0 WHAT'S AN OS?

An OS is a VM (extends and simplifies interface to physical machine), resource manager, and a control program.

1 PROCESSES AND THREADS

A process is any program that's running. The heap is allocated at runtime by `sbrk()` and `mmap()` usually in C it is `malloc()`.
New → Ready ↔ Running, Exit. Then we have blocked.

1.1 PCB

Stores:
- Process state (ready, running, blocked)
- PC, CPU registers to be saved on interrupt
- Scheduling information
- Memory management info (page tables)
- Accounting information (CPU time, memory)
- I/O status information Always stored in memory.

1.2 System calls

```
printf("Hello world\n");
libc:
 %rax = sys_write;
 SYSENTER;
```

1.3 Threads

A thread has its own PC and stack, but everything else is for the process.

2 SYNCHRONIZATION

2.1 Critical Section Problem

- Mutual exclusion
- Fairness, no starvation, or bounded waiting
- Performance or progress

2.2 Condition Variables

Only call the API when you're holding the mutex. A condition variable stores a wait queue. `cond_wait(*cv, *mutex)`
- Releases `mutex`, add me to `cv`'s queue, sleeps me. On return, reacquire mutex `cond_signal(*cv)` or `cond_broadcast(*cv)`
- Wakes up 1 random or all enqueued threads

Pattern:
```
lock(mutex);
while(!condition){
 wait(cond, mutex) };
// do stuff
signal(cond); unlock(mutex);
```
The producer/consumer problem involves a pipe.

2.3 Semaphores

Initialize `sem.count`. **On call:** 1 for *lock*, 0 for *requirement*, −1 for double requirement. **1 or more: GO** otherwise **0 or less: STOP** A wait call can never cause `Sem.count` to go negative.
```
Wait(sem):
 while(Sem.count <= 0);  // spin
 Sem.count--;
Signal(Sem)
 Sem.count++;
```

2.4 Deadlock

Conditions: Mutex, Hold & Wait, No preemption, **Circular wait** (fix by resource acquisition ordering)

3 SCHEDULING

Which thread or process runs over time?
- **Arrival timestamp**: when a thread becomes ready
- **Service time**: time required to run thread to completion
- **Wait time**: time between arrival and being scheduled (put onto CPU)
- **Turnaround time**: Wait + Service
- **I/O Burst**: State of requesting I/O
- **CPU Burst**: State of needing CPU

3.1 Policies

- **FCFS** – first come first served; suffers from convoy but very easy
- **SJF** – when I have the chance run thread w/ shortest expected time
- **RR** – time quantum q, circular ready queue
- **MLFQ** – always run highest priority; jobs start at high priority to gather data; priority can change
  - Aging ↑
    - Priority goes *down* if the CPU was used a lot **recently**
  - I/O, interactive, q not used up ↑
  - Shortest service ↑
- **Proportional-share / Lottery**: Each group is assigned tickets, hold a lottery, processes can loan tickets. Add `num_tickets` to PCB, generate random `winner` value, loop over all processes adding their ticket count to the counter circularly until `counter > winner`
- **UNIX**: $\text{CPU}_j(i) = U_j(i)/2 + \text{CPU}_j(i-1)/2$ $P_j(i) = \text{base}_j + \text{CPU}_j(i-1)/2 + \text{nice}_j$, lower $P_j(i)$ means higher priority
- **Linux 2.4** – Credit based; always choose highest credit and −1 credit on running process per timer interrupt. Suspend if 0. Refill when no runnable processes with `credits_left / 2 + base` credits.
- **Linux 2.5** The $O(1)$ scheduler. Each process gets time quantum on priority. Two arrays: ACTIVE, EXPIRED. Processes selected to run on ACTIVE (on time quantum runout, move to EXPIRED) until it becomes empty, then swap the two arrays.
- **Linux 2.6**: 140 Queues, one for each priority level, 0-99 for real-time and 100-139 for timesharing. Each priority level has its own time slice; a thread stays active until it uses it up. After running, priority and time slice recalculated and moved to set of expired queues. When all active ready, swap

4 MEMORY MANAGEMENT

Should support enough active processes to keep CPU busy and use memory efficiently, minimizing overhead, whilst keeping:
- **Relocation** – no dependency on physical memory availability
- **Protection** from unwanted access
- **Sharing**
- **Logical organization** (VMA)
- **Physical organization**

4.1 Partitioning

- **Fixed partitioning** – allocate blocks ahead of time that processes can occupy, but a process can only occupy one
- **Dynamic partitioning** – `malloc` w/ relocation
- **Paging**

4.2 Base, Limit

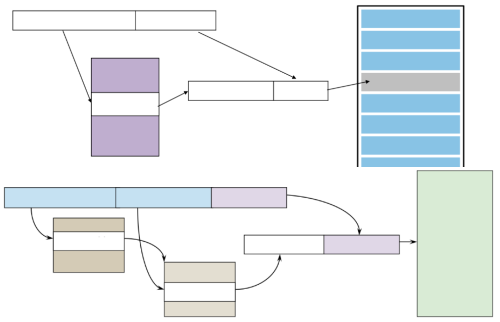
ABSENT IN PAGING. **2 hardware registers, "base" and "limit"**. These registers are affected by a context switch. On memory reference, use base address and compare with limit. Illegal then trap. Store in PCB.

4.3 Malloc Placement Algorithms

- **First**: → / Fragments get put at start
- **Next**: Use a clock handle on malloc
  - Fragments free space rapidly
- **Best**: Closest in size
  - Fragments become small and unusable
- **Worst**: Largest block
- **Quick**: Keep track of free list for common block sizes
  - Fast allocation, harder to coalesce / join together

4.4 Page Tables

Each process has its own page table.



A PTE is a row in a page table. Each PTE stores at least a valid bit and the frame no.

4.5 Address Translation

```
PAGE_NUMBER = VADDR / PAGE_SIZE
PAGE_OFFSET = VADDR % PAGE_SIZE
VADDR = (PG_NO*PG_SIZE) + PG_OFSET
PADDR = (F_NO * PG_SIZE) + PG_OFSET
```

0	0000	4	0100	8	1000	12/C	1100
1	0001	5	0101	9	1001	13/D	1101
2	0010	6	0110	10/A	1010	14/E	1110
3	0011	7	0111	11/B	1011	15/F	1111

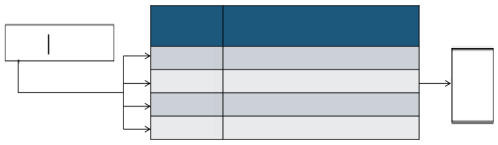
0	0x0	48	0x30	1024	0x400
16	0x10	128	0x80	2048	0x800
32	0x20	256	0x100	4096	0x1000

A row of the page directory tells us the **frame number** of the secondary page table we must target. A PDE is very similar to a PTE, so

reading the frame no. will require you to multiply it by the page size. Remember the size of each entry!

4.6 TLB

Hardware cache that stores recently used virtual addresses and their PTEs (→ frame number). Small but **very fast**.



The **TLB must change entries on a process context switch.** TLBs exploit locality. There will be TLB misses (a.k.a. faults).

4.7 Demand paging

Evictions, done in A3. Only write dirty pages to disk (pages that have been modified *since* it was written to disk).

4.8 Paging Policies

Cold misses (first-time accesses) are not capacity misses!

- **Belady's**: Replace page that will not be used for the longest
- **FIFO** (suffers from Belady's anomaly)
- **Exact LRU**: one used last gets replaced
- **Clock**: LRU but better. Persistently stores clock handle % frame count. Each frame gets a reference bit. They start at 1; when referenced also set bit to 1. If handle touches 1, set to 0. If 0, evict it. Handle at "where I will check next" when not moving, so once I'm finished analyzing a frame I move the handle.
- **s2q**: Single-use pages get evicted quickly otherwise they are harder to evict.

Thrashing is when page replacement algorithms take too long

4.9 VMA

Stack, heap, shared library `mmap` regions, heap, data, code. VMAs track: R/W/X, `(start, size)`
FYI: `sbrk` changes memory an entire process can access.

4.10 Working Set

A page is in the working set ⇔ it was referenced in the last D references – in interval $(t - D, t]$. "portion actively used"

4.11 Sharing, Copy on Write

Can map different virtual addresses to same physical addresses. Used on `vfork()`, saves overhead on OS copying data
Copy on write defers large copies until I actually need to copy (by writing) shared pages are read-only and writing to them traps and copies then writes.
By the way `mmap` lets you map a contiguous region of a file into memory so you can do what you want on it. You update something there, it eventually gets put on the file.

5 FILE SYSTEMS

Ops:

- Creation, Write, Reading, Delete (whole), Truncate
- Repositioning (changes current file position for next read or write operation)

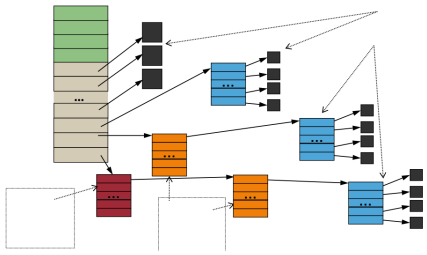
5.1 Directories

A directory entry holds the inode number and the file name. A soft link is like a shortcut and holds the true path to the file.

5.2 Data Blocks, Allocation

Everything but extent uses blocks

- Contiguous
  - Malloc but with blocks
- Linked
  - Block links next block; used in FAT
- Indexed
  - A4



5.3 Inodes

An inode holds metadata. Examples:

- Is directory?
- Owner
- Size (bytes)
- Access, created, last modified, freed?
- **Pointers to file content**

5.4 File Access Example

To access `/one/two/three`

1. Read super block for inode of `/`
2. Read inode for `/` and get the directory data block
3. Search for `one` in that directory block, and go to its inode
4. Repeat

5.5 Dealing with Hard Disks

Seek time: moving my arm Latency time: wait for the disk rotation
Aim to reduce seek times. Put related things close to each other, perhaps on the same ring.

5.6 Consistency

1. Inodes may only point to data blocks marked allocated by bitmap
   - If an inode is inactive change the data pointer
2. A data block can't be pointed to more than once
3. All allocated inodes must be in some DE. Put an allocated inode in `lost+found` if no directory refers to it. FSCK does this.
4. Inode Hard Link count == no. directory entries. This includes `.` and `..`
5. Directories must start with `.` and `..` and the directory tree must be acyclic

5.7 Reliability on Failure

When updating a file, I can read to: **Data bitmap, Inode, Data block**. If power fails and my writes have these two cases:

- **Data block** only (data lost, consistent otherwise)
- **Data bitmap and Inode** only (garbage data, consistent otherwise) Then I still have a consistent FS. Everything else leads to inconsistency.

5.8 Journaling

Start with some journal inodes and some data blocks allocated for journals. We update the journal entry **before** writing to the true data block. Our journal entry looks like this:

- `TxBegin`, Inode, Bitmap, Data block, `TxEnd`

If we want to add one data block in the file:

1. **Journal write:** Write the transaction (inode, bitmap, data). Fail here, don't retry.
   - `TxEnd` **MUST** be done last in this step
   - Step 2 may not start until `TxEnd` is written
   - Enforce with barrier
2. **Checkpoint:** Write the blocks. Here onwards, do "redo logging"
3. **Free step:** Mark transaction as free

Any failure anywhere in this process is recoverable. Ext3 tries to batch multiple file system operations into one transaction. Journaling ensures FS consistency. Recovery complexity is in size of journal.

5.9 Metadata journaling

Speeds up journaling. Only FS metadata (inodes) are updated, nothing else is. **Always write data before writing metadata.**

6 SSDS

To update a bit in a valid page, you'll have to update the whole block. All erased pages are set to 1. You can harmlessly set bits to 0 but you cannot set bits to 1 without collateral damage (hence constraints). Write amplification
A page can either be `VALID`, `ERASED`, `INITIAL`, or `TRIMMED`. You can erase an entire block and set the bits of any erased page. Think about what you can do with these constraints.

6.1 FTL (File translation Table)

We have logical and physical blocks, and our device stores a map between logical → physical.
Logical addresses are translated to physical addresses. The device stores a map for this. However, it must also garbage collect. It must also store this map on device in an OOB area. This does help with wear leveling, though.

6.2 Physical organization

A block stores 128 pages.
Physical organization - an SSD has multiple chips, a processor, and volatile memory. In the SSD, hierarchy goes like: Die, Plane, Block, Page, Cell, Transistors