# CSC369 Notes

Jenci Wei

Fall 2023

## Contents

# 1 Memory API

Types of Memory

1. **Stack memory**: allocations are managed implicitly by the compiler
    - Also called **automatic memory**
2. **Heap memory**: allocations are explicitly handled by the programmer

The `malloc()` call

- Pass in a size to ask for on the heap
- Returns a pointer to the space if it succeeds
- Returns `NULL` if it fails

```c
#include <stdlib.h>
...
void *malloc(size_t size);
```

- Use macros for the parameter, e.g. `sizeof(double)`
    - `sizeof` is a *compile-time operator* (not a function), i.e. the size is determined at compile time and then the number is substituted
- Returns a pointer of type `void`
    - It is best to *cast* the result to the desired type

The `free()` call

- To free heap memory that is no longer in use
- Takes in the pointer returned by `malloc()`

Some languages have support for *automatic memory management*

- We don't have to manually free space
- A *garbage collecor* frees space that is no longer in use

Common errors

- **Buffer overflow**: not allocating enough memory
- **Uninitialized read**: forgetting to initialize allocated memory
- **Memory leak**: forgetting to free memory
- **Dangling pointer**: freeing memory before we're done with it
- **Double free**: freeing memory repeatedly
- **Invalid free**: passing some other value (not returned by `malloc()`) to `free()`

Other calls

- `calloc()` allocates memory and also zeroes it before returning
- `realloc()` makes a new larger region of memory, copies the old region into it, and returns the pointer to the new region

## 2 Intro to Operating Systems

Von Neumann Model of Computing

1. The processor *fetches* an instruction from memory
2. The processor *decodes* the instruction
3. The processor *executes* the instruction

The **operating system (OS)** is in charge of making sure the system operates correctly and efficiently in an easy-to-use manner

- The OS does this through **virtualization**
  1. The OS takes a *physical* resource (e.g. processor, memory, disk)
  2. Transforms it into a more general, powerful, and easy-to-use *virtual* form of itself

  – The OS is also known as a **virtual machine**
- The OS provides some interfaces (APIs) that we can call
  – The OS provides a *standard library* to applications
- The OS is also known as a **resource manager**
  – Each of the CPU, memory, and disk is a **resource** of the system
  – It is the OS's role to *manage* those resources

Virtualizing the CPU

- The OS creates the illusion that the systemn has a very large number of virtual CPUs
- **Virtualizing the CPU**: allows many programs to seemingly run at once

Virtualizing Memory

- Different programs could independently use the same memory address
- **Virtualizing memory**: each process accesses its own private **(virtual) address space**, which the OS maps onto the physical memory of the machine
- The memory reference within one running program does not affect the address space of other processes

Concurrency

- Things could go unexpected when multiple processes are running at the same time

Persistence

- Data can be easily lost in system memory
  – **Volatile**: data is lost when the power is turned off
- We need hardware and software to be able to store data **persistently**
- The software in the OS that usually manages the disk is the **file system**, which is responsible for storing **files** the user creates
- The OS does not create a private, virtualized disk for each application
- It is often assumed that user want to *share* information that is in files

Goals of OS

- Abstraction: makes the system convenient and easy to use
- Protection between applications: prevents one application from affecting another (isolation)
- Reliability: because if the OS fails, all the applications running on the system fail as well

Bootstrapping

- Hardware stores small program in non-volatile memory
- BIOS: basic input/output system
- Knows how to access simple hardware devices (e.g. disk, keyboard, display)
- When power is first supplied:
    1. Bootloader is loaded from boot sector of hard drive
    2. Bootloader finds and loads OS code into memory
    3. Starts running OS code

OS Startup

1. Initialize internal data structures
2. Create first process (init)
3. Switch ode to user and start running first process
4. Wait for something to happen (interrupt/exception)
5. OS is entirely driven by external events (i.e. external to the OS)

## 3 Processes

**Process**: a running program

- **Time sharing** of CPU allows users to run as many concurrent processes as they want
- In reality the CPU is only running one process at a time
- **Machine state**: what a program can read or update when it is running
  - Includes memory, registers (e.g. program counter, stack pointer), and I/O devices

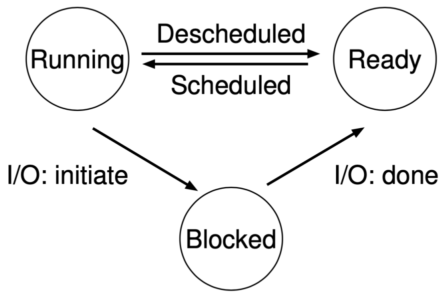
Process API

- **Create**: create a new process
- **Destroy**: destroy a process
- **Wait**: wait for a process to finish
- **Miscellaneous control**: including suspend/resume processes
- **Status**: get information about a process, e.g. how long it has run for, its state, etc.

Process Creation

1. Load the code and any static data (e.g. initialized variables) into memory
   - Programs initially reside on disk in some kind of *executable format*
   - Can be done **eagerly** (all at once before running the program) or **lazily** (as needed)
2. Allocate memory for the program's **(run-time) stack**
   - E.g. fill in the parameters to the `main()` function, i.e. `argc` and `argv`
3. Allocate memory for the program's **heap**
   - Used for explicitly requested dynamically-allocated data
   - The heap can be initially small, and the OS can allocate more memory as needed
4. Perform some initialization tasks
   - E.g. initialize three open *file descriptors* for standard input, output, and error
5. Start the program running at an entry point (e.g. `main()`)

Process States



- **Running**: the process is running on a processor, i.e. it is executing instructions

- **Ready**: the process is ready to run but the OS has chosen not to run it at the moment
- **Blocked**: the process is not ready to run until some other event takes place (e.g. initiated an I/O request to a disk)

Data Structures

- To track the state of each process, the OS maintains a **process list**
- **Process control block (PCB)**: the individual structure that stores information about a process
  - An **address space**, which contains the code and data for the executing program, and an execution stack
  - Program counter indicating the next instruction
  - Process state
  - A set of general-purpose registers with current values (must be saved at an interrupt)
  - CPU scheduling information (e.g. priority)
  - Memory management info (e.g. page tables)
  - Accounting information (e.g. resource use info)
  - A set of OS resources (e.g. open files, network connections, signals, etc.)
  - Process ID

## 4 Process API

The `fork()` system call

- Creates a new process
- The new process (called the **child**) is almost an exact copy of the calling process (called the **parent**)
- The child starts executing at the same point as the parent
- `fork()` returns `0` for the child and the child's process ID for the parent
- The output is not deterministic as the parent and child may run in any order

The `wait()` system call

- Waits for a child process to terminate

The `exec()` system call

- Run a program that is different from the calling program
- Given the name of an executable and some arguments, it *loads* code (and static data) from that executable and overwrites its current code segment (and current static data) with it
- The entire memory space of the program is re-initialized
- Does not create a new process, instead, it transforms the current running program into a different running program
- A successful call of `exec()` never returns

The `exit()` system call

- On exit, a process voluntarily releases all resources
- The OS does *not* discard them immediately (it may retain the return value)
- May result in zombie, which retains its PID
- Zombie remains until its parent cleans it up

Process Control and Users

- The `kill()` system call sends *signals* to a process
- There are shortcuts in the shell to send signals to processes
- A process should use the `signal()` system call to "catch" various signals
- For security, users are not allowed to send signals to processes they do not own

## 5 Direct Execution

Limited Direct Execution

- **Direct execution**: run the program directly on the CPU
- We need to ensure that the program does not do anything illegal
- We want to be able to stop the process and run another for virtualization

Restricted Operations

- **User mode**: code that runs in user mode is restricted in what it an do (e.g. issuing I/O requests)
- **Kernel mode**: the mode that the OS runs in, which can run whatever it likes
- When a user process wants to perform a privileged operation (e.g. reading from disk), it performs a **system call**
  - System call: a function call that invokes the OS
  - Kernel must verify arguments that it is passed
  - All user pointers must be verified (i.e. using `copy_from_user()`, `copy_to_user()`)
- To execute a system call, a program must execute a special **trap** instruction
  - This instruction jumps into the kernel and the system can perform such privileged operations (if allowed)
  - The processor pushes the program counter, flags, and a few other registers onto a per-process **kernel stack**
- When finished, the OS calls a special **return-from-trap** instruction, which returns into the calling user program and reduces the privilege level back to user mode
  - The kernel stack values are popped off
- To determine what code to execute upon a trap, the kernel sets up a **trap table** at boot time
  - Determines what code to run when each kind of interrupt takes place
  - The OS then informs the hardware the locations of the **trap handlers**
  - The instruction to tell the hardware where the trap tables are is a privileged operation
- To specify the exact system call, a **system-call number** is assigned to each system call
  - The user code places the desired system-call number in a register or at a specified location on the stack
  - The OS examines this number and executes the corresponding code
  - User code cannot specify an exact address to jump to, it can only request a particular service

Boundary Crossings

1. Getting to kernel mode
   - Boot time (not crossing, starts kernel)
   - Hardware interrupt
   - Software exception
   - Explicit system call

2. Kernel to user
    - OS sets up registers, MMU, mode for application
    - Jumps to next application instruction

Two Phase in the **Limited Direct Execution (LDE)** Protocol

1. Boot time
    - The kernel initializes the trap table, and the CPU remembers its location for subsequent use
2. When running a process
    - Before using a return-from-trap instruction, the kernel sets up a few things (e.g. allocating memory), and switches the CPU to user mode
    - When the process wants to issue a system call, it traps back to the OS, which handles it and returns control via a return-from-trap to the process
    - When the process returns (e.g. by calling the `exit()` system call), the OS cleans up

Switching Between Processes

- A process is running on the CPU means that the OS is not running, yet the OS wants to switch between processes
- **Cooperative approach**: the OS trusts the processes of the system to behave reasonably
    - Processes that run for too long are assumed to periodically give up the CPU so that the OS can run
    - Processes can transfer control of the CPU to the OS by making a *system call*
    - Applications also transfer control to the OS by doing something illegal (e.g. dividing by zero, accessing memory that it shouldn't be able to access)
        * It will generate a **trap** to the OS
        * Then the OS will have control of the CPU
    - When a process gets stuck in an infinite loop, then we would have to reboot the machine
- **Non-cooperative approach**: timer interrupt
    - A timer device raises an interrupt every so often
    - When the interrupt is raised, the currently running process is halted, and a pre-configured *interrupt handler* in the OS runs
    - When the interrupt occurs, the hardware has to save enough of the state of the program that was running so that it can resume later
- When the OS regains control, it could choose to continue running the same process, or switch to a different one
    - The decision is made by the **scheduler**
- If the decision is to switch, the OS executes a **context switch**
    - The OS saves a few register values (e.g. stack pointer, program counter, etc.) from the current process, and loads a few register values from the next process
    - The OS finally executes a return-from-trap instruction, which resumes the next process and completes the context switch

Concurrency

- The OS needs to know what happens when an interrupt occurs during interrupt or trap handling
- The OS could *disable interrupts* during interrupt processing
  - This might lead to lost interrupts

Interrupt

- **Interrupt**: a hardware signal that causes the CPU to jump to a predefined instruction (interrupt handler)
- May be caused by hardware or software
- Any illegal instruction executed by a process causes a software-generated interrupt (aka an exception)

Workflow with Interrupts

1. OS fills in interrupt table at boot time, sets IDTR (interrupt descriptor table register) to point to it
2. CPU execution loop (fetch instruction at PC, decode instruction, execute instruction, repeat)
3. Interrupt occurs (signal from hardware)
4. CPU changes mode, disables interrupts
5. Interrupted PC value is saved
6. IDTR + interrupt number is used to set PC to start of interrupt handler
7. Execution continues

## 6 Threads

A **Multi-threaded** program has more than one point of execution

- Each thread is like a different process, except that they share the same address space and can access the same data
- Each thread has its own private set of registers
- Need **thread control blocks (TCBs)** to store the state of each thread of a process
- When a context switch is performed between threads, the address space remains the same
- There will be one stack per thread
  - What's on the stack will be placed in the **thread-local** storage
- On one processor, the OS decides which thread to run

Use of Threads

- **Parallelization**: transforming a single-threaded program into a multi-threaded program to run faster
- Avoids blocking program progress due to slow I/O

Uncontrolled Scheduling

- When two threads are incrementing the same variable (adding 1 to the variable 1 million times), the result is not deterministic (almost never equal 2 million)
- When we increment a variable, we are actually doing three things
  1. Load the value of the variable into a register
  2. Add 1 to the register
  3. Store the value of the register back into the variable
- When a timer interrupt goes off when a thread finishes step 2, the OS saves the state of the thread and switches to another thread
- The new thread still has the old value of the variable in its register, and it increments it a few times
- When another context switch occurs, the old thread still has the old value of the variable in its register and it performs step 3 and stores the old value back into the variable
- This is a **race condition**, which makes the result **indeterminate** (i.e. not known in advance, different between runs)
- **Critical section**: a piece of code that accesses a shared variable/resource and must not be concurrently executed by more than 1 thread
- We want to have **mutual exclusion**, which guarantees that if one thread is executing within the critical section, the others will be prevented from doing so

Atomicity

- If we have a "super instruction" that combines the above three instructions, then at a timer interrupt, it either not run or run to completion
- Usually we won't have such an instruction
- **Atomic**: running as a unit, i.e. "all or none"

- We could ask a hardware for a few useful instructions upon which we can build a general set of **synchronization primitives**

Waiting for Another

- Sometimes one thread needs to wait for another thread to finish (e.g. when a thread is waiting for a disk I/O to complete)

User-Level Threads

- Kernel-level threads make concurrency much cheaper than processes, however, it is still expensive
- Can implement threads at user level to make them cheap and fast
- Kernel-level threads are managed by the OS
- User-level threads are managed entirely by the run-time system
- All thread operations can be done by procedure call
- User-level threads are invisible to the OS, thus OS may make poor decisions
  - Scheduling a process with idle threads
  - Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
  - De-scheduling a process with a thread holding a lock

Comparison to Process

- Processes are safer and more secure, since each process ahs its own address space
- A thread crash takes down all other threads
- A thread's buffer overrun creates a security problem

## 7 Thread API

Thread Creation

- Use the `pthread_create()` function

```c
#include <pthread.h>
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg);
```

- `thread` is a pointer to a `pthread_t`, which we use to interact with the thread
- `attr` specifies any attributes this thread might have
  - E.g. stack size, scheduling priority, etc.
  - An attribute is initialized with `pthread_attr_init()`
- `start_routine` is the function that the thread will run (i.e. a **function pointer**)
- `arg` is the argument to be passed to the function where the thread begins execution

Thread Completion

- To wait for thread completion, use the `pthread_join()` function

```c
int pthread_join(pthread_t thread, void **value_ptr);
```

- `thread` is the thread to wait for
- `value_ptr` is a pointer to the return value that we expect to get back
  - We pass the pointer to that value, not just the value itself

Locks

- Locks are used for providing mutual exclusion to critical sections

```c
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Usage: (missing initialization)

```c
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // operate on critical section
pthread_mutex_unlock(&lock);
```

- If no other thread holds the lock when `pthread_mutex_lock()` is called, then the thread will acquire the lock and enter the critical section
- If another thread holds the lock, the thread trying to grab the lock will not return from the call until it has acquired the lock
- All locks must be properly initialized

- Two ways to initialize locks
  1. Use `PTHREAD_MUTEX_INITIALIZER`

     ```c
     pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
     ```

  2. Make a call to `pthread_mutex_init()` (can be done at runtime)

     ```c
     int rc = pthread_mutex_init(&lock, NULL);
     assert (rc == 0); // check for success
     ```

- For the second method, a call to `pthread_mutex_destroy()` should be made when we are done with the lock
- Failure checking calls are provided within the pthreads library

```c
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex, const struct timespec *abs_timeout);
```

- The `trylock` version returns failure if the lock is already held
- The `timedlock` version returns after a timeout or after acquiring the lock (whichever comes first)

Condition Variables

- **Condition variables** are useful when some kind of signaling must take place between threads

```c
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

- To use a condition variable, we need to have a lock associated with this condition – when calling the above routines, this lock should be held
- `pthread_cond_wait()` puts the calling thread to sleep, and waits for some other thread to signal it, e.g.

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (!ready) {
   pthread_cond_wait(&cond, &lock);
}
pthread_mutex_unlock(&lock);
```

  – The while loop (instead of an if statement) is necessary

- The code (from another thread) to wake the above thread up is like

```c
pthread_mutex_lock(&lock);
ready = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```

- In the wait call, in addition to putting the calling thread to sleep, the lock passed in is released

- Before returning after being woken, the `pthread_cond_wait()` re-acquires the lock
- Do *not* use a flag instead of a condition variable

Compiling

- Include the header `pthread.h`
- Explicitly link with the pthreads library when compiling, i.e.

```
gcc -o main main.c -Wall -pthread
```

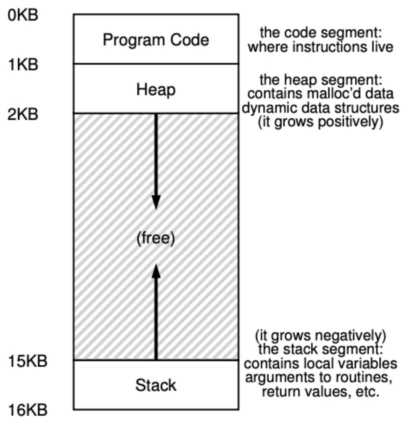
## 8 Address Spaces

Multiprogramming and Time Sharing

- **Multiprogramming**: multiple processe are ready to run at the same time, and the OS would switch between them
- **Time sharing**: when we switch between processes, we leave them in memory (rather than swapping them out to disk)

Address Space

- **Address space**: an abstraction of physical memory, the running program's view of memory
- Contains all of the memory state of the running program, e.g.
  - The *code*: the instructions
  - The *stack*: the local variables and function calls
  - The *heap*: dynamically allocated memory


0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows positively)
(it grows negatively) the stack segment: contains local variables arguments to routines, return values, etc.

- The heap and the stack may grow and shrink while the program runs
  - We place them like this to allow growth

Goals of Virtualizing Memory

- **Transparency**: virtual memory should be implemented in a way that is invisible to the running program (hard to notice)
- **Efficiency**: in terms of both time and space
- **Protection**: protect processes from one another and protect OS from processes
  - *Isolation* among processes

# 9 Locks

Locks

- A **lock** ensure that a critical section executes atomically
- E.g.

```
lock_t mutex; // some globally-allocated lock
...
lock(&mutex);
// critical section
unlock(&mutex);
```

- Need to declare a **lock variable**, e.g. `lock_t mutex`
- The lock variable holds the state of the lock
- It is either **available/unlocked/free** where no thread holds the lock, or **acquired/locked/held**, where exactly one thread holds the lock
- `lock()` tries to acquire the lock
  - If no other thread holds the lock, the thread will acquire the lock and enter the critical section
  - The thread is the **owner** of the lock
  - If another thread then calls `lock()` on the same lock variable, it will not return while the lock is held by another thread
  - Other threads are prevented from entering the critical section while the first thread that holds the lock is in there
- `unlock()` releases the lock
  - If no other threads are waiting for the lock, the state of the lock is changed to free
  - If there are threads waiting, one of them will notice this change of the lock's state, acquire the lock, and enter the critical section

Pthread Locks

- **Mutex**: short for **mutual exclusion**, the name that the POSIX library uses for a lock
- Equivalent code to the above:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&lock);
// critical section
pthread_mutex_unlock(&lock);
```

Evaluating Locks

1. *Mutual exclusion*: does the lock work?
2. *Fairness*: does each thread contending for the lock get a fair chance to acquire it?
3. *Performance*: time overheads added by using the lock

Controlling Interrupts

- For single-processor systems, a solution is to disable interrupts for critical sections
  - Locking disable interrupts, unlocking enable interrupts
- Requires us to allow any calling thread to perform a *privileged operation*, trusting that this facility is not abused, which is a bad idea
- Does not work on multiple processors, since even if interrupts are disabled, threads on other processors can still run
- Turning off interrupts can lead to interrupts becoming lost
- This approach can be used within the OS itself

Using Flags (Loads/Stores)

- Use a flag to indicate whether some thread has possession of a lock
- If another thread calls lock while the flag is set, it *spin-wait* in a while loop until the flag is cleared
- Could be incorrect since an interrupt could happen before a thread sets the flag
- Spin-waiting is inefficient
  - On a single processor, the spinning thread does not do anything

Test-and-Set

- Hardware support for locking
- **Test-and-set** (or **atomic exchange**) is an instruction defined by the pseudocode

```
int TestAndSet(int *old_ptr, int new) {
   int old = *old_ptr; // fetch old value at old_ptr
   *old_ptr = new; // store value of 'new' into old_ptr
   return old; // return the old value
}
```

- *Tests* the old value (returned) and *sets* the memory location to a new value
- Can build a lock as follows

```
typedef struct __lock_t {
   int flag;
} lock_t;

void int(lock_t *lock) {
   // 0: locak is available, 1: lock is held
   lock->flag = 0;
}

void lock(lock_t *lock) {
   while (TestAndSet(&lock->flag, 1) == 1)
      ;  // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
   lock->flag = 0;
}
```

- If the test-and-set gives 0, then we don't enter the while loop, and acquire the lock by atomically setting the flag to 1
- If the test-and-set gives 1, then we enter the while loop, and spin-wait until the flag is set to 0
- Known as a **spin lock**
- On a single processor, requires a *preemptive scheduler*, otherwise the thread spins forever

Spin Lock

- Provides mutual exclusion
- Spin locks don't provide fairness since they may spin forever
- Performance is bad on single processor because if every thread (except for one) is spinning on a lock, then whenever the scheduler doesn't choose the one, the other threads will spin
- On multiple processors, performance is good if # threads equals # processors, because spinning to wait for a lock held on another processor doesn't waste many cycles

Compare-and-Swap

- **Compare-and-swap** (or **compare-and-exchange**) is an instruction defined as follows:
  - Takes in a pointer `ptr`, a value `expected`, and a value `new`
  - Test whether the value at `ptr` is equal to `expected`
  - If so, update the value at `ptr` to `new`
  - If not, do nothing
  - Return the old value at `ptr`
- Can build a lock similar to the test-and-set lock, modifying the `lock()` function as follows:

```c
void lock(lock_t *lock) {
   while (CompareAndSwap(&lock->flag, 0, 1) == 1)
      ;  // spin
}
```

- If the flag is 0, then we atomically set it to 1, acquiring the lock
- Otherwise, we spin-wait until the flag is 0

Load-Linked and Store-Conditional

- **Load-linked** is an instruction that fetches a value from memory and places it in a register

```c
int LoadLinked(int *ptr) {
   return *ptr;
}
```

- **Store-conditional** is an instruction that stores a value from a register into memory, but only succeeds if no other writes to that memory location have occurred since the load-linked
  - If it succeeds, then it returns 1 and updates the value
  - If not, then it returns 0 and does not update the value

```c
int StoreConditional(int *ptr, int value) {
    if (no update to *ptr since LoadLinked to this address) {
        *ptr = value;
        return 1; // success
    } else {
        return 0; // failure
    }
}
```

- Can build a lock as follows:
  1. A thread spins waiting for the flag to be set to 0
  2. When the flag is 0, the thread tries to acquire the lock via the store-conditional
  3. If it succeeds, the thread has atomically changed the flag's value to 1, and acquired the lock
- If one thread calls `lock()` and executes the load-linked, returning 0 as the lock is not held
- If it is interrupted before it can execute the store-conditional, then another thread executes the load-linked instruction, also getting 0 and continuing
- At this point, two threads have each executed the load-linked and each are about to attempt the store-conditional
- Only one of them will succeed, which is the one that called load-linked most recently
- Locking/unlocking defined as follows:

```c
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ;  // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // success
        // otherwise try again
    }
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Fetch-and-Add

- **Fetch-and-add** is an instruction that atomically increments a value while returning the old value at a particular address, defined as follows:

```c
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

- Can be used to build a **ticket lock** as follows:

```c
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

- When a thread wants to acquire the lock, it atomically increments the ticket number and spins until the turn number equals the ticket number
- Ensure progress for all threads, since once a thread is assigned its ticket value, it will be scheduled at some point

Yield

- When a thread is about to spin, it could just give up the CPU to another thread
- Yield is a system call that moves the caller from the *running* state to the *ready* state
- The thread **deschedules** itself
- Still expensive since context switch is costly

Sleeping

- Can use a queue to keep track of which threads are waiting to acquire the lock
- `park()`: puts the calling thread to sleep
- `unpark(threadID)`: wakes up a thread with the given ID
- Implementation does not entirely avoid spinning, but it does reduce it

```c
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_Init(m->q);
}

```

```
void lock(lock_t *m) {
   while (TestAndSet(&m->guard, 1) == 1)
       ; // acquire guard lock by spinning
   if (m->flag == 0) {
      m->flag = 1; // acquire lock
      m->guard = 0;
   } else {
      queue_add(m->q, gettid());
      m->guard = 0;
      park(); // block
   }
}

void unlock(lock_t *m) {
   while (TestAndSet(&m->guard, 1) == 1)
       ; // acquire guard lock by spinning
   if (queue_empty(m->q)) {
      m->flag = 0; // release lock
   } else {
      unpark(queue_remove(m->q)); // unblock
   }
   m->guard = 0;
}
```

- After unlock, we don't reset the flag, since the thread that we unpark would wakes up with the lock
- Problem: when a thread is enqueued but before parking, a context switch occurred – then its unpark could happen before park, permanently sleeping it
- Could use a system call `setpark()`, which indicates that the thread is about to park, so that if the situation above happens, the subsequent park returns immediately

Futex

- Supported by the Linux kernel
- `futex_wait(address, expected)` puts the calling thread to sleep if the value at `address` is equal to `expected`, otherwise returns immediately
- `futex_wake(address)` wakes up a thread sleeping on `address`

Two-Phase Locks

- In the first phase, the lock spins for a while, hoping that it can acquire the lock
- If the lock is not acquired during the first phase, a second phase is entered, where the caller is put to sleep, and only wakes up when the lock becomes free later
- The following Linux futex lock is a two-phase lock that spins *once* in the first phase

```
void mutex_lock(int *mutex) {
   int v;
   if (atomic_bit_test_set(mutex, 31) = 0) {
      return; // lock acquired since high bit is 0
   }
   atomic_increment(mutex); // increment lock count
   while (1) {
      if (atomic_bit_test_set(mutex, 31) == 0) {
         atomic_decrement(mutex);
```

```
            return; // lock acquired
        }
        // wait first to make sure the futex value is negative (locked)
        v = *mutex;
        if (v >= 0)
            continue;
        futex_wait(mutex, v); // it's locked, sleep
    }
}

void mutex_unlock(int *mutex) {
    if (atomic_add_zero(mutex, 0x80000000)) {
        // adding 0x80000000 to counter results in 0 iff there are no waiting threads
        return;
    }

    // there are waiting threads, wake one up
    futex_wake(mutex);
}
```

- The high bit of mutex is used to indicate whether the lock is free or not, so that negative means the lock is held, and positive means the lock is free
- The low bits are used to count the number of threads waiting for the lock

# 10 Locked Data Structures

Example: Concurrent Counter

- Want to have a counter which we can increment (+1) and decrement ($-1$)
- To make it concurrent we add a lock, which is acquired when we modify the counter and released when we are done
- The result is that when we have more threads, there is a slowdown (not scalable)
- We want **perfect scaling**, which is when the work done by $n$ threads is $n$ times the work done by 1 thread under the same time

Approximate Counter

- Represent a single logical counter via numerous *local* physical counters, one per CPU core, as well as a single *global* counter
- One lock per each counter (since we assume there may be $\geq 1$ thread on each core)
- When a thread running on a core wants to increments the counter, it increments its local counter
- The local counter is synchronized via the local lock
- The local values are periodically transferred to the global counter, by acquiring the global lock and incrementing it by the local counter's value, then resetting the local counter to 0
- The local-to-global transfer depends on the threshold $S$
  - When $S$ small, it behaves like the non-scalable counter
  - When $S$ large, then the global value might be off
  - Syncing makes it not as scalable

Concurrent Linked Lists

- Have a lock on the list, then whenever we want to modify the list (write) or traverse it (read), we acquire the lock
- Do not unnecessarily lock, i.e. when mallocing a new node
- Only lock when we are modifying the list
- Does not scale well

Hand-Over-Hand Locking

- Have a lock per node of the list
- When traversing the list, the code first grabs the next node's lock and the releases the current node's lock
- Theoretically faster but acquiring and releasing lock is expensive

Concurrent Queues

- Have a lock on the head node and on the tail node, so that enqueueing and dequeueing can be done concurrently

Concurrent Hash Table

- Use a lock per hash bucket, which enables many concurrent operations to take place

## 11 Condition Variables

Condition Variable

- **Condition variable**: an explicit queue that threads can put themselves on when some state of execution (i.e. condition) is not as desired (by waiting on the condition)
- Some other thread, when it changes state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signalling on the condition)
- To declare a condition variable:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- A condition variable has two operations associated with it: `wait()` and `signal()`
  - `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)` is executed when a thread wants to put itself to sleep
    - ∗ Assumes the mutex is locked when `wait()` is called
    - ∗ Releases the lock and puts the calling thread to sleep (atomically)
    - ∗ When the thread wakes up, it must re-acquire the lock before returning to the caller
  - `pthread_cond_signal(pthread_cond_t *c)` is executed when a thread has changed something in the program and wants to wake a sleeping thread waiting on the condition
- It is best to hold a lock while signalling when using condition variables
- Must hold a lock while waiting (due to precondition)

Lost Wake-Up Problem

- Due to race conditions, signal may be called before wait, so the waiting thread waits forever
- Add a shared variable to indicate whether signal was sent

The Producer/Consumer (Bounded Buffer) Problem

- $\geq 1$ producer threads and $\geq 1$ consumer threads
- Producers generate data items and place them in a buffer
- Consumers grab data items from the buffer and consume them in some way
- Need a shared buffer, could use an integer for simplicity

```
void *producer(void *arg) {
    int i;
    int loops = (int) arg;
    for (i = 0; i < loops; i++) {
        put(i);
    }
}

void *consumer(void *arg) {
    while (1) {
        int tmp = get();
        printf("%d\n", tmp);
    }
}
```

- Assume there is a `count` variable which indicates whether the buffer is empty (0) or full (1)
- Only put data into the buffer when `count` is 0, and only take data out of the buffer when `count` is 1
- Signalling a thread only indicates that the state has changed, but does not guarantee that when the woken thread runs, the state will still be as desired
- **Mesa semantics**: when a waiting thread is signalled, it does not necessarily run immediately (more commonly used)
  - With condition variables, always use while loops (instead of if statements)
- **Hoare semantics**: when a waiting thread is signalled, it runs immediately
- A consumer should only wake producers, and vice versa
  - Use two condition variables
- Working example:

```c
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    int loops = (int) arg;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == 1) {
            pthread_cond_wait(&empty, &mutex);
        }
        put(i);
        pthread_cond_signal(&fill);
        pthread_mutex_unlock(&mutex);
    }
}

void consumer(void *arg) {
    int i;
    int loops = (int) arg;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == 0) {
            pthread_cond_wait(&fill, &mutex);
        }
        int tmp = get();
        pthread_cond_signal(&empty);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

- Can generalize to multiple buffer slots

Covering Condition

- Sometimes when we signal, there are some threads that would block and some that would work – we want to be able to signal the work ones

- We could use `pthread_cond_broadcast()`, which wakes up all waiting threads
- Guarantees that any threads that should be woken are
- Expensive since it wakes up a lot of threads and then puts them back to sleep

## 12 Semaphores

Semaphore

- A **semaphore** is an object with an object with an integer value that we can manipulate with two routines: `sem_wait()` and `sem_post()`

```c
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // initialize it to value 1

int sem_wait(sem_t *s) {
   decrement the value of semaphore s by 1
   wait if value of semaphore s is negative
}

int sem_post(sem_t *s) {
   increment the value of semaphore s by 1
   if there are 1 or more threads waiting, wake one
}
```

- When the value of the semaphore is negative, it is equal to the number of waiting threads

Binary Semaphores (Locks)

- Can use a semaphore as a lock

```c
sem_t m;
sem_init(&m, 0, 1);

sem_wait(&m);
// critical section
sem_post(&m);
```

- **Binary semaphore**: a semaphore used as a lock
- Difference: lock has the notion of an owner, can only be released by its owner

Semaphores For Ordering

- Can use a semaphore to wait for an event to happen (similar to the use of condition variables)
- E.g. parent waiting for child

```c
sem_t s;

void *chiLd(void *arg) {
   printf("child\n");
   sem_post(&s); // signal that child is done
}

int main() {
   sem_init(&s, 0, 0);
   pthread_t c;
   pthread_create(&c, NULL, child, NULL);
   sem_wait(&s); // wait for child
   printf("parent\n");
   return 0;
}
```

The Producer/Consumer (Bounded Buffer) Problem

- Consider the number of resources we want to give away immediately after initialization, and use it to initialize the semaphore
- Initialize the empty and full semaphores

```c
sem_t empty, full, mutex;
sem_init(&empty, 0, MAX);
sem_init(&full, 0, 0);
sem_init(&mutex, 0, 1);
```

- The producer and consumer code is as follows:

```c
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty); // wait for empty slot
        sem_wait(&mutex); // lock
        put(i);
        sem_post(&mutex);
        sem_post(&full); // signal full slot
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full); // wait for full slot
        sem_wait(&mutex); // lock
        int tmp = get();
        sem_post(&mutex);
        sem_post(&empty); // signal empty slot
        printf("%d\n", tmp);
    }
}
```

Reader-Writer Locks

- There can only be one writer at a time
- There can be multiples readers at a time as long as there is no writer

```c
typedef struct _rwlock_t {
    sem_t lock; // binary semaphore
    sem_t writelock; // allows 1 writer or multiple readers
    int readers; // number of readers in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
```

```
        rw->readers++;
        if (rw->readers == 1) {
            // first reader gets writelock
            sem_wait(&rw->writelock);
        }
        sem_post(&rw->lock);
    }

    voide rwlock_release_readlock(rwlock_t *rw) {
        sem_wait(&rw->lock);
        rw->readers--;
        if (rw->readers == 0) {
            // last reader lets it go
            sem_post(&rw->writelock);
        }
        sem_post(&rw->lock);
    }

    void rwlock_acquire_writelock(rwlock_t *rw) {
        sem_wait(&rw->writelock);
    }

    void rwlock_release_writelock(rwlock_t *rw) {
        sem_post(&rw->writelock);
    }
```

- First reader acquires the write lock
- Any thread that wishes to acquire the write lock will have to wait until all readers are finished
- Not fair since the readers can starve the writers

The Dining Philosophers

- Assume there are 5 philosophers sitting around a table
- Between each pair of philosophers is 1 fork (i.e. 5 in total)
- The philosophers either think (does not need a fork) or eat (needs two forks, left and right)
- For each thread with identifier $p = 0, 1, 2, 3, 4$

```
    while (1) {
        think();
        get_forks(p);
        eat();
        put_forks(p);
    }
```

- Want to ensure no philosophers starve, and concurrency is high (i.e. as many philosophers can eat at the same time as possible)
- If we lock the fork on the left and lock a fork on the right, then we can have a deadlock (i.e. every philosopher grabbed one fork and no progress can be done)
- If one philosopher locks in a different order (right then left), then the problem could be solved

Thread Throttling

- **Throttling**: limiting the number of threads concurrently executing a piece of code
- Can initialize a semaphore to the maximum number of threads, and put `sem_wait()` and `sem_post()` around the region

## 13 Concurrency Bugs

Non-Deadlock Bugs

- Atomicity violation
- Order violation

Atomicity-Violation Bugs

- The code region is intended to be atomic, but the atomicity is not enforced during execution
- E.g. one thread checking if a field is not null, then does something to the field; another thread sets the field to null
  - Checking not-null and doing something to the field should be atomic
- Can add locks to fix this

Order-Violation Bugs

- The code region is intended to be executed in a certain order, but the order is not enforced during execution
- E.g. one thread creating an object, another thread assumes the object is there and uses it
  - Creating the object should happen before the object is used
- Can use condition variables to fix this

Deadlock Bugs

- Deadlock occurs when a thread 1 is holding lock $L_1$ and waiting for lock $L_2$, and thread 2 is holding lock $L_2$ and waiting for lock $L_1$
- Four conditions for a deadlock to occur:
  1. *Mutual exclusion*: thread claim exclusive control of resources (locks) that they require
  2. *Hold-and-wait*: threads hold resources (locks) allocated to them while waiting for additional resources (locks)
  3. *No preemption*: resources (locks) cannot be forcibly removed from threads that are holding them
  4. *Circular wait*: there exists a circular chain of threads such that each thread holds resources (locks) that are being requested by the next thread in the chain

Circular Wait

- It is best to write code that never induces a circular wait
- Can provide a *total ordering* on lock acquisition
- If that's too difficult, can provide a *partial ordering* instead (which could still be difficult)

Hold-and-Wait

- Can be avoided by atomically acquiring all locks at once
- Requires a global lock for the lock acquisition section
- Requires us to know which locks must be held and to acquire them ahead of time, which breaks encapsulation

No Preemption

- Can use `pthread_mutex_trylock()` to avoid this

```
top:
pthread_mutex_lock(L1);
if (pthread_mutex_trylock(L2) != 0) {
   pthread_mutex_unlock(L1);
   goto top;
}
```

- **Livelock**: two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks
  - Could add a random delay after each iteration of the loop
- Breaks encapsulation if the two lock acquisitions are far apart
  - A jump could be complex to implement
  - Need to free all resources that are acquired after first lock is acquired

Mutual Exclusion

- Can work around with hardware support (e.g. use compare-and-swap to update a value)

Deadlock Avoidance via Scheduling

- Avoidance requires some global knowledge of which locks threads might grab during their execution
- Schedule in a way that guarantees no deadlock
- E.g. if two threads may cause a deadlock, then schedule them so that they don't run at the same time
- Sometimes we may need to run many threads sequentially, which limits concurrency

Detect and Recover

- Ostrich Algorithm: ignore the problem and hope it doesn't happen often
- Allow deadlocks to occasionally occur, and take some action when they do
- E.g. if an OS freeze once a year, then we manually reboot it
- Can periodically run a deadlock detector, then restart the program if a deadlock is detected (commonly used in databases)

# 14 Scheduling

Intro

- **Workload**: the processes running in the system
- **Job**: process
- Some assumptions we make are unrealistic, but we relax them as we go

Workload Assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. Once started, each job runs to completion
4. All jobs only use the CPU (i.e. no I/O)
5. The runtime of each job is known

Scheduling Metrics

- **Turnaround time**: the time at which the job completes minus the time at which the job arrived in the system:
$$T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$$
- Since we assume that all jobs arrive at the same time, $T_{\text{arrival}} = 0$
- Turnaround time is a *performance metric*
- We could also have a *fairness metric*

First In, First Out (FIFO)

- Can have a **first in, first out (FIFO)** scheduling algorithm (also known as **first come, first served (FCFS)**)
- Average waiting time often long
- When we relax assumption 1, we may have a job that arrives first, and runs for a long time, blocking all other (shorter) jobs from running (i.e. **convoy effect**)

Shortest Job First (SJF)

- Runs the shortest job first, then the next-shortest, etc.
- If we assume that all jobs arrive at the same time (assumption 2), this is the optimal algorithm
- Starvation is possible
- When we relax assumption 2, short jobs could arrive after a long job was started, so they still have to wait

Shortest Time-to-Completion First (STCF)

- Relax assumption 3 so that we could preempt jobs
- **Shortest time-to-completion first (STCF)** is an algorithm that adds preemption to SJF (also known as **preemptive shortest job first (PSJF)**)

- Whenever a new job enters the system, the scheduler determines which of the remaining jobs has the least time left, and schedules that one
- Optimal under the current set of assumptions (i.e. 4 and 5)

Response Time

- **Response time** is a metric defined as the time from when the job arrives in a system to the first time it is scheduled:
$$T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$$
- The previous algorithms are not good for response time, since a job is only scheduled after some other jobs have finished

Short Term Scheduling

- Scheduling happens frequently (also called **dispatching**)
- Needs to be efficient (i.e. fast selection of next process to run, fast queue manupulation, fast context switches)

Round Robin (RR)

- **Round robin (RR)**: runs a job for a *time slice* (aka *scheduling quantum*), and then switches to the next job in the run queue; repeatedly does so until the jobs are finished
- RR is also known as **time-slicing**
- The length of a time slice must be a multiple of the timer-interrupt period
- The shorter the time slice, the better the performance under the response time metric
- However making the time slice too short would make the cost of context switching too high
- Should make the time slice long enough to *amortize* the cost of switching without making it so long that the system is no longer responsive
- Long quantum makes RR similar to FCFS
- RR is not good for the turnaround time metric
- Generally any *fair* policy (e.g. RR) performs poorly in terms of turnaround time

Incorporating I/O

- Relax assumption 4 so that jobs can do I/O
- A job won't be using the CPU during the I/O, and it's *blocked* waiting for I/O completion
- When the I/O completes, an interrupt is raised, and the OS moves the process back to the ready state
- A job may run for a period of time, issues an I/O request, then run for another period of time, etc.
- Consider each contiguous period of CPU time as a subjob, then can use STCF base on the subjobs
  - Allows for *overlap*, i.e. the CPU being used by one process while waiting for the I/O of another process to complete
  - Treating each *CPU burst* as a job enables "interactive" processes to run frequently
  - While the interactive jobs are performing I/O, other CPU-intensive jobs run

Relaxing All Assumptions

- Realistically the scheduler would *not* know the length of each job
- Makes approaches like SJF/STCF impossible

Proportional-Share Scheduling

- Group processes by user
- Ensure that each group receives a proportional share of the CPU
- Lottery scheduling
  - Each group is assigned “tickets” according to its share
  - Hold a lottery to find the next process to run
  - Each group can assign tickets to its processes as it chooses
  - A process can “loan” its tickets to another process (e.g. for priority inheritance)

## 15 Multi-Level Feedback Queue

Multi-Level Feedback Queue (MLFQ)

- Multiple level of queues, uses feedback to determine the priority of each job
- Want to optimize both *turnaround time* and *response time*
- MLFQ has a number of distinct *queues*, each assigned a different *priority level*
- At any given time, a job that is ready to run is on a single queue
- MLFQ uses priorities to decide which job should run
- If more than 1 job is on a given queue (i.e. with same priority), we use round-robin scheduling among those jobs

Basic Rules of MLFQ

1. If Priority($A$) > Priority($B$), then $A$ runs
2. If Priority($A$) = Priority($B$), then $A$ and $B$ run in RR

- MLFQ varies the priority of a job based on its *observed behaviour*
  - E.g. if a job relinquishes the CPU while waiting for input from the keyboard, then MLFQ would keep its priority high
  - If a job uses the CPU intensively for long periods of time, MLFQ would reduce its priority
- MLFQ learns about processes as they run, and then uses the history of the job to predict its future behaviour

Changing Priority

- A job's **allotment** is the amount of time a job can spend at a given priority level before the scheduler reduces its priority
- Priority adjustment rules
  3. When a job enters the system, it is placed at the highest priority
  4. If a job uses up its allotment while running, then its priority is reduced; if a job gives up the CPU (e.g. by performing an I/O operation) before the allotment is up, it stays at the same priority level (i.e. its allotment is reset)
- This tries to approximate SJF by first assuming the job is short, which would be quick to complete – if not, then we don't prioritize it
- Interactive jobs that perform I/O frequently would stay at a high priority
- Could have *starvation* if there are too many interactive jobs in the system, which would consume all CPU time, starving long-running jobs
- Can *game the scheduler* by performing I/O right before allotment is used up, which monopolizese the CPU
- If a long-running jobs suddenly becomes interactive, then it would not be treated like other interactive jobs (they have high priority)

Priority Boost

- Can periodically boost the priority of all jobs in the system
- Add a rule to the previous:
   5. After some time period $S$, move all the jobs in the system to the topmost queue
- Processes are guaranteed not to starve (i.e. when they make it to the top queue, they will share the CPU by RR)
- If a CPU-bound job becomes interactive, the scheduler treats it properly once it has received the priority boost
- If $S$ is set too high, long-running jobs could starve
- $S$ is determined by the system administrator, or machine learning methods

Prevent Gaming of Scheduler

- Can perform better accounting of CPU time at each level of the MLFQ
- When a process performs I/O, the scheduler keeps track of how much CPU time the process has used
- Can rewrite rule 4 as follows:
   4. Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced

Tuning MLFQ

- We need to parameterize such a scheduler, e.g.
   - Number of queues
   - Time slice per queue
   - Allotment
   - Frequency of priority boosting
- High-priority queues usually have shorter time slices than low-priority queues
- Can use `nice` to increase/decrease priority of a job
   - This is only an **advice**, the OS can ignore it if it wants

Priority Inversion

- A low-priority task may prevent a high-priority task from making process
- E.g. a high-priority task is waiting for a lock held by a low-priority task, which is pre-empted by a medium-priority task
- Can let the low-priority task temporarily **inherit** the priority of the high-priority task it is blocking

Credit-Based Algorithm

- The scheduler chooses process with the most credits (i.e. highest priority)
- At every timer interrupt, the running process loses 1 credit
- When a process's credit reach 0, it is suspended
- If no runnable processes have credits, perform *recrediting*

```
    credits = credits / 2 + base
```

- Prevents (starvation due to influx of new threads)

O(1) Scheduler

- Recrediting takes time proportional to the number of processes
- O(1) scheduler: two arrays of runnable processess, *active* and *expired*, when active is empty, swap the two arrays
- Also use hardware instruction to find first non-empty queue

Completely Fair Scheduler (CFS)

- Divide CPU cycles among threads in proportion to their weights
- A thread accumulates virtual runtime (vruntime) when it runs
- Threads with higher priority accumulate vruntime slower
- Thread with lowest vruntime is scheduled next
- Track per-thread wait time
- New process is assigned lowest vruntime in system, ensuring that it is quickly scheduled after creation

## 16 Address Translation

Hardware-Based Address Translation

- Also known as **address translation**
- The hardware transforms each memory access (e.g. fetch, load, store), changing the *virtual* address provided by the instruction to a *physical* address
- Goal: to create an illusion that the program has its own private memory

Assumptions

1. The user's address space must be placed *contiguously* in physical memory
2. The size of the address spae is less than the size of physical memory
3. Each address space is exactly the same size

Dynamic (Hardware-Based) Relocation

- **Dynamic relocation**: also known as **base and bounds**
- Need two hardware registers with each CPU, one **base** and one **bounds** (or **limit**)
- Each program is written and compiled as if it is loaded at address 0
- When a program starts running, the OS decides where in physical memory it should be loaded and sets the base register to that value
- Any memory address would be translated by the processor:

$$\text{physical address} = \text{virtual address} + \text{base}$$

- The bounds register is used for protection: the processor first check that the memory reference is within bounds (i.e. legal)
- If the process generates a virtual address not in $[0, \text{bounds})$, then the hardware raises an exception
- **Memory management unit (MMU)**: the part of the processor that helps with address translation (including the base and bounds registers on the CPU)
- The bounds register may either hold the size of the address space, or it could hold the physical address of the end of the address space
- The instructions of modifying base and bounds registers are privileged

Operating System Issues

- When a process is created, the OS needs to find space for its address in memory
- When a process is terminated, the OS needs to reclaim the memory
- When a context switch occurs, the OS must save and restore the base and bounds pair
  - The base and bounds registers are usually stored in the process control block (PCB) or the process structure
- The OS could move a process's address space by descheduling the process, copies the address space from the current location to the new location, then update the saved base register
- The OS must provide *exception handlers*, which is installed at boot time

## 17 Segmentation

Intro

- Though the space between stack and heap is not being used by the process, it is still taking up physical memory
- Using vanilla base and bounds is wasteful
- It is hard to run a program when the entire address space doesn't fit into memory

Segmentation: Generalized Base/Bounds

- **Segmentation**: have a base and bounds pair per logical *segment* of address space
- **Segment**: a contiguous portion of the address space
- We have three logically-different segments: code, stack, heap
- Can place each of them in different parts of the physical memory
- When performing translation, we first subtract the desired virtual address by the virtual address of the start of the segment to get an offset; then add the offset to the physical address of the start of the segment to get the physical address
- When we try to refer to an illegal address, we get a **segmentation fault**

Knowing Which Segment We Are Referring To

- Need to know the offset into a segment, and to which segment an address refers
- *Explicit approach*: chop address space into segments based on the top few bits of the virtual address
  - E.g. if we have 3 segments, we can use the top 2 bits to determine which segment we are referring to, and the rest to determine the offset
- Using the top bits limits each segment to a maximum size
  - E.g. by using the top 2 bits, a 64KB address would be split into 4 segments of 16KB each
- *Implicit approach*: the hardware determines the segment by noticing how the address was formed
  - If the address was generated from the program counter, then it is within the code segment
  - If the address is based off the stack or base pointer, then it must be in the stack segment
  - Otherwise it is in the heap segment

Stack

- Grows backwards (e.g. start at 28KB [16KB] and grows back to 26KB [14KB] in physical [virtual] address space)
- Hardware needs to know which way the segment grows
- Have the offset be negative if the segment grows backwards

Support For Sharing

- It may be useful to *share* certain memory segments between address spaces
- Can adds a few bits per segment, indicating whether or not a program can read/write/execute the segment

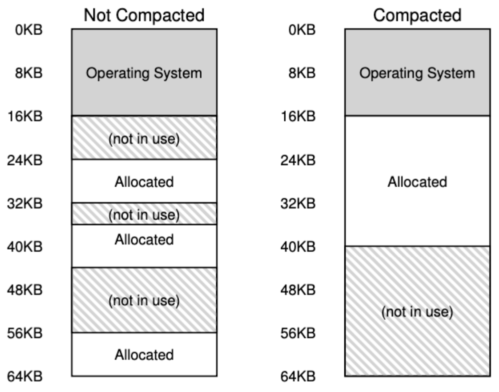
- The hardware needs to check whether a particular access is permissible (raise an exception if not)

Fine-Grained vs. Coarse-Grained Segmentation

- **Course-grained segmentation**: chops up the address space into relatively large, course chunks
  - E.g. code, stack, heap
- **Fine-grained segmentation**: allow address spaces to consist of a large number of small segments
  - Requires a **segment table** to keep track of the base and bounds of each segment

OS Support

- The unused space between the stack and the heap does not need to be allocated in physical memory
- Upon a context switch, the segment registers are saved and restored
- The heap segment itself may need to grow
  - In which case the memory allocation library performs a system call to grow the heap (e.g. `sbrk()`)
  - The OS would then update the segment size register, and inform the library of success
  - The library would then be able to allocate space for the new object
  - The OS could reject the request if no more physical memory is available, or if the calling process already has too much
- When a new address space is created, the OS has to be able to find space in physical memory for its segments
- Since different segments have different sizes, the physical memory would become full of little holes of free space, making it difficult to allocate new segments or to grow existing ones
  - **External fragmentation**: the memory is fragmented into little holes
- We could have **compact** physical memory by rearranging the existing segments



  - Compactation is expensive since copying segments is expensive
- Can use a free-list management algorithm to mitigate external fragmentation

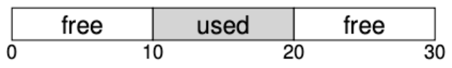
# 18 Free Space

Assumptions

- Assume a basic interface (e.g. `malloc()` and `free()`)
  - **Free list**: the generic data structure used to manage free space in the heap
- Assume that we are primarily concerned with *external fragmentation*
  - In contrast, if an allocator hands out chunks of memory bigger than that requested, any unasked for (thus unused) space in such a chunk is considered **internal fragmentation**
- Assume that once memory is handed out to a client, it cannot be relocated to another location in memory
  - Makes compaction of free space impossible
- Assume that the allocator manages a contiguous region of bytes
  - Realistically, an allocator could ask for that region to grow
  - But for simplicity we assume that region is a single fixed size throughout its life

Splitting and Coalescing

- Assume the following 30-byte heap:



- The free list for this heap would have 2 elements on it (bytes 0-9, bytes 20-29)
- A request for anything > 10 bytes will fail
- A request for exactly 10 bytes will succeed
- If we have a request for 1 byte of memory, the allocator will perform **splitting**:
  1. Find a free chunk of memory that can satisfy the request, and split it into two
  2. The first chunk is returned to the caller, and the second chunk remains on the list

  - E.g. if the allocator decides to use the second element on the list, then the call to `malloc()` would return 20, then the second element of the list would become 21-29
- If we have a request that frees the (10 bytes of) used memory, the allocator will perform **coalescing**:
  - Look carefully at the addresses of the chunk we are returning as well as the nearby chunks of free space
  - If the newly freed space sits right next to one (or two) existing free chunks, merge them into a single larger free chunk

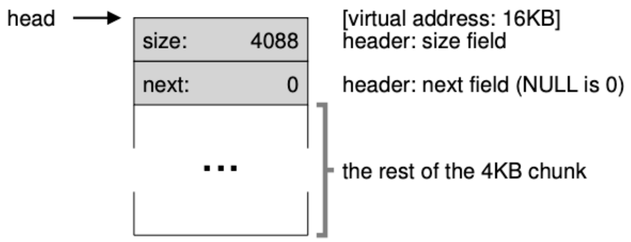
Tracking the Size of Allocated Regions

- Interface to `free()` does not take a size parameter, which requires the malloc library to determine the size
- Most allocators stores extra information in a **header** block which is kept right before the handed-out chunk of memory
- The header contains the following information:

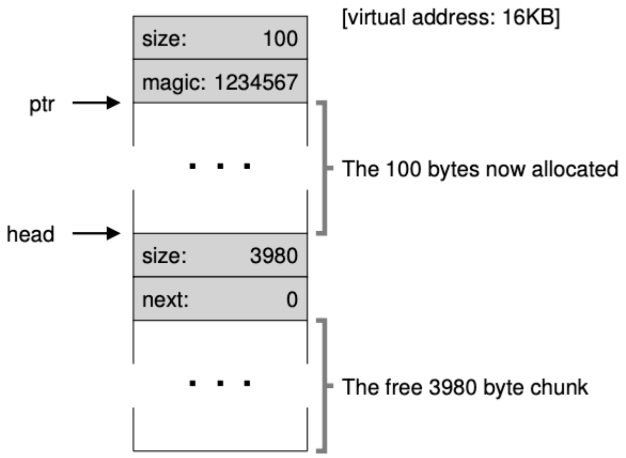
  - Size of the allocated region
  - Additional pointers to speed up deallocation
  - A magic number to provide additional integrity checking
  - Etc.
- To obtain the header, we subtract a value from the pointer returned by `malloc()`
- The size of the free region is the size of the header plus the size of the space allocated to the user

Embedding a Free List

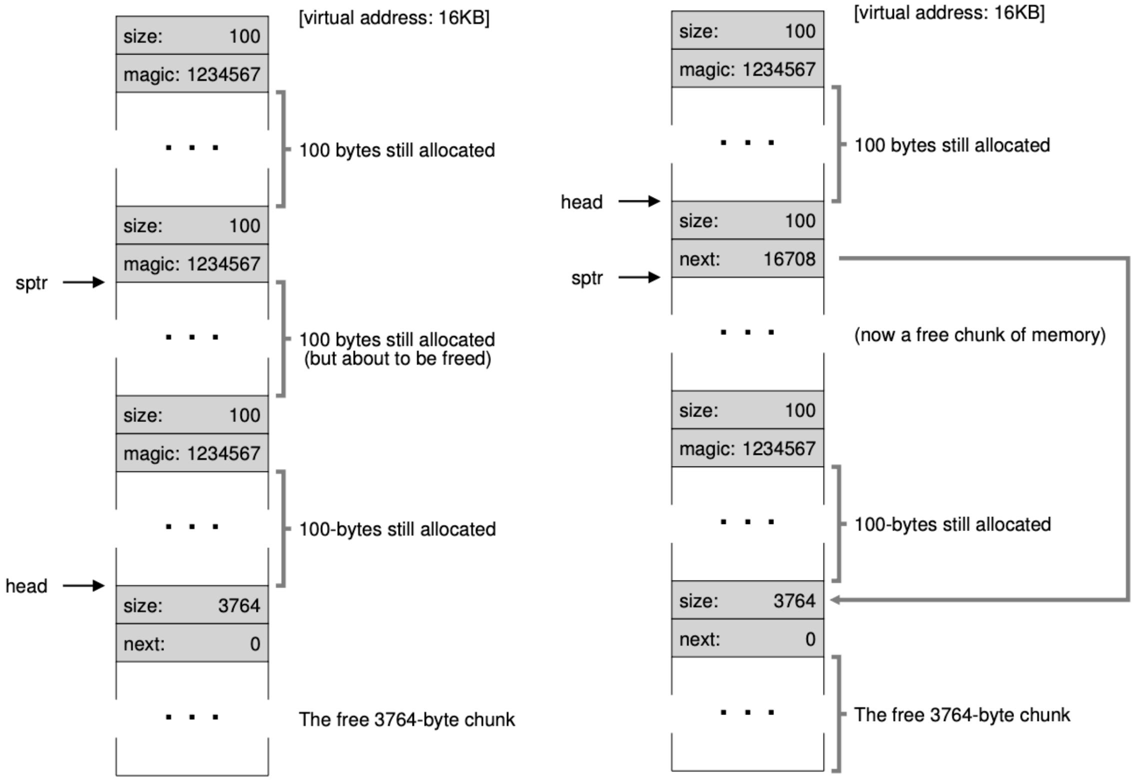
- Within the memory-allocation library, we need to build the free list *inside* the free space
  - Since we cannot just call `malloc()`
- Start with a single free chunk of memory



- When we allocate, we overwrite the free list with the header, and the free list sits on top of the new free space



- To coalesce the list, we go through the list and merge neighbouring chunks
- If we want to free memory, then we overwrite the header with the free list structure, and have the free list's next pointer to point to the next free chunk


[virtual address: 16KB]
size: 100
magic: 1234567
100 bytes still allocated
sptr
100 bytes still allocated (but about to be freed)
100-bytes still allocated
head
size: 3764
next: 0
The free 3764-byte chunk
next: 16708
(now a free chunk of memory)

Growing the Heap

- If the heap runs out of space, we can fail the request or grow the heap
- To grow the heap, we can make a system call (i.e. `sbrk()`), then allocate the new chunks from there

Strategies For Managing Free Space

- Goal: fast, and minimize fragmentation
- Any strategy could perform badly given the wrong set of inputs

Best Fit

- Search through the free list, find chunks of free memory that are as big or bigger than the requested size
- Then return the one that is the *smallest* in that group of candidates (called the *best-fit chunk*)
- Tries to reduce wasted space
- Searching for the correct free block can be expensive

Worst Fit

- Opposite to best fit, find the largest chunk and return the requested amount
- Keep the remaining (large) chunk on the free list
- Tries to leave big chunks free
- Requires searching the entire free list

- Empirically performs badly

First Fit

- Finds the first block that is big enough and returns the requested amount to the user
- Fast
- May pollute the beginning of the free list with small objects
- For the order, can use **address-based ordering**, which keeps the list ordered by the address of the free space, making coalescing eaasier

Next Fit

- Keep an extra pointer to the location within the list where one was looking last
- First fit based on that pointer
- Spreads the searches for free space throughout the list more uniformly
- Performance similar to first fit

Segregated Lists

- If a particular application has one (or a few) popular-sized request that it makes, keep a separate list just to manage objects of that size
- All other requests are forwarded to a more general memory allocator
- Reduces fragmentation for objects of popular sizes
- Allocation and free requests can be fast for objects of popular sizes because they all have the same size so searching is easy
- Need to decide how much memory goes to the size-specific pool vs. the general pool
- **Slab allocator**
  - When the kernel boots up, allocate a number of *object caches* for kernel objects taht are likely to be requested frequently (e.g. locks, etc.)
  - The object caches are each segregated free lists
  - When a given cache is running low on free space, it requests some *slabs* of memory from a more general memory allocator
  - When the reference counts of the objects within a given slab all go to 0, the general allocator can reclaim them from the specialized allocator (done when the VM system needs more memory)

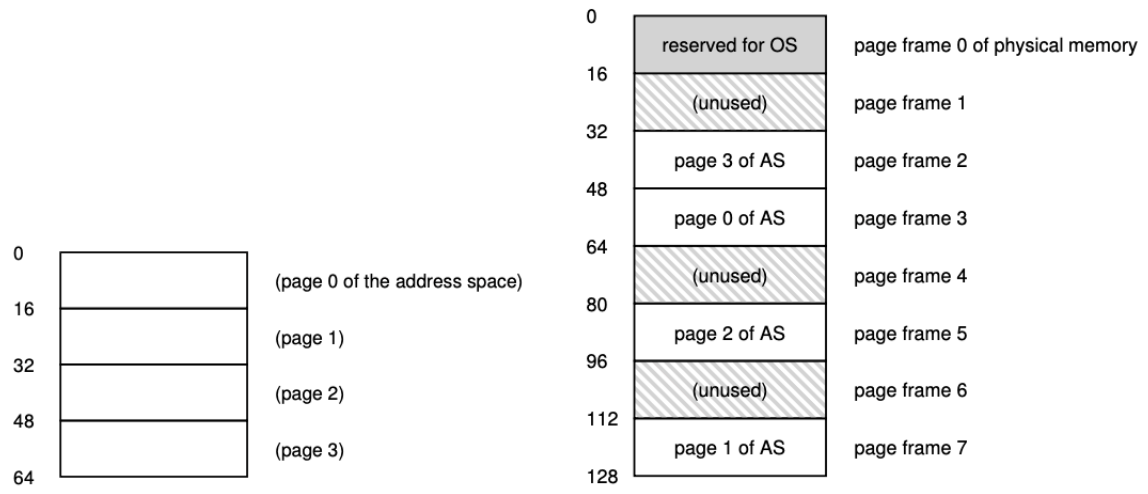
Buddy Allocation

- Free space is thought of as one big space of size $2^N$
- When a request for memory is made, the search for free space recursively divides free space by 2 until a block that is big enough to accommodate the request is found (i.e. a further split would result in a block that is too small)
- Has the problem of *internal fragmentation* since we are only allowed to give out power-of-two-sized blocks
- When we free memory, we coalesce recursively
- It is simple to determine the buddy of a particular block (their address only differs by 1 bit)

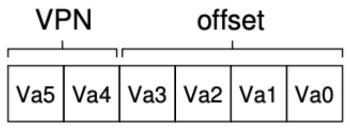
## 19 Paging

Overview

- **Paging**: chop up space into fixed-sized pieces
- Instead of splitting up a process's address space into some number of variable-sized logical segments, we divide it into fixed-sized units, each of which we call a **page**
- We view physical memory as an array of fixed-sized slots called **page frames**, each of these frames can contain a single virtual-memory page



- Paging offers *flexibility* since the system would be able to support the abstraction of an address space effectively, regardless of how a process uses the address space
- Paging also offers *simplicity*, since the OS could keep a free list of all free pages, and can grab the desired number of pages off the list
- **Page table**: a per-process data structure that the OS holds
  - Store **address translations** for each of the virtual pages of the address space, letting us know where in physical memory each virtual page is located
- To translate a virtual address, we need to know two components
  - **Virtual page number (VPN)**
  - Offset within the page



- Using the virtual page number, we can index the page table and find the **physical frame number (PFN)** (aka **physical page number (PPN)**)
- Can then translate the virtual address by replacing the VPN with the PFN
- The offset stays the same, because it tells us which bite *within* the page we want

Storage of Page Tables

- The page table can be very big (e.g. 32-bit address space with 4KB pages would have $2^{20}$ entries)
- We store page tables in memory (i.e. not in the CPU)

Page Table Content

- Page table maps virtual addresses (i.e. VPNs) to physical addresses (i.e. PFNs)
- **Linear page table**: an array indexed by VPN, where we look up the page-table entry (PTE) at an index in order to find the desired PFN
- A PTE contains many bits
  - A **valid bit** to indicate whether the translation is valid (i.e. unused space between stack and heap would be marked *invalid*)
  - **Protection bits** to indicate whether or not the page is readable/writable/executable
  - A **present bit** to indicate whether this page is in physical memory or on disk (i.e. *swapped out*)
  - A **dirty bit** to indicate whether the page has been modified since it was brought into memory
  - A **reference bit** (aka. **accessed bit**) to track whether a page has been accessed
    - Determine which pages are popular so that we can keep them in memory

Paging Cost

- For each memory reference, paging requires us to perform one extra memory reference in order to first fetch the translation from the page table
- We are doubling the cost of each memory reference
- Memory references are costly

## 20 Translation Lookaside Buffer

Intro

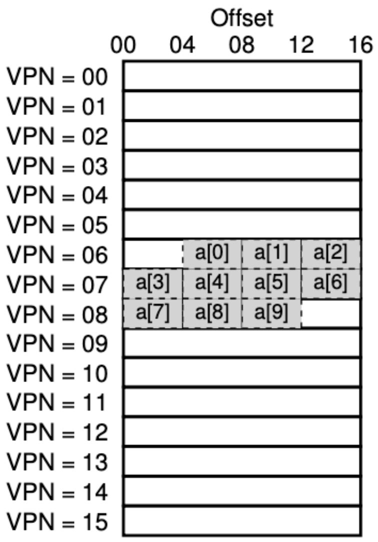
- Paging requires going to memory for translation information before every instruction fetch or explicit load/store, which is slow
- **Translation lookaside buffer (TLB)**: a part of the MMU that caches popular virtual-to-physical address translations (i.e. an **address-translation cache**)
- Upon each virtual memory reference, the hardware first checks the TLB to see if the desired translation is held therein

TLB Basic Algorithm

1. Extract the VPN from the virtual address
2. Check if the TLB holds the translation for this VPN
3. If so, we have a **TLB hit**, we then extract the PFN from the relevant TLB entry, concatenate that onto the offset from the original virtual address to form the physical address
4. If not, then we access the page table to find the translation, then update the TLB with the translation, and finally we retry the instruction (and will end up with a TLB hit)

- Want to avoid misses as much as we can

Accessing Array



- When we iterate over all elements of the array as illustrated above, we will TLB miss on indices 0, 3, 7, then hit on the rest
- TLB improves performance in this case due to *spatial locality*, i.e. the elements of the array are packed tightly into pages
- If we access this array again, then we would TLB hit on all indices due to *temporal locality*, i.e. the quick re-referencing of memory items in time

TLB Miss

- Complex instruction sets (CISC) hardwares use hardware to handle TLB miss
  - The hardware knows where the page tables are located in memory, as well as their format
  - On a miss, the hardware would “walk” tha page table, find the correct page-table entry and extract the desired translation, update the TLB with the translation, and retry the translation
  - Intel x86 architecture does this by using a fixed *multi-level page table*
- Reduced instruction set computers (RISC) hardwares use a **software-managed TLB**
  - On a TLB miss, the hardware raises an exception, which transfers control to the OS by jumping to a trap handler
  - The trap handler looks up the translation in the page table, uses privileged instructions to update the TLB, and return from trap
  - Then the hardware retries the instruction
  - The OS needs to be careful not to cause an infinite chain of TLB misses to occur
- Software-managed TLB is more flexible as the OS can use any data structure it wants to implement the page table
- It is also more simple as the hardware only raises an exception when the TLB misses

TLB Contents

- TLB is **fully associative**, i.e. any translation can be anywhere in the TLB
- A TLB entry consists of the VPN, PFN, and some other bits
- The hardware searches the entire TLB in order to find the desired translation
- Other bits
  - **Valid** bit: whether the entry has a valid translation or not
  - **Protection** bit: determines how a page can be accessed (read/write/execute)
  - Address-space identifier, dirty bit, etc.

Context Switches

- The TLB contains virtual-to-physical translations that are only valid for the currently running process
- We can **flush** the TLB on context switches, i.e. emptying it before running the next process (achieved by setting all valid bits to 0)
  - If the OS switches between processes frequently, this cost could be high since we would have misses after each flush
- Some hardware systems provide an **address space identifier (ASID)** field in the TLB
  - Acts like a process identifier (PID), but has fewer bits
  - The TLB could then hold translations from different processes at the same time without confusion
  - The OS would need to set some privileged register to the ASID of the current process on a context switch
  - When two processes share a page, there could be 2 entries for 2 different processes with 2 different VPNs but the same PFN

Replacement Policy

- When we are installing a new entry in the TLB, we have to replace an old one
- **Least-recently-used (LRU)** entry: assume that the entry that was used least recently is the one that is least likely to be used again in the near future
- **Random** policy: simple and avoids corner-case behaviours

# 21 Advanced Page Tables

Intro

- Page tables are too big, which consume too much memory
- A 32-bit address space (i.e. $2^{32}$ bytes) with 4KB (i.e. $2^{12}$ bytes) pages would require $2^{20}$ entries
- Each entry being 4 bytes results in a 4MB page table

Bigger Pages

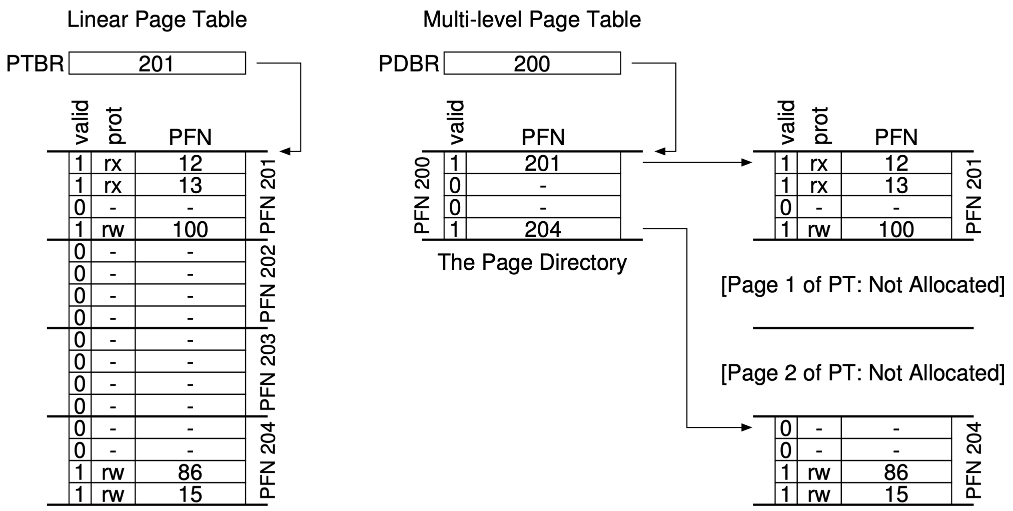
- Can reduce the size of the page table by using bigger pages
- If we have 18-bit VPN and 14-bit offset, then we can have 1MB per page table
- However big pages lead to waste within each page (i.e. *internal fragmentation*)

Paging and Segments

- Instead of having 1 page table per process, we can have 1 per logical segment of the process
- Can have the base register to hold the physical address of the page table of the current segment
- Can have the bounds register to indicate the end of the page table (i.e. how many valid pages it has)
- The virtual address would be split into 3 parts: segment number, VPN, offset
- On a TLB miss, the hardware uses the segment bits to determine which base and bounds pair to use
- Unallocated segments no longer consume page
- Suffers from external fragmentation since page tables could be of arbitrary size

Multi-Level Page Tables

- First chop up the page table into page-sized units
- If an entire page of the page-table entries (PTEs) is invalid, then don't allocate that page of the page table at all
- Use the **page directory** to keep track of which page is valid (and where it is in memory)



- The page directory consists of a number of **page directory entries (PDE)**
- A PDE has a valid bit and a page frame number (PFN), similar to a page table entry (PTE)
- If the PDE is valid, then at least one of the pages of the page table that the entry points to is valid
- Multi-level page table only allocates page-table space in proportion to the amount of address we are using
- If carefully constructed, each portion of the page table fits within a page
- However, on a TLB miss, we need to perform 2 memory references to find the translation
- An example of **time-space trade-off**, where we got smaller tables but at the cost of more memory references
- The VPN would be broken down into the page-table index and the page-table index

More Than 2 Levels

- We could have a deeper tree
- The VPN could be broken down into PD index 0, PD index 1, and the page table index
- Requires additional memory accesses to look up a valid translation

Inverted Page Tables

- Instead of having one page table per process, we can keep a single page table that has an entry for each physical page of the system
- The entry tells us which process is using this page, and which virtual page of that pocess maps to this physical page
- Can us a hash table for lookups

Swapping Page Table to Disk

- Can place page tables in kernel virtual memory, allowing the system to swap some of these page tables to disk when memory pressure is tight

# 22 Swapping Mechanisms

Swap Space

- **Swap space**: the space where we reserve on the disk for moving pages back and forth from memory
- The OS needs to remember the **disk address** of a given page
- Assume that the size of the swap space is very large

The Present Bit

- When the hardware looks in the page table entry (PTE) while translating a virtual address, it may find that the page is *not present* in physical memory using the **present bit**
- Accessing a page that is not present in memory is called a **page fault**, which triggers the **page fault handler**

The Page Fault

- All systems handle page faults in software
- If a page is not present and has been swapped to disk, the OS will need to swap the page into memory in order to service the page fault
- When the OS receives a page fault for a page, it looks in the PTE to find the address, and issues the request to disk to fetch the page into memory
- When the disk I/O completes, the OS updates the page table to mark the page as present, update the PFN field of the page-table entry (PTE) to record the in-memory location of the page, and retry the instruction
- The next attempt may generate a TLB miss, which would be handled by the TLB miss handler
- A final attempt would succeed

Memory Being Full

- When the memory is full, the OS would **page out** one or more pages to make room for the new page(s) to **page in**
- **Page-replacement policy**: the process of picking a page to kick out

Page Fault Control Flow

- If the page is both present and valid,m then the TLB miss handler can grab the PFN from the PTE, retry the instruction, and succeed
- If the page is not present, then the page fault handler is run
- If the page is invalid, then the OS terminates the process

Replacement

- We want to keep a small portion of memory free
- Most OS have a **high watermark (HW)** and a **low watermark (LW)** to help decide when to start evicting pages from memory
- When there are fewer than LW pages available, a background thread starts evicting pages until there are HW pages available

- The background thread is called the **swap daemon** or **page daemon**
- We could perform a number of replacements at once to optimize performance
- Instead of performing a replacement directly, we could instead check if there are any free pages available
  - If not, then we inform the background paging thread that free pages are needed
  - When the thread frees up some pages, we could page in the desired page

## 23 Swapping Policy

Cache Management

- Main memory holding a subset of pages can be viewed as a *cache*
- Replacement policy aims to minimize *cache misses* (maximize *cache hits*)
- **Average memory access time (AMAT)** can be calculated as follows:

$$\text{AMAT} = T_M + \Pr(\text{miss}) \cdot T_D$$

  - $T_M$: cost of accessing memory
  - $T_D$: cost of accessing disk
  - Notice that we are *always* paying the cost of accessing memory
  - Usually $T_D \gg T_M$

Optimal Replacement Policy

- Leads to the fewest number of misses overall
- Replace the page that will be accessed *furthest in the future*
- **Cold-start miss** (aka **compulsory miss**): the inevitable miss(es) when the page is first accessed
  - Sometimes ignore compulsory misses when calculatig hit rate
- Impossible to implement since the future is unknown

FIFO Policy

- Pages are placed in a queue when they enter the system
- Always evict the page on the tail of the queue
- Simple to implement
- However does not recognize the importance level of each page
- **Belady's anomaly**: with FIFO policy, as cache gets larger, the hit rate *decreases*

Random Policy

- Picks a random page to evict

Least Recently Used Policy

- The more recently a page was accessed, the more likely it will be accessed again
- Based on the principle of *locality*
- **Least frequently used (LFU)**: replace the page that was least frequently used
- **Least recently used (LRU)**: replace the page that was least recently used
- Opposite algorithms such as **most frequently used (MFU)** and **most recently used (MRU)** exist, which don't generally work well
- However, scanning the entire page table (of  1 million pages) to find the least-recently-used page is expensive

Approximating LRU

- Each page has a **use bit** (aka **reference bit**) that is set to 1 when the page is accessed
- **Clock algorithm**: having a *clock hand* point to some particular page
  - When a replacement must occur, the OS checks the use bit of the page pointed to by the clock hand
  - If 1, then the page was recently used, so the OS clears the bit to 0, and advances the clock hand
  - The algorithm continues until it finds a use bit that is set to 0, i.e. the page is not recently used

Considering Dirty Pages

- If a page has been modified and is thus dirty, it must be written back to disk to evict it, which is expensive
- If a page is clean, then it can be evicted without writing it back to disk
- Therefore we want to evict clean pages first
- Each page has a **modified bit** (aka **dirty bit**) that is set to 1 when the page is written
- Can modify the clock algorithm so that it first evicts pages that are both clean and not recently used

Other VM Policies

- **Page selection** policy: when to pring a page into memory
- **Demand paging**: bring the page into memory when it is accessed, i.e. on demand
- **Prefetching**: guess that a page is about to be used, and bring it into memory ahead of time
  - Should only be done when there is reasonable chance of success
  - E.g. if a code page $P$ is brought into memory, then code page $P+1$ will likely be accessed soon
- Need another policy to determine how the OS writes pages out to disk
- Can write one at a time, however, we can collect pending writes together in memory and write them to disk in one efficient write
  - Called **clustering** or **grouping** of writes
  - Disk drives perform a single large write more efficiently than many small ones

Thrashing

- **Thrashing**: the condition where the system is constantly paging (i.e. since the memory is full, the OS is constantly swapping pages in and out)
  - The system would not make much progress
- Given a set of processes, a system could decide not to run a subset of processes, hoping that the reduced set of processes' **working sets** (i.e. the pages that they are using actively) fits in memory and thus can make process
  - Known as **admission control**
  - It is sometimes better to do less work well than to try to do everything poorly
- Can run a **out-of-memory killer** where this daemon chooses a memory-intensive process and kills it
  - May kill processes that we don't want to be killed

# 24 Files and Directories

Intro

- **File**: a linear array of bytes, each of which we can read or write
  - Each file has some kind of low-level name, known as its **inode number (i-number)**
  - The OS does not know about the structure of a file (e.g. picture or text)
- **Directory**: contains a list of (user-readable name, low-level name) pairs
  - Each directory also has an inode number
  - Can build a **directory tree (directory hierarchy)** by placing directories within other directories
  - The directory hierarchy starts at a **root directory** (`/`), and uses some kind of **separator** to name subsequent **subdirectories** until the desired file or directory is named
- The file name may sometimes be separated by a period (e.g. temp.txt)
  - What's behind the period is usually the type of the file (e.g. .txt, .jpg, etc.)
  - This is just a convention, and the OS does not enforce it

Creating Files

- Can use the `open()` system call, with the `O_CREAT` flag
  - Can also use the `creat()` system call, which is a wrapper around `open()` with flag `O_CREAT | O_WRONLY | O_TRUNC`
  - `O_WRONLY` opens the file for write only
  - `O_TRUNC` truncates the file if it already exists
- `open()` returns a file descriptor
  - Once a file is opened, we can use the file descriptor to read or write the file
  - A file descriptor is a *capability*, i.e. an opaque handle that gives us the power to perform certain operations
  - File descriptors are managed by the OS on a per-process basis

Reading and Writing Files

- To read an existing file, use `cat` on terminal
- Can use `strace` to find out the system calls used by a program

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE) = 3
read(3, "hello\n", 4096)     = 6
write(1, "hello\n", 6)       = 6
hello
read(3, "", 4096)            = 0
close(3)                     = 0
...
```

  - `cat` first opens a file for reading
    * `O_RDONLY`: read only flag

        * `O_LARGEFILE`: use 64-bit offset
        * `open()` returns a file descriptor 3
        * FDs 0, 1, 2 are used by stdin, stdout, and stderr, respectively
    - `cat` then uses the `read()` system call to repeatedly read bytes from the file
        * 3 is the file descriptor
        * Second argument points to the buffer where the bytes should be placed
        * The result of the read is `"hello\n"`
        * Third argument is the size of the buffer
    - `cat` then uses a `write()` system call to write the bytes to the terminal (file descriptor 1 stdout)
    - `cat` attempts to read more from the file, but gets 0 bytes back, indicating the end of file
    - Then the program calls `close()` to indicate that it is done with the file "foo"

- Writing a file involves a similar set of steps, where `write()` is repeatedly called

Non-Sequential Reading/Writing

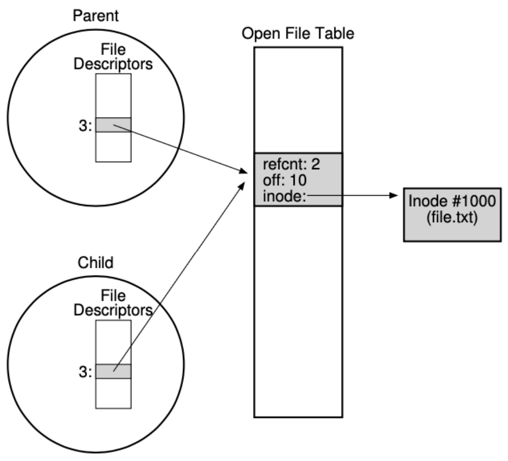
- Want to access a file starting from some random offset
- Can do so using `lseek()`

```c
off_t lseek(int fildes, off_t offset, int whence);
```

    - First argument is the file descriptor
    - Second argument is the offset, which positions the file offset to a particular location within the file
    - Third argument determines how the seek is performed
        * `SEEK_SET`: the offset is set to offset bytes
        * `SEEK_CUR`: the offset is set to its current location plus offset bytes
        * `SEEK_END`: the offset is set to the size of the file plus offset bytes

- The OS tracks a "current" offset
- When a read/write of $N$ bytes takes place, $N$ is added to the current offset
- `lseek()` can be used to change the current offset
- Each file can have two different file descriptors, independently tracking the current offset

Shared File Table Entries



- Can have multiple processes reading the same file at the same time
- An entry in the open file table can be shared (e.g. from `fork()`)
- In which case the offset is shared between the two processes
- Only when both processes close the file (or exit) will the shared file table entry be removed
- The `dup()` call allows a process to create a new file descriptor that refers to the same underlying open file as an existing descriptor
  - Useful for file redirection
  - The two file descriptors can be used interchangeably

Writing Immediately With `fsync()`

- Normally when we write to a file, the OS buffers the data in memory, and only writes it to disk every so often
- We want to be able to force a write
- Can use `fsync(int fd)`, which when called, forces all dirty data to the disk from that file descriptor
- We might also need to `fsync()` the directory of the desired file in order to apply the change to the directory

Renaming Files

- Can use `mv` on the terminal
- `mv` uses the system call `rename(char *old, char *new)`
- `rename()` is usually atomic, making it resistent to system crashes
- When we use an editor to change a file, to ensure resistance to crash, the following is done:
  - The new version of the file is written under a temporary name
  - It is forced to disk using `fsync()`
  - The temporary file is renamed to the original file name, and the old version of the file is deleted

Getting Information About Files

- The file system keeps file **metadata**, which stores information about each file
- Use `stat()` or `fstat()` system calls to see metadata for a certain file
- Metadata includes the size, low-level name, ownership information, accessed/modified information, etc.
- Such information is kept in the **inode** for the file

Removing Files

- Can use `rm` on the terminal
- `rm` uses the `unlink()` system call

Making Directories

- Can use `mkdir` on the terminal
- `mkdir` uses the `mkdir()` system call

- When a directory is created, it starts empty, but has two entries
  - `.`: the directory itself
  - `..`: the parent directory

Reading Directories

- Can use `ls` on the terminal
- Can use system calls `opendir()`, `readdir()`, and `closedir()`

Deleting Directories

- Can use `rmdir` on the terminal
- `rmdir` requires that the directory is empty

Hard Links

- An entry can be made in the file system tree through system call `link()`
- `link()` takes two arguments: an old pathname and a new one
- When we "link" a new file name to an old one, we create another way to refer to the same file
- On terminal, we can use `ln`
- The file is not copied, we just have two human-readable names pointing to the same inode
- Can see inode number by passing `-i` flag to `ls`
- When we remove a file using `unlink`, we can still find the file using other links
- When unlinking a file, the OS decrements the **reference count (link count)** within the inode number
- When the reference count reaches 0, the file system also frees the inode and related data blocks

Symbolic Links

- We cannot hard link directories, and cannot hard link files in other disk partitions (because inode nuymbers are only unique within a particular file system, not across file systems)
- To create a **symbolic link (soft link)**, use `ln -s` on the terminal
- The symbolic link is actually a file itself, of a different type
- A symbolic link holds the pathname of the linked-to file
- When `ls -l`, the first character being `l` indicates a symbolic link
- Symbolic links might cause **dangling reference**, when the original file is not present anymore

Permission Bits and Access Control Lists

- When `ls -l`, the first string starting from the second character indicates permission bits
- First three bits indicate the permissions for the owner of the file
- Second three bits indicate the permissions for the group of the file
- Third three bits indicate the permissions for everyone else
- The owner can change permissions using `chmod` (i.e. change **file mode**)

- The execute bit for directories indicate whether or not we can access the files within the directory (i.e. using `cd`)
- **Access control list (ACL)** is a more general way to represent who can access a given resource
  - Enables a user to create a specific list of who can and cannot read/write/execute a set of files

Making and Mounting a File System

- To make a file system, use `mkfs`
- Once such a file system is created, it needs to be accessible within the uniform file system tree
- Can use `mount` to mount a file system, which makes the underlying system call `mount()`
- Mount takes an existing directory as a target **mount point**, and pastes the new file system onto the directory tree at that point
- E.g.

  ```
          prompt> mount -t ext3 /dev/sda1 /home/users
  ```

  makes `home_users` to refer to the root of the newly mounted directory
- `mount` unifies all file systems into one tree

## 25 File System Implementation

Overall Organization

- Need to divide the disk into blocks
- Can have all blocks having one size (e.g. 4KB)
- Assume we have a disk of 64 blocks, addressed from 0 to 63
- Most of the space in file system is user data, let block 8 onwards be user data
- Let blocks 3 to 7 be the inode table, which stores metadata for each file
- We need some way to track whether inodes or data blocks are free or allocated
  - Can use a **free list** that points to the first free block, which points to the next free block, etc.
  - Can use a **bitmap**, where each bit represents a block, and a 1 indicates that the block is free
- Let block 1 be the inode bitmap, and block 2 be the data bitmap
- Let block 0 be the **superblock**, which contains information about this particular file system
  - E.g. how many inodes, how many data blocks, etc.
  - When mounting the file system, the OS reads the superblock in order to initialize various parameters

File Organization and the Inode

- **Inode**: short for **index node** (historically all nodes are arranged in an array)
- Each inode is implicitly referred to by a number, known as the **i-number**, which is the low-level name of the file
- Given the i-number, we can calculate where on the disk the corresponding inode is located
- Need to know how an inode refers to its corresponding data blocks
  - Can store a **direct pointer** inside the inode
  - Not a good idea when file is large (i.e. spans many blocks)
- Can store an **indirect pointer** that points to a block containing more pointers, each of which points to user data
  - An inode may have some fixed number of direct pointers (e.g. 12) and 1 indirect pointer
  - If a file grows, then an indirect block is allocated, and the indirect pointer is updated to point to the new block
- Can also utilize **double indirect pointer**, which points to a block containing indirect pointers
- **Multi-level index**: the tree of pointers described above

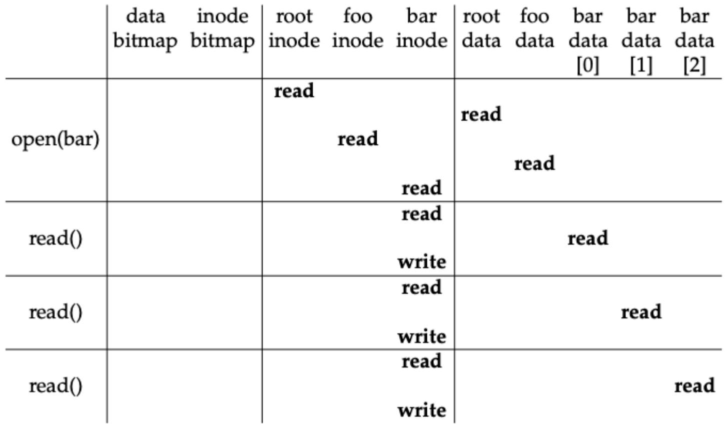
Directory Organization

- A directory contains a list of (entry name, inode number) pairs
- Deleting a file can leave an empty space in the middle of the directory, which can be marked for reuse
- Directories are stored like regular files (i.e. has an inode, and data blocks)

Free Space Management

- When we create a file, we have to allocate an inode for that file
  - The file system search through the bitmap for an inode that is free, and allocate it to the file
  - The file system has to mark the inode as used, and eventually update the on-disk bitmap with the correct information
  - Same thing happens when we wnat to allocate a data block

Reading a File From Disk

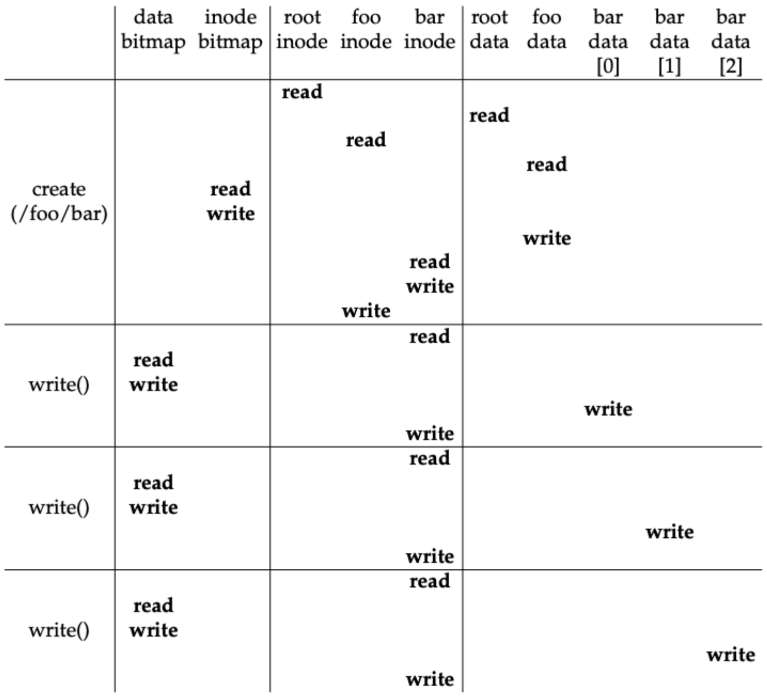


| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data [0] | bar data [1] | bar data [2] |
|---|---|---|---|---|---|---|---|---|---|---|
| open(bar) | | | read | | | | | | | |
| | | | | | | read | | | | |
| | | | | read | | | | | | |
| | | | | | | | read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

- When we issue an `open()` call, the file system first finds the inode for the file
  - It traverses the pathname to look for the desired inode, starting from `/`
  - The root inode number is well known (usually 2)
  - The file system looks inside the root directory to find the inode number for the next component of the pathname
  - The pathname is recursively traversed until the desired inode is found
  - The final step reads the file's inode into memory
- Once open, the program can issue a `read()` system call
  - The first read (at offset 0) reads the first block of the file, consulting the inode to find the location of such a block
  - Updates the inode with the new last accessed time, updates the in-memory open file table for this file descriptor, updates the file offset such that the next read will read the next block, etc.
- When closing the file, the file descriptor should be deallocated, which does not involve disk I/Os

Writing a File to Disk

- Writing a file may allocate a block
- Each write to a file generates 5 I/Os
  1. Read the data bitmap
  2. Write the bitmap
  3. Read the inode
  4. Write the inode
  5. Write the actual block itself



| | 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 | | | | | |

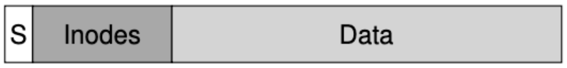
Caching and Buffering

- Can use system memory to cache important blocks
  - Can use a fixed-size cache
  - The cache is allocated at boot time to be 10% of the total memory (static partitioning)
  - Could be wasteful if the cache is not used
- Can use a **dynamic partitioning** approach, where we integrate virtual memory pages and file system pages into a **unified apge cache**
- Can use **write buffering** to batch writes together by delaying some writes
  - Can additionally buffer writes in memory, where the OS schedules the subsequent I/Os to increase performance
  - If some writes are buffered when the file is deleted, then we won't have to write them to disk
  - Drawback: if the system crashes, the buffered writes are lost
- If we cannot afford to lose buffered writes, we can use `fsync()`
  - Can use direct I/O interfaces that work around the cache
  - Can also use raw disk interface that bypasses the file system entirely

## 26 Fast File System

Intro

- The old Unix file system has the following data structure:



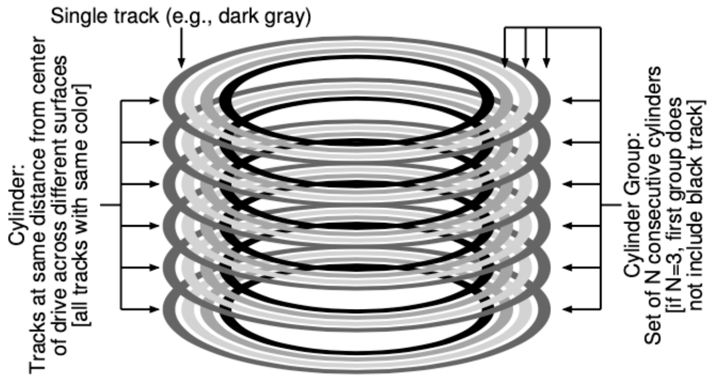
- Treated disk like random access memory
- Positioning the disk head was expensive
- The file system is *fragmented*, since a data block could be spread out across the disk
  - Can be mitigated with disk defragmentation tools
- Block sizes being too small leads to transferring data from disk being inefficient

Fast File System (FFS)

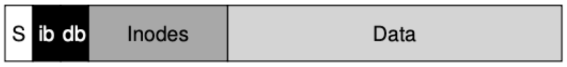
- The file system structures and allocation policies are designed to be "disk aware"

Cylinder Group

- FFS divides the disk into a number of **cylinder groups**
  - A **cylinder** is a set of tracks on different surfaces of a hard drive that are the same distance from the center of the drive
  - FFS aggregates $N$ consecutive cylinders into a group


Single track (e.g., dark gray)
Cylinder:
Tracks at same distance from center of drive across different surfaces [all tracks with same color]
Cylinder Group:
Set of N consecutive cylinders [if N=3, first group does not include black track]

- The geometry of the disk is unknown to the file system, so file systems organize the drive into **block groups**
  - Each block group is a consecitive portion of the disk's address space
- By placing two file within the same group, they are spatially close to each other on the disk
- Within one cylinder group, we have the following structure:



  - Keep a copy of the super block in each group in case some are corrupted
  - Per-group **inode bitmap** (ib) and **data bitmap** (db) are used to manage free space

Allocating Files and Directories

- Want to keep related stuff together and unrelated stuff apart
- For directories, FFS finds the cylinder group with a low number of allocated directories and a high number of free inodes
  - The former is to balance directories across groups
  - The latter is to be able to allocate a bunch of files
- For files, FFS does two things
  1. Makes sure to allocate the data blocks of a file in the same group as its inode, so that seeking from inode to data is fast
  2. Places all files that are in the same directory in the cylinder group of the directory they are in

Managing Large Files

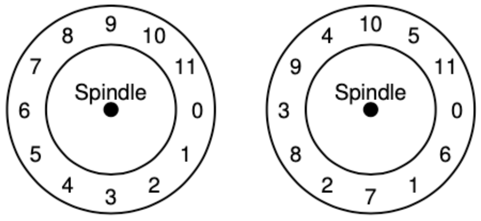
- Without a different rule, a large file would entirely fill the block group, preventing subsequent "related" files from being placed within this block group
- After some $x$ blocks are allocated into the first block group, FFS places the next chunk of $x$ blocks into the next block group, and so on
- Can choose the next chunks to be closed to the previous chunk

Sub-Blocks

- Using 4KB blocks is wasteful for small files (causes internal fragmentation)
- Divide some blocks into 8 **sub-blocks** of 512 bytes each
- By the time the file grows to 4KB, FFS will find a 4KB block, copy the sub-blocks into it, and free the sub-blocks for future use
- This is generally avoided with buffered writes

Parameterization

- FFS can figure out for a particular disk, how many blocks it should skip in order to find the next block group
  - Sometimes, when we are accessing block 1 after accessing block 0, the disk has already rotated to the next block group so that it requires 1 full rotation to access block 1
- Layout the block differently



- Slow (need to rotate twice to read everything once)
- Can read the entire track and buffer it in an internal disk cache (called **track buffer**)

# 27 File System Checker and Journaling

Crash Consistency Problem

- Need to update 2 on-disk structure, $A$ and $B$
- Can only serve 1 request at a time – either $A$ or $B$ will reach the disk first
- If the system crashes or loses power after one write completes, the on-disk structure will be left in an **inconsistent** state

Crash Example

- Consider appending a data block to an existing file
- Requires 3 writes:
  1. Write to the data block
  2. Update the inode's size and pointer
  3. Update the data block bitmap
- Case: only (1) succeeds
  - No inode points to it and it is not marked as allocated
  - As if nothing has happened, so no problem
- Case: only (2) succeeds
  - The inode points to garbage data
  - The inode says that the data block is allocated, however the bitmap says that it is free
  - **File system inconsistency**
- Case: only (3) succeeds
  - The bitmap indicates that the data block is allocated, but there is no inode pointing to it
  - **Space leak** – this block will never be used by the system
- Case: (2) and (3) succeed
  - System metadata is consistent
  - The inode points to garbage data
- Case: (1) and (2) succeed
  - Inode points to the right data
  - File system inconsistency
- Case: (1) and (3) succeed
  - File system inconsistency
  - We don't know who the data belongs to, since no inode points to it
- Impossible to do writes atomically because disk only commits 1 write at a time

File System Checker (FSCK)

- Let inconsistencies happen and fix later
- `fsck`: unix tool for finding inconsistencies and repairing them

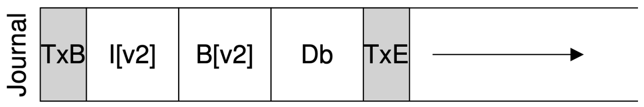
- Cannot fix the cases where the file system looks consistent but the inode points to garbage data
- `fsck` does the following:
  1. Check if the superblock looks reasonable
     - Mostly sanity checks (e.g. file system size $\geq$ number of blocks allocated)
     - If it looks corrupt, then let the system use another copy of the superblock
  2. Scan the inodes, indirect blocks, double indirect blocks, etc. to know which blocks are currently allocated
     - Use this knowledge to update the bitmap
     - All nodes that look like are in used are marked as allocated in the inode bitmaps
  3. Check each inode for corruption or other problems
     - E.g. makes sure that each allocated inode has a valid type field
     - If there are problems with the inode fields that are not easily fixed, then the inode is cleared by `fsck` (and the inode bitmap updated)
  4. Verify the link count for each allocated inode
     - Scans through the entire directory tree (starting at the root directory)
     - If there is a mismatch, then update the inode link count
     - If an inode has no directory referring to it, it is moved to the `lost + found` directory
  5. Check for duplicate pointers, i.e. two inodes referring to the same block
     - If one inode is obviously bad, it is cleared
     - If both are good, then the pointed-to block is copied, giving each inode its own copy
  6. Check for bad block pointers
     - A pointer is considered "bad" if it obviously points to something outside its valid range (e.g. an address that refers to a block greater than the partition size)
     - In which case `fsck` removes the pointer from the inode or indirect block
  7. Perform integrity che3cks on the contents of each directory
     - Makes sure `.` and `..` are the first entries
     - Each inode referred to in a directory entry is allocated
     - No directory is linked to more than once in the entire hierarchy
- `fsck` is too slow
  - Scanning the entire disk is very expensive

Journaling (Write-Ahead Logging)

- When updating a disk, before overwriting the structures in place, first write down a little note (somewhere else on disk) discribing what we are going to do
  - The note is the "write ahead" part and we write to a structure called "log"
- If a crash takes place, we can look at the note and then try again
- The file system reserves a journal space right after the super block

Data Journaling

- When we are doing the same data/inode/bitmap write, we first write the following to the journal:



- TxB (transaction begin) tells us about the update
    - Contains information about the pending update to the file system (e.g. final addresses fo the blocks)
    - Also contains a **transaction identifier (TID)**
  - Middle three blocks contain the exact contents of the blocks themselves
    - Known as **physical logging**, where we are putting the exact physical contents of the update in the journal
  - TxE (transaction end) marks the end of the transaction
- Once the transaction is on disk, we can overwrite old structures in the file system, called **checkpointing** (i.e. bringing it up to date with the journal)
  - Issue writes to their disk locations
- Journal writing requires 5 writes, which is slow
  - Can issue them as 1 big sequential write, however, this is unsafe
  - The disk may internally perform scheduling and complete small pieces of a big write in any order
  - If TxB and TxE are written but any of the middle blocks are not, then the transaction still looks valid
- Best we can do is 2 writes: everything except TxE, and TxE
  - The disk guarantees that any 512-byte write is atomic
  - TxE needs to be a single 512-byte block
  - The second write (TxE) is known as **journal commit**

Recovery

- If a crash happen before journal commit, then we can skip the pending update
- If a crash happens after journal commit, then we can recover the update
  - When the system boots, the file system recovery process scaans the log and looks for transactions that have committed to the disk
  - These updates are **replayed** (in order)
  - This method is called **redo logging**
- Since recovery is a rare operation (only happens when the system crashes), we can afford to do it slowly

Batching Log Updates

- We can batch multiple updates into a single transaction
- Avoids the overhead of journaling

Making the Log Finite

- When the log becomes full, two things happen:
  1. Recovery takes longer
  2. No more transactions can be committed to the disk
- Can treat the log as a circular data structure (circular log), where we overwrite old transactions
- Once a transaction has been checkpointed, the file systeem should free the space it was occupying within the journal

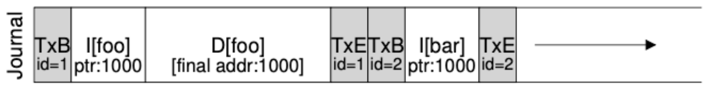
- Can mark the oldest and newest non-checkpointed transactions in the log in a **journal superblock**

Metadata Journaling

- For each write to disk, we need to write to the journal first, which doubles write traffic
- Can use **ordered journaling** (aka **metadata journaling**), where user data is not written to the journal
- Only the inode and bitmap updates are written to the journal
- When actually writing, we need to write data blocks first
  - Otherwise when we write the inode first, it may point to garbage data
- Workflow
  1. Data write: write data to final location
  2. Journal metadata write: write the begin block and metadata to the log, wait for writes to complete
  3. Journal commit: write the transaction commit block to the log, wait for the write to complete
  4. Checkpoint metadata: write the contents of the metadata update to their final locations within the file system
  5. Free: later, mark the transaction free in journal superblock

Block Reuse

- If we first have a directory `foo`, add an entry to `foo`, delete `foo`, then create a new file `bar` reusing the same block as `foo` was previously using, then we have a problem



- Directory data is considered metadata, so it is journaled
- When a crash occurs and the journal is replayed, the write of directory data is replayed, overwriting the user data of `bar` with old directory contents
- We could never reuse a block until the delete of the block is checkpointed out of the journal
- We could also add a **revoke** record, where deleting the directory would cause a revoke record to be written to the journal
  - When replaying the journal, the system first scans for revoke records, any revoked data is never replayed

Other Approaches

- Soft updates: carefully order all writes to the file system to ensure that the on-disk structures are never left in an inconsistent state
  - E.g. writing data block before inode
  - Can be challenging to implement
- Copy on write (COW): never overwite files or directories in place, places new updates to previously unused locations on disk
- Backpointer-based consistency (BBC): add an additional back pointer to every block in the system (i.e. data block to inode)
  - Check consistency using both forward pointer and back pointer

# 28 Flash-Based SSDs

Storing Bits

- Flash chips store 1 or more bits in a single transistor
- The level of charge trapped within the transistor is mapped to a binary value
- **Single-level cell (SLC)** flash: 1 bit is stored within a transistor (i.e. 1 or 0)
  - Best performance, but most expensive
- **Multi-level cell (MLC)** flash: 2 bits are stored within a transistor (i.e. 00, 01, 10, 11)
  - “Low”, “somewhat low”, “somewhat high”, “high”
- **Triple-level cell (TLC)** flash: 3 bits per cell

Banks/Planes

- Flash chips are organized into **banks** or **planes** which consist of a large number of cells
- A bank is accessed in two different sized units
  1. **Blocks** (aka **erase blocks**), which are typically 128 KB or 256 KB
  2. **Pages**, which are a few KB in size
  - Many pages within one block

Basic Flash Operations

- **Read (a page)**: read any page by specifying the read command and the page number
  - Very fast
  - **Random access**: being able to access any location uniformly quickly
- **Erase (a block)**: destroy the contents of the block (by setting each bit to value 1)
  - Need to erase a *block* before writing to a *page*
  - Before erasing a block, we need to copy the data we care about to another location
  - Erasing too often can **wear out** the flash
  - Expensive
- **Program (a page)**: once a block has been erased, the program command can be used to change some of the 1s within the page to 0s
  - Write the desired contents of a page to the flash
  - Moderately expensive
- Can think of each page having a state
  - Pages start in `INVALID` state
  - By erasing the block, we set the state of the page to `ERASED`
  - When we program a page, we set the state to `VALID`

Reliability of Flash

- Wear out

  - When a flash block is erased and programmed, it slowly accrues a little bit of extra charge
  - When the extra charge build up, it becomes increasingly difficult to differentiate between a 0 and a 1
  - When it becomes impossible, the block becomes unusable

- Disturbance
  - When accessing a particular page within a flash, it is possible that some bits get flipped in neighbouring pages
  - Such bit flips are known as **read disturbs** or **program disturbs**

Raw Flash and Flash-Based SSDs

- An SSD contains the following:
  - Some number of flash chips (for persistent storage)
  - Some amount of volatile memory (for caching and buffering)
  - Control logic for device operation
- The control logic need to turn client read/writes into internal flash operations
  - The **flash translation layer (FTL)** takes read/write requests on *logical blocks* (i.e. device interface) and turns them into low-level read/erase/program commands on the underlying *physical blocks/physical pages* (i.e. actual flash device)
  - Utilize multiple flash chips in parallel
  - Reduce **write amplication**:

$$\frac{\text{total write traffic (in bytes) issued to the flash chips by the FTL}}{\text{total write traffic (in bytes) issued by the client to the SSD}}$$

  - **Wear leveling**: try to spread writes across the blocks of the flash as evenly as possible to avoid wear out, so that all of the blocks wear out at roughly the same time

Direct Mapped Approach

- A read to logical ppage $N$ is mapped directly to a read of physical page $N$
- A write to logical page $N$:
  - The FTL reads in the entire block that page $N$ is contained
  - Erase the block
  - Program the old pages as well as the new one
- Writing is very expensive, severe write amplification
- Certain blocks wear out very fast

Log-Structured FTL

- Most FTLs are **log-structured**
- **Logging:** upon a write to logical block $N$, the device appends the write to the next free spot in the currently-being-written-to block
- To allow for subsequent reads, the device keeps a **mapping table**, which stores physical address of each logical block on the system

- This approach produces **garbage**: old versions of data taking up space
- The device has to periodically perform **garbage collection (GC)** to free space for future writes
- Making in-memoy mapping tables is expensive when device is large

Garbage Collection

- The pages marked `VALID` may have garbage in them
- When we want to reclaim blocks, we first find a block that contains 1 or more garbage pages, read in the live (non-garbage) pages from that block, write out those live pages to the log, and then erase the block
- Can use the mapping table to determine whether each page within the block holds live data
- Garbage collection is expensive since it requires many reads and writes
- Ideally want to reclaim a block that consists of only dead pages, in which case we can just erase the block
- Some SSDs **overprovision** the device, i.e. adding extra flash capacity so that cleaning can be delayed and pushed to the background

# 29 Security

Security Goals and Policies

1. **Confidentiality**: if some piece of information is supposed to be hidden from others, then it should be
2. **Integrity**: if some piece of information or component of a system is supposed to be in a particular state, then it should not be changed by others
    - Such information must be created by a particular party (not an adversary)
3. **Availability**: if some information or service is supposed to be available for use, then an attacker cannot prevent its use

- **Non-repudiation**: when someone told us something, they cannot later deny that they did so

Designing Secure Systems

1. **Economy of mechanism**: keep the system as small and simple as possible
2. **Fail-safe defaults**: if policies can be set to determine the behaviour of a system, have the default for those policies to be more secure
3. **Complete mediation**: check if an action to be performed meets security policies every single time the action is taken (may be expensive)
4. **Open design**: Assume that the adversary knows every detail of the design of the system
5. **Separation of privilege**: require separate parties or credentials to perform critical actions (e.g. two-factor authentication)
6. **Least privilege**: give a user/process the minimum privileges required to perform the actions
7. **Least common mechanism**: for different users/processes, use separate data structures or mechanisms to handle them (e.g. each process gets its own page table)
8. **Acceptability**: The system should not be so inconvenient that people try to circumvent it

# 30 Authentication

Intro

- **Principal**: the party asking for something
- Knowing who's requesting an OS service is crucial for security
- **Agent**: the process or other active computing entity performing the request on behalf of a principal
- **Object**: what is being requested (nothing to do with OOP)
- If the OS determined that the process is permitted access, the OS can remember the decision
  - **Credential**: a record of a decision
  - Once a principal has been authenticated, systems will always rely on that authentication decision

Attaching Identities to Processes

- For a multi-user system, we can assign identities to processes based on which human user they belong to
- When a user first starts interacting with a system, the OS can start a process up for that user
- The user *log in* to provide identity information to the OS

Authenticate By What You Know

- A password is a secret known only to the party to be authenticated
- Since the system knows the password, it may leak out
- The system could store a *hash* of the password, not the password itself
- **Cryptographic hash**: hash that makes it infeasible to figure out the original password
  - Hard to design
- **Dictionary attack**: guessing the password using a list of common passwords
  - Can shut off access to an account when too many failed attempts are made
  - Can also slow down the process of password checking
- Before hashing a new password and storing it in the password file, we can generate a big random number and concatenate it to the password, has the result and store that
  - Also need to store that big random number
  - **Salt**: the random number
  - With a salt, the attacker can no longer create one translation table of passwords to hashes
  - Need one translation table per salt, which can be computationally infeasible

Authentication By What You Have

- **Multi-factor authentication**
  - **Two-factor authentication**
- **Dongle**: a small device that can be plugged into a computer to provide authentication
- If we don't authenticate frequently, then this method degrades to "authentication by what you know"

- The user may not always have "what you have" (i.e. dongle, phone, etc.)
  - "What you have" may fall into the wrong hands

Authentication By What You Are

- DNA, facial recognition, etc.
- E.g. for facial recognition, the system relies on the fact that they person took the picture is the owner
- There may be scenarios where, e.g. lighting is dim, person wearing a mask, etc.
- Since the comparison of what is on file vs. what is presented is not mathematical, need to consider sensitivity and specificity
  - For protection, want false positive rate to be low
- Require specific hardware that not all devices carry (e.g. retina reader, fingerprint reader, etc.)
- Don't want an untrusted machine to check biometric information, otherwise it can be stolen

Authenticating Non-Humans

- If a web server could toggle a light, we only want the real web server to be able to do so
- Can assign a password to the web server
  - Only certain users have this permission
  - In Linux/Unix, this is specified by the `sudo` command
- Sometimes we want to allow a group of users to perform an action
  - Can check membership when someone tries to do something that requires group membership

# 31 Access Control

Intro

- Two steps when there is a request:
    1. Figure out if the request fits within the security policy
    2. If so, perform the operation; if not, ensure it isn't done
- **Access control**: step 1
- **Subject**: the entity (e.g. user, process) making the request
- **Object**: the thing (e.g. file, device) that the subject wants to access
- **Access**: some particular mode of dealing with the object (e.g. reading, writing)
- **Authorization**: determining whether a subject is allowed to perform a particular access on an object
- The subject can freely access objects belonging to them (e.g. a process accessing its virtual memory)

Solutions

- **Capability-based system**: each subject has "keys" to access certain objects
- **Access control list (ACL) system**: the system maintains a list dictating whether each subject can access each object

Access Control Lists (ACLs)

- Each object has an ACL associated with it
- ACL is a part of file metadata
    - If we have an entry for each user, then the ACL can be very large
- ACL takes 9 bits (read/write/execute for user/group/everyone)
    - Cheap to query ACL
    - Not very expressive, so some systems use an extension of this approach
- If we want to query all files accessible to some principal, then we need to query the ACL of every file

Capabilities

- When a user requests an access, either the user provides a capability permitting the access as a parameter, or the OS finds it
- A capability is a set of bits
    - Anyone can create any bunch of bits and call it a capability
    - Once someone has a working capability, they can duplicate it (and give it to others)
- Want capabilities to be unforgeable
- Never let the process see the capability, store it in the kernel
    - When a process wants to give permission to another process, it asks the kernel to do so
    - Keeping all capabilities for all objects in the kernel is expensive
- Instead of storing capabilities in the kernel, we can cryptographically protect them

- Determining the entire set of principals who can access a resource becomes expensive
- With capabilities, it is easy for a process to give its child a *subset* of its capabilities

Mandatory and Discretionary Access Control

- **Discretionary access control**: who gets access to a resource is determined by the owning user
- **Mandatory access control**: some elements of the access control decisions are mandated by an authority, who can override the owner

Practicalities of Access Control Mechanisms

- Infeasible to have human users to set access control permissions for every single file
- Instead, we can let the user to specify a default set of permissions for all files they create
- **Role-based access control (RBAC)**: assign permissions to roles, and assign roles to users
  - When a user has multiple roles, they can switch roles as needed to perform different tasks
- **Privilege escalation**: a user temporarily gains additional privileges
  - E.g. `setuid` in Linux
  - Those privileges are only granted during the run of the program
  - Need to ensure that those privileges are not abused
  - Can gain superuser privileges using `sudo` in Linux