capacity.

---

# Challenges w/ SMP:

1. Lock - preemption/lock-holding problem.
2. Job shootout.
3. Threads w/ producer-consumer semantics. Not, but schedules is agnostic.
4. Multiple resource scheduling, but low stren.
- Transfer speed: ~150 MB/s.
- Notes:
  - Dev.
  - Interface.
  - Write (start sector, count, buffer).
- DMA for direct transfers to/from memory.

![Diagram of DMA and memory interface]

- Block size: 16-256 KB
- Pages/block: 1 page = 512 - 4096 bytes.
- Quirks:
  - Write operations at block level, not page level. To rewrite page, block must be erased & re-written.
  - Finite writes
  - Life-time 8% block.
  - After 10^5/10^6 writes, block unreliable (wear-out).

3. Tapes:
- 9 tracks across tape.
- 8 bytes + parity
- ~MB/s.

---

Flash:
- Solid-state (semi-conductor based).
- Non-volatile / persistent.
- No moving parts.
- Faster than disk. ~6 Gbps. ~5-10x.
- Impl. via NAND or NOR gates.

Note: (i) Interface to use device, provided by device driver, is blocks.
(ii) Interface to user via FS is logical locations / offsets in files.
- Decoupling & consistent interfaces.
File: named logical storage unit.
- files on disk are independent of processes.
- attributes
  - name
  - identifiers
  - type
  - size
  - protection/ownership
  - time stats.
- operations (FS actions)
  - create
  - write
  - read
  - delete
  - truncate
  - reposition/seek
  - state req’d to manage files:
    - File system metadata (on disk)
    - OS state

Applications

User

open, close, read, write, seek.

Operating System

File system API

Kernel

File system

DD interface

Device driver

write block

move head to <cyl, sector, track>

read, write

Setup DMA

Where to store file information/attributes?

- On disk!

- File control block (FCB) / inode.
  (index node)

boot block
super block
inode list
data blocks

File system layout on disk
- tcb inode is an index into the inode list number.
- every file has a inode allocated on creation.
- multiple files may point to same inode.

# in-memory state of file systems.
1. in-memory inode table.
   - stores all file info for lookup & verification.
   - read from disk.
   - stores all file info for lookup & verification.
2. system-wide file table.
   - stores a list of all open files appropriate w/ a pointer to in-memory inode entry.
   - count # times each file opened.
   - note: same file can be opened multiple times by same/diff processes.
3. per-process file table.
   - list of files opened by a process.
   - pointer to the system file table.

* dentry: directory entry.
  - maps file object to inode.
  - no secondary state, all in-memory.
  - each path component has a dentry.

multiple file objects per physical file.

Linux example:

struct inode
struct file
struct file->array [NR_OPEN+1].

struct file *fdp = fopen / open.

index in this array.
File Systems:
- A system to organize & manage files.
- Two main tasks:
  - File (system) management.
  - IO performance.

FS = the on-device view.

Boot block | Super block | Inode block(s) | Data blocks

File: Unit of data.
- Byte stream
- Records.

File system state has to be persistent.

Super block & Inode blocks
File system metadata.

App's:
- FS - API read, write
- Device driver
  - DMA setup
  - Dev commands
    - Device controller

FCB / Inode:
- File control block
- Index node
- Metadata about each file:
  - Name, inode #,
  - Size, timestamps,
  - Ownership,
  - Location,
  - Type

System-wide:
- File descriptor table
- Inode table

File list:

* * *
FS in-memory state:
- in-memory inode store
- cache
- per-process file table
- system-wide file descriptor table

FS on-disk state:
- boot, superblock, inodes list, data blocks

Several such devices can be "mounted".

Mount: - mount pt. in directory structure

File system used for

Why needed? - org. on disk.

OS has to invoke appropriate FS operations to access content on disk & update.

OS can get bloated! What about new FS integration is time consuming & critical.

VFS: virtual file system extensively used by Linux.
VFS: Virtual file system.
- Data structure & methods used to decouple system-call functionality and file system implementation details.
- Avoid multiple directory & file structures and their implementation.
- Ease of integration of new FS.

- Linux VFS has 4 main objects: common file model.
  - inode: represents a file
  - file: represents an open file
  - superblock: represents a file system
  - entry: represents a directory entry
  - Each object is defined.
  - Also, has a set of operations that use & manipulate these objects.
  - The common file model maps naturally to the Unix-like operating systems & file systems.
  - Other FS that need to be work with Linux, such as FAT & NTFS etc, need to adhere to the common file model, even if their internal state/objects do not rely on the common file model.
  - The common model operates on the 4 basic objects & operations implemented by the FS.
Allocation methods for disk blocks to files:

- Super block manages all blocks (tracks)
- Properties: (a) minimize seek time, sequential & random access!
  (b) dynamic sizing, allocation.

1. contiguous allocation:
- Each file occupies a contiguous set of blocks.
- Example:
  - File | Start block | Count
  - f₁  6   | 3
  - f₂ 10  | 6
  - f₃ 26  | 2

  - Yes: 1. Spatial locality, arm/head movement access for subsequent blocks is minimized.
  - 2. Sequential & direct access.
  - block = start block + \[ \text{offset} \div \text{size} \]

- Yes: 1. Need to reserve blocks
  - How to determine file size?
  - External fragmentation.
  - 2. Dynamic sizing needs extra block band efforts.

- Fixes: de-fragmentation to reclaim blocks to resize files.

2. linked allocation:
- Link together blocks assigned to file, no requirement of contiguous alloc.
- Each block stores address/index of next block.
- Metadata: file | start block | endblock

  - Quiz: 23 69
  - Next block info takes up some space (4 bytes).

- Yes: 1. No external frag.
  - 2. Dynamic on-demand sizing.
(3) File Allocation Table (FAT)
- part of file system metadata.
- contains one entry per block, indexed by block number.
- each entry contains block number of next block in file.

- file startBlock: size...
- quiz 23

- \[ <23, 67, 86> \]
- up to data blocks.

 tres: - FAT can be loaded in-memory.
- direct access can be faster!
- still needs to touch all blocks.
- dynamic on-demand.
- no external fragmentation.

(4) Indexed allocation.
- to overcome block-walks for direct-access
- maintain a separate index-block which stores
  pointers to all blocks on disk.

- file ... index block

- quiz 23

 tres: FAT + quick direct access.

 tres: - index block needs space!
- 4-byte entries \( \Rightarrow \) 1024 blocks (4096 blk size).

Soln: (i) Linked allocation \( \Rightarrow \) index blocks.
(ii) Multi-level index (like paging!)
- first level index block points to second
  level index blocks.

(iii) Combined scheme:
- direct & multiple levels \( \Rightarrow \) indirect.

\[ \text{max size} \]

\[ \text{4 KB x 1024 x 1024} \]

\[ \text{1024 x 1024} \]
recap
- Allocation

VFS
- 10 Performance
- Lab 10
- Ext 7-6

recap VFS
- in-mem. stat

VFS

op

vfs

write (fp, buf, size)
close (fp)

write (fp, buf, size)
close (fp)

write -> sys_write -> fs_write -> disk

user

vfs

fs

phy

Page/disk caching
Buffer cache
mmem.

fp

per-process file table,
file desc:

system-wide
file desc:

inode list
(cache)

inode list
(cache)

fp = open()

p

index.

struct file

inode

store offset
file lock
count etc.

- ptr. to
inode

- ptr. to
dentry

- points to
file object.
File system consistency.

The issue: Several copies/locations of file system meta-data and data.

- Page cache, in-memory meta-data, on-disk metadata.
- WM disk caches (NFS cache).
- Synchronous vs. asynchronous writers and
  not updates/writes are not atomic.
- System crashes are non-deterministic.

E.g.: file open/create.

1. - make data blocked used.
2. - allocate inode/allocate inode.
   - pt. file object to inode.
3. - update directory data vs. file details.

Several steps - not atomic!
1. fsck - file system check.
   - walk through meta data to ensure that files & meta data are consistent.
   - eg: file size in inode & FAT entries match.
   - used blocks in bitmap & FAT match.

2. JFS: journaling file system
   - a "journal" is maintained for all non-committed file changes to the file system.
   - metadata and data:

   - updates in journal area
     "atomic"
     start End/commit: change state of data/metadata

   - mmap - memory mapped files
     - mmap (void* addr, size, prot, flags, file, offset).
     - creates a virtual address region whose contents are from offset to offset + size from file ptr's "fd".

   - fuses:
     - bulk copy of disk data to memory.
     - double copy eliminated, disk buffer user.
     - reads are very efficient (no seeks!)
     - system call overhead decreases.

   - Page cache wrinkles:
     - synchronous vs. asynchronous writes
     - write back vs. write through vs. direct io
RAID: Redundant Array of Independent Disks.

RAID 0: Striping, no redundancy, no checksum.
- increases bandwidth, no fault tolerance.
- reads could be faster.

RAID 1: Mirroring.
- fault tolerance, 2x cost, [A1, A2, ..., An], [A1, A2, ..., An]
- N-1 additional disks.

RAID 2: Hamming code, Bit-level striping.
- N-1 additional disks.

RAID 3: byte-level striping, single checksum disk.
- [A1, A2, A3, P1-3]
- single block fail.

RAID 4: block-level RAID 3.
- reads perform better, multiple blocks read in parallel.

RAID 5: block-level striping, 1/5 in situ parity.
- no extra disk.
- [A0, A1, A2, A3, A4], [B0, B1, B2, B3, B4], [C1, C2, C3, C4, C5]

RAID 6: RAID 5 + 2 parity blocks.

Disk/IO scheduling:
- how to service IO requests (for blocks) in request queue?

Seek time
- rotational latency
- bandwidth
BIOS

basic input output system
(post: power on self test)

Load OS/Boot loader (boot sector)
program
read first sector
readable disk

Read OS from disk
(blocks of OS in boot sector / MBR)
jump to kernel code in "real" mode

Read rest of kernel, execute in protected mode
First user process
- OS conceptual requirements, services.
- Basic abstractions.
  - Basic abstractions.
  - Data matching.
- Processes, interrupts, scheduling.
- Memory.
  - Memory.
- File systems.
  - File systems.
- xuw file systems.
  - xuw file systems.
- Security.
- Booting & arch specifics.