1 Introduction

OS is a layer of systems software that provides application programming interface (API) to use hardware.

- Has privileged access to the hardware directly
- **Abstraction**: Hides the hardware complexity
  - **Level of indirection**: solve by introducing intermediate layer
- Manages and **shares** hardware among applications
- **Protect** from malicious or buggy applications
- **Isolating** one application from another

1.1 OS as three pieces

- **Virtualization**: How to make application believe it has each resource to itself?–Process, CPU scheduling, Virtual memory
- **Concurrency**: How to handle concurrent events correctly and efficiently?–Threads, synchronization
- **Persistence**: How to make information survive power loss?–Storage, file systems

2 Kernel and Protection

Motivation: Protect kernel from buggy or malicious application

2.1 Design

- **Privileged instruction**: Preventing applications from executing some important instructions
- **Memory protection**: Preventing applications from reading/writing other applications' or kernel's memory
- **(Timer) interrupt**: OS must regain control from applications

2.2 Privileged instruction

2.2.1 Dual model kernel (Limited direct execution)

- Kernel mode: full privileges
- User mode: CPU checks every instructions before executing them, and execute only those granted by OS kernel

On the x86, mode stored in EFLAGS register.

2.2.2 Handling privileged instructions

- Change mode bit in EFLAGs register
- Change which memory locations a user program can access
- Execute kernel code. May send commands to I/O devices.

2.2.3 Attempts to execute a privileged instruction

- **Exception**: Kernel is notified of unexpected/malicious behavior
- **System call**: Programming interface to the services provided by the OS. Cause a mode switch to kernel, and kernel executes privileged instructions required for the process

2.3 Memory protection

- Software-based: Kernel intervenes all memory accesses and allows only valid ones (Very slow)
- **Hardware-based**
  1. If target memory to access is illegal, hardware raises exception
  2. Kernel takes the control and do proper actions, *e.g.* kill

2.3.1 Virtual address

Virtual address is invented to separate the view of the address space of a process (V1) and hardware (V2).

- A process can freely access any addresses of V1
- An address in V1 is translated to an address in V2
  - Translation is done by Memory-Management Unit (MMU) in hardware using a table.
  - The table is configured by operating system kernel

2.4 Timer interrupt

During the boot sequence, the kernel start the timer hardware. The timer periodically raises an interrupt every few milliseconds.

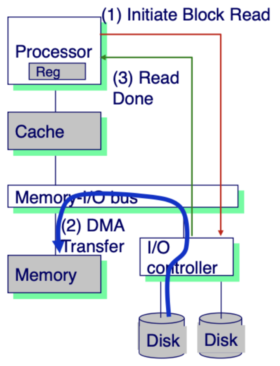
2.5 IO design

I/O devices and CPU can execute concurrently.

- Each device **controller** is in charge of a particular device type
- Each device has a local buffer
- CPU issues specific commands to I/O devices
- CPU moves data between main memory and local buffers

2.5.1 DMA (Direct Memory Access)

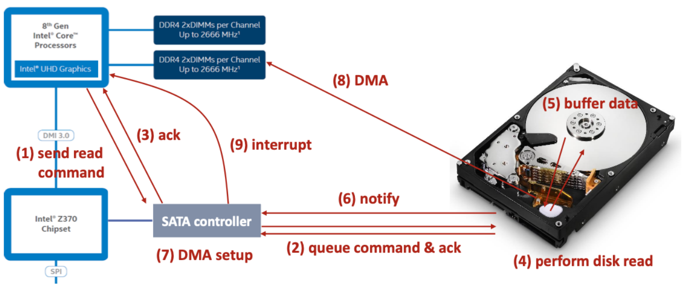
- Used for high-speed I/O devices to transmit information at close to memory speeds
- IO controller transfers blocks of data from the local buffer directly to main memory (or vice versa) without CPU intervention
- While transferring data, CPU executes a task



How does the kernel notice an I/O has finished?

- Blocking: Kernel waits until I/O is done
- Non-blocking: Kernel can do other work in the meantime

2.5.2 Disk I/O example



1. CPU initiates a read command to retrieve data from the disk
2. The command is sent to the SATA controller, which queues the command for execution
3. The SATA controller sends an acknowledgment back to the CPU to confirm that the command has been queued.
4. The hard disk drive performs the actual read operation by accessing the data stored on its magnetic platters.
5. The data is temporarily stored in a buffer on the hard drive itself. Buffering is used to accommodate the speed difference between the disk rotation speed and the data transfer rate.
6. The disk sends a notification to the SATA controller that the data is ready to be transferred.
7. The SATA controller sets up a DMA operation.
8. Data is transferred from the disk's buffer directly to the system's main memory via DMA.
9. The SATA controller sends an interrupt to the CPU, to handle the completion of the read operation.

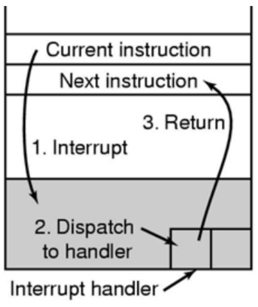
2.6 Mode switch

User → Kernel: Interrupts, exceptions, system calls

Kernel → User:

- New process/thread starts
  - Jump to first instruction in program/thread
- Return from interrupt, exception, system call
  - Resume suspended execution
- Process/thread context switch
  - Resume some other process
- User-level upcall (UNIX signal)
  - Notify user process of some event that needs to be handled right away, *e.g.* time expiration, asynchronous I/O completion
  - Implemented same as interrupts: signal handlers, signal stack (in user address space), save/store registers, signal masking

2.7 Handling interrupts and exceptions



1. Preserves the state of CPU in **interrupt stack**
   - Per-processor, located in kernel memory
2. Transfers control to corresponding interrupt handler, also called interrupt service routine (ISR)
3. Kernel executes a interrupt handler
4. Restore saved state (a process state) of CPU from interrupt stack
5. Transfers control from interrupt handler to the user code
   - x86 does not allow direct jump from kernel to user-level
   - Kernel must use `iret` instruction

2.7.1 Interrupt masking

- Interrupt handler runs with interrupts off. Re-enabled when interrupt completes.
- OS kernel can also turn interrupts off, *e.g.* when determining the next process/thread to run. On x86,
  - CLI: disable interrupts
  - STI: enable interrupts
  - Only applies to the current CPU (on a multicore)
  - Privileged instructions

2.8 OS trap

To execute a system call, a program must execute a special **trap** instruction.

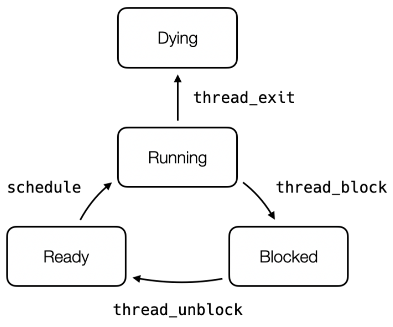
- Causes an exception, which invokes a kernel handler
- Passes a parameter indicating which system call to invoke
- At boot time, set the trap table which tell the hardware what code to run when certain exceptional events occur

3 Processes

- Process is an abstraction of **machine**.
- Process is an **instance** of a program in execution.
- Process is a basic unit of **protection**.

3.1 Implementation

PCB (Process Control Block) represents a process:



- CPU registers
- PID, PPID, process group, priority, process state, signals
- CPU scheduling information
- Memory management information
- Accounting information
- File management information
- I/O status information
- Credentials

3.1.1 Implementing `fork()`

- Creates and initializes a new PCB
- Creates and initializes a new address space
- Initializes the address space with a copy of the entire contents of the address space of the parent
- Initializes the kernel resources to point to the resources used by the parent e.g., open files
- Places the PCB on the ready queue
- Returns the child's PID to the parent, and zero to the child

3.1.2 Implementing `exec()`

- Loads the program into the process's address space
- Initializes hardware context and args for the new program
- Places the PCB on the ready queue
- Does not return after executing program

3.1.3 Why separate `fork()` and `exec()`?

Allow changes in child process after `fork()` but before `exec()`.

- **I/O Redirection**: Before calling `exec()`, the shell can change file descriptors in the child process. This allows redirecting output to a file (`command > output.txt`) or taking input from a file (`command < input.txt`).
  - After `fork()`, before calling `exec()`, shell closes `STDOUT` via `close(1)` and opens `output.txt` and use as `STDOUT`
- **Pipes**: The shell can set up a data pipe between two processes. This is done after the `fork()` but before the `exec()`, allowing commands like `command1 | command2`

3.2 UNIX I/O API (POSIX model)

- **Uniformity**: Unix treats everything as a file
  - All operations on all files, devices use the same set of system calls: open, close, read, write
- Open before use
- Open returns a handle (file descriptor) for later calls on file
- Byte-oriented (not block-oriented)
- Kernel-buffered read/write
- Explicit close: To garbage collect the open file descriptor

3.3 File descriptor

- An integer that represents a file, a pipe, a directory and a device
- A process uses a file descriptor to open a file and directory.
- Each process has its own file descriptor table.
- 0 (Standard input), 1 (Standard output), 2 (Standard error)

3.3.1 `cat` example

Shell is a program that `fork()` and `exec()` the command with argument. For example, when `ls -l`, shell calls `fork()` and `exec("ls'", "ls -l")`.

4 Threads

Bottleneck is phenomenon where the performance of an entire system is limited by one or more components/resources.

> Bottleneck shifts from CPU to IO

We address the IO bottleneck via concurrency. We run more tasks in CPU that does not use IO.

4.1 Motivation for new abstraction

- A single process cannot benefit from multi-cores
- Expensive creation, communication, and context switching

A **thread** is a single execution sequence that represents a separately schedulable task.

- No protection between threads for performance.
  - Code, data, heap are shared
  - A thread has its own stack

	Process	Thread
Unit of	Protection	Execution
Abstraction of	Machine	CPU
Switch overhead	High (∵ Memory/IO)	Low (only CPU)
Creation overhead	High	Low
Protection	CPU, Memory/IO	CPU
Sharing overhead	High	Low

4.2 Interface: Pthreads (POSIX Threads)

In Pthreads implementation, `fork()` duplicates only a calling thread. In the Unix international standard,

- `fork()` duplicates all parent threads in the child
- `fork1()` duplicates only a calling thread

Normally, `exec()` replaces the entire process.

4.2.1 Thread cancellation

We may need to terminate a thread before it completes.

- Asynchronous cancellation: Terminates the target immediately
  - What happens if the target thread is holding a resource?
- Deferred cancellation: The target is terminated at the cancellation points. Periodically check if I should be cancelled.

Pthreads API supports both cancellation.

4.2.2 Signal handling

Where should a signal be delivered? Pthread only delivers to a single thread found in a process that is not blocking the signal (Signal handlers are per process, signal masks are per thread).

4.3 Implementation

4.3.1 Kernel-level (OS-managed) threads

- All thread operations are implemented in the kernel
- Thread creation and management requires system calls
- OS schedules all the threads

Limitations are:

- Cheaper than creating processes, but can still be too expensive
- Thread operations are all system calls
- Must maintain kernel state for each thread → place limit on the number of simultaneous threads
- Different needs by programmers, languages, etc

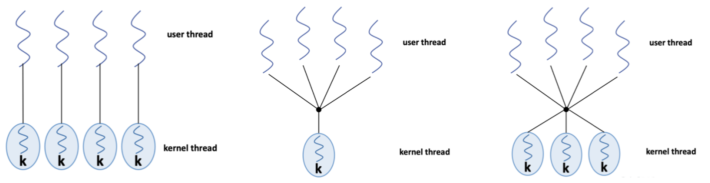
4.3.2 User-level threads

- A library linked into the program manages the threads
- Threads are invisible to the OS
- Thread operations are procedure calls (no kernel involvement)
- Small and fast: 10-100x faster than kernel-level

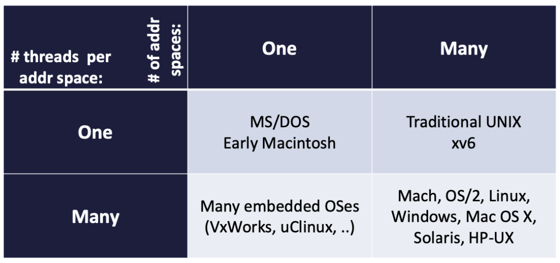
Limitations are:

- Rely on non-preemptive scheduling
- OS can schedule poorly as it is not aware of threads
- To prevent blocking all threads, blocking system calls should be emulated in the library via non-blocking calls → Requires coordination between kernel and thread manager
- Cannot leverage multi-core CPUs

4.4 Threading models



- One-to-one: For every thread that an application creates, OS creates a corresponding thread at the kernel level (Most popular)
- Many-to-One: Systems that do not support kernel-level threads
- Many-to-Many: Allows OS to create a sufficient number of kernel threads



# threads per addr space: / # of addr spaces:	One	Many
One	MS/DOS Early Macintosh	Traditional UNIX xv6
Many	Many embedded OSes (VxWorks, uClinux, ..)	Mach, OS/2, Linux, Windows, Mac OS X, Solaris, HP-UX

4.5 Linux Thread implementation

In Linux, a **task** refers to the basic unit of execution. Linux kernel treats both processes and threads as tasks, using the same underlying data structure `task_struct` to manage them.

Threads are created using `clone()` with flags that determine what is shared, *e.g.* address space. Threads in a process are simply tasks that share certain resources.

5 CPU scheduling

5.1 Introduction

5.1.1 Policy vs. Mechanism

- Policy: What should be done?
- Mechanism: How to do (implement) something?

Separating policy from mechanism is a key principle in OS design. Policies are likely to change depending on workloads, so a general mechanism, separated from policy, is more desirable. It enables more modular OS and extensible systems.

For a scheduling problem,

- Mechanism: How to transition?
- Policy: When to transition? To whom?

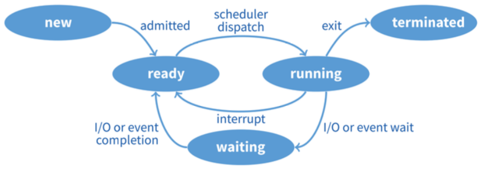
5.1.2 Scheduling mechanism

- Pick a task from run queue according to scheduler algorithm
- Kick out the running task from CPU
- Make the selected task run in CPU
- Context switch if running task ≠ selected task

5.2 When does OS invoke scheduler?

5.2.1 Preemption

- Non-preemptive: waits for running process to voluntarily yield CPU; processes should be cooperative. Invoke scheduler when
  - CPU is yielded: Running → Waiting/Terminated
- Preemptive: can interrupt a process and force a context switch. Invoke scheduler when
  - CPU is yielded (same)
  - New job enters queue: New/Running/Waiting → Ready



5.3 Workload assumptions

A1 Each job runs for the same amount of time
A2 All jobs arrive at the same time
A3 Once started, each job runs to completion
A4 All jobs only use the CPU (no I/O)
A5 The run time of each job is known

5.4 Scheduling metrics

- Turnaround time: Time to complete, $T_{\text{finish}} - T_{\text{arrival}}$
- Response time: Time to first response, $T_{\text{response}} - T_{\text{arrival}}$

If scheduling algorithm is identical,

- Non-preemptive scheduling has better turnaround time
- Preemptive scheduling has better response time

5.5 Scheduling policy

5.5.1 FIFO

First come, first served (FCFS). **But** when we relax A1, average turnaround time can be large if small jobs wait behind long ones (**Convoy effect**).

5.5.2 Shortest Job First (SJF)

Non-preemptive. Run the shortest job first, then the next shortest, and so on. Given A2-5, achieves optimal turnaround time. **But** when we relax A2, Convoy effect.

5.5.3 Shortest Time-to-Completion First (STCF)

Add preemption to SJF. When a new job enters, schedule the job which has least time left. Given A4-5, achieves optimal turnaround time. **But** bad response time, A5 is unrealistic.

5.5.4 Round Robin (RR)

Runs a job for a time slice (or scheduling quantum), then switches to the next job in the run queue. Can be either preemptive or non-preemptive. Time slice is

- Multiple of the timer-interrupt period (10~100ms)
- Trade-off between response time and cost of context switching
  - Longer time slice amortize the cost but worse response time

Typically higher turnaround time than SJF, but better response time.

5.6 Towards general scheduling policy

A4 relaxed. Overlap computation with I/O. → Treat each CPU burst as an independent job. A5 relaxed. No *a priori* knowledge on the workloads.

- Optimize turnaround time
- Minimize response time for interactive jobs

5.6.1 Priority scheduling

- Each job has a (static) priority
- Choose the job with the highest priority to run next
- Can be either preemptive or non-preemptive

Starvation problem: If there is an endless supply of high priority jobs, no low priority job will ever run.

- FIFO: no priority
- SJF and STCF: priority is run time/time left
- RR: priority is order of queue → No starvation

5.6.2 Multi-level Feedback Queue (MLFQ)

1. Distinct queues for each priority level. Priority scheduling between queues, RR in the same queue

Jobs can be classified as below.

- Interactive: short-running, require fast response time
- CPU-intensive: need a lot of CPU time, don't care response time

Dynamic priority change

2. When a job enters, it is placed at the highest priority
3a. Job uses up an entire time slice → reduce priority
3b. Job gives up CPU before the time slice is up → keep priority

→ MLFS approximates SJF. Limitations are:

- Starvation
- May abuse scheduler by relinquishing CPU just before expiration
- A program may change its behavior over time

Avoid starvation via priority boost:

4. After some time period S, move all jobs to the topmost queue.

3a/3b revised to avoid scheduler abusing. Count the total runtime.

3. Job uses up its time allotment at a given level → reduce priority

6 Virtual Memory

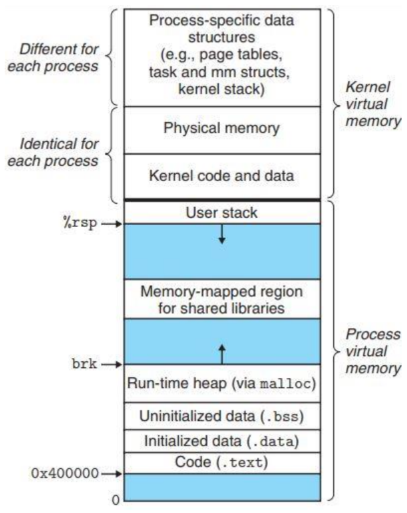
6.1 Goals

- Protection: Protect processes and the OS from another process
  - Isolation: a process can fail without affecting other processes
  - Cooperating processes can share portions of memory
- Transparency: Processes are not aware that memory is shared → Convenient abstraction
- Efficiency
  - Space: Minimizes fragmentation due to variable-sized requests
  - Time: Gets some hardware support

6.2 Virtual address space

Each process has its own virtual address space used for memory references: large, contiguous, and private.

- Kernel space: invisible to user code
- User space: specific to the process
  - Static area (code/data): allocated on `exec()`
  - Dynamic area (heap/stack): allocated at runtime and grow/shrink


Different for each process
Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
Identical for each process
Physical memory
Kernel code and data
Kernel virtual memory
%rsp
User stack
Memory-mapped region for shared libraries
brk
Run-time heap (via malloc)
Uninitialized data (.bss)
Initialized data (.data)
Code (.text)
0x400000
0
Process virtual memory

Address translation is performed at runtime. Supports **lazy allocation**:

- Physical memory is dynamically allocated or released on demand
- Programs execute without requiring their entire address space to be resident in physical memory

6.3 VM APIs

6.3.1 `libc`

Used by applications to manage memory in a heap.

- `void* malloc(size_t size)`: allocate a `size` bytes memory region on the heap
  - Return: on success, pointer to memory, on fail, a null pointer
- `void free(void* ptr)`: free a memory region
- `void *calloc(size_t num, size_t size)`: allocate memory for an array of `num` elements of `size` and zeroes it before returning
- `void *realloc(void *ptr, size_t size)`: change the size of memory block to `size`

6.3.2 System calls

`malloc` uses `brk` system call to ask OS to expand heap. Programmers should never directly call either `brk` or **sbrk**. `break` is the location of the end of the heap in address space and those calls expand the program's `break`.

- `int brk(void *addr)` sets `break` to `addr`
- `void *sbrk(intptr_t inc)` increments heap by `inc`

6.4 Memory mapping

A dynamically allocated virtual memory area may have a backing store which holds the actual data.

- File mapping: maps to a file region and the content of the file can be read from/written to using load/store instructions
- Anonymous mapping: not backed by a file, maps to a memory area filled with 0 (zero-page mapping)

7 Address translation

7.1 Introduction

7.1.1 Static relocation (software-based)

OS rewrites each program before loading it in memory. Changes addresses of static data and functions. Limitations are:

- No protection: can read memory of OS or other processes
- Cannot move address space after it has been placed

7.1.2 Dynamic relocation (hardware-based)

MMU performs address translation on every memory reference instructions.

- Protection is enforced by hardware: if the virtual address is invalid, the MMU raises an exception
- OS passes the information about the valid address space of the current process to the MMU

7.1.3 OS responsibilities

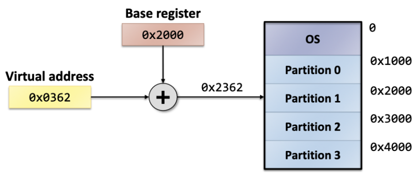
Generally manage memory via free list.

- Allocate memory for new processes
- Reclaim memory from terminated processes
- Save/store base-and-bounds pair in PCB upon context switch
- Dealing with external fragmentation
  - **Compaction** rearranges the existing segments in physical memory. Costly because it stops running process, copy data, and change segment register value.

7.2 Fixed partitions

Physical memory is broken up into fixed sized partitions. Hardware requirement is only base register.

- Physical address = virtual address + base register
- Base register loaded by OS on context switch

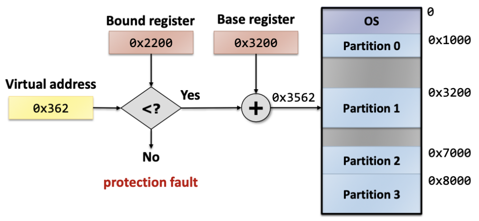


The number of partitions = degree of multiprogramming.

- Pros: Easy to implement, fast context switch
- Cons: Internal *frag*, One partition size cannot fit all

7.3 Base and bound

Variable-sized partitions. Hardware requirements are base register and bound register for protection. Bound register can hold the size (check first) or physical address (addition first).



- Pros: Simple, inexpensive, variable size ➞ less internal *frag*
- Cons: External *frag*, no partial sharing

7.4 Segmentation

Divide address space into logical segments: code, data, stack, etc.

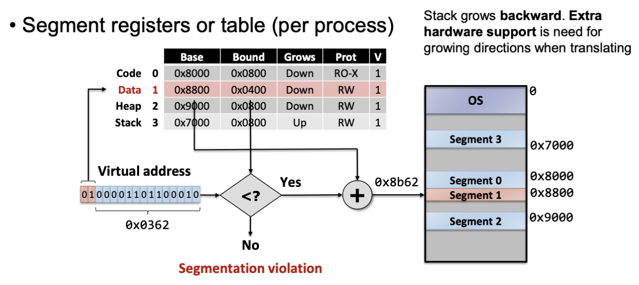
- Each segment are managed independently
  - Placed and grow/shrink separately
  - Protected: separate read/write/execute protection bits
- Users view memory as a collection of base-and-bound regions ➞ Virtual address is <Segment X, Offset>

7.4.1 Addressing

- Explicit: Use a part of virtual address as a segment number, reamaining bits as offset within the segment
- Implicit: determines segment by the type of memory reference

- PC-based addressing: code segment
- SP- or BP-based addressing: stack segment

7.4.2 Implementation


• Segment registers or table (per process)
Stack grows **backward**. Extra **hardware support** is need for growing directions when translating
Base | Bound | Grows | Prot | V
Code 0 | 0x8000 | 0x0800 | Down | RO-X | 1
Data 1 | 0x8800 | 0x0400 | Down | RW | 1
Heap 2 | 0x9000 | 0x0800 | Down | RW | 1
Stack 3 | 0x7000 | 0x0800 | Up | RW | 1
Virtual address
01000011011000010
0x0362
<?
Yes
No
Segmentation violation
+
0x8b62
OS
0
Segment 3
0x7000
Segment 0
0x8000
Segment 1
0x8800
Segment 2
0x9000

7.4.3 Support for sharing

Segment can be shared between address space, e.g. code sharing (shared libraries). To share, the shared segment must have the identical **protection bits**. May support few more bits per segment to indicate permissions of read, write and execute.

7.4.4 Pros

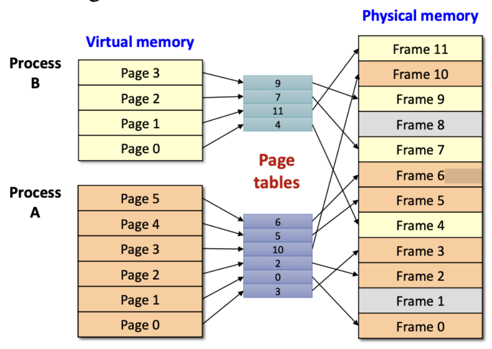
- Enables sparse allocation of address space
- Easy to protect segments: valid bit, segment-level protection bits
- Better sharing than base and bounds
- Supports dynamic relocation of each segment

7.4.5 Cons

- Each segment allocated contiguously → External fragmentation
- Bigger segment table in main memory (Space overhead)
  - Slow to search through → May use hardware cache for speed
- Internal fragmentation: Free objects in the middle of a heap

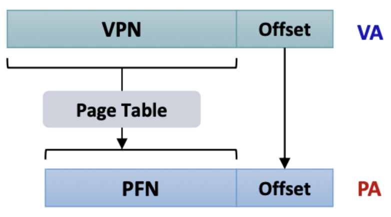
7.5 Paging

Avoid external fragmentation



- Process address space is split into fixed-sized unit called a **page**.
- Physical memory is split into **page frame**.
- Page table per process is needed to translate the virtual address to physical address.

7.5.1 Address translation



Virtual Page Number (VPN) is an index to page table. Page table determines Page Frame Number (PFN). Usually, |VPN| ≥ |PFN|.

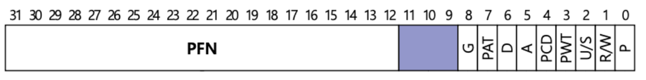
7.5.2 Page tables

- Page tables are managed by OS.
- One Page Table Entry (PTE) per page in virtual address space.
- Page tables can get awfully large. For 32-bit address space with 4KB pages, which means 12 bits for offset, we will need $2^{2}0$ entries. For 4 bytes PTE, it is 4MB per process.

7.5.3 Protection

- Process-level: Separate page table. On context switch, an MMU register is set to point to the base address of the current page table, *e.g.*, CR3 in x86.
- Page-level: Associate protection bits with each PTE.
  - Valid bit indicates that the page is in the process' address space and in use. Invalid means the page is not allocated.
  - Protection bits for valid pages indicate read/etc permissions

7.5.4 Page flags



An x86 Page Table Entry(PTE)

- P (present): in physical memory or on disk (swapped out)
- R/W: read/write bit
- U/S (supervisor): user-mode, kernel-mode
- A (accessed bit): has been accessed since it was brought into memory
- D (dirty bit): has been modified since it was brought into memory

7.5.5 Demand Paging

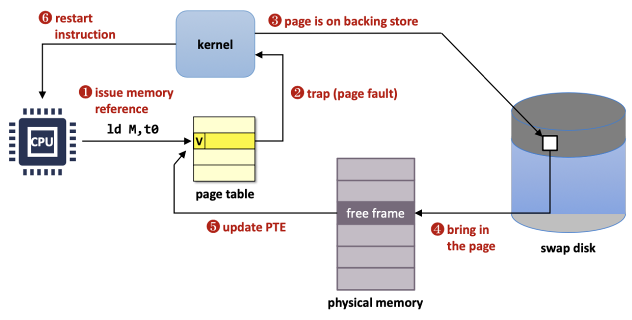
OS uses main memory as a (page) cache of all the data allocated by processes in the system.

- Bring a page into memory only when it is needed
- Pages can be evicted from their physical memory frames
- Evicted pages go to disk (only dirty pages are written)
- Movement of pages is transparent to processes

Thus less I/O and memory needed, faster response, and can accommodate more processes.

7.5.6 Page fault

An exception raised by CPU when accessing invalid PTE.



- Major: page is valid but not loaded into memory
  - OS maintains information on where to find the contents
  - Require disk I/Os
- Minor: can be resolved without disk I/O
  - Used for lazy allocation, *e.g.* accesses to stack/heap pages
  - Accesses to prefetched pages, etc.
- Invalid: Segmentation violation, the page is not in use

7.5.7 Pros

- No external fragmentation
- Fast to allocate and free
  - Allocation: Maintain a list or bitmap for free page frames → no need to find contiguous free space
  - Free: no need to coalesce with adjacent free space
- Easy to *page out* portions of memory to disk

  - Page size is chosen to be a multiple of disk block sizes
  - Use valid bit to detect reference to paged-out pages
  - Can run process when some pages are on disk
- Easy to protect and share pages

7.5.8 Cons

- Internal fragmentation
- Memory reference overhead: requires one extra memory reference to first fetch the translation from the page table
- Storage needed for page tables → Advanced table

Additional `AccessMemory`

```
VPN = (vaddr & VPN_MASK) >> SHIFT
pteaddr = PTBR + (VPN * sizeof(PTE))
PTE = AccessMemory(pteaddr)
if (PTE.valid)
    offset = vaddr & OFFSET_MASK
    paddr = (PTE.PFN << PFN_SHIFT) | offset
    reg = AccessMemory(paddr)
```

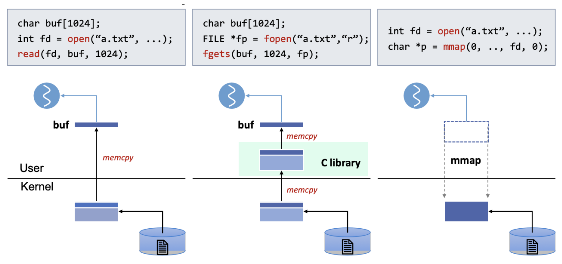
7.6 Memory mapping

`mmap` system call creates a new mapping in the virtual address space of the calling process. This maps files or devices into memory, which often enables efficient file access.

```
void *mmap(void *ptr, size_t length, int prot, int flags,
     int fd, off_t offset)
```

- `addr`: the starting address for the new mapping (should be aligned to page boundary)
  - If `NULL`, the kernel chooses the address
  - Otherwise, kernel takes it as a hint about where to place mapping
- `length`: the length of the mapping
- `prot`: protection info, *e.g.* `PROT_EXEC, PROT_READ`
- `flags`: mapping flags, *e.g.* `MAP_PRIVATE, MAP_SHARED`
- `fd/offset`: file descriptor/offset used for file mapping

7.6.1 Comparisons with file I/O



```
char buf[1024];
int fd = open("a.txt", ...);
read(fd, buf, 1024);
```

```
char buf[1024];
FILE *fp = fopen("a.txt","r");
fgets(buf, 1024, fp);
```

```
int fd = open("a.txt", ...);
char *p = mmap(0, .., fd, 0);
```

- Read system call: data is read from the file on disk into a kernel buffer, then copied to the user-space buffer `buf` through `memcpy`
- Standard I/O: same but additional copy to C library's buffer
- Memory mapping: does not directly copy data

7.6.2 Implementation

- Initially, all pages in mapped region are marked as invalid
- OS reads a page from file whenever invalid page is accessed
- PTEs map virtual addresses to page frames holding file data
- <`vaddr base + n`> refers to `offset + n` in file

Several processes can map the same backing store in their own virtual address space.

- `MAP_SHARED`: Modifications to shared pages are visible to all involved processes. OS writes to a page and it is written to the file when evicted from physical memory
- `MAP_PRIVATE`: Modifications are not visible to other processes. File is not modified. Copy-On-Write (COW) which defer copy:
  - Create shared mappings to same page frames in physical memory
  - Shared pages are protected as read-only
  - When write, allocate new space in physical memory and write to it
  - Usage: `fork()`, allocating data and heap pages, etc. Efficient when child calls `exec()` immediately after `fork()`

7.6.3 Pros

- Uniform access for files and memory: both use pointers
- Less memory copying
- Several processes can map the same file → Shared memory

7.6.4 Cons

- Process has less control over data movement
- Does not generalize to streamed I/O (pipes, sockets, etc.)

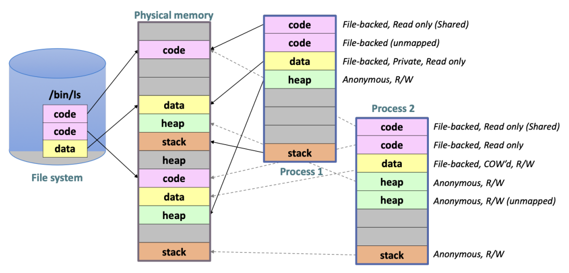
7.6.5 Shared memory example

Allows (unrelated) processes to share data using direct memory reference. PTE maps to the same physical frame, but may have different protection values. Must update all PTEs when page becomes invalid.

```
int fd = shm_open("/shm1", O_CREAT | O_EXCL | O_RDWR,
    0600);
ftruncate(fd, 4096); // set shmem size because a new
    shared memory object has a size of 0 bytes
int *p = (int *) mmap(0, 4096, PROT_READ | PROT_WRITE,
    MAP_SHARED, fd, 0);
for (int i = 0; i < 1024; i++) p[i] = i;
close(fd);
```

If we map shared memory at

- Different virtual address: flexible (no address space conflicts), but pointers inside the shared memory are invalid
- Same virtual address: less flexible, but shared pointers are valid



7.6.6 Memory trace example

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  1024
```

10 memory accesses per loop = 4 instruction fetches + 1 explicit update of memory (`movl`) + 5 page table accesses to translate those

8 Advanced page tables

↔ Linear page table

Most of the page table is unused, full of invalid entries.

8.1 Paging with segmentation

Divide virtual address space into variable-length segments. Divide each segment into fixed-sized pages.

- Each segment has a page table
- Segment table: tracks base (physical address) and limit (number of PTEs) of the page table for each segment → must be changed on context switch

8.1.1 Pros

- Reduce the memory consumption for page table
- Segments can grow without any reshuffling
- Increases flexibility of sharing: share either page or segment

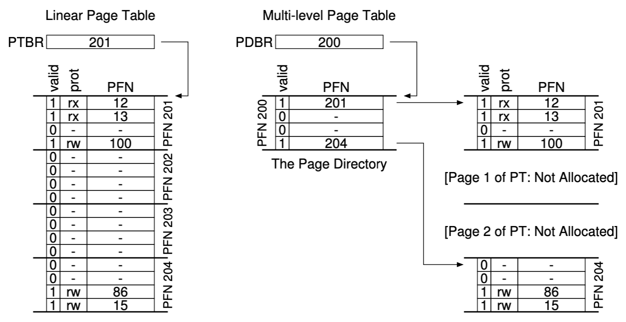
8.1.2 Cons

- Page tables can be still large, *e.g.* large but sparse-used heaps
- External fragmentation due to arbitrary-sized page tables: finding contiguous space for them would be complicated

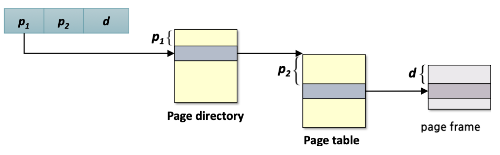
8.2 Multi-level page table

Consider page table as tree of page-sized units.

- If entire page of PTEs is invalid, don't allocate that page at all.
- Page directory (PD): Track the validity of page of PTEs
- Allow each page table to be allocated non-contiguously



8.2.1 Examples



Consider 30-bit virtual address with 21-bit VPN and 9-bit offset, thus 512B page size. For 4B PTE, it's $128 = 2^7$ PTEs per page.

- Two-level: We can have $2^{21-7} = 2^{14}$ PD entries
- Three-level: $2^{21-7-7} = 2^7$ PD0 entries and 2^7 PD1 entries

Intel x86_32 use 32-bit virtual address with 12-bit offset, thus 4KB page, and 4B PTE. → 2^{10} PTEs per page → $2^{32-12-10} = 2^{10}$ PDEs.

Intel x86_64 use 48-bit virtual address with 12-bit offset, thus 4KB page, and 8B PTE → 2^9 PTEs per page. 36-bits are split to 4 levels: PML4, PDPT, PD, PT. Each entry value is 40-bits, so physical address is 40-bit + 12-bit offset = 52-bit.

8.2.2 Pros

- Compact while supporting sparse-address space
- Page-table space in proportion to the amount of address space actually used
- Easier to manage physical memory
- Each page table usually fits within a page
- Easier for hardware to walk though page tables
- No external fragmentation

8.2.3 Cons

- More memory accesses for address translation
- More complex hardware than linear page-table

9 Translation lookaside buffer (TLB)

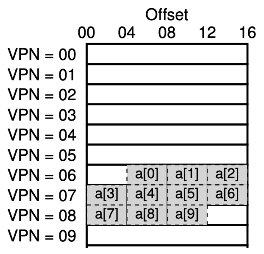
For each memory reference, page tables adds up memory lookups equal to its level. **TLB** is part of MMU and is a **hardware cache** of popular VPN-**PTE** translation (usually PTE not just PFN).

- Small: typically 16~256 entries
- Usually fully associative: all entries looked up in parallel
  - May be set associative to reduce latency
- Replacement policy: LRU (Least Recently Used)
- Performance metric: hit rate, lookup latency, etc

9.1 Locality

- Temporal: an instruction or data item that has been recently accessed will likely be re-accessed soon
- Spatial: if a program accesses memory, it will likely soon access memory near that

9.1.1 Spatial locality: Array example



3 misses and 7 hits → TLB hit rate is 70%

9.2 Handling TLB miss

Software-managed TLB:

- CPU traps into OS upon TLB miss
- OS lookup page table and loads right PTE into TLB
- CPU ISA has (privileged) instructions for TLB manipulation
- Page tables can be in any format convenient for OS (flexible)

Hardware-managed TLB:

- CPU knows where page tables are in memory *e.g.* CR3 register
- MMU *walks* the page table and loads right PTE into TLB
  - OS has already set up the page tables so that hardware can access it directly → OS is not involved in this step
- Page tables have to be in hardware-defined format

At this point, there is a valid PTE for the address in the TLB. TLB restarts translation.

9.3 TLB on context switches

- Flush TLB on each context switch
  - Hardware-managed: automatically flush when PTBR is changed
  - Some architectures support the pinning of pages into TLB
- Track which entries are for which process
  - Tag each TLB entry with an ASID (Address Space ID)
  - A privileged register holds the ASID of the current process

9.4 TLB on multi-core

Each core typically has its own TLB. TLBs of different cores must be coherent. Page table change may leave stale entry (no longer valid) in TLB, and flushing the local TLB is not enough. TLB coherence is not maintained by hardware, so OS should.

→ TLB shootdown

- Initiating core sends IPI (Inter-Processor Interrupt) to remote cores
- Remote cores invalidate their TLBs (may flush entire TLB)
- IPI may take several hundreds of cycles (heavy)

9.5 Implications

Modern applications have large memory footprint and low memory access locality. TLB coverage should scale with memory size.

TLB coverage = # TLB entries × Page size

- Superpage: larger page *e.g.* 2MB with more offset bits and less level
  - TLB handles page and superpage in same way
  - Should allocated contiguously → internal fragmentation
- Increase # entries

9.5.1 Multi-level TLBs

Access the fastest first level (L1) TLB first. If miss, then check the second level (L2), which is slightly slower but has a larger capacity.

For example Intel Haswell has L1 ITLB with 128 entries (4-way set associative), L1 DTLB with 64-entries (4-way), and L2 STLB with 1024 entries (8-way).

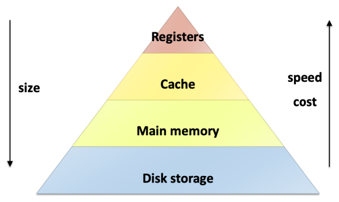
10 Swapping

10.1 Introduction

Process only uses small amount of address space at a moment. Memory reference within processes also follows locality.

→ Consider physical memory as a cache for disks

Support processes when not enough physical memory. Key of *abstraction* is that user program should be independent of the amount of physical memory.



Memory hierarchy: Each layer acts as *backing store* for layer above.

10.1.1 Swap design

- Memory overlays (old systems): programmers manually move pieces of code/data in/out of memory as wanted
- Process-level swapping
- **Page-level swapping**

10.1.2 Swap space

Disk space reserved for moving pages back and forth.

- Swap size = *max* size of memory that can be in use
- Block size is same as the page size
- Can be a dedicated partition or a file in the file system

10.2 Page eviction mechanism

Swap means allocating new swap space and *write* the page to it. *Drop* means just removing it from memory.

- Code: *drop*–can recover it from the executable file on disk
- Data: *swap* if modified (dirty) page, *drop* if unmodified (clean)
- Clean anonymous: *drop* if it has previously been swapped or is zero page, otherwise *swap*
- Dirty anonymous: *write* if it has previously been swapped, otherwise *swap*

10.3 Swap policy

- Lazy: waits until memory is entirely full (unrealistic)
- Threshold-based: OS wants to keep a small portion of memory free
  - 2 thresholds: HW (high watermark), LW (low watermark)
  - Swap daemon (or page daemon): background thread responsible for freeing memory, *e.g.* `kswapd` in Linux
    - If # free pages < LW, swap daemon evicts
    - If # free pages > HW, swap daemon sleeps

10.4 Page-replacement policy

10.4.1 Goal

AMAT (Average Memory Access Time)

$$AMAT = P_{\text{Hit}} \times T_M + P_{\text{Miss}} \times T_D$$

$P_{\text{Hit}}, P_{\text{Miss}}$ indicate hit and miss rate, and T_M, T_D indicate the cost of accessing memory and disk. Miss penalty T_D is so high ($>\times100{,}000$), so goal is to minimize P_{Miss}.

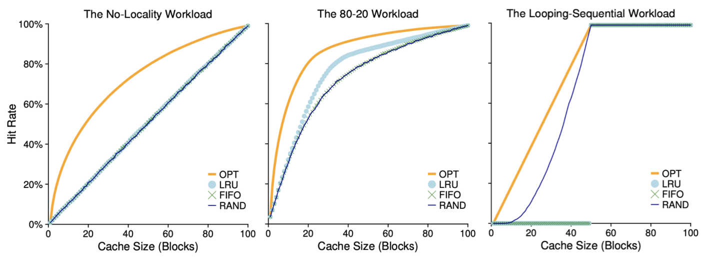
10.4.2 Policies

Replace the page that

- Optimal (baseline): will not be used for longest time in future.
- FIFO: has been in memory the longest
  - Locality: some pages may always be needed. Limitations:
  - Belady's anomaly: fault rate often increases given more memory
- Random
- Least recently used (LRU)
- Least frequently used (LFU)

10.4.3 Workload examples

- No locality: Random
- 80-20: 80% of the reference are made to 20% of the page
- Looping sequential: Refer to 50 pages in sequence repeatedly



- If cache is large enough to fit entire workload, policy doesn't matters
- LRU is worst in looping workload

10.4.4 Clock: approximated LRU

Implementing true LRU require update in usage list for every access → expensive, complex hardware support

Clock only requires a **use** bit for hardware support

- When page is referenced, use bit ← 1 via hardware
- Hardware never clears the bit; it is responsibility of OS

```
Require: Pages arranged in a circular list.
Require: hand ← some page to begin with
  while use bit of hand is 1 do
      Clear bit
  Evict hand
```

Clock < LRU in 80-20 workload but > FIFO, RAND.

10.5 Advanced ideas

- **Prefetching**: OS guess soon-to-used page and bring in ahead of time
- **Clustering/Grouping**: Collect a number of pending writes together in memory and write them to disk in one write.
  - Single large write is more efficient than many small ones
- **Thrashing**: phenomenon when memory is oversubscribed, *i.e.* physical memory is not enough to hold all the working sets (pages) of processes → Most of time is spent by paging in/out
  - Possible solution: Decide not to run some processes

11 Synchronization

11.1 Introduction

The execution of the two threads can be interleaved, assuming preemptive scheduling. Concurrent threads accessing a shared resource create a race condition.

- Output of the program is nondeterministic
- Hard to debug (*Heisenbugs*)

→ Synchronization mechanisms to control access to shared resources.

- Synchronization restricts the concurrency
- Scheduling is not under programmer's control

11.1.1 Shared resources among threads

	Stored in	Shared
Local variables	Data on stack	✗ (has own stack)
Global variables	Static data segment	✓
Dynamic objects	Heap	✓ (through pointer)

Also, processes can share memory through `shmem`.

11.1.2 Critical section

A piece of code that accesses a shared resource, usually a variable or data structure. Needs **mutual exclusion**:

- All-or-nothing: Execute the critical section atomically
- Only one thread at a time can execute in the critical section
- All other threads are forced to wait on entry
- When a thread leaves a critical section, another can enter

11.1.3 Requirements

- **Correctness**
  - Mutual exclusion
  - Progress (deadlock-free): must allow one to proceed
  - Bounded waiting (starvation-free): must eventually allow all waiting thread to enter
- **Fairness**: Each thread gets a fair chance at acquiring the lock
- **Performance**: Time overhead both w/o and with contentions

11.2 Interrupt disabling

On a single-core processor, two threads cannot disable interrupts simultaneously. But inappropriate as synchronization method:

- Application can monopolize processor → Only available to kernel
- Does not work on multiprocessors
- When critical section is long, important interrupts can be delayed or lost, *e.g.* timer, disks
- Slower than executing atomic instructions on modern CPUs

11.3 Locks

A lock is an object (in memory) that provides mutual exclusion with the following two operations:

- `acquire()`: wait until lock is free, then grab it
  - Does not return until the caller holds the lock
  - Thread can spin (spinlock) or block (mutex)
  - Spinlocks are horribly wasteful (primitive)
- `release()`: unlock and wake up any thread waiting in `acquire()`

Call `acquire()` before entering a critical section, and `release()` after leaving it.

- Lock is initially free
- At most one thread can hold a lock at a time

11.3.1 Software-only solutions

```
struct lock { int held = 0; }
void acquire(struct lock *l) {
    while (l->held);
    l->held = 1; }
void release(struct lock *l) {
    l->held = 0; }
```

- Incorrect: between check and set, another thread can acquire lock
- Performance: endlessly checks value of flag (spin-wait) → waste

Peterson's algorithm: solution for 2 processes

Assume that each load and store instruction is atomic.

```
int turn;
int interested[2];
void acquire(int process) {
    int other = 1 - process;
    interested[process] = 1;
    turn = other;
    while (interested[other] && (turn==process); }
void release(int process) {
    interested[process] = 0; }
```

11.3.2 Hardware support of atomic instructions

Test-and-set (`xchg` in x86)

```
int TestAndSet(int *v, int new) {
    int old = *v;
    *v = new;
    return old; }
void acquire(struct lock *l) {
    while (TestAndSet(&l->held, 1)); }
```

Compare-and-swap (`cmpxchg` in x86)

```
int CompareAndSwap(int *v, int expected, int new) {
    int old = *v;
    if (old == expected) *v = new;
    return old; }
void acquire(struct lock *l) {
    while (CompareAndSwap(&l->held, 0, 1)); }
```

Fetch-and-add (`xadd` in x86)

```
int FetchAndAdd(int *v, int a) {
    int old = *v;
    *v = old + a;
    return old; }
```

We implement ticket locks which provide bounded waiting.

```
struct lock { int ticket = 0; int turn = 0; };
void acquire(struct lock *l) {
    int myturn = FetchAndAdd(&l->ticket, 1);
    while (l->turn != myturn); }
void release(struct lock *l) {
    l->turn = l->turn + 1; }
```

11.4 Condition variable

- Provide a mechanism to wait for events
- Used with **mutexes**, a blocking lock:
  - Threads are blocked when it is held by another thread
  - Manipulating some condition related to a CV should be done inside the critical section
- Memoryless: no internal states other than queue of waiting thread
  - No memory of earlier calls of signal or broadcast

11.4.1 Operations

- `cond_wait(cond_t *cv, mutex_t *mutex)`
  - Assumes mutex is held when it is called
  - Puts the caller to sleep and releases mutex (atomically)
  - When awoken, reacquires mutex before returning
- `cond_signal(cond_t *cv)`: wakes a single thread
- `cond_broadcast(cond_t *cv)`: wakes all waiting threads

11.4.2 Joining threads

```
mutex_t m, cond_t c;
int done = 0;
void *child(void *arg) {
    thread_exit(); return NULL; }
int main(int argc, char *argv[]) {
    thread_t p;
    thread_create(&p, NULL, child, NULL);
    thread_join();
    return 0; }
void thr_exit() {
  mutex_lock(&m);
  done = 1;
  cond_signal(&c);
  mutex_unlock(&m); }
void thr_join() {
  mutex_lock(&m);
  while (done == 0) cond_wait(&c, &m);
  mutex_unlock(&m); }
```

⚠ w/o `done`: child signals before parent waits → parent stuck forever

⚠ `if` instead of `while`: condition may change after being signalled and before resuming execution

11.4.3 Bounded buffer (producer/consumer) problem

There is a set of resource buffers shared by producers and consumers. Producer inserts, consumer removes resources. Wake up the consumer thread when an item is ready.

```
struct item buf[MAX];
int front=0, tail=0, loop = MAX_LOOP;
mutext_t m; cond_t full, empty;
void *producer(void *arg) {
    for(i=0;i<loop;i++) put(i); }
void *consumer(void *arg) {
    for(j=0;j<loop;j++){
        int tmp = get(); printf("%d\n", tmp); } }
void put(data) {
    mutex_lock(&m);
    while (tail-front == MAX) cond_wait(&full, &m);
    tail = (tail+1) % MAX; buf[tail] = data;
    cond_signal(&empty);
    mutex_unlock(&m); }
void get() {
    mutex_lock(&m);
    while (front == tail) cond_wait(&empty, &m);
    front = (front+1) % MAX; item = buf[front];
    cond_signal(&full);
    mutex_unlock(&m);
    return item; }
```

⚠ Single CV instead of `full, empty`: consumer could wake other consumers, and vice-versa

11.5 Semaphores

A synchronization primitive higher level than locks

- Does not require busy waiting
- Object with an integer value (state)
- State cannot be directly accessed by user program, but determines the behavior of semaphore operations

11.5.1 Operations

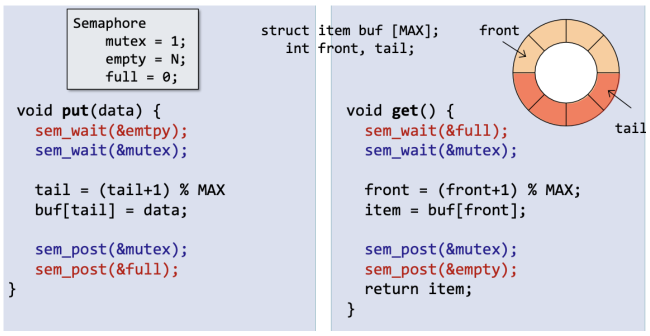
- `sem_wait()`: decrement value and wait until value ≥ 0, *a.k.a.* `down(), P(), wait()`
- `sem_post()`: increment value then wake up a single waiter, *a.k.a.* `up(), V(), signal()`

11.5.2 Use cases

Value initialized to

- 1: Binary semaphore = mutex ≃ lock
- 0: CV (but unlike CV, semaphore has memory)
- *N*: Counting semaphore
  - Represents a resource with many units available

11.5.3 Bounded buffer problem with semaphore



```
Semaphore
    mutex = 1;
    empty = N;
    full = 0;

void put(data) {
  sem_wait(&emtpy);
  sem_wait(&mutex);

  tail = (tail+1) % MAX
  buf[tail] = data;

  sem_post(&mutex);
  sem_post(&full);
}
```

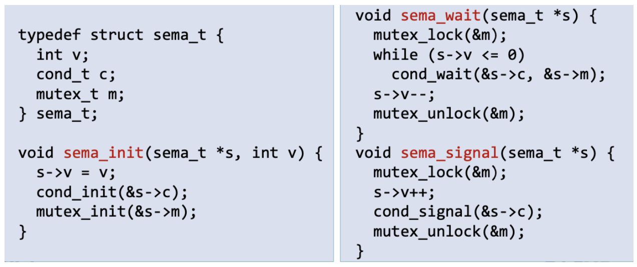
```
struct item buf [MAX];
  int front, tail;

void get() {
  sem_wait(&full);
  sem_wait(&mutex);

  front = (front+1) % MAX;
  item = buf[front];

  sem_post(&mutex);
  sem_post(&empty);
  return item;
}
```

Semaphores and CVs with mutexes have same expressive power. Implementing semaphores using CVs and mutexes:



```
typedef struct sema_t {
  int v;
  cond_t c;
  mutex_t m;
} sema_t;

void sema_init(sema_t *s, int v) {
  s->v = v;
  cond_init(&s->c);
  mutex_init(&s->m);
}
```

```
void sema_wait(sema_t *s) {
  mutex_lock(&m);
  while (s->v <= 0)
    cond_wait(&s->c, &s->m);
  s->v--;
  mutex_unlock(&m);
}
void sema_signal(sema_t *s) {
  mutex_lock(&m);
  s->v++;
  cond_signal(&s->c);
  mutex_unlock(&m);
}
```

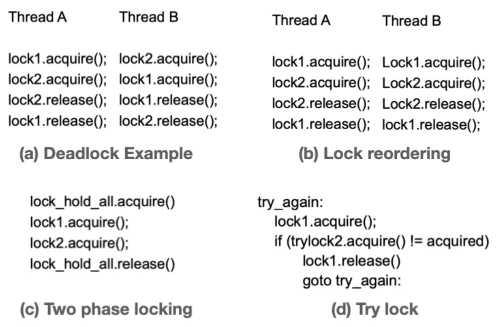
11.6 Bugs

11.6.1 Deadlock

- **Resource**: any (passive) thing needed by a thread to do its job
  - CPU, disk space, memory, lock
  - May preemptable, *i.e.* can be taken away by OS, or not
- **Starvation**: thread waits indefinitely

Conditions of deadlock (**all** 3 are necessary):

1. Circular waiting
2. Wait while holding
3. No preemption



```
Thread A            Thread B

lock1.acquire();    lock2.acquire();
lock2.acquire();    lock1.acquire();
lock2.release();    lock1.release();
lock1.release();    lock2.release();
```

(a) Deadlock Example

```
Thread A            Thread B

lock1.acquire();    Lock1.acquire();
lock2.acquire();    Lock2.acquire();
lock2.release();    Lock2.release();
lock1.release();    lock1.release();
```

(b) Lock reordering

```
lock_hold_all.acquire()
lock1.acquire();
lock2.acquire();
lock_hold_all.release()
```

(c) Two phase locking

```
try_again:
  lock1.acquire();
  if (trylock2.acquire() != acquired)
      lock1.release()
      goto try_again:
```

(d) Try lock

Pro-active approaches: make the conditions not occur

1. Lock ordering: always acquire locks in a fixed order
2. Two phase locking: atomically hold all locks
   ⚠ Reduced concurrency, risk of starvation
3. `trylock()`: instead of blocking, return immediately with error
   - If fails to acquire `lock2`, thread releases `lock1` and retries entire sequence → ensures thread holding `lock2` can proceed

   ⚠ Busy waiting, risk of starvation, performance overhead
   ⚠ Livelock: threads continuously change their state in response to each other w/o making any actual progress

Reactive approaches: when deadlock happens, do a corrective action

- Abort processes: all deadlocked or one at a time until cycle is eliminated (needs to rerun detection after each abort)
- Preempt resources: force their release
  - Need to select process and resource, *e.g.*, a lock to preempt
  - Need to rollback process to previous state
  - Need to prevent starvation

11.6.2 Other real-world bugs

- Atomicity violation
- Order violation: operations in multiple threads done in wrong order

→ Use locks/mutexes properly

12 Storage

12.1 Modern system architecture

- CPUs connected by QPI links
- CPU and DRAM connected by memory bus
- I/O devices (NVMe, SSD, NTC) connected to CPU via PCIe bus

12.2 Typical I/O device

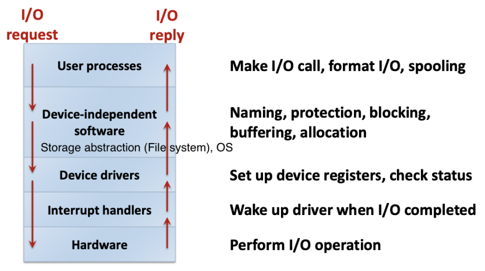
3 main interfaces allow CPU to communicate with I/O devices:

- **Control**
  - Special instructions: direct CPU instructions
  - Memory-mapped I/O: device control registers mapped into system memory
- **Data transfer**
  - Programmed I/O (PIO): CPU-managed read/write operations
  - Direct Memory Access (DMA): bypass CPU
- **Status check**
  - Polling: CPU actively checks device status register
  - Interrupts: device signals CPU upon status change/completion

Can be classified to:

- **Block** device: stores information in fixed-size (typically 512B or 4KB) blocks, each with its own address
  - Can read or write each block independently
  - Disks, tapes, etc.
- **Character** device: delivers/accepts a stream of characters (bytes)
  - Not addressable and no seek operation supported
  - Printers, networks, mouse, keyboard

12.3 I/O stack


I/O request
I/O reply
User processes
Make I/O call, format I/O, spooling
Device-independent software
Storage abstraction (File system), OS
Naming, protection, blocking, buffering, allocation
Device drivers
Set up device registers, check status
Interrupt handlers
Wake up driver when I/O completed
Hardware
Perform I/O operation

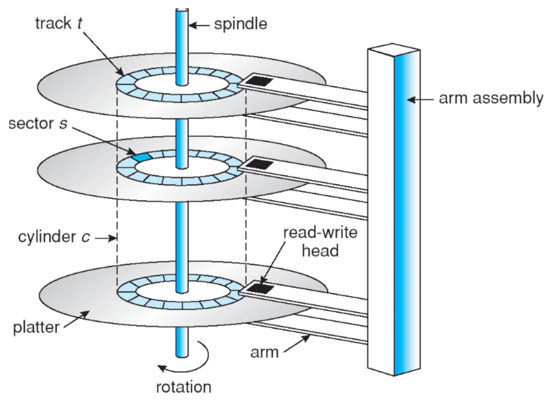
12.4 Device drivers

- Device-specific code to control each I/O device
- Many devices → Provides abstraction
- Implementation
  - Statically linked with the kernel
  - Selectively loaded into the system during boot time
  - Dynamically loaded into the system during execution

12.5 Hard disk drive (HDD)

- Secondary storage: anything other than primary memory (DRAM)
- Does not permit direct execution of instructions or data retrieval via machine load/instructions *e.g.*`movl`
- Abstracted as an array of sectors (typically 512B or 4KB)
- Large (>100GB), cheap, persistent, slow–*ms* (↔ 100*ns* of DRAM)
- No write amplification, media does not wear down (↔ SDD)

12.5.1 Architecture



How to retrieve data?

- Mechanical: Put arm in specific track and rotate disk so that head of arm locate in specific sector (slow)
- Electronics: Put data of sector to disk buffer then DMA to DRAM

12.5.2 Interfacing with HDDs

- Cylinder-Head-Sector (CHS) scheme
  - Each block is addressed by <Cylinder #, Head #, Sector #>
  - OS needs to know all disk geometry parameters → complex

→ Logical block addressing (LBA) scheme
  - Disk is abstracted as a logical array of blocks
  - Address a block with a logical block address (LBA)
  - Hardware disk maps an LBA to its physical location

  → Physical parameters of a disk are hidden from OS

12.5.3 Performance factors

- Seek time T_{seek}: move disk arm to correct cylinder
  - Depends on the cylinder distance (not purely linear cost)
  - Average seek time is roughly one-third of the full seek time
- Rotational delay T_{rotation}: rotate sector under head
  - Depends on rotations per minute (RPM)
  - 5400, 7200 RPM are common, 10K or 15K RPM for servers
- Transfer time T_{transfer}: surface → disk controller → host

$T_{\text{seek}} > T_{\text{rotation}}$ (mechanical) » T_{transfer} (electronics)

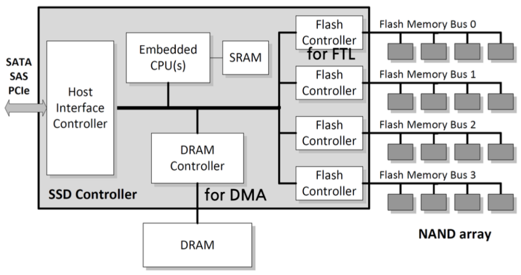
→ Sequential accesses are much better than random
- Distant LBAs lead to longer seek time

12.5.4 Disk scheduling

Reorder a stream of I/O requests to reduce disk head movement

- Work conserving schedulers
- Non-work-conserving schedulers: often wait if system anticipates another request will arrive

12.6 Solid-state drive (SSD)

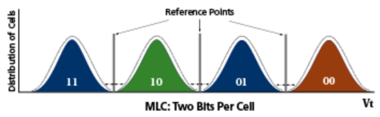


- Collection of blocks, each having number of pages (↔ sector of HDD)
  - Page size depends on vendor but it is getting larger
- Embedded CPU (and SRAM): manages logical to physical address mapping, wear leveling, etc
- ↔ HDD: Faster sequential read/write, very faster random read-/write, higher cost-per-capacity

12.6.1 NAND flash

Page consist of flash cells, a minimum unit containing information. Classified by number of reference points which decide the bit.

	bit/cell
SLC (single level)	1
MLC (multi level)	2
TLC (triple level)	3



Higher level, larger capacity, lower reliability (bit error more likely)

12.6.2 Constraints

- No in-place update ➞ Sector remapping (address mapping)
- Bit errors ➞ Use of error correction codes (ECCs)
- Bad blocks ➞ Bad block remapping
- Limited program/erase cycles ➞ Wear-leveling

12.6.3 Erase-before-write

Three operations: read, write/program (1➞ 0), and erase (0➞ 1). To update a cell, the entire block containing that cell must be erased before it can be rewritten. Program unit is page but erase unit is multiple pages called *erase block*.

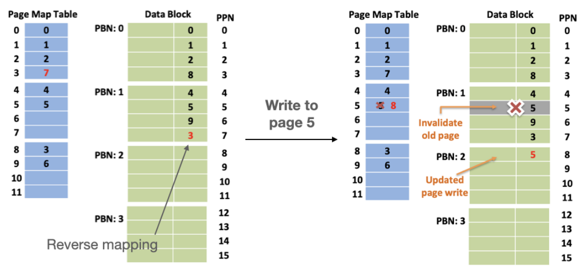
12.6.4 Flash translation layer (FTL)

Previously implemented in OS (software), now in hardware

- Translate LBA ➞ physical address
- Translate read/write ➞ read/write/erase

12.6.5 Address mapping

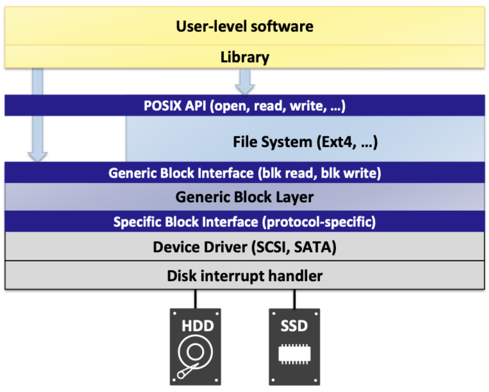
Required since flash pages cannot be overwritten



12.6.6 SSD support in OS

- Turn off *defragmentation* for SSDs
- Simpler I/O scheduler
- Align file system partition with SSD layout
- Flash-aware file systems, *e.g.* F2FS in Linux
- Larger block size (4KB)

13 File system


User-level software
Library
POSIX API (open, read, write, ...)
File System (Ext4, ...)
Generic Block Interface (blk read, blk write)
Generic Block Layer
Specific Block Interface (protocol-specific)
Device Driver (SCSI, SATA)
Disk interrupt handler
HDD
SSD

13.1 Introduction

Storage is abstracted as logical array of blocks. Block interface:

- `identify`: returns N
- `read`(start sector #, # of sectors, buffer addresses)
- `write`(start sector #, # of sectors, buffer addresses)

A file system provides a higher-level abstraction over LBAs by organizing data into files and directories. It maps <filename, data, metadata> ➞ <set of blocks>

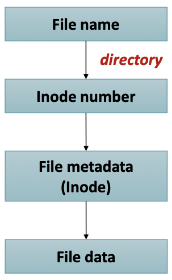
13.2 File

File is an abstraction of set of blocks.

- Named collection of related info recorded on persistent storage
  - Content (data): sequence of bytes
  - Inode: metdata or file attributes
    - ∗ File size, block locations, owner, access control lists, etc
    - ∗ Associated inode number (internal file ID)
  - Name: full pathname from the root specifies a file
- Level of indirection: inode ➞ blocks

Directory provides a structured way to organize files.

- A special file used to map a user-readable file name to its inode number: a list of <file name, inode number>
- **Hierarchical directory tree**: directories can be placed within other directories



13.3 Interface

```
int open(char *pathname, int flags, mode_t mode);
int creat(char *pathname, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int close(int fd);
int fsync(int fd);
int rename(char *oldpath, char *newpath);
int unlink(char *pathname);
int stat(char *path, struct stat *buf);
int link(char *oldpath, char *newpath);
int symlink(char *oldpath, char *newpath);
int mount(char *source, char *target, char *fstype,
          unsigned long mountflags, void *data);
int umount(char *target);
```

13.3.1 Pathname translation

- Open directory → Search entry → Get location
- Permissions are checked at each step
- Spends a lot of time walking down directory paths → OS caches prefix lookups to enhance performance

13.3.2 File system mounting

A file system must be mounted before it can be available to processes on the system.

- Windows: to drive letters, *e.g.* C:, D:
- Unix: to an existing empty directory *root* (/)
  - Different file systems can be mounted in the same tree
  - Forms a large, single directory tree

13.3.3 Hard vs. symbolic links

- Hard link: `ln old.txt new.txt`
  - Both pathnames use the same inode number
  - Cannot tell which name was the *original*
  - Inode maintains the number of hard links
  - Deleting (unlinking) a file decreases the link count
  - Inode is removed only when the link count becomes 0
  - Does not work across a file system boundary
- Symbolic (or soft) link: `ln -s old.txt new.txt`
  - Reference to another file or directory in form of absolute or relative pathname, *e.g.*, shortcut in Windows
  - Different file type
  - Subject to the dangling reference

13.3.4 Ensuring persistence

FS uses memory (page cache) to buffer writes for performance

- `fsync()` flushes all dirty data including metadata to disk, and tells disk to flush its write cache to the media too
- `fdatasync()` does not flush modified metadata

Periodic flush or `fsync()` call

13.3.5 Prefetch (read ahead)

FS preloads data blocks to the file cache. Can overlap this (I/O) with execution on previous block.

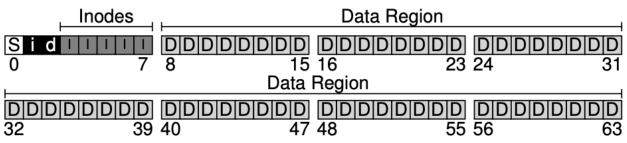
13.3.6 Consistency/Sharing Semantics

Semantics	Write to open file is visible
Unix	Immediately
AFS ACL	Once a file is closed
Immutable-shared-files	Cannot modify shared file

13.4 Implementation: VSFS

Divide the disk into blocks, multiple of sector size

- Data region: store user data (most of blocks)
- Inode table: store inode (file metadata)
  - Inode is fix-sized, inode table size determines the max # fies
- Data/inode bitmap: each bit indicates whether corresponding block/inode is free (0) or in-use (1)
- Superblock: information about this particular FS
  - FS type, block size, # of blocks/inodes/bitmap blocks
  - When mounting FS, OS will read superblock first



13.4.1 Allocation strategies

1. The amount of fragmentation (mostly external)
2. Ability to grow file over time
3. Performance of sequential accesses
4. Speed to find data blocks for random accesses
5. Metadata overhead to track data blocks

	Contiguous	Linked	FAT	Indexed	Multi-level	Extent
Meta	#, length	#	#, FAT	Array of *		* to index
1	Severe	No				Okay
2	Hard	Easy				Easy
3	Great	Depends on data layout				Good
4	Great	Poor	Better	Better	Okay	Okay
5	Little	Yes (*)	Yes (FAT)	Large	Less	Small

# indicate starting block

- **Contiguous**: allocate each file to contiguous blocks
- **Linked**: each block contains pointer to next block
- **File allocation table (FAT)**: keep link info in on-disk FAT
  - Cached in main memory to avoid disk seeks
  - Cannot scale with large FS
- **Indexed**: allocate blocks and keep pointers in index block
  → wasted space for unneeded pointers (-1)
- **Multi-level indexing**: metadata contain few direct & indirect pointer
  - Direct: point to data blocks directly
  - Indirect: point to data block containing pointers (may multi level)
    → Does not waste space for unneeded pointers
  - ⚠ Need to read indirect blocks of pointers to calculate addresses (extra disk read)
  - → Keep indirect blocks cached in main memory
  - VSFS inode has 12 direct pointers and 1 indirect pointer
    - 4B disk address → 1024 pointers per 4KB block
    - Max file size = (12 + 1024) × 4KB = 4144KB
- **Extent-based**: multiple contiguous regions (extents) per file (B+tree)
  - Index node points multiple leaf nodes which contain #, extent size

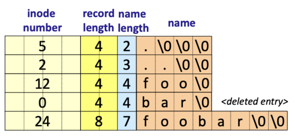
13.4.2 Directory organization

- Large directories just use multiple data blocks
- Use bits in inode to distinguish directories from files

Organization varies:

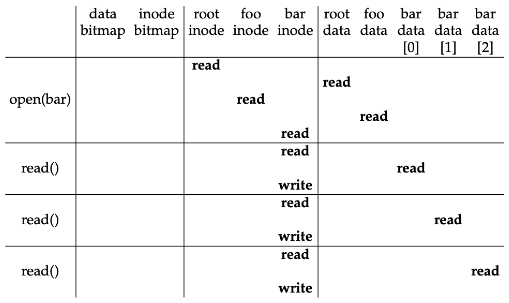
- Table (fixed length entries) or linear list: Linear search to find entry
- Tree: may sort entries to decrease the average search time and to produce a sorted directory listing easily
- Hash table: fast, should be scalable as # files increases (∵ collision)

VSFS organize directory as linear list of <file name, inode number>



inode number	record length	name length	name								
5	4	2	.	\0	\0	\0					
2	4	3	.	.	\0	\0					
12	4	4	f	o	o	\0					
0	4	4	b	a	r	\0	<deleted entry>				
24	8	7	f	o	o	b	a	r	\0	\0	

13.4.3 Read

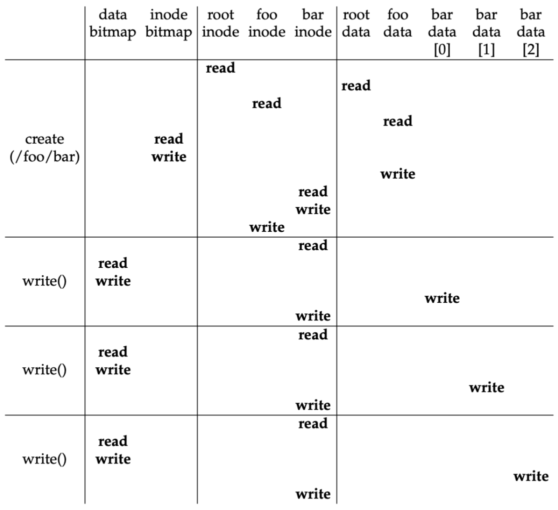


	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

Figure 40.3: **File Read Timeline (Time Increasing Downward)**

`read()` check if block is in cache. If not, read from disk, insert into cache, return to user.

13.4.4 Write



	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)			read							
						read				
				read						
							read			
		read write								
							write			
					read write					
				write						
write()					read					
	read write									
								write		
					write					
write()					read					
	read write									
									write	
					write					
write()					read					
	read write									
										write
					write					

Figure 40.4: **File Creation Timeline (Time Increasing Downward)**

13.5 First Unix FS (Ken Thompson)

- Super block: head of freelists of inodes and data blocks
- Inode list: referenced by index, all same size
- Data blocks: belongs to only one file

13.5.1 Too slow

1. Blocks too small
   - File index too large
   - Require more indirect blocks
   - Transfer rate low (get one block at time)
2. Unorganized freelist
   - Consecutive file blocks not close together → seek cost for even sequential access
   - Aging: becomes fragmented over time
3. Poor locality
   - Inodes far from data blocks
   - Inodes for directory not close together
   - Poor enumeration performance *e.g.* `ls`

13.6 Fast file system (FFS)

Design FS structures and allocation polices to be *disk aware*

1. Bigger blocks → increased bandwidth
   ⚠ Internal fragmentation
   → BSD FFS: allow chopping blocks to *fragments*
     – Only for little files or ends of files
     – Fragment size specified at the time that the FS is created
     – Limit number of fragments per block to 2, 4, or 8
     – Multiple fragments fill single block → Allocate whole new block
2. Use bitmap of free blocks
   - Easier to find contiguous blocks
   - Small, so usually keep entire thing in memory
   - Trade space for time (need blocks for bitmap)
3. Cylinder groups (clustering) tries to
   - Put sequential blocks in adjacent sectors
   - Keep inode in same group as file data
   - Keep all inodes/files in a directory in same group
   - Balance directories across groups
   - Large file are partitioned into chunks and distributed over multiple groups

13.7 Modern FS

- Large block size (4KB)
- Block groups: do not export disk geometry info (↔ cylinder)
- Superblock replication for reliability

13.8 Crash consistency

FS may perform several disk writes to complete a single system call. But, disk only guarantees atomicity of a single sector write. If FS is interrupted between writes due to power loss, kernel panic (system crash), or transient hardware malfunctioning, the on-disk structure may be left in an inconsistent state.

→ Move FS from one consistent state to another atomically

13.8.1 Crash cases

File creation requires updating three blocks.

- Inode & Data bitmap (B)
- Inode for new file (I)
- Parent directory data block (D): Da →Da+Db

OS can reorder operations so 7 crash scenarios:

- BID ✓ due to page cache or internal disk write buffer
- BID' ✓ just lost data
- BI'D ✗ read gets garbage, duplicated pointers if Db is later allocated
- B'ID: ✗ space leak, Db will never used
- B'I'D: ✗ metadata is consistent but read gets garbage
- BI'D': ✗ duplicated pointers if Db is later allocated
- B'ID': ✗ space leak, Db will never used

13.8.2 FSCK (File System Checker)

Unix tool for finding inconsistencies in a FS and repairing them. Run before FS is mounted and made available. After crash, scan whole FS to find and fix:

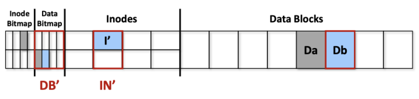
- Inconsistent bitmap and inode: Trust inode, modify bitmap
- Duplicate pointers: One inode is cleared or copy to give each inode its own
- Inconsistent link count: Fix the count of inode
- Lost file (B'ID): Create a temporary file in `/lost+found`

Problems are:

- Too slow: need to read all allocated blocks and entire directory
- Requires intricate knowledge of the FS
- Not always obvious how to fix file system image
- Don't know *correct* state, just consistent one

13.8.3 Journaling (Write-ahead logging)

Record a log, or journal, of changes made to on-disk data structures to a separate location (*journaling area*). Fast as it requires to scan only the journaling area.



1. Journal write: Write journal header block (TxB), IN', DB', and Db
2. Journal commit: Write journal commit block (TxE)
3. Checkpoint: Write updates to their final locations

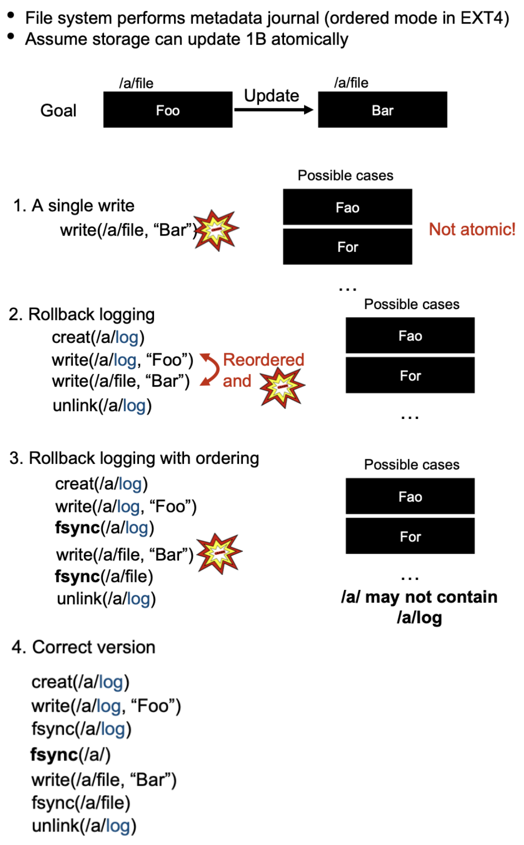
If a crash occurs between

- 1 and 2: Discard the journal
- 2 and 3: Roll-forward recovery (redo logging) which overwrite on-disk locations using the journal (regardless of 7 scenarios)

13.8.4 Optimize journaling

- Circular log: Mark transaction free and reuse journal space
- Batching log updates: Buffer all updates into a global transaction
- Journal checksums: Eliminate write barrier between journal write & commit
- Metadata/ordered journaling: Db is not written in journal
  - Force data write before journal is committed to avoid garbage

13.8.5 Application's crash consistency



- File system performs metadata journal (ordered mode in EXT4)
- Assume storage can update 1B atomically

1. A single write

```
write(/a/file, "Bar")
```

2. Rollback logging

```
creat(/a/log)
write(/a/log, "Foo")
write(/a/file, "Bar")
unlink(/a/log)
```

3. Rollback logging with ordering

```
creat(/a/log)
write(/a/log, "Foo")
fsync(/a/log)
write(/a/file, "Bar")
fsync(/a/file)
unlink(/a/log)
```

4. Correct version

```
creat(/a/log)
write(/a/log, "Foo")
fsync(/a/log)
fsync(/a/)
write(/a/file, "Bar")
fsync(/a/file)
unlink(/a/log)
```