CS 350 Operating Systems

Keven Qiu
Instructor: Bernard Wong
Winter 2024

Contents

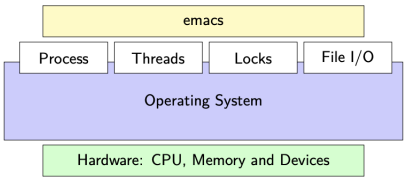
Chapter 1

Introduction

Definition: Operating System (OS)

The layer between applications and hardware.

It usually provides abstractions for applications. This includes managing and hiding details of hardware and can access hardware through low level interfaces unavailable to applications. It often provides protection.



Primitive OS are just a library of standard services (no protection). The system runs one program at a time and there are no bad users or programs.

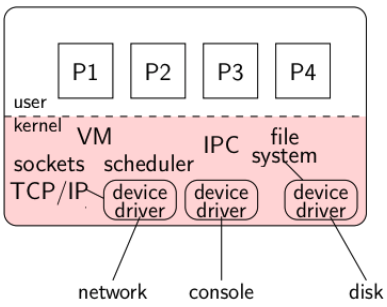
Multitasking is the idea of running more than one process at once. An ill-behaved process can go into an infinite loop and never relinquish the CPU. It can also scribble over other processes' memory. The OS provides mechanisms to address these problems:

- Preemption: take CPU away from looping.
- Memory protection: protect process' memory.

Multi-user OS use protection to serve distrustful users/apps. With n users, the system is not n times slower. Users can use too much CPU (need policies), total memory usage is greater than in the machine (must virtualize), or a super-linear slowdown with increasing demand (thrashing).

The OS structure: Most software runs as user-level processes and the OS kernel runs in privileged mode.

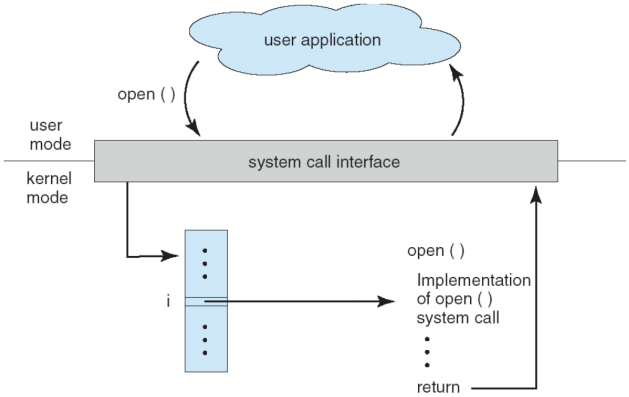
Protection:



- Pre-emption: give application a resource, take it away if needed elsewhere.
- Interposition/mediation: place the OS between application and "stuff". Track all pieces that the application is allowed to use. On every access, look in the table to check that access is legal.
- Privileged & unprivileged modes in CPUs: applications are unprivileged and the OS is privileged. Protection operations can only be done in privileged mode.

Definition: System Calls

Applications can invoke the kernel through system calls. These are special instructions that transfers control to the kernel.



The goal for system calls is to do things applications cannot do in unprivileged mode. The kernel supplies well-defined system call interface.

Applications set up syscall arguments and trap to kernel. The kernel performs operation and returns result.

Higher-level functions are built on the syscall interface. E.g. `printf`, `scanf`, `gets`, etc.

These call the POSIX/UNIX interface. E.g. `open`, `close`, `read`, `write`, etc.

Chapter 2

Processes

Definition: Process

An instance of a program running and the environment for running the program.

Modern OSes run multiple processes simultaneously.

- Multiple processes increase CPU utilization: overlap one process' computation with another.
- Multiple processes can reduce latency: running A then B may require more time than running A and B concurrently.

Each process has its own view of the machine. It has its own address space, open files, and virtual CPU.

2.1 User View of Processes

2.1.1 Creating Processes

`int fork(void);` creates a new process that is an exact copy of the current one. It returns the process ID of the new process in the parent. It returns 0 in the child.

The only part that is shared between the parent and child are the open files, otherwise the address space and code are its own.

`int waitpid(int pid, int *stat, int opt);` waits for a child process to terminate. `pid` is the process to wait for, or -1 for any. `stat` contains the exit value, or signal. `opt` is usually 0 or `WNOHANG`. `waitpid` returns the process ID that it waited for, or -1 if error.

2.1.2 Deleting Processes

`void exit(int status);` exits the current process. By convention, status of 0 is success and non-zero is error.

`int kill(int pid, int sig);` sends the signal `sig` to process `pid`. `SIGTERM` is the most common value and kills the process by default. `SIGKILL` is stronger and always kills the process.

2.1.3 Running Programs

`int execve(char *prog, char **argv, char **envp);` executes a new program. `prog` is the full path name of the program to run, `argv` is the argument vector that gets passed to main, and `envp` are the environment variables (PATH, HOME, etc.).

It is generally called through wrapper functions.

- `int execvp(char *prog, char **argv);` searches PATH for `prog` and use current environment.
- `int execlp(char *prog, char *arg, ...);` list arguments one at a time and finish with `NULL`.

2.1.4 Manipulating File Descriptors

`int dup2(int oldfd, int newfd);` closes `newfd`, if it was a valid descriptor and makes `newfd` an exact copy of `oldfd`.

Two file descriptors will share the same offset, i.e. `lseek` will affect both.

2.1.5 Pipes

This allows reading and writing between two processes like the parent and child.

`int pipe(int fds[2]);` returns two file descriptors in `fds[0]` and `fds[1]`. `fds[0]` reads and `fds[1]` writes. When the last copy of `fds[1]` is closed, `fds[0]` will return EOF. `pipe` will return 0 on success and -1 on error.

Operations on pipes:

- read/write/close.
- When `fds[1]` is closed, `read(fds[0])` returns 0 bytes.

- When `fds[0]` is closed, `write(fds[1])` kills the process with SIGPIPE or if the signal is ignored, it fails with EPIPE.

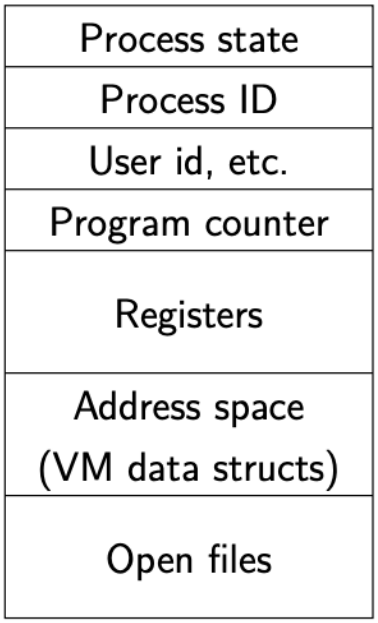
Most calls to fork are followed by execve. We can also combine into one spawn system call.

Without fork, we requires tons of different options.

2.2 Kernel View of Processes

2.2.1 Implementing Processes

The OS keeps a data structure for each process called the Process Control Block (PCB). It is called `proc` in Unix, `task_struct` in Linux, and `struct Process` in COS.



PCB

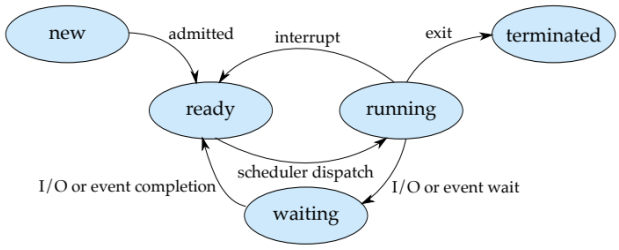
It tracks the state of the process (running, ready, blocked, etc.) and includes information necessary for it to run such as registers, virtual memory mappings, etc. There are also other data about the process like credentials, signal mask, controlling terminal, priority, etc.

Process States:

- New/Terminated: at beginning/end of life.
- Running: currently executing.
- Ready: can run, but kernel has chosen a different process to run.

- Waiting: needs async even (e.g. disk operation) to proceed.

If 0 processes are runnable, run an idle loop or halt CPU, otherwise if there is 1, run that process. If > 1 processes are runnable, it must make a scheduling decision.



Scanning the process table for the first runnable process is expensive.

We can use FIFO/round-robin to pick which process to run first.

Preemption: We can preempt a process when the kernel gets control. A running process can vector control to the kernel, a period timer interrupt, device interrupt, or changing running process to another (context switching).

Context Switching: Typical things include

- Saving program counter and integer registers.
- Save floating point or other special registers.
- Save condition codes.
- Change virtual address translations.

Some non-negligible costs are saving/restoring floating point registers, flushing the TLB, and usually causes more cache misses.

Chapter 3

Threads

Definition: Thread

A schedulable execution context.

An execution context contains a program counter, memory, etc. to tell where to execute code from.

Multi-threaded programs share the address space.

Definition: Kernel Thread

A thread that is scheduled for execution by the kernel.

Each kernel thread is assigned to one CPU core.

Definition: User Thread

A thread that is scheduled for execution by a user space threading library.

POSIX Thread API:

- `int pthread_create(pthread_t *thr, pthread_attr_t *attr, void *(*fn)(void *), void *arg);`: creates a new thread identified by `thr` with optional attributes, run `fn` with `arg`.
- `int pthread_exit(void *return_value);`: destroys current thread and return a pointer.
- `int pthread_join(pthread_t thread, void **return_value);`: wait for thread `thread` to exit and receive the return value.
- `void pthread_yield();`: tell the OS scheduler to run another thread or process.

We can implement `pthread_create` as a system call. To add `pthread_create` to an OS:

1. Start with process abstraction in kernel.
2. `pthread_create` like process creation with features stripped out:
    - Keep same address space, file table, etc. in new process.
    - `rfork`/`clone` syscalls actually allow individual control.

This is faster than a process, but still very heavy weight.

Limitations of Kernel-Level Threads:

- Every thread operation must go through the kernel: syscall takes 100 cycles, function calls take 2 cycles. This results in 10-30x slower threads when implemented in a kernel.
- One-size fits all thread implementation.
- General heavy-weight memory requirements.

User Threads: Implement a user-level library where there is one kernel thread per process.

Implementing User-Level Threads:

1. Allocate a new stack for each `pthread_create`.
2. Keep a queue of runnable threads.
3. Replace blocking system calls.
4. Schedule periodic timer signal (`setitimer`)

Limitations of User-Level Threads:

- Cannot take advantage of multiple CPUs or cores.
- A blocking system call blocks all threads.
- A page fault blocks all threads.
- Possible deadlock if one thread blocks another.

User Threads on Kernel Threads: There are multiple kernel-level threads per process. This is sometimes called $n : m$ threading; have n user threads per m kernel threads.

Limitations of $n : m$ Threading:

- Many of the same problems as $n : 1$ threading.
- Hard to keep same number of kernel threads as available CPUs.
- Kernel does not know relative importance of threads.

3.1 Case Study (Go Language)

3.1.1 Go Routines

Go Routines are very light-weight. Running 100k go routines is practical. It runs on a segmented stack and the OS thread typically allocated 2 MiB fixed stacks.

Go routines are on top of kernel threads with the $n : m$ model.

Each kernel-level thread finds and runs a go routine (user-level thread). Every logical core is owned by a kernel thread when running. It converts blocking system calls when possible to non-blocking ones in the runtime. This yields the CPU to another core.

3.2 Implementing Threads in CastorOS

AMD64/x86-64 Calling Conventions:

- Registers are divided into 2 groups: functions are free to clobber caller-saved registers (`%r10, %11`), but must restore callee-saves ones to original value upon return (`%rbx, %r12-%r15`).
- `%rsp` register is always the base of the stack.
- Local variables are stored in registers and on stack.
- Function arguments go in caller-saved registers and on the stack: the first six arguments are `%rdi, %rsi, %rdx, %rcx, %r8, %r9`.
- Return value is `%rax` and `%rdx`

3.2.1 Threads vs. Procedures

Threads may resume out of order.It cannot use LIFO stack to save state. A general solution is to use one stack per thread.

Threads switch less often.

Threads can be involuntarily interrupted.

- Synchronous: procedure call can use compiler to save state.
- Asynchronous: thread switch code saves all registers.

More than one thread can run at a time:

3.2.2 CastorOS Threads

CastorOS supports both kernel and user threads.

`Thread *Thread_KThreadCreate(void (*f)(void *), void *arg);` creates a kernel thread associated with the process.

`Thread *Thread_UThreadCreate(Thread *oldThr, uint64_t rip, uint64_t arg);` creates a userspace thread (and associated kernel thread).

All thread switches go through `Sched_Scheduler()` and `Sched_Switch()`. `Sched_Switch()` calls `Thread_SwitchArch` that runs `switchstack`. `switchstack` switches from one stack to other while saving and restoring registers.

Chapter 4

Concurrency

Recall that a process is an instance of a running program and a thread is an execution context. The POSIX thread API contains `pthread_create()`, `pthread_exit()`, `pthread_join()`.

4.1 Critical Sections

Definition: Critical Section

Part of a concurrent program in which a shared object is accessed.

```
int total = 0;
void add() {
    for (int i = 0; i < n; i++)
        total++;
}
void sub() {
    for (int i = 0; i < n; i++)
        total--;
}
```

Let thread 1 run add and thread 2 run sub.

1. Schedule 1 (add then sub): This increments then decrement, so the total is 0.
2. Schedule 2 (Alternate assembly instructions): Both loads, then add 1/sub 1 in the register, but the store in total is -1.
3. Schedule 3 (Load/add, sub, store, store): Total gets 1.

To prevent race conditions, we can enforce mutual exclusion on critical sections in the code.

Desired properties of a solution:

- Mutual exclusion: only one thread can be in the critical section at a time.
- Progress: if no process is currently in the critical section, one of the processes trying to enter will get in.
- Bounded waiting: once a thread T starts trying to enter the critical section, there is a bound on the number times other threads get in.
- Progress vs. Bounded waiting: if no thread can enter critical section, we don't have progress. If thread A is waiting to enter the critical section while B repeatedly leaves and re-enters, we don't have bounded waiting.

4.1.1 Peterson's Solution

```
// t is the current thread's id
int not_turn = 0; // not this thread's turn to enter CS
bool w[2] = {false, false}; // w[i] if thread i wants to enter CS
int total = 0;
void add() {
    for (int i = 0; i < n; i++) {
        w[t] = true;
        n_turn = t;
        while (w[1-t] && n_turn == t);
        total++;
        w[t] = false;
    }
}
void sub() {
    for (int i = 0; i < n; i++) {
        w[t] = true;
        n_turn = t;
        while (w[1-t] && n_turn == t);
        total--;
        w[t] = false;
    }
}
```

Mutual exclusion: both threads cannot be both in the critical section, `n_turn` prevents this.

Progress: If T_{1-i} is not in the critical section, it can't block T_i.

Bounded waiting: If T_i wants to lock and T_{1-i} tries to re-enter, T_{1-i} will set `n_turn=1-i`, allowing T_i in.

Peterson is expensive. This implementation requires *Sequential Consistency*.

4.2 CPU and Compiler Consistency

Definition: Sequential Consistency

The result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified in the program.

The boils down to two requirements

1. Maintaining program order on individual processors.
2. Ensuring write atomicity.

Without sequential consistency, multiple CPUs can be worse than preemptive threads.

Sequential consistency thwarts hardware optimizations. It cannot reorder overlapping write operations, reduces opportunities for data prefetching, and makes cache coherence more expensive.

Sequential consistency thwarts compiler optimizations. It causes code motion, caching value in register, common subexpression elimination, and loop blocking.

Often we reason about concurrent code assuming sequential consistency. However, for low-level code, we need to know the memory model. For most code, avoid depending on the memory model.

Hardware-Specific Synchronization Instructions: Used to implement synchronization primitives like locks. Example using `xchg`:

```
Xchg(value, addr) {
    old = *addr;
    *addr = value;
    return(old);
}

Acquire(bool *lock) {
    while (Xchg(true, lock) == true);
}

Release(bool *lock) {
    *lock = false; // give up lock
}
```

If `Xchg` returns true, the lock was already set and we continue to loop. If it returns false, then the lock was free, so we have acquired it.

This is known as a *spinlock*, since a thread busy-waits (loops) in Acquire until the lock is free.

This implementation is almost correct without sequential consistency. This is because Release can be inlined when it is called, which can interfere with other stores. We can fix it by adding a fence or replacing `*lock=false` with `Xchg(false, lock)`.

Spinlocks provide mutual exclusion and progress, but no bounded waiting. A thread has no bound in calling `Xchg`. Another inefficiency is that spinlocks keep spinning until a lock can be acquired. Using mutexes, we let the thread know when a lock is ready.

4.3 Mutexes and Condition Variables

Most operating systems provide locks (also known as a mutex). They enforce mutual exclusion.

```
mutex_lock(lock);
// critical section
mutex_unlock(lock);
```

Spinlocks spin, locks block. A thread that calls `spinlock_acquire` spins until the lock can be acquired. A thread that calls `mutex_lock` blocks until the lock can be acquired. Mutexes utilize spinlocks for implementation.

Sometimes a thread needs to wait for something, such as

- a lock to be released by another thread.
- data from a relatively slow device.
- input from a keyboard.
- busy device to become idle.

When a thread blocks, it stops running. The scheduler chooses a new thread to run. A blocked thread is signaled and awakened by another thread.

4.3.1 Wait Channels

To implement thread blocking, wait channels are used. It manages a list of sleeping threads and abstract details of the thread scheduler. It is a queue.

- `void WaitChannel_Lock(WaitChannel *wc);` locks wait channel operations and prevents a race between sleep and wake.
- `void WaitChannel_Sleep(WaitChannel *wc);` blocks calling thread on wait channel wc.

- `void WaitChannel_WakeAll(WaitChannel *wc);` unblocks all threads sleeping on the wait channel.
- `void WaitChannel_Wake(WaitChannel *wc);` unblocks one thread sleeping on the wait channel.

There can be many different wait channels, holding threads that are blocked for different reasons.

PThread Mutex API:

- `int pthread_mutex_init` initializes a mutex.
- `int pthread_mutex_destroy` destroys a mutex.
- `int pthread_mutex_lock` acquires a mutex.
- `int pthread_mutex_unlock` releases a mutex.
- `int pthread_mutex_trylock` attempts to acquire a mutex, returns 0 if successful, -1 otherwise (`errno == EBUSY`).

All global data should be protected by a mutex. Global means it is accessed by more than one thread with at least one write.

> **Compiler/Runtime Contract**
>
> Assuming no data races, the program behaves sequentially consistent.

There are problems when trying to acquire a mutex a thread already owns. It has to rely on something else to wake the thread up.

Improved Producer/Consumer: In the producer, we add a `mutex_unlock` and `mutex_lock` in the loop to wait for the consumer to consume data. Similarly in the consumer.

This is not a good approach since the loop keeps spinning.

4.3.2 Condition Variables

To handle spinning, we want to inform the scheduler of which threads can run. This is done with condition variables.

We have the functions

- `int pthread_cond_init`: initializes with specific variables.
- `int pthread_cond_wait`: atomically unlock mutex m and sleep until c is signaled.

- `int pthread_cond_signal`
- `int pthread_cond_broadcast`: wake one/all threads waiting on c.

Now in the while loop, we can do a condition wait for the condition required. We can signal at the end of the critical section to signal a condition.

We use a while loop, not an if statement. This is because we want to always recheck the condition on wake-up.

We cannot separate the mutex and condition variables because the the producer can wait forever. So we have the mutex unlock and lock in the condition wait call.

4.4 Semaphores

Definition: Semaphore

A way to hold a counter.

Initialized with an integer N, `int sem_init(sem_t *s,..., unsigned int n);` and provides two functions:

- `sem_wait(sem_t *s);`: decreases the counter.
- `sem_post(sem_t *s);`: increases the counter.

`sem_wait` will return only N more times than `sem_post` called.

If $N = 1$, then this is a binary semaphore which is a mutex, with `sem_wait` as lock and `sem_post` as unlock.

Semaphores can give elegant solutions to some problems.

```
Semaphore_Wait(Semaphore *sem) {
    Spinlock_Lock(&sem->sem_lock);
    while (sem->sem_count == 0) {
        WaitChannel_Lock(sem->sem_wchan);
        Spinlock_Unlock(&sem->sem_lock);
        WaitChannel_Sleep(sem->sem_wchan);
        Spinlock_Lock(&sem->sem_lock);
    }
    sem->sem_count--;
    Spinlock_Unlock(&sem->sem_lock);
}
Semaphore_Post(Semaphore *sem) {
```

```
    Spinlock_Lock(&sem->sem_lock);
    sem->sem_count++;
    WaitChannel_Wake(sem->sem_wchan);
    Spinlock_Unlock(&sem->sem_lock);
}
```

4.4.1 Hand-Over-Hand Locking

Allows for fine-grained locking and useful for concurrent data structure manipulation.

Can only hold at most two locks: the previous and next locks. They must also be ordered in a sequence; i.e. we cannot go backwards.

Example: Suppose we have locks A, B, C.

```
lock(A)
// Operate on A
lock(B)
unlock(A)
// Operate on B
lock(C)
unlock(B)
// Operate on C
unlock(C)
```

If we acquire a lock in A, B, C order and another thread acquires it in C, B, A order, then we get into a case where one thread holds onto A and B and the other thread holds onto B and wants to acquire A. This is known as a *deadlock* (circular dependency).

4.4.2 Monitors

It is a programming language construct. It is less error prone than raw mutexes, but less flexible.

Definition: Monitor

A class where only one procedure executes at a time.

It provides condition variables like functionality.

Upon entering a monitor, we need to acquire a lock and upon exiting, it automatically releases the lock.

If there is a synchronized method in a class, then Java will create a monitor for the whole object. The lock is automatically released after object is destroyed.

The monitor implementation is `wait`, `notify`, `notifyAll`.

4.5 Data Races

We may care more about speed than accuracy.

Definition: Race Condition

The output of the code is determined by the ordering of he threads, how long the threads are running, and how many processors you may have.

Lockset algorithm: for each global memory location, keep a lockset. On each access, remove any locks not currently held. If the lockset becomes empty, abort; no mutex can protect the data.

4.5.1 Detecting Data Races

The tool does not know which lock is used to provide mutual exclusion for which variable. Even if the lock is protecting thread 1, it does not mean thread 1 always happens before thread 2, but if at any point it cannot determine an order that must be obeyed, there is a good chance there is no lock enforcing an ordering between the threads.

Chapter 5

Synchronization

5.1 Deadlock Problem

```
mutex_t m1, m2;
void f1(void *ignored) {
    lock(m1);
    lock(m2);
    /* critical section */
    unlock(m2);
    unlock(m1);
}
void f2(void *ignored) {
    lock(m2);
    lock(m1);
    /* critical section */
    unlock(m1);
    unlock(m2);
}
```

Definition: Deadlock Condition

1. Limited access (mutual exclusion): resource can only be shared with finite users.
2. No preemption: once resource granted, it cannot be taken away.
3. Multiple independent requests (hold and wait).
4. Circularity in graph of requests.

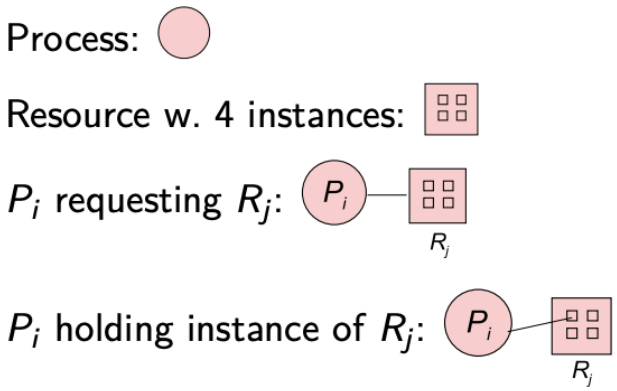
All of conditions 1-4 are necessary for a deadlock to occur.

Two approaches to dealing with deadlock:

1. Pro-active: prevention.
2. Reactive: detection and corrective action.

5.2 Resource-Allocation Graph

We can view the system as a graph. The processes and resources are the nodes and resource requests/assignments are edges.



A graph with a deadlock:

If the graph has no cycles, there are no deadlocks. If the graph contains a cycle, then there is a deadlock if only one instance per resource. Otherwise, there may or may not be a deadlock.

We can prevent deadlocks with partial order on resources, e.g. always acquire mutex m_1 before m_2, statically assert lock ordering, or dynamically find potential deadlocks.

Definition: Amdahl's Law

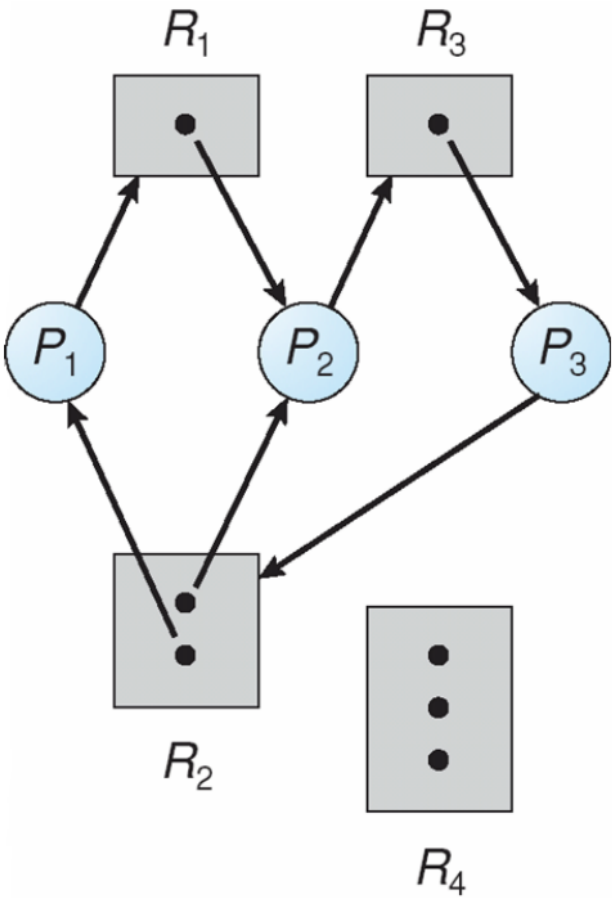
Let $T(1)$ be the time one core takes to complete the task, B be the fraction of the job that must be serial, and n be the number of cores, then

$$T(n) = T(1)\left(B + \frac{1}{n}(1 - B)\right)$$

5.3 Multicore Caches

Performance requires caches. Caches, however, create an opportunity for cores to disagree about memory.

Bus-based approaches makes the CPU listen to the memory bus. These limit scalability though.



Modern CPUs use networks. The cache is divided into chunks of bytes called cache lines. 64 bytes is a typical size.

5.3.1 3-State Coherence Protocol (MSI)

Each cache line is one of three states:

1. Modified/Exclusive state: One cache has a valid copy. That copy is stale (needs to be written back to memory), and must invalidate all copies before entering this state.
2. Shared state: One or more caches have a valid copy.
3. Invalid state: Cache does not have any data.

Transitions can take 100-2000 cycles.

Core and Bus Actions: The core has three actions on cache lines

1. Read

2. Write
3. Evict

Every transition requires bus communications, so we should try to avoid state transitions whenever possible.

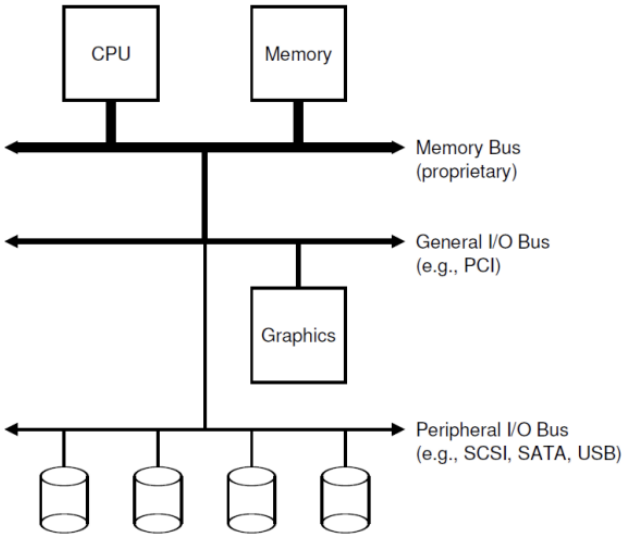
Lessons for multithreaded design:

1. Avoid false sharing.
2. Align structures to cache lines.
3. Pad data structures.
4. Avoid contending on cache lines.

Chapter 6

I/O and Disks

CPU accesses physical memory over a memory bus. Some devices are connected to the system via a general I/O bus. Slow devices are connected to a peripheral bus.



Device drivers communicate with devices by accessing/modifying their registers. It can access device registers through two options:

1. Special I/O instructions: device registers are assigned port numbers.
2. Memory-mapped I/O: each device register has a physical memory address.

Definition: Device Driver

A part of the kernel that interacts with a device.

We want mutual exclusion so we do not have multiple threads writing to same device at the same time.

```
// only one writer at a time
P(output device write semaphore)
// trigger write operation
write char to device data register
repeat {
    read writeIRQ register
} until status is completed
// make device ready again
write writeIRQ register to completion
V(output device write semaphore)
```

This keeps running repeat which is similar to the problem of spinlocks. We can instead use interrupts to avoid polling.

Definition: Direct Memory Access (DMA)

Used for block data transfers between devices and memory.

Under DMA, the CPU initiates the data transfer and is notified when the transfer is finished (by interrupt).

6.1 Anatomy of a Disk

Stack of magnetic platters. The rotate together on a central spindle at 3600-15000 RPM.

The disk arm rotate around pivot. Arms contain disk heads.

Platters are divided into concentric tracks. A stack of tracks of fixed radius is a cylinder. Heads record and sense data along cylinders. Generally only one head active at a time.

The disk an array of numbered sectors. Each sector is the same size. Sectors are the unit of transfer between the disk and memory.

A seeks consists of up to four phases:

- Speedup: accelerate arm to max speed or half way point.
- Coast: at max speed.
- Slowdown: stops arm near destination.
- Settle: adjusts head to actual desired track.

Very short seeks are dominated by settle time. Short seeks are dominated by speedup. Average seek time is roughly $\frac{1}{3}$ of the full seek time.

6.1.1 Basic Cost Model for Disk I/O

Moving data involves:

- Seek time: move the read/write heads to the appropriate cylinder.
- Rotational latency: wait until the desired sectors spin to the read/write heads.
- Transfer time: wait while the desired sectors spin past the read/write heads.

Request service time is the sum of seek time, rotational latency, and transfer time.

6.2 Disk Performance

Placement and ordering of requests is a huge issue. We try to achieve contiguous accesses where possible. If multiple requests come, let the OS order requests to minimize seek times.

6.3 Scheduling

Definition: First Come First Served (FCFS)

Process disk requests in the order they are received.

Some advantages are easy to implement and good fairness. Disadvantages include not being able to exploit request locality and increases average latency, decreasing throughput.

Definition: Shortest Positioning/Seek Time First (SPTF/SSTF)

Always pick request with shortest seek time.

Advantages are it exploits locality of disk requests and higher throughput. The disadvantage is the starvation. We can improve this by giving older requests higher priority.

Definition: Elevator Scheduling (SCAN)

Sweep across the disk, servicing all requests passed.

This is similar to SPTF, but the next seek must be in the same direction. We switch directions only if there are no further requests.

Advantages include taking advantage of locality and bounded waiting. Disadvantages include the cylinders in the middle get better service and might miss locality SPTF could exploit.

Definition: VSCAN(r)

A continuum between SPTF and SCAN.

Like SPTF, but slightly changes effective positioning time. If the request is in the same direction as previous seek: $T_{eff} = T_{pos}$. Otherwise, $T_{eff} = T_{pos} + rT_{max}$. When $r = 0$, we get SPTF and when $r = 1$, we get SCAN. Typically $r = 0.2$. Advantages/disadvantages depend on r and are similar to SPTF and SCAN.

6.4 Flash Memory

Now we use flash memory or solid state drives. Solid state means no moving parts. DRAM requires constant power to keep values.

The SSD is logically divided into blocks and pages. 2, 4, or 8 KB pages and 32 KB-4 MB blocks. Reads and writes are at page level. Pages are initialized to 1s. It can transition from 1 to 0 at pages level (write new page).

6.4.1 Writing and Deleting from Flash Memory

Naive solution: read whole block into memory, re-initialize block (bits back to 1), update block in memory, write back to SSD.

Using translation layer solution: mark page to be deleted/overwritten as invalid, write to an unused page, update translation table. This requires garbage collection.

Wear leveling: blocks in SSDs have limited number of write cycles. If a block becomes unwriteable, it becomes read-only, and if certain percentage of blocks become read-only, then the entire disk becomes read-only. SSD controller spreads the write cycles over all blocks.

Chapter 7

File Systems

Definition: Files

Persistent, named data objects.

Data consists of a sequence of numbered bytes. File may change size over time and each file has associated meta-data.

Definition: File Systems

The data structures and algorithms used to store, retrieve, and access files.

The translation name (file) and offset goes through the file system and gives the disk address.

7.1 File Interface

- `open`: returns a **file identifier**, which is used in subsequent operations to identify the file.
- `close`: kernel tracks while file descriptors are currently valid for each process. close invalidates a valid file descriptor.
- `read`: copies data from a file into a virtual address space.
- `write`: copies data from a virtual address space into a file.
- `lseek`: enables non-sequential reading/writing.

Each file descriptor (open file) has an associated file position. The position starts at byte 0 when the file is opened. The read and write operations start from the current file position and updates as bytes are read/written. `lseek` changes the file position associated with a descriptor.

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=0; i<100; i++) {
read(f,(void *)buf,512);
}
close(f);
```

7.2 Directories and File Names

Definition: i-Number

A unique identifier (within file system) for a file or directory.

Definition: Directory

Maps file names (strings) to i-numbers.

Given an i-number, the file system can find the data and meta-data for the file.

Directories are ways for applications to group related files. Since directories can be nested, we can view it as a tree, with a single root directory and files as leaves.

7.3 Links

Definition: Hard Link

An association between a name and an i-number.

Each entry in a directory is a hard link. When a file is created, so is a hard link to that file.

Once a file is created, additional hard links can be made to it. Example: link(/docs/a.txt, /foo/myA) creates a new hard link myA in /foo that links to the i-number of file /docs/a.txt.

Linking to an existing file creates a new pathname for that file. Each file has a unique i-number, but may have multiple pathnames.

It is not possible to link to a directory, this is to avoid cycles.

Hard links can be removed. Once the last hard link to a file is removed, the file is also removed.

7.4 Multiple File Systems

It is not uncommon for a system to have multiple file systems. Some kind of global file namespace is required. DOS/Windows uses two-part file names: the file system name and the pathname within the file system. Unix creates single hierarchical namespace that combines the namespaces of two file systems. It does this using the Unix mount system call.

7.5 File System Implementation

The following needs to be stored persistently:

- file data
- file meta-data
- directories and links
- file system meta-data

The following are non-persistent information:

- open files per process
- file position for each open file
- cached copies of persistent data

Consider a 256 KB disk with sector size of 512 bytes. Memory is usually byte addressable and disk is usually sector addressable. There are 512 total sectors on this disk.

We group every 8 consecutive sectors into a block. There is better spatial locality (fewer seeks, reduces the number of block pointers. 4 KB block is a convenient size for demand paging. There are 64 total block on the disk.

7.5.1 Very Simple File System

Most of the blocks should be for storing user data. We need some way to map files to data blocks.

We can do this by creating an array of i-nodes, where each i-node contains the meta-data for a file. The index into the array is the file's index number (i-number).

Assume each i-node is 256 bytes, and we dedicate 5 blocks for i-nodes. This allows for 80 total i-nodes.

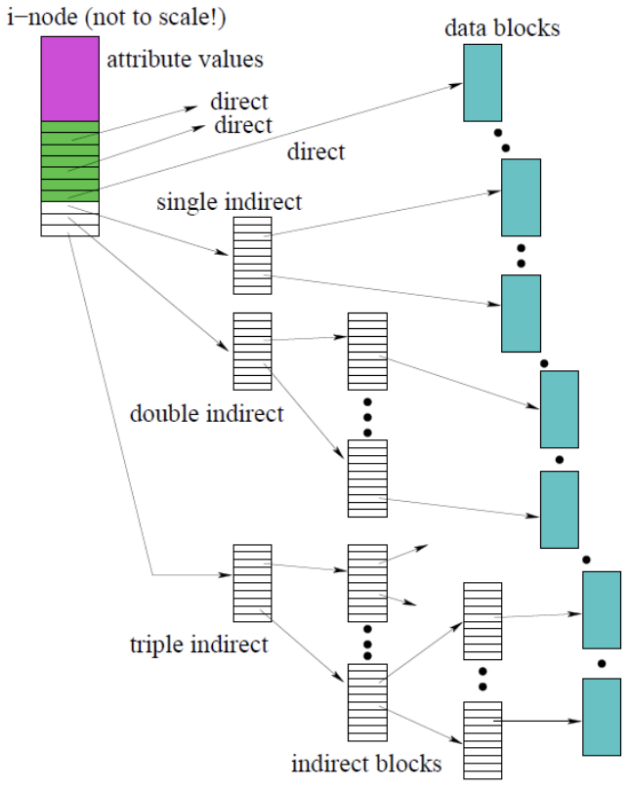
We also need to know which i-nodes and blocks are unused. We can use a bitmap for each. A block size of 4 KB means we can track 32K i-nodes and 32K blocks. The i-bmap is the bitmap is for i-nodes. The d-bmap is the bitmap for the data blocks.

We reserve the first block as the superblock. This contains meta-information about the entire file system.

Definition: i-Node

A fixed size index structure that holds both file meta-data and a small number of pointers to data blocks.

The fields of an i-node include: file type, file permissions, file length, number of file blocks, time of last file access, time of last i-node update, last file update, number of hard links to this file, direct data block points, single/double/triple indirect data block pointers.



Assume disk blocks can be referenced based on a 4 byte address, 2^{32} blocks, 4 KB blocks. The maximum disk size is 16 TB. In VSFS, an i-node is 256 bytes. Assume there is enough room for 12 direct pointers to blocks and each pointer points to a different block for storing user data. Pointers are ordered, i.e. first pointer points to first block in file, etc. The maximum file size if we only have direct pointers is $12 \cdot 4$ KB.

In addition to 12 direct pointers, we can have indirect pointers.

Definition: Indirect Pointer

A pointer that points to a block full of direct pointers.

4 KB block of direct pointers is 1024 pointers. The maximum file size is $(12+1024)\cdot 4$ KB $=$ 4144 KB.

This is enough for any file that can fit in our 256 KB disk, but if the disk were larger, we add a double indirect pointer.

Definition: Double Indirect Pointer

Pointer that points to a 4 KB block of indirect pointers.

The size is $(12 + 1024 + 1024 \cdot 1024) \cdot 4$ KB, which is just over 4 GB in size.

We use indirect pointers to enable bigger file size. If we just had all direct pointers, we would be wasting memory on a bunch of null pointers if the file is small. Indirection is also cost inefficient, since we have to follow lots of levels of indirection.

7.5.2 In-Memory Structures

There is an open file table per process. It stores which file descriptors are open, which files each descriptor refers, and which current file position for each descriptor.

Each file descriptor in the file table points to the system wide open file table. The system stores an i-node cache and a block cache.

7.6 Reading From a File

To read the file `/foo/bar`:

1. Call the open system call.
2. It first reads the root i-node/i-node 2. i-node 1 is usually for tracking bad blocks.
3. Read the directory information from root, find the i-number for foo, read the foo i-node.
4. Read the directory information from foo, find the i-number for bar, read the bar i-node.
5. It reads the bar i-node again; there is a permission check, allocate a file descriptor in the per-process descriptor table, increment counter for this i-number in the global open file table.
6. Find the block using direct/indirect pointer and read data.

7. Update i-node with new access time.

8. Update the file position in the per-process descriptor table.

9. Close the file, deallocates the file descriptor and decrements the counter for this i-number in the global open file table.

	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** **write**					**read**		
read()			**read** **write**						**read**	
read()			**read** **write**							**read**

7.7 Creating a File

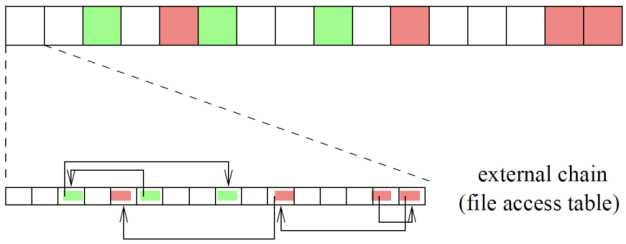
To write a file, the following diagram shows the steps:

	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** **write**	**read**	**read** **write**	**read** **write**	**read** **write**	**read**			
write()	**read** **write**				**read** **write**			**write**		
write()	**read** **write**				**read** **write**				**write**	
write()	**read** **write**				**read** **write**					**write**

We need to read before writing the bar i-node to prevent writing other i-nodes. The last step of create needs to write to foo i-node because the file size grew when adding the bar i-number.

7.8 Different File System Implementations

The implementation we have been using is called per-file indexing, the i-node points to the blocks of the data. Other implementations include chaining (linked list implementation) and external chaining.



External chaining uses a file access table (FAT) which stores all pointer information in the table. This allows us to not read each data block to get a pointer.

A single logical file system operation may require several disk I/O operations.

Deleting a file: remove entry from directory, remove file index (i-node) from i-node table, mark file's data blocks free in free space index.

We want our file system to be file consistent. The data should be consistent.

There are special-purpose consistency checkers. This will run after a crash to find and attempt to repair inconsistent file system data structures.

Now there is journaling. This records file system meta-data changes in a journal (log), so that sequences of changes can be written to disk in a single operation. Only do operations that have a begin and a commit (end), if the system crashes.

Chapter 8

Virtual Memory

Every physical address identifies one byte of physical memory. The size of the address gives you the maximum size of physical memory. So for 256 KB total physical memory, the addresses are 18 bits since 2^{18} is 256 KB.

> If physical addresses have P bits, the maximum amount of addressable physical memory is 2^P bytes.

The actual amount of physical memory on a machine may be less than the maximum amount that can be addressed.

8.1 Virtual Addresses

The kernel provides a separate, private virtual memory for each process. The virtual memory of a process holds the code, data, and stack for the program that is running in that process.

If virtual addresses are V bits, the maximum size of a virtual memory is 2^V bytes.

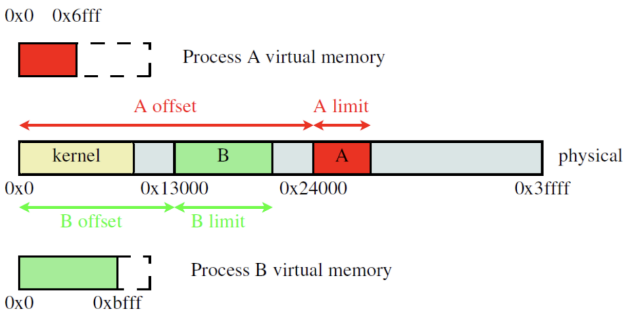
Running applications see only virtual addresses, and each process cannot access other process' virtual memories.

Each virtual memory is mapped to a different part of physical memory. Virtual memory is not real, so when trying access via load or store, it is translated to a physical address. Address translation is performed in hardware, using information provided by the kernel.

8.2 Dynamic Relocation

The CPU contains a memory management unit (MMU), for translation, with a relocation register and a limit register.

The relocation register holds the physical offset R for the running process' virtual memory. The limit register holds the size L of the running process virtual memory.



Translation: To translate a virtual address v to a physical address p: if $v \geq L$, generate exception, else $p = v + R$.

The kernel mains a separate R and L for each process and changes the values in the MMU registers when there is a context switch.

8.2.1 Properties

Each virtual address space correspond to a contiguous range of physical addresses. The kernel is responsible for deciding where each virtual address space should map in physical memory.

A problem is external fragmentation since there a holes in memory where a process' memory is unused.

Example: Process A has limit register $0x7000$, relocation register $0x24000$.

1. $v = 0x102c$: Since $v < L$, then $p = 0x2502c$.
2. $v = 0x8800$: Since $v \geq L$, then exception.
3. $v = 0x0000$: Since $v < L$, then $p = 0x24000$.

8.3 Pages

Definition: Pages

Fixed-size chunks of virtual memory. The page size is equal to the frame size.

In our virtual memory, page size is 4 KB. If the virtual memory of a process is 64 KB, then there are 16 pages.

Each page maps to a different frame. Any page can map to any frame.

Definition: Internal Fragmentation

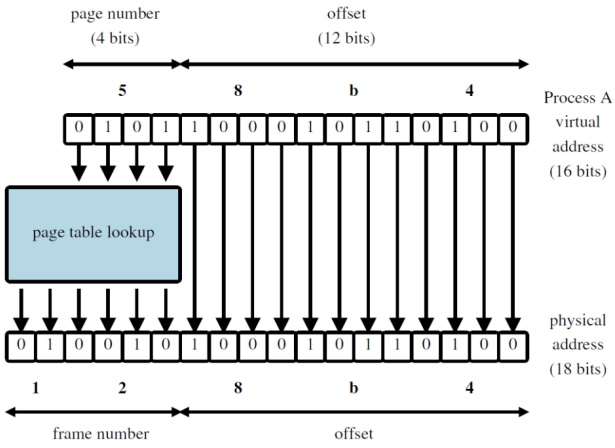
Utilizing a partial amount of a page, but the rest of the page is unused.

Instead of using relocation registers, we use a page table for translation.

The MMU includes a page table register, which points to the page table for the current process.

MMU Virtual Address Translation

1. Determines the page number and offset of the virtual offset. (page number is the virtual address divided by page size, offset is the virtual address modulo the page size)
2. Looks up the page's table entry (PTE) in the current process page table using the page number.
3. If the PTE is not valid, raise an exception.
4. Otherwise, combine page's frame number from the PTE with the offset to determine physical address. (Physical address is (frame number $\times$ frame size) + offset)



Since the page size is 4 KB, we can represent is with 12 bits (since 2^{12} is 4 KB). So the last 12 bits is the offset. The first remaining 4 bits is the page number.

We then lookup page number in the page table to get frame number, then the physical address is frame number and offset.

Example:

1. $v = 0x102c$: page number is $0x1$ and is valid, lookup frame number $0x26$, so $p = 0x2602c$.
2. $v = 0x9800$: Exception since $0x9$ was invalid in the page table.

3. $v = 0x0024$: page number is $0x0$ and is valid, lookup frame number $0x0f$, so $p = 0x0f024$.

8.3.1 Other Information in PTEs

A field is write protection bit. This can be set by the kernel to indicate that a page is read-only.

Another field are bits to track page usage. Reference bit tells whether the process has used this page recently. Dirty bit tells whether this page has been changed.

A page table has one PTE for each page in the virtual memory.

Page table size = number of pages × size of PTE.

Number of pages = virtual memory size / page size.

Example: If virtual memory is 64 KB with 4 KB pages, then there are 16 pages. If each PTE has size 32 bits (4 bytes), so page table size is $16 \times 4 = 64$ bytes.

Summary: Page tables are kernel data structures. The kernel manages MMU registers on address space switches. It Creates and manages page tables, manages physical memory, and handles exceptions raised by the MMU.

The MMU (hardware) translates virtual addresses to physical addresses and checks for and raises exceptions when necessary.

8.4 Translation Lookaside Buffer

Execution of each machine instruction may involve multiple memory operations (one to fetch instruction, one or more for instruction operands). Address translation through a page table adds one extra memory operation for each memory operation performed during instruction execution.

Definition: Translation Lookaside Buffer (TLB)

A small, fast, dedicated cache of address translations, in the MMU.
Each TLB entry stores a page number to frame number mapping.

If the page number is in the TLB, this is a TLB hit, otherwise it has to look it up and add it to TLB. If the MMU cannot distinguish TLB entries from different address spaces, the the kernel must clear or invalidate the TLB on each context switch.

8.4.1 Segmentation

Limitations of Simple Address Translation Approaches A sort application uses 1.2 MB. But on MIPS, $V = 32$ so the virtual memory size is 2^{32} bytes (4 GB) and on x86-64, $V = 48$ so the virtual memory is 2^{48} bytes (256 TB).

A kernel that used simple dynamic relocation would have to allocate 2 GB of contiguous physical memory for sort. A kernel that uses simple paging would require a page table with 2^{20} PTEs to map sort's address space.

Instead of mapping the entire virtual memory to physical, we can provide a separate mapping for each segment of the virtual memory that the application actually uses. Instead of a single offset and limit for the entire address space, the kernel maintains an offset and limit for each segment.

With segmentation, a virtual address can be split into two parts: segment ID and offset within the segment.

With k bits for the segment ID, we can have up to 2^k segments and 2^{V-k} bytes per segment. The kernel decides where each segment is placed in physical memory. Fragmentation of physical memory is still possible.

> **Translating Segmented Virtual Addresses**
>
> MMU has a relocation register and a limit register for each segment. Let R_i be the relocation offset and L_i be the limit offset for the ith segment. To translate virtual address v to a physical address p: Split v into segment number s and address within segment a, if $a \geq L_s$, generate exception, else, $p = a + R_s$.

Example: Limit register 0: $0x2000$, Relocation register 0: $0x38000$, Limit register 1: $0x5000$, Relocation register 1: $0x10000$.

- $v = 0x1240$: $0x1 = 1$, $s = 0$, offset $= 0x1240$, $p = 0x39240$.
- $v = 0xa0a0$: $0xa = 1010$, $s = 1$ (first bit), offset $= 0x20a0$ (strip off first bit, look at rest), $p = 0x120a0$.

Another approach is maintaining a segment table.

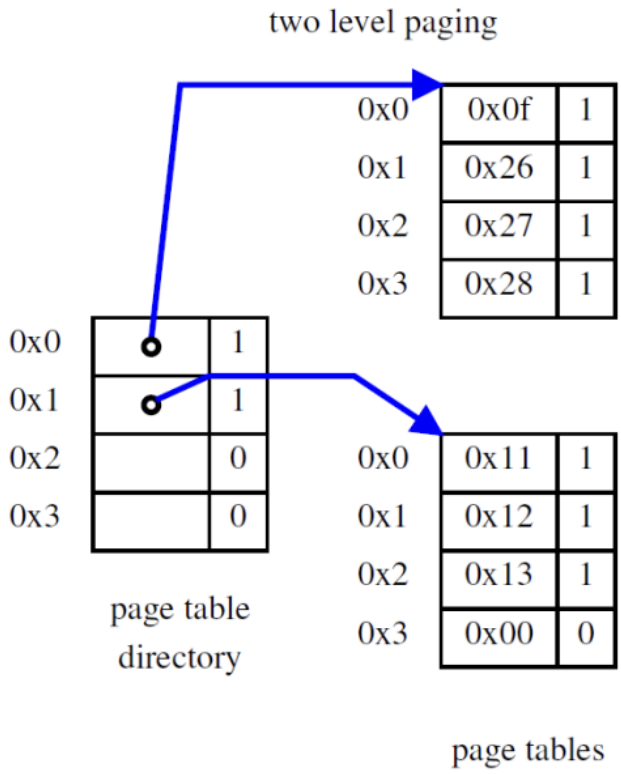
8.5 Two-Level Paging

Instead of having a single page table to map an entire virtual memory, we can split the page table into smaller page tables, and add a page table directory.

Each virtual address will have 3 parts:

- Level one page number: used to index the directory.

- Level two page number: used to index a page table.
- Offset within the page.



For example: for page number 3, in binary is 0011, so we follow 0x0 (first two bits), follow into the first page table, then $11 = 0x3$, which returns $0x28$.

The MMU's page table base register points to the page table directory for the current process. Each virtual address has 3 parts (p_1, p_2, o). To translate:

1. Index into the page table directory using p_1 to get a pointer to a 2nd level page table.
2. If the directory entry is not valid, raise an exception.
3. Index into the 2nd level page table using p_2 to find the PTE for the page.
4. If the PTE is not valid, raise an exception.
5. Otherwise, combine the frame number from the PTE with o to determine the physical address being accessed.

8.5.1 Limits

The goal of two-level paging was to keep individual page tables small. Suppose we have 40 bit virtual addresses ($V = 40$) and that the size of a PTE is 4 bytes, page size is 4 KB (2^{12} bytes).

The offset is 12 bits (page size is 2^{12} bytes, so 12 bits). P_2 needs to be 10 bits (each PTE is 4 bytes, page table size is 4 KB, so $2^{12}/2^2 = 2^{10}$).

We'd like to limit each page table's size to 4 KB, then we can introduce a third level so that P_2, P_3 has 10 bits, and P_1 is 8 bits.

We can solve large directory problems with additional levels of directories.

Properties of Multi-Level Paging:

- Can map large virtual memories by adding more levels.
- Individual page tables/directories can remain small.
- Can avoid allocating page tables and directories that are not needed for programs that use a small amount of virtual memory.
- Can avoid allocating page tables and directories that are not needed for programs that use a small amount of virtual memory.
- TLB misses become more expensive as the levels go up.

8.6 Virtual Memory for the Kernel

We want the kernel to live in virtual memory, there are two challenges:

- Bootstrapping: Since the kernel helps to implement virtual memory, how can the kernel run in virtual memory when it is starting?
- Sharing: Sometimes data need to be copied between the kernel and application programs? How do you get data in between different virtual address space?

  We can make the kernel's virtual memory overlap with process' virtual memories.

8.6.1 Exploiting Secondary Storage

Goals:

- Allow virtual address spaces that are larger than the physical address space.
- Allow greater multiprogramming levels by using less of the available primary memory for each process.

Method:

- Allow pages from virtual memories to be stored in secondary storage, i.e. on disks or SSDs.
- Swap pages between secondary storage and primary memory so that they are in primary memory when they are needed.

8.6.2 Resident Sets and Present Bits

When swapping is used, some pages of each virtual memory will be in memory, and others will not be in memory.

Definition: Resident Set

The set of virtual pages present in physical memory is the resident set of a process.

To track which pages are in physical memory, each PTE needs to contain an extra bit called the present bit. If the present bit is 1, it is in memory, and 0 if it is not in memory.

When a process tries to access a page that is not in memory, the problem is detected because the page's present bit is 0. The MMU detects this when it checks the page's PTE and generates an exception. This event is called a *page fault*.

When a page fault happens, it is the kernel's job to

1. Swap the page into memory from secondary storage, evicting another page from memory.
2. Update the PTE by setting the present bit.
3. Return from the exception so that the application can retry the virtual memory access that caused the page fault.

8.6.3 Replacement Policies

A simple replacement policy is the FIFO policy. When the kernel needs to evict a page from physical memory, it replaces the page that has been in memory the longest.

An optimal page replacement policy for demand paging is called MIN. It replaces the page that will not be referenced for the longest time. MIN requires knowledge of the future. Real programs do not access their virtual memories randomly. They exhibit locality.

Definition: Temporal Locality

Programs are more likely to access pages that they have accessed recently than pages that they have not.

Definition: Spatial Locality

Programs are likely to access parts of memory that are close to parts of memory they have accessed recently.

A scheme is Least Recently Used (LRU). We evict the page that has been least recently paged.

The kernel is not aware which pages a program is using unless there is an exception. The makes it difficult for the kernel to exploit locality by implementing a replacement policy like LRU. The MMU can solve this by tracking page accesses in hardware by adding a use/reference bit to each PTE. This bit is set by the MMU each time the page is used. This can be read and cleared by the kernel.

8.6.4 The Clock Replacement Algorithm

The clock algorithm (second chance algorithm) is a simple algorithm that exploits the use bit. The algorithm is identical to FIFO, except that a page is skipped if its use bit is set.

Chapter 9

Scheduling

We are given a set of jobs to schedule. Only one job can run at a time. For each job, we are given job arrival time a_i and job run time r_i.

For each job, we define

Definition: Response Time

Time between the jobs arrival and when the job starts to run.

Definition: Turnaround Time

Time between the jobs arrival and when the job finishes.

9.1 First Come First Served

Jobs run in order of arrival. This is bad when long jobs come first, then we will have bad turnaround/response time.

9.2 Round Robin

The preemptive variant of FCFS. Each job get a small time quantum to run. This is bad when we want to minimize turnaround time, round robin is worse than FCFS.

9.3 Shortest Job First

Run jobs in increasing order of runtime. Sometimes idling improves response time.

9.4 Shortest Remaining Job First

Preemptive variant of shortest job first. Arrive jobs preempt running job. We select one with shortest remaining time. Starvation is possible. This always minimizes average turnaround time.

9.5 CPU Scheduling

The jobs to be scheduled are the threads. However, we don't know the runtimes of threads. The objective of the scheduler is to achieve balance between responsiveness, fairness, and efficiency.

One of the most common algorithms is the Multi-Level Feedback Queues. It gives good responsiveness for interactive threads, non-interactive threads make as much progress as possible.

The algorithm: n round-robin ready queues where the priority of $Q_i > Q_j$ if $i > j$. Threads in Q_i use quantum q_i and $q_i \leq q_j$ if $i > j$.

The scheduler selects a thread from the highest priority queue to run. Threads in Q_{n-1} are only selected if Q_n is empty. Preemptive threads are put onto the back of the next lower-priority queue. When a thread from Q_n is preempted, it is pushed onto Q_{n-1}.

A thread wakes after blocking, it gets put onto the highest-priority queue.