CS 162: Operating Systems & Systems Programming Lecture Notes

Week 1: Lecture 1 What is an Operating System? (1/19)

What is an Operating System?

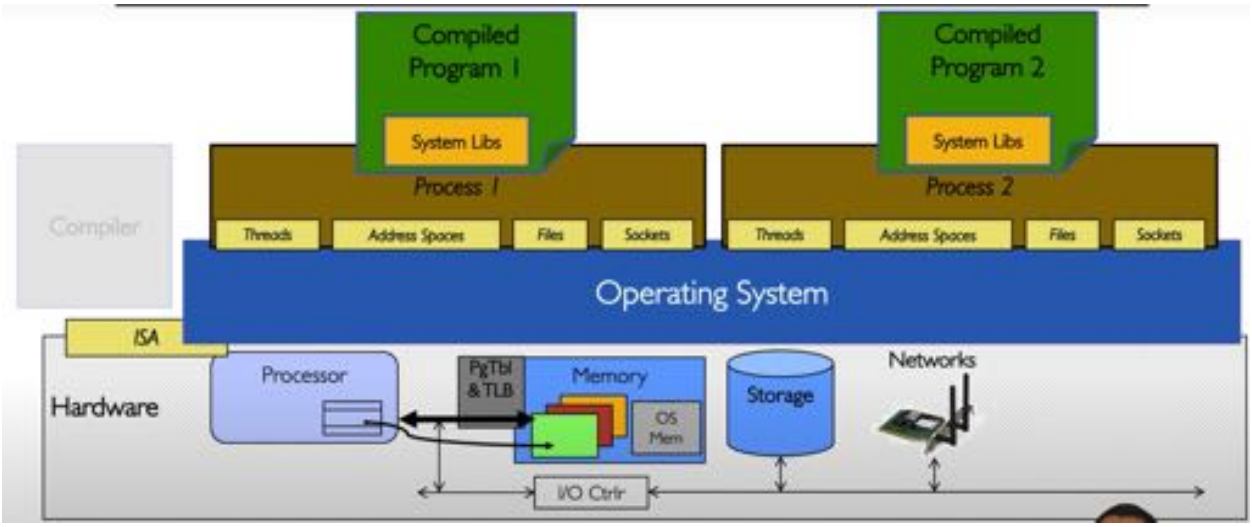
- *Special layer of software that provides application software access to hardware resources*
    - Convenient abstraction of complex hardware devices
    - Protected access to shared resources, security
- **Illusionist**: provide clean, easy to use abstractions of physical resources
- The internet is the most connected, more computers per person now
- For search, find DNS server and go to data center to retrieve
-

What does an Operating System do?

- Memory management, I/O management, CPU Scheduling
- Each running program runs its own process, processes provide nicer interfaces

What's in a Process?

- Contains: Address Space, One or more threads, open files, open sockets
-



Operating System as a Referee

- Protects direct access to storage, by using seg faults
- Isolates processes from each other, protects itself, even though running on same hardware

Operating System as Glue

- Storage, Window System, UI, Networking

Operating Systems

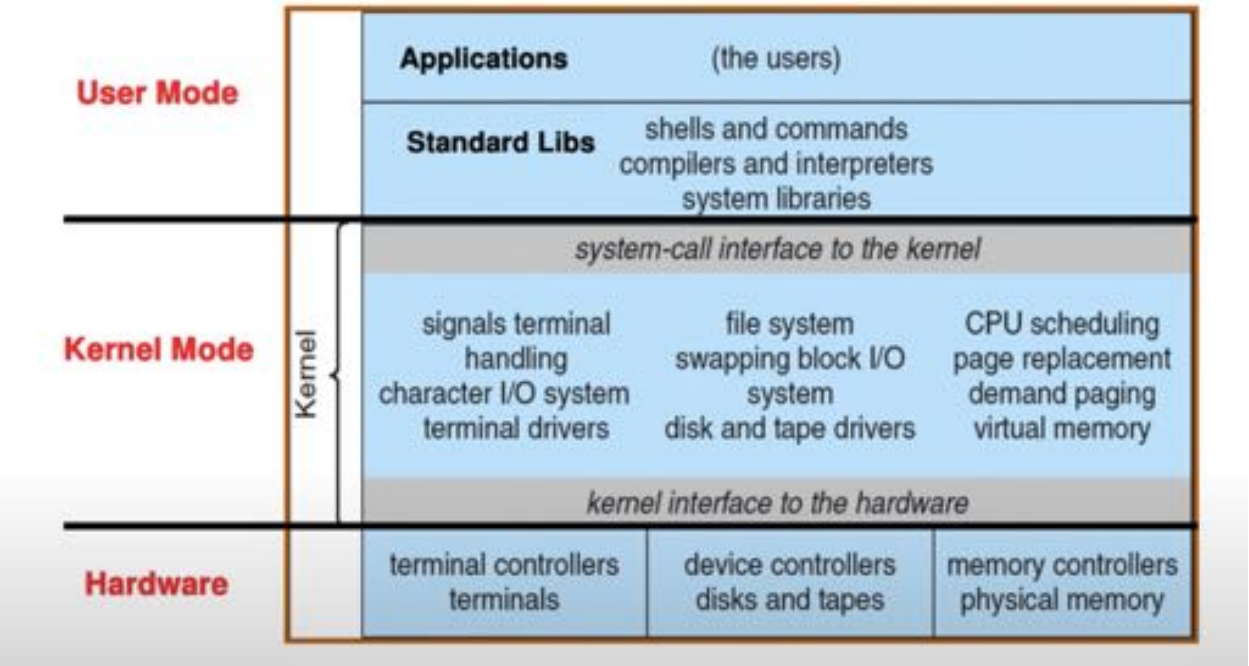
- Challenge: Complexity

- Remove SW/HW quirks by fighting complexity
- Optimize for convenience, utilization, reliability

Basic Tool: Dual-Mode Operation

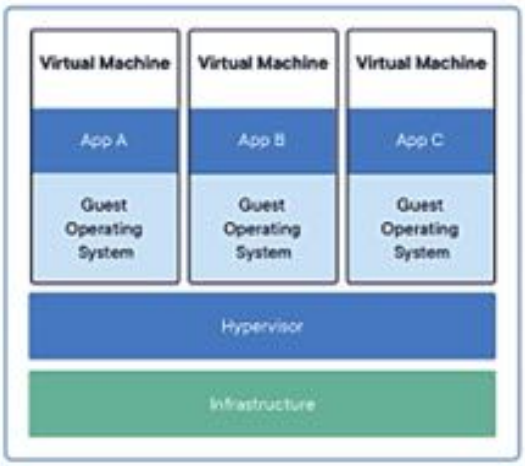
1. Kernel Mode ("supervisor mode")
2. User Mode

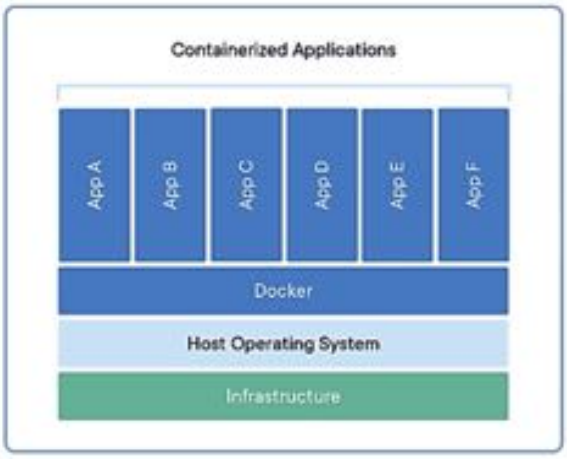
- Certain operations prohibited when running in user mode
- Carefully controlled transitions between user mode and kernel mode



-

- Hypervisor exports a virtual machine, create a place to create





Week 1: Lecture 2 Four Fundamental OS Concepts (1/21)

Review

- Device drivers are caused by interface between hardware device drivers

OS Abstracts Underlying Hardware to help Tame Complexity

- Processor -> Threads
- Memory -> Address Space
- Disks, SSDs, -> Files
- Networks -> Sockets
- Machines -> Processes

<u>Four Fundamental OS Concepts</u>

1. Thread: Execution Context
2. Address Space
3. Process: an instance of a running program
4. Dual Mode Operation/Protection

Thread of Control

- Single unique execution context
- Program Counter, Registers, Execution Flags, Stack, Memory State
- A thread is executing on a processor (core) when it is resident i nthe processor registers
- *Suspended* when state is not loaded into processor

Illusion of Multiple Processors

- Multiplex in time
- Threads are virtual cores
- gSaved in chunk of memory called Thread Control block

Multiprogramming - Multiple Threads of Control

- Thread Control Block (TCB)
    - Holds contents of registers when thread not running
- Where are TCBs stored
    - In the kernel

<u>Address Space</u>

Address Space

- Set of accessible addresses and state associated with them
- 32 bit processor 2^32 64 bit 2^64

Multiprogramming

- All vCPU's share non-CPU resources (memory, I/O devices)

Simple Multiplexing has no Protection

- Allow it to access any place in OS

OS must protect user programs from others and itself

Simple Protection: Base and Bound

- Program address must be greater than Base and smaller than Bound
- Need to know the amount of space

Another idea: Address Space Translation

- Translator for virtual address and physical address

- Break the entire virtual address space into equal size chunks (pages)
- All pages same size so easy to place,
- Hardware translates address using page table
- Need special hardware to translate

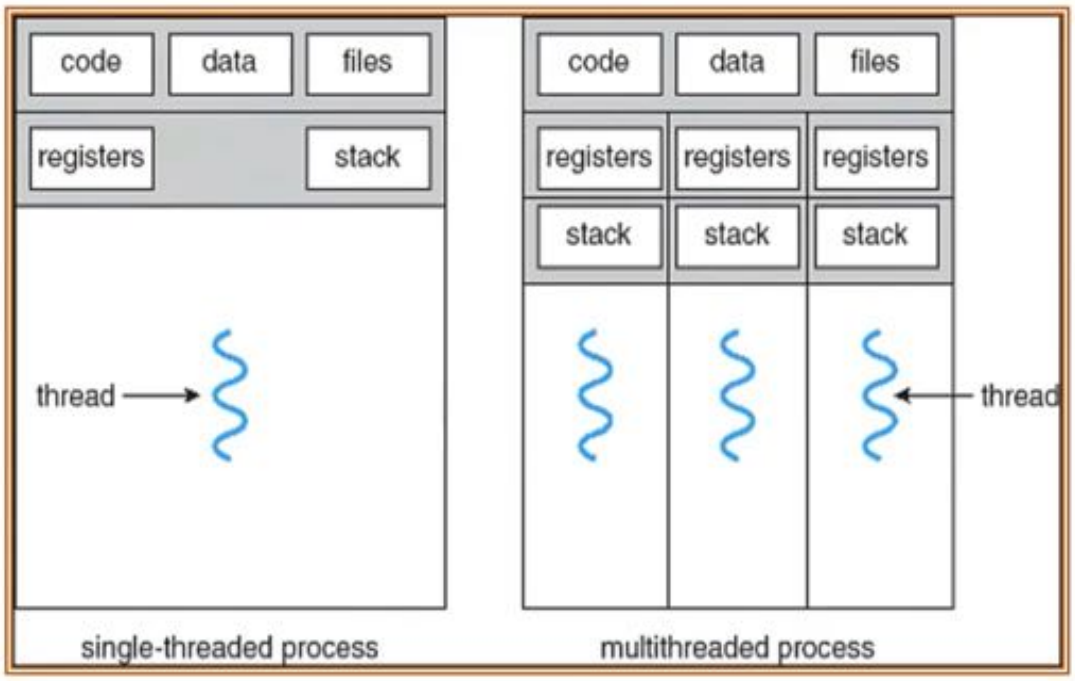
Process

Process

- Execution environment with restricted rights
- Protected from each other, OS protected

Single Multithreaded Processes

- Threads encapsulate *concurrency*
- Address spaces encapsulate *protection*
- Multiple threads per address space
    - Parallelism: take advantage of actual hardware parallelism
    - Concurrency: ease of handling I/O



Hardware must support privilege levels to prevent thread from modifying page table register

Dual Mode Operation

Hardware provides at least 2 modes

- Kernel Mode (“supervisor”)
- User Mode
    - Certain operations are prohibited

3 types of user -> Kernel Mode Transfer

- Syscall
    - Process requests a system service, exit

        - Like a function call but "outside the process"
    - Interrupt
        - External asynchronous event triggers context switch, Timer I/O
    - Trap or execution
        - Something went wrong

Simple B&B: OS gets ready to execute process

- userPC and osPC, (uPC, PC)
- Privileged Inst: set special registers

Unprogrammed control transfers

- User -> Kernel mode transitions are unprogrammed control transfers, require support of lookup tables

Context switching

- Interrupt: process gives control to kernel, save current int TLB, set PC, load new PC and run from uPC

We have basic mechanism to

- Switch between user processes and the kernel,
- The kernel can switch among user processes
- Protect OS from user processes and processes from each other

Week 2: Lecture 3 Abstractions 1: Threads and Processes (1/26)

What Threads Are

- Abstraction of single execution sequence that represents a separately schedulable task

Motivation for Threads

- Operating systems must handle multiple things at once
    - Processes, interrupts, background system maintenance
- Networked servers must handle concurrent requests
- Parallel programs for performance
- Programs with user interface need threading for responsiveness

Multiprocessing vs Multiprogramming

- Multiprocessing: Multiple CPUs (cores)
- Multiprogramming: multiple jobs/processes
- Multi Threadings: multiple threads/processes

Running concurrently

- Scheduler runs threads in any order and interleaving

Concurrency is not Parallelism

- Concurrency is about handling multiple things at once
- Parallelism is about doing multiple things at the same time (multiple cores)
- Parallel -> concurrent but not necessarily other way around

Thread Mask I/O Latency

- 3 states

- Running - running
- Ready - eligible to run, not running
- Blocked - ineligible to run

Multithreaded Programs

- Compiles C program creates a process that is executing that program
- New process has one thread

OS Library API for Threads

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void*), void *arg);
```

– thread is created executing *start_routine* with *arg* as its sole argument.
– return is implicit call to pthread_exit

```
void pthread_exit(void *value_ptr);
```

– terminates the thread and makes *value_ptr* available to any successful join

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

– suspends execution of the calling thread until the target *thread* terminates.
– On return with a non-NULL *value_ptr* the value passed to *pthread_exit()* by the terminat
thread is made available in the location referenced by *value_ptr*.

- What happens when pthread_create() is called in a process
- Int pthread_create() {
  - Asm code … syscall # into %eax
  - Put argos into registers %ebx
  - Special trap instruction
    - Kernel
      - Get args from regs
      - Dispatch to system func
      - Do work to spawn the new thread
      - Store return value in %eax
- }

Fork-Join pattern

- Create (fork) collection of sub-threads passing them argos
- May not exit in the order they were created

Shared State

- Shared: heap, global variables, code
- Per thread State: Thread Control block (TCB), Stack info, saved registers, stack, thread metadata

Memory Layout with Two threads

- Two sets of CPU registers
- Two sets of stacks

Interleaving and Nondeterminism

Non-determinism:

- Scheduler can run threads in any order, witch threads at any time
- Independent threads
    - No state shared with other threads, deterministic

Race Conditions

- Can get different threads

Relevant Definitions

- Synchronization: coordination among threads
- Mutual Exclusion: ensuring only one thread does a particular task at a time
- Critical Section: code exactly one thread can execute a one
- Lock: an object can only access at one time

Processing

Bootstrapping

- First process is started by the kernel: init process
- First process then creates new processes

Process Management API

1. exit: terminate a process
    a. Automatically called in main
2. fork: copy the current process
    a. Copy the currently process, new process has different pid, new process contains a single thread
    b. Parent process
        i. getpid() > 0: parent
        ii. getpid() == 0: child
            1. Can make different processes for the parents and child
        iii. getpid() < 0: failed
    c. Processes do not share the same variables, no race condition
3. exec: change the program being run by the current process
    a. Child process calls execv
4. wait: wait for a process to finish
    a. Parent process creates new program and waits for new program to finish
5. kill : send a signal (interrupt-like notification) to another process
6. signaction : set handlers for signals
    a. Determine what a program does when it receives a specific signal
    b. If there is no handler, the process dies

Week 2: Lecture 4 Abstractions 2: File (1/28)

OS Library Issues Syscalls

- Use an wrapper for calling into syscall

Pthread

- POSIX thread library
- POSIX: Portable Operating System Interface
    - Interface for application programmers
    - Requries standard system call interface

Unix POSIX Idea: Everything is a “file”

- Based on system calls: open, read, write, close
- ioctl() for custom configuration that doesn’t quite fit
- File System Abstraction

File System Abstraction

- File
    - Named collection of data in a file system, sequence of bytes
    - File metadata: info about the file
- Directory
    - Folder containing files & directories
    - Hierarchical naming, links and volumes

Connecting Processes, FIle Systems, and Users

- Every process has a current working directory
- Relatives paths are relative to CWD

<u>The FIle Abstraction</u>

C High Level FIle API

- Operates on “streams” unformatted sequences of bytes
- FILE data structure after fopen

stdio.h

- FILE *stdin - normal source of input, can be redirected
- FILE *stdout - normal source of output, can also be redirected
- FILE *stderr - diagnostics and errors

C High-Level File API

- fputc(int c, FILE *fp)
- Fputs
- Fgetc, fgets
- EOF: end of file

Block by Block

- Fread, fwrite

```c
#define BUFFER_SIZE 1024
int main(void) {
  FILE* input = fopen("input.txt", "r");
  FILE* output = fopen("output.txt", "w");
  char buffer[BUFFER_SIZE];
  size_t length;
  length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
  while (length > 0) {
    fwrite(buffer, length, sizeof(char), output);
    length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
  }
  fclose(input);
  fclose(output);
}
```

System Programming

- Remember to check for errors perror(<message>)

Positioning the Pointer

- Int fseek(FILE *stream, long int offset, int whence);
- Long int ftell(FILE *stream) : tells you where the value is
- Void rewind (FILE *stream)

Low-Level File I/O

- Everything is a file: simple composition of programs
- Open before use
- Methods
    - Int open, creat, close
- Returns a file descriptor,
- Write data to open file using file descriptor

Kernel Buffering

- Reads are buffered inside kernel

Other Operations

- Ioctl: set features to specific devices
- Duplication descriptors (dup), pipe to channel (pipe)

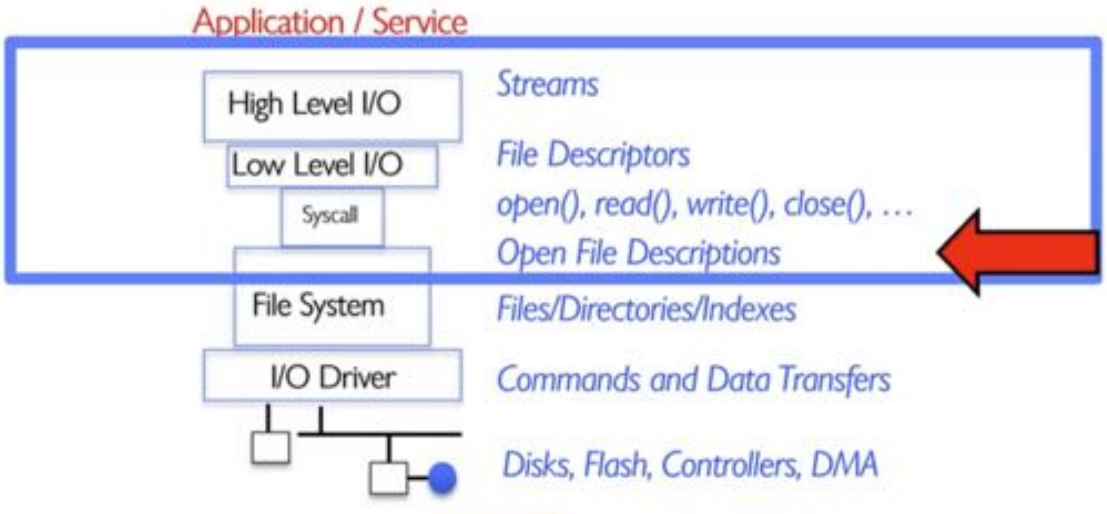
What's in FILE?

- File descriptor (from call to open)
- Buffer (array)
- Lock (use FILE concurrently)
- Fflush before reopening same file

Why Buffer in Userspace? Functionality!

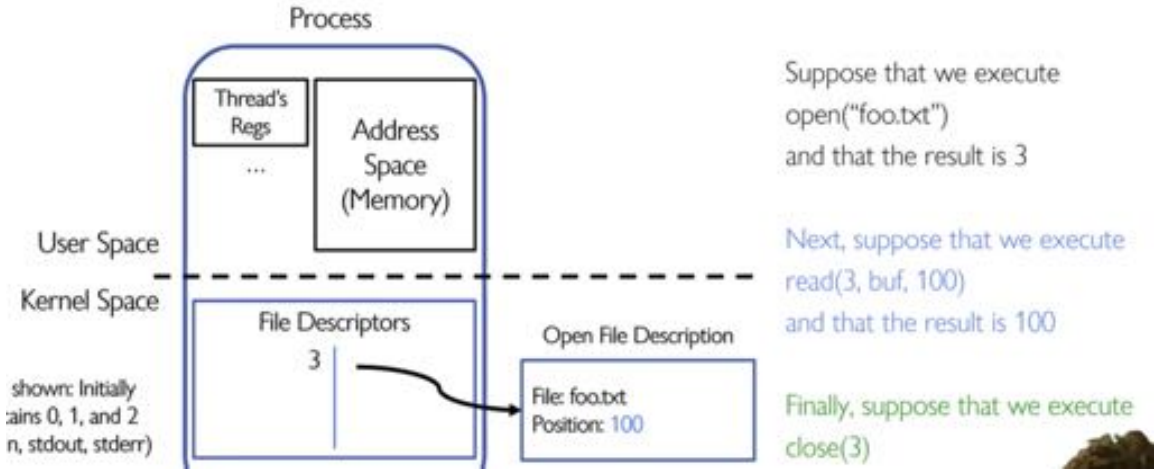
- System call operations less capable, slower than normal C function

Process State for File descriptors

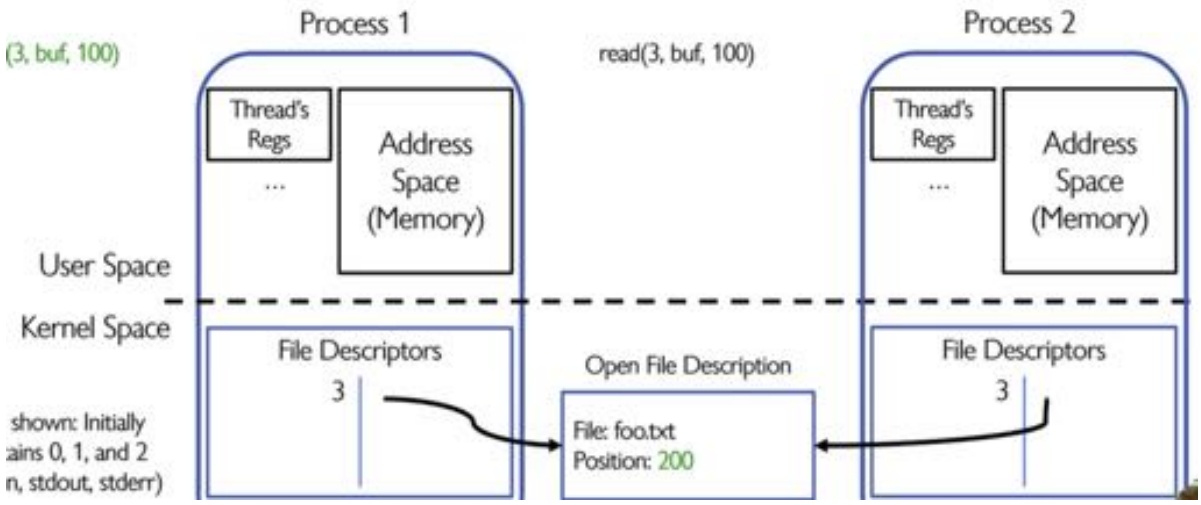


State maintained by the Kernal

- For each process, kernel maintains mapping from file descriptor to open file description
- Finds were the file data on disk and current position within the file



Fork copies address space but shares file descriptors



Shared resources for fork

Pitfalls with OS Abstractions

Don't fork in a process that already has multiple threads

- Child process always has just a single thread
- Its safe if you call exec in the child

Don't mix low level and high level file I/O

- The fread will have a local user level buffering chunck while low level has kernel file level positioning

Be careful with fork() with FILE*

Week 3: Lecture 5 IPC, Pipes and Sockets (2/2)

A quick programmer's viewpoint

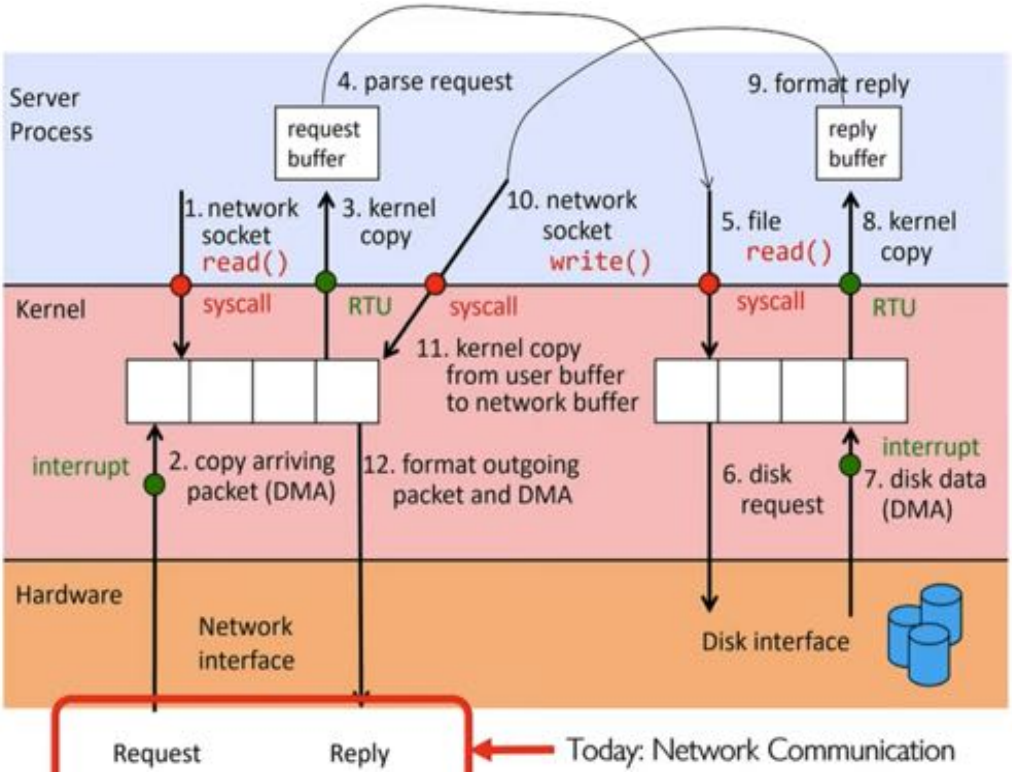
Communication between processes and across the world File I/O

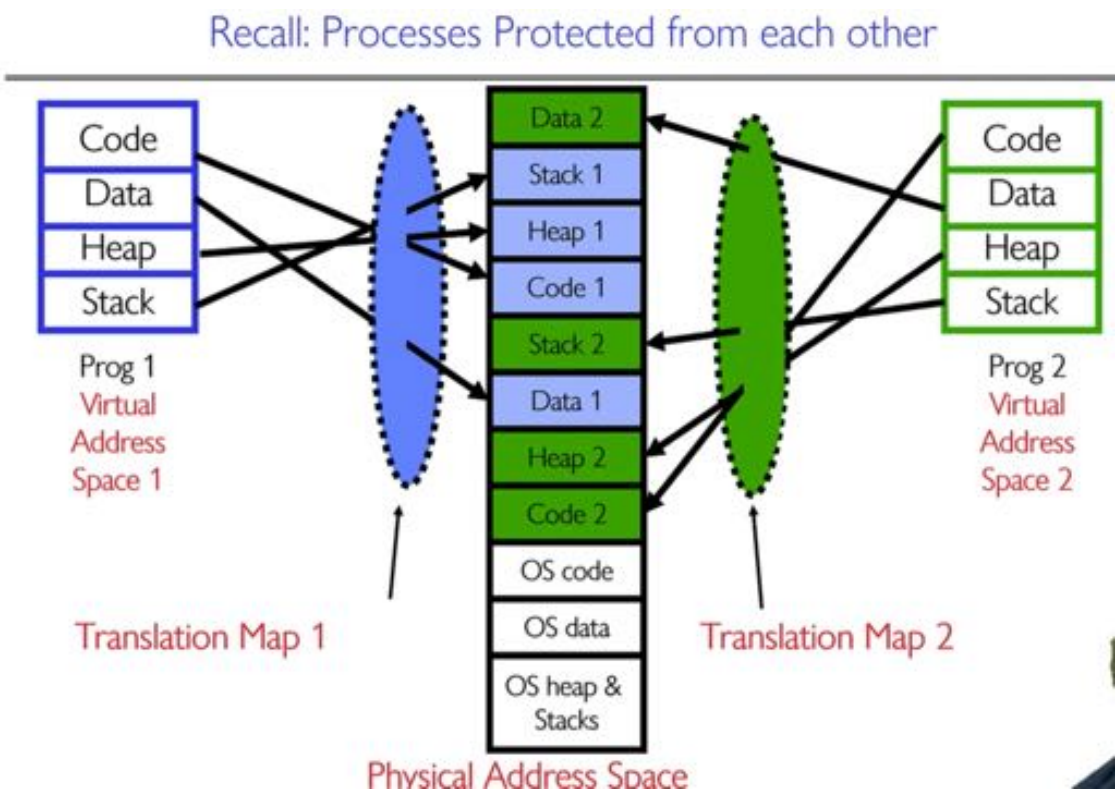
- Uniformity -- Everything Is a FIle
- Open before use
- Explicit close

Web Server

Server Process

1. Network socket read() wait





Communication between processes (Another Option)

- Ask Kernel to help, in memory queue, accessed by system calls
- Data written by A is held in memory until B uses it

Pipe

```
write(wfd, wbuf, wlen);
```


```
n = read(rfd, rbuf, rmax);
```

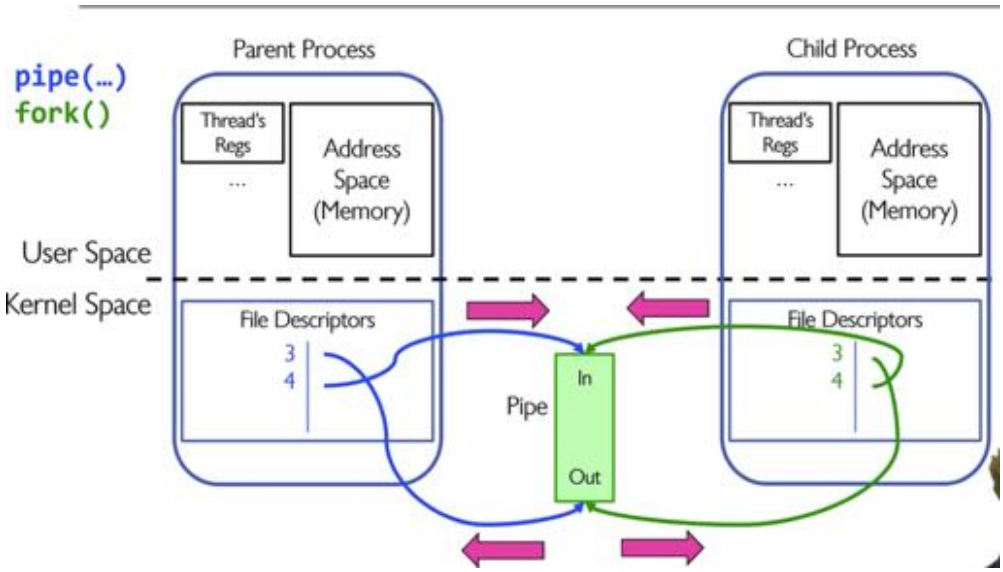
- Producer (A) tries to write when buffer full, it blocks
- If consumer (B) tries to read when buffer empty, it blocks

Int pipe(int fields[2])

- Allocates two new file descriptors in the process

Pipes Between Processes

- pipe()
- Fork
- Depending on pid,
- Write or read



Once we have communication, we need a protocol

- Protocol is agreement on hwo to communicate
- Syntax: how a communication is specified structured
- Semantics: what a communication means
- Described by state machine

Client-server communication

- Client is sometimes on,
- Server is always on listening to requests

Network Connection

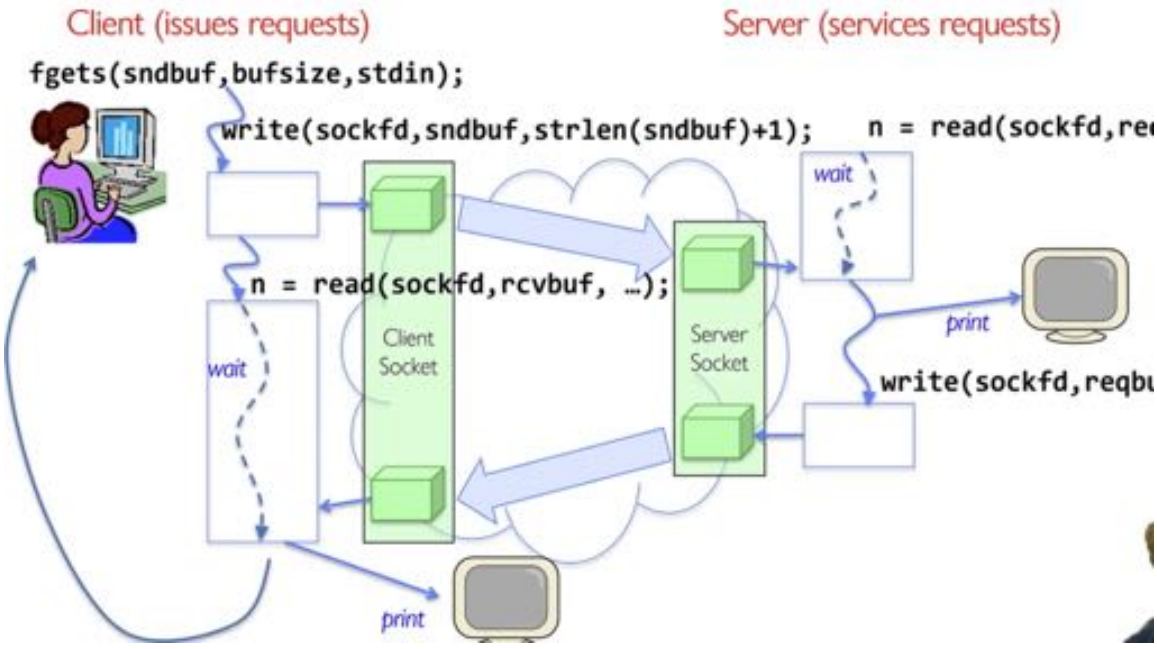
- Bidirectional stream of bytes between two processes “TCP” Connections

The Socket Abstraction: Endpoint for communication

- Abstraction for one endpooint of a network connection
- Standarized by POSIX
- Same abstraction for any kind of network
    - Local, the internet,

Sockets: More Details

- Looks just like a file with a file descriptor
    - Maybe need messaging facility
- Each socket has a receiving and sending queue



We assume that the stream is sequential

Socket Creation

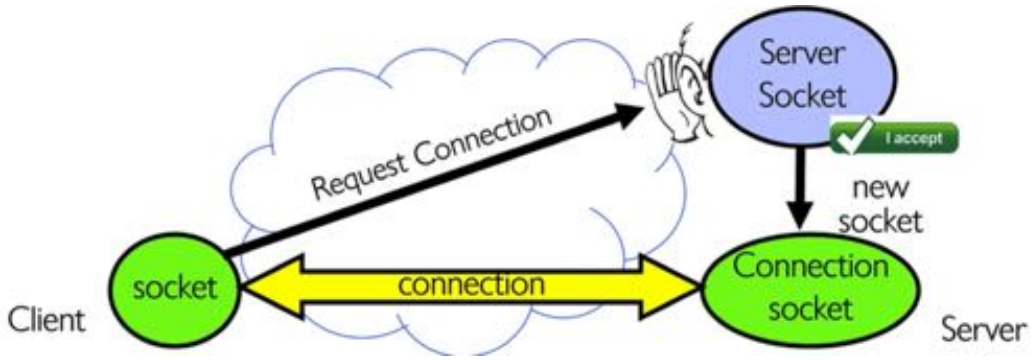
- File systems have a permanent objects in structured name space
- Pipes: one way communication between processes on same physical machine

How do we name the process

- IP to refer to socket

- Port Number distinguishes between different processes on same machine

Connection Setup over TCP/IP

- Special kind of socket: server socket
  - Has file descriptor
  - Can't read or write
  - listen(): start allowing clients to connect
  - accept(): create a new socket for particular client
- 

Sockets in Concept

1. Server socket binded to address
2. Listen for connection
3. Client socket created and connect to server
4. Server accepts syscall
5. Write request, read request, write response, read response
6. Close

Client protocol

- Lookup_host
- socket(server-ai_family, server->ai_socktype, server->aiprotocol)
- connect()
- run_client()
- close()

Server Protocol (v1)

- setup_address(port_name)
  Socket
- bind () : bind socket to specific port
- listen ()

Running code from different users in the same process

- Want a new process for each accepting connection, When receive, fork a new process

Concurrent server can handle and service a new connection before the previous client disconnects

Connection Setup over TCP/IP

1. Source IP
2. Destination IP

3. Source Port Number
4. Destination Port Number
5. Protocol (TCP)

Thread Pools

- Unbounded Theads
    - Too popular, throughput sinks
- Allocate a bounded pool of worker threads, represeenting the maximum level of multiprogramming
- When servicing a request, use a thread from the pool, If no thread available, queue request

Conclusion

- Interprocess Communication (IPC)
    - Communication facility between protected environments
- PIpes are abstraction of single queue
- Sockets are an abstraction of two queues, one in each direction

Week 3: Lecture 6 Synchronization 1: Concurrency & Mutual Exclusion (2/4)

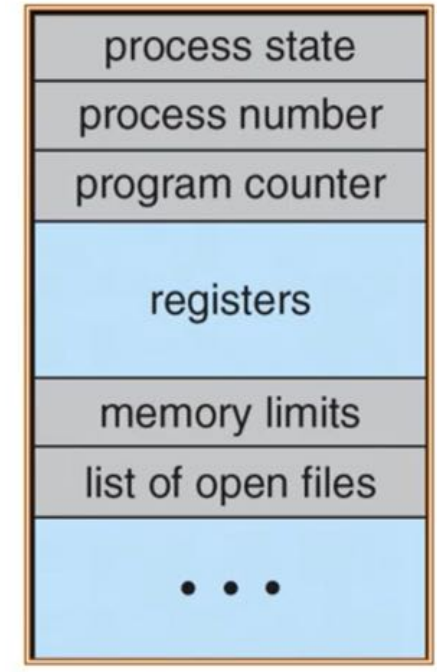
Connection Setup over TCP/IP

- Often Client port "randomly" assigned
- Server port has well known ports 80 (web), 443 (Secure web)

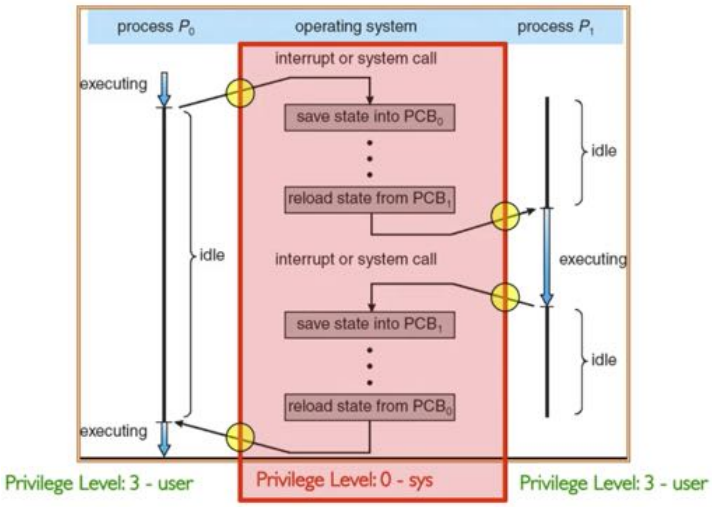
Multiplexing Processes: Process Control Block

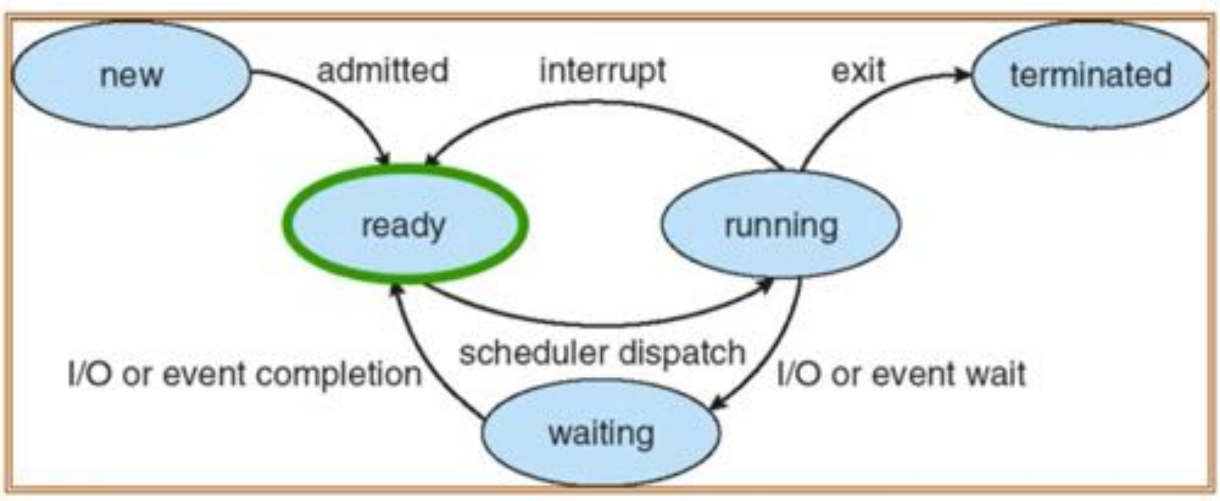
Process Control Block

- Kernel represents each process as a process control block (PCB)
    - Status, register state, process ID, execution time, memory space
- Transition from privilege level to user to system
- Context Switch



Process Control Block

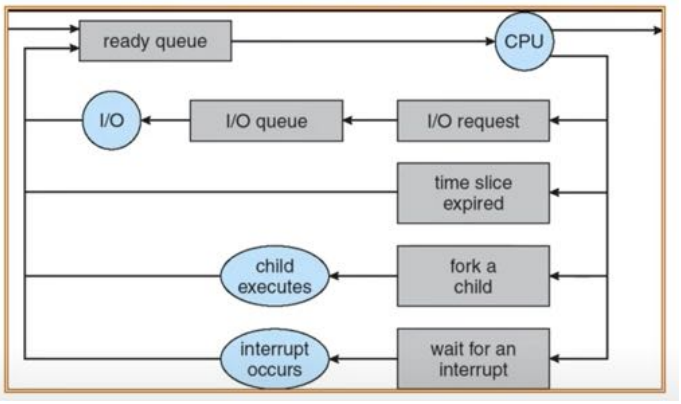


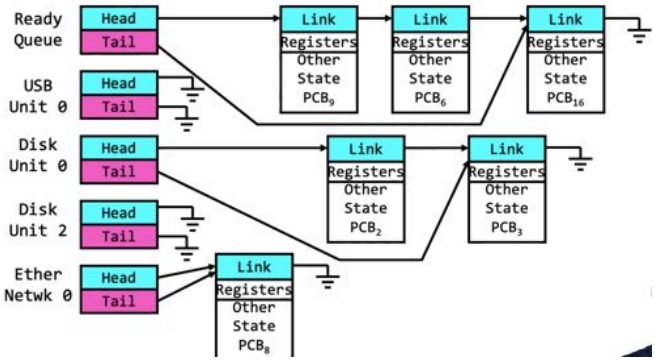


Lifecycle of a process or thread

- New: being created
- Ready: waiting to run
- Running: being executed
- Waiting: waiting for some event to occur
- Terminated: finished execution

Scheduling: all about Queues





Scheduler:

- Mechanism for deciding which processes/thread receive the CPU

Single and Multithreaded Processes

- Threads encapsulate concurrency: Active component
- Address spaces encapsulate protection : Passive part
- Each thread has their own stack

The dispatch Loop

- Loop {
    - RunThread(),
        - Load state into CPU, load environment, jump to PC
        - Internal events, thread returns control voluntarily

        - External events thread gets preempted
    - ChooseNextThread,
    - SaveStateOfCPU(curTCB)
    - LoadStateOfCPU(newTCB)

Internal Events

- Blocking on I/O
    - Act of requesting I/O Implicitly yields the CPU
- Waiting on signal from other thread
    - Thread asks to wait and thus yields the CPU
- Thread executes a yield()
    - THread volunteers to give up CPU

```
int pthread_create(pthread_t *thread, const pthread_attr_t *at
                   void *(*start_routine)(void*), void *arg);
```

– thread is created executing *start_routine* with *arg* as its sole argument.

– return is implicit call to pthread_exit

```
void pthread_exit(void *value_ptr);
```

– terminates the thread and makes *value_ptr* available to any successful join

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

– suspends execution of the calling thread until the target *thread* terminates.

– On return with a non-NULL *value_ptr* the value passed to *pthread_exit()* by the terminating thread is made available in the location referenced by *value_ptr*.

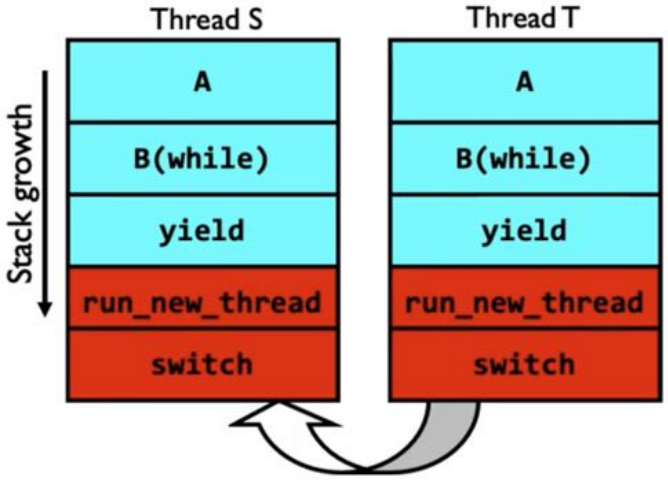
```
void pthread_yield(void);
void sched_yield(void);
```

- – Current thread *yields* (gives up) CPU so that another thread can run

- Stack for yieldthing Thread
    - run_new_thread()
        - pickNewTHread
        - switch(curThread, newThread
        - ThreadHouse Keeping
    - Dispatcher switch to a new thread
        - Save anything next thread may trash

What Do the Stacks Look Like?

- switch(rCur, tNew)
    - Unload old thread
    - Load and execute new thread
    - Save registers



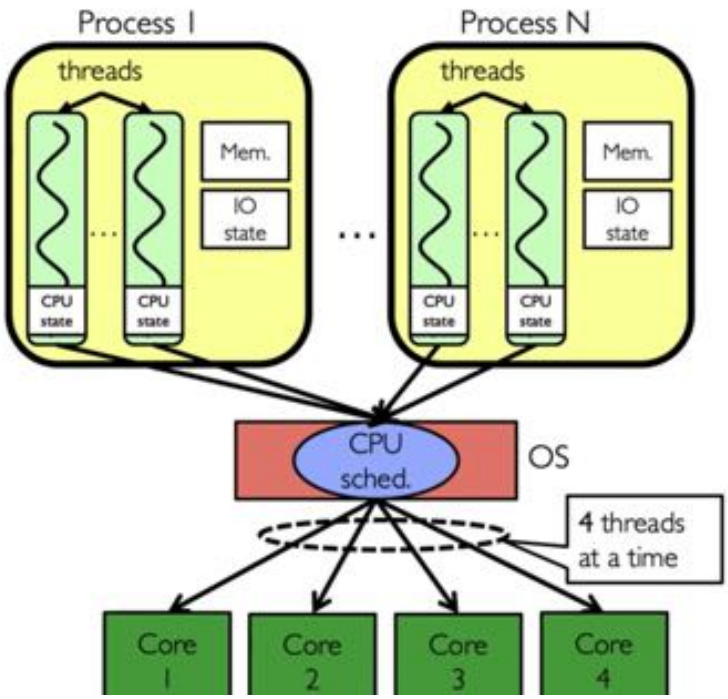
Thread S's switch returns to Thread T's (and vice versa)

- Cannot exhaustively test switch code
- Frequency of context switch: 10-100ms
- Switching between processes: 3-4 usec
- Switching between threads: 100ns

Processes vs. Threads

- Switch overhead:
  - Same process: low
  - Different proc: high
- Protection
  - Same proc: low
  - Different proc: high
- Sharing overhead
  - Same proc: low
  - Different proc: high
- Parallelism : no

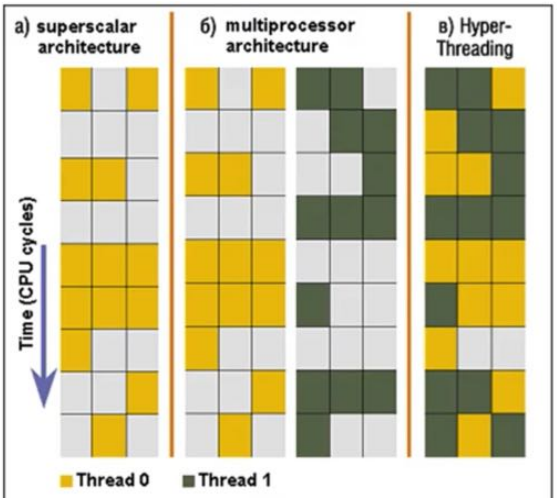
Mutlicore



- Switch overhead:
  - Same process: lov
  - Different proc.: hig
- Protection
  - Same proc: low
  - Different proc: higl
- Sharing overhead
  - Same proc: low
  - Different proc, simultaneous core:
  - Different proc, offloaded core: hig
- Parallelism: yes

Hyperthreading

- Hardware scheduling technique, superscalar processors can execute multiple instructions that are independent,
- Hyperthreading duplicates register state to amke a second thread



- Interweave the two

What happens when thread blocks on I/O?

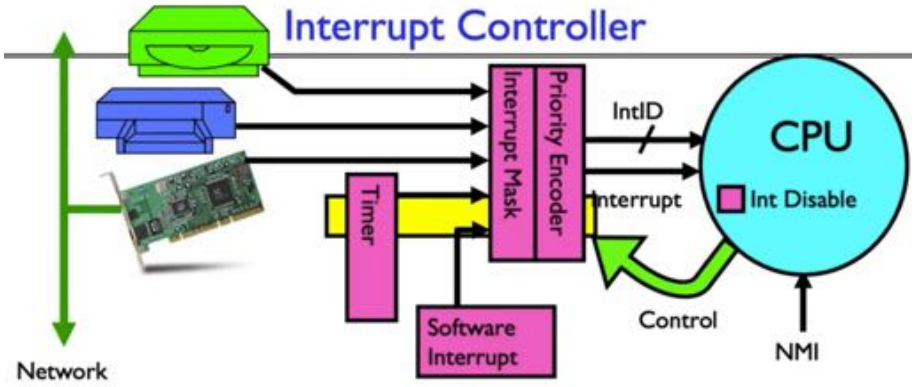
- Sys call, read, run new thread and switch while waiting
- Wait for signal/join

External Events

External Events

- Find a way that dispatcher cna regain control
- Interrupts: signals from hardware or software that stop the running code and jump to kernel
- TImer: like an alarm clock that goes off every milliseconds

Interrupt Controller



- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Interrupt identity specified with ID line
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
- CPU can disable all interrupts with internal flag

- Non-maskable interrupt line can't be disabled

Example: Netowrk INterrupt

- After receiving interrupt, do pipeline flush,
- Save PC, switch to kernel mode
- After finishing interrupt handler
- Restore PC, enable all ints user mode

An interrupt is a hardware-invoked context switch

- No separate step to choose what to run next, interrupt handler runs immediately

Use of Timer Interrupt to Return control

- Can have it check for pi

How do we initalize TCB and Stack?

- Initialize Register fields of TCB
- Stack pointer made to point at stack

- Don't need to initialize stack data
- Thread Root
    - Root for the thread routine
    - Startup Housekeeping()
    - UserModeSwitch
    - Call fcnPtr(fcnArgPtr)
    - ThreadFinish()
- How do we make a new thread
    - Setup TCB/kernel thread to pointa t new user stack and ThreadRoot code
    - Put pointers to start function and args

Correctness with Concurrent Threads

- Non-determinism
    - Any order, any time
    - Take program make sure it doesn't block
    - Shared state can get corrupted

Atomic Operation

- An operation that always runs to completion or not at all
- It is indivisible
- Many instructions are not atomic
- Locks: lock, unlock, wait
    - Only one thread gets to run at a time

Synchronization

- Using atomic operations to ensure cooperation between threads

Mutual excursion

- One thread excludes the other while doing its task

Critical Section

- Critical section is result of mutual exclusion

Locks

- Restricts parallelism but makes sure variables always consistent
- No races at operation level
- Concurrency errors caused the death of nuber of patietns by misconfiguign radiation production

Conclusion

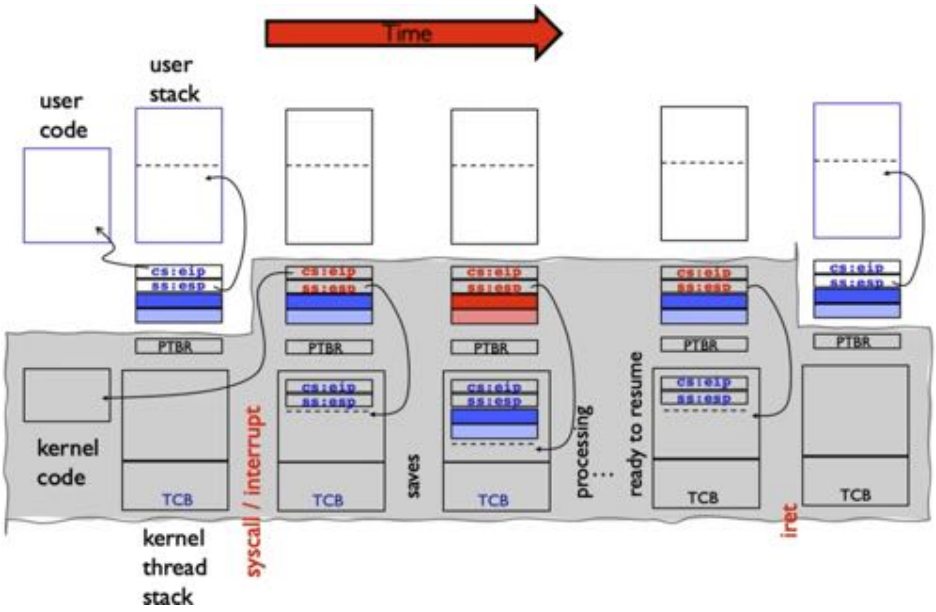
- Concurrency accomplished by multiplexing CPU time
- TCB + Stacks hold complete state of thread for restarting

Week 4: Lecture 7 Synchronization 2: Semaphores, Lock, Atomic Instr (2/9)

Hardware Context Switch Support

-

- Syscall/Intr (U → K)
  - PL 3 → 0;
  - TSS ← EFLAGS, CS:EIP;
  - SS:ESP ← k-thread stack (TSS PL 0);
  - push (old) SS:ESP onto (new) k-stack
  - push (old) eflags, cs:eip, <err>
  - CS:EIP ← <k target handler>
- Then
  - *Handler then saves other regs, etc*
  - *Does all its works, possibly choosing other threads, changing PTBR (CR3)*
  - kernel thread has set up user GPRs
- iret (K → U)
  - PL 0 → 3;
  - Eflags, CS:EIP ← popped off k-stack
  - SS:ESP ← popped off k-stack



Producer-Consumer with a bounded Buffer

- Producer put things into a shared buffer, consumer take them out, synchronization to coordinate producer/consumer

Circular Buffer Data Structure

- Mutex lock = <initially unlocked>
- Producer :acquireds lock,
- Don't want to waste CPU, busy waiting, acquiring and releasing locks

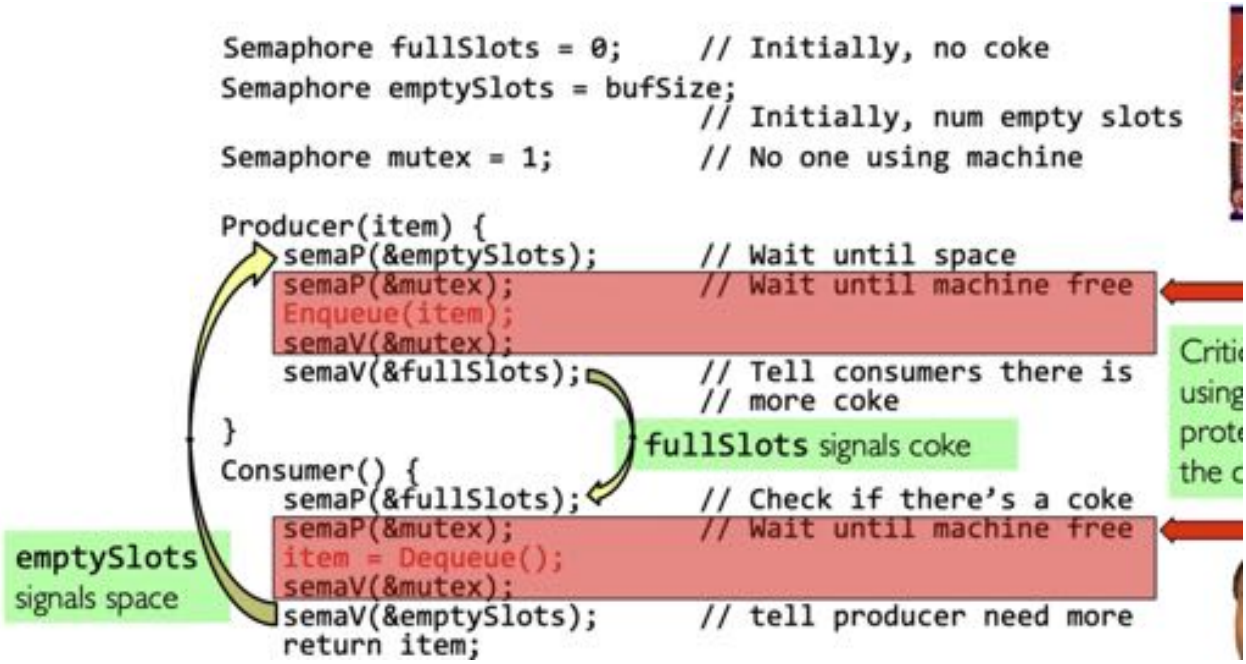
Semaphores

- First defined by Dijkstra
- Main synchronization primitive used in original UNIX
- Only operations allowed are P and V, can't read or write value

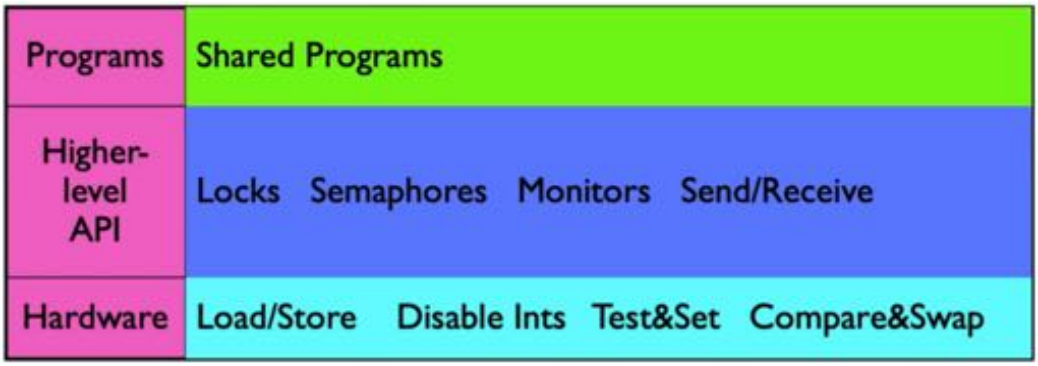
Two Uses of Semphores

- Mutual Exlusion (initial value = 1)
  - Binary Semaphore, mutex
  - Can be used for mutual exclusion, just like a lock
- Scheduling Constraints (initial value = 0)
  - Allow thread 1 to wait for a signal from thread 2

Full solution to bounded buffer



- Consumer is worried about occupied slots, Producer cares about empty slots
- Order of P is important, can cause deadlock
- Order of V doesn't matter, scheduling efficiency
- Not dependent on number of consumers or producers



Example: Too Much Milk

- Communication
- Lock: Prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
- Put a key on the refrigerator
- Never more than one person buys, someone buys if needed
- Worse since fails intermittently
- Starvaing, I'm not getting milk, you're getting milk
- Leave note A, while note B, wait, check milk and buy milk if necessary

- Leave note B
- Acquire milklock (atomic lock for milk)
- If no milk
    - Buy milk
- Release milklock

How to Implement Locks

Locks

- Prevent someone from doing something
- Lock before entering critical section
- Unlock when leaving,
- Wait if locked, sleep
- Atomic load/store, complex and error prone
- Hardware Lock instruction

Naive use of Interrupt Enable Disable

- Avoid internal events, disable interrupts
- Can't let the user does this, never get control back if lock and loop

New Lock Implementation

- Disable interrupts when we check and set lock value

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

Critical Section

- Disabling interrupt for a very short time

How to enable after sleep()

- Interrupts are disabled when you call sleep
- Responsibility of next thread to reenable ints

Atomic Read-Modify-Write Instructions

Previous Solution

- Doesn't work well on mutliprocessor

Atomic instruction sequences

- Read a values and write atomically
- Test&set
    - Is a memory address equal to result
    - Set if it is not 1
- swap(&address, register)

    - Swap register's value to value at "address"
- compare&swap(&address, reg1, reg2)
    - If memory == reg1 put reg2 in memory, otherwise don't change memory
- load-linked&store-conditional(&address)
    - Powerful compare&swap
    - Load value from memory, do a conditional store if changed at all since last check

Conclusion

- Atomic Operations
    - Operation that runs to completion or not at all,
    - Primitives on which to construct various synchronization primities

Week 4: Lecture 8 Synchronization 3: Atomic, Monitors, Reader/Writer (2/11)

Read-Modify-Write

- For user programs because we can't disable interrupts for user programs

Implementing Locks with test&set

- If lock is free, test&set reads 0 and sets lock=1, so lock is now busy
    - It returns 0 so while exists
- If lock is busy, test&set reads 1 and sets lock=1
    - It returns 1 so while loops continues
- When we set thelock = 0, someone else can get the lock

```
// (Free) Can access this memory location from user space!
int mylock = 0; // Interface: acquire(&mylock);
                //            release(&mylock);

acquire(int *thelock) {
   while (test&set(thelock)); // Atomic operation!
}
release(int *thelock) {
   *thelock = 0;              // Atomic operation!
}
```

Very Inefficient , priority inversion: if busy-waiting thread has higher priority than thread holding lock -> no progress

Better Locks using test&set

```
int guard = 0; // Global Variable!
int mylock = FREE; // Interface: acquire(&mylock);
                   //            release(&mylock);

acquire(int *thelock) {
   // Short busy-wait time
   while (test&set(guard));
   if (*thelock == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
      // guard == 0 on wakup!
   } else {
      *thelock = BUSY;
      guard = 0;
   }
}
```

```
release(int *thelock) {
   // Short busy-wait time
   while (test&set(guard));
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      *thelock = FREE;
   }
   guard = 0;
```

Use a guard instead of disabling interrupts

- Replaced disable interrupts -> while (test&set(guard))
- Enable interrupts -> guard = 0

Linux futex: Fast userspace Mutex

- Uaddr points to a 32 bit value in user space
- Futex_op
  - FUTEX_WAIT, FUTEX_WAKE, FUTEX_FD
- Futex: kernelspace wait quee attached to userspace atomic integer
- Idea: Userspace lock is syscall-free in uncontended case
- Lock has three states
  - Free
  - Busy, no waiters
  - Busy, possibly with some waiters

```
typedef enum { UNLOCKED,LOCKED,CONTESTED } Lo
Lock mylock = UNLOCKED; // Interface: acquire
                        //            release

acquire(Lock *thelock) {
   // If unlocked, grab lock!
   if (compare&swap(thelock,UNLOCKED,LOCKED))
      return;

   // Keep trying to grab lock, sleep in fute
   while (swap(mylock,CONTESTED) != UNLOCKED)
      // Sleep unless someone releases hear!
      futex(thelock, FUTEX_WAIT, CONTESTED);
}

release(Lock *thelock) {
   // If someone sleeping,
   if (swap(thelock,UNLOCKED) == CONTESTED)
      futex(thelock,FUTEX_WAKE,1);
```

Using more atomics and futex

<u>More Synchonizaiton</u>

Semaphores

- Down or P wait for semaphore to be positive, then decrements
- Up or V: an atomic operation that increments by 1 waking up a waiting P if any

Bounded Buffer: Correctness constraints for solution

Monitors are better!

- Problem is that semaphores are dual purpose
    - They are used for both mutex and scheduling constraints
- Monitors:
    - Lock and zero or more condition variables for managing concurrent access to shared data
    - Some languages like Java provide this natively
    - Most others use actual locks and condition variables
    - A monitor is paradigm for concurrent programming

Condition Variables

- A queue of threads waiting for something inside a critical section
- Allow sleeping inside critical section by atomically releasing lock at time we go to sleep
- Can't wait inside critical section

Operations

- Wait(lock)
    - atomically release lock and go to sleep
- Signal()
    - wake up one waiter
- Broadcast
    - wake up all waitiers

```
lock buf_lock;                  // Initially unlocked
condition buf_CV;               // Initially empty
queue queue;


Producer(item) {
   acquire(&buf_lock);          // Get Lock
   enqueue(&queue,item);        // Add item
   cond_signal(&buf_CV);        // Signal any waiters
   release(&buf_lock);          // Release Lock
}

Consumer() {
   acquire(&buf_lock);          // Get Lock
   while (isEmpty(&queue)) {
      cond_wait(&buf_CV, &buf_lock); // If empty, sleep
   }
   item = dequeue(&queue);      // Get next item
   release(&buf_lock);          // Release Lock
   return(item);
}
```

Mesa vs Hoare monitors

- Hare monitors

    - Waitier gives up lock, processor back to signaler when it exits critical section or if it waits again
- Mesa monitors
    - Signaler keeps lock and processo
    - Waiter placed on ready queue with no special priority
    - Need to check condition again after wait

Readers/Writers Problem

Wwant many readers at the same time, one writer at a time

Correctness constraints

- Readers can access database when no writers
- Writers can access database when nonreaders or writers
- Only one thread manipulates state variables at a time

```
Reader() {
  // First check self into system
  acquire(&lock);
  while ((AW + WW) > 0) {  // Is it safe to read?
    WR++;                  // No. Writers exist
    cond_wait(&okToRead,&lock);// Sleep on cond var
    WR--;                  // No longer waiting
  }
  AR++;                    // Now we are active!
  release(&lock);
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  acquire(&lock);
  AR--;                    // No longer active
  if (AR == 0 && WW > 0)   // No other active readers
    cond_signal(&okToWrite);// Wake up one writer
  release(&lock);
}
```

- Lock used to protect state variables which keep track of who is active and waiting reader/writer
- Writers, given priority

```
Writer() {
  // First check self into system
  acquire(&lock);
  while ((AW + AR) > 0) { // Is it safe to write?
    WW++;                 // No. Active users exist
    cond_wait(&okToWrite,&lock);  // Sleep on cond var
    WW--;                 // No longer waiting
  }
  AW++;                   // Now we are active!
  release(&lock);
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  acquire(&lock);
  AW--;                   // No longer active
  if (WW > 0){            // Give priority to writers
    cond_signal(&okToWrite);// Wake up one writer
  } else if (WR > 0) {    // Otherwise, wake reader
    cond_broadcast(&okToRead); // Wake all readers
  }
  release(&lock);
```

Conclusion

- Atomic operations run to completion or not at all, primitives on which to construct various synchronization primitives
- Can use hardware atomicity primitis

Semaphores

- Two operations P(), V()
- Separate semaphore for each constraint

Monitors

- A lock plu one or more condition variables
- Acquire lock before accessing shared data
- wait(), signal() broadcast

Week 5: Lecture 9 Synchronization 4: Process Structure, Device Drivers (2/16)

Want to reduce busy waiting

- Must hold the lock when doing condition variable ops

Readers/Writers Solution

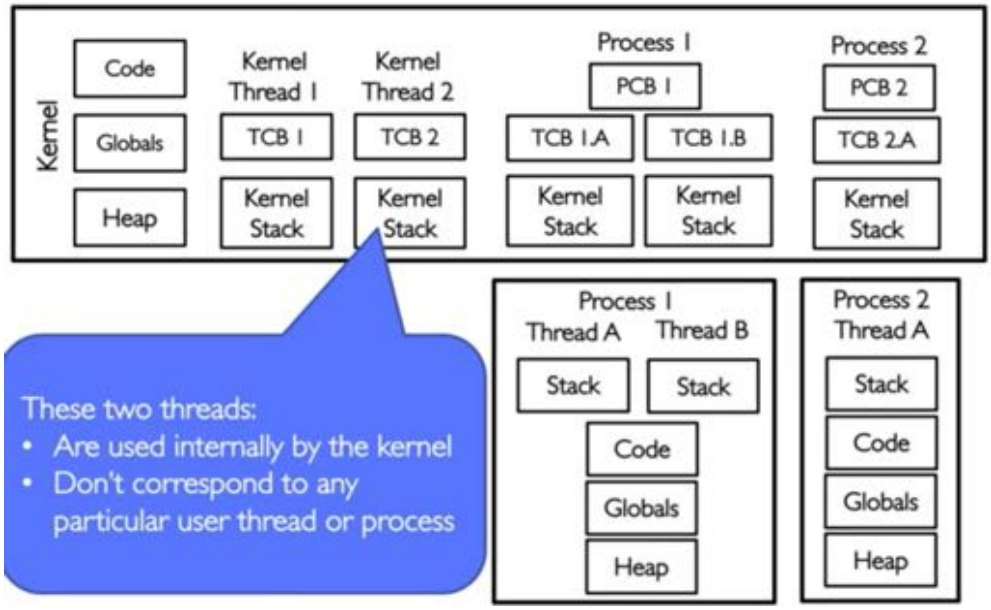
- Signal to waiting writers
- Broadcast to waiting readers

Can we construct Monitors from Semaphores

- Not legal to look at contents of semaphore queue
- There is a race condition -- signaler can slip in after lock release and before waiter executes semaphore

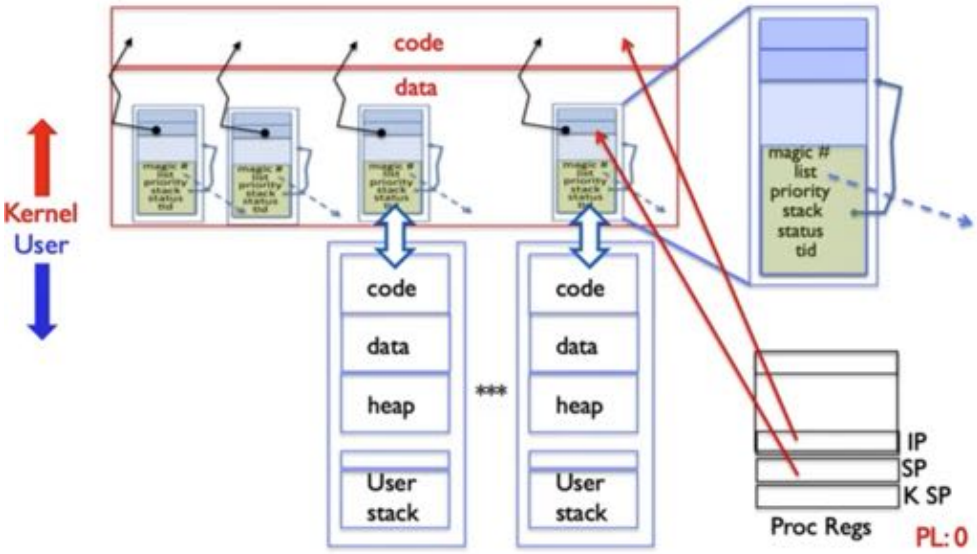
DIfferent languages handle releasing locks in different ways
Multithreaded Process

- Kenel maintains threads and processes
- Linux and pintos embedded into linked list list elem
-



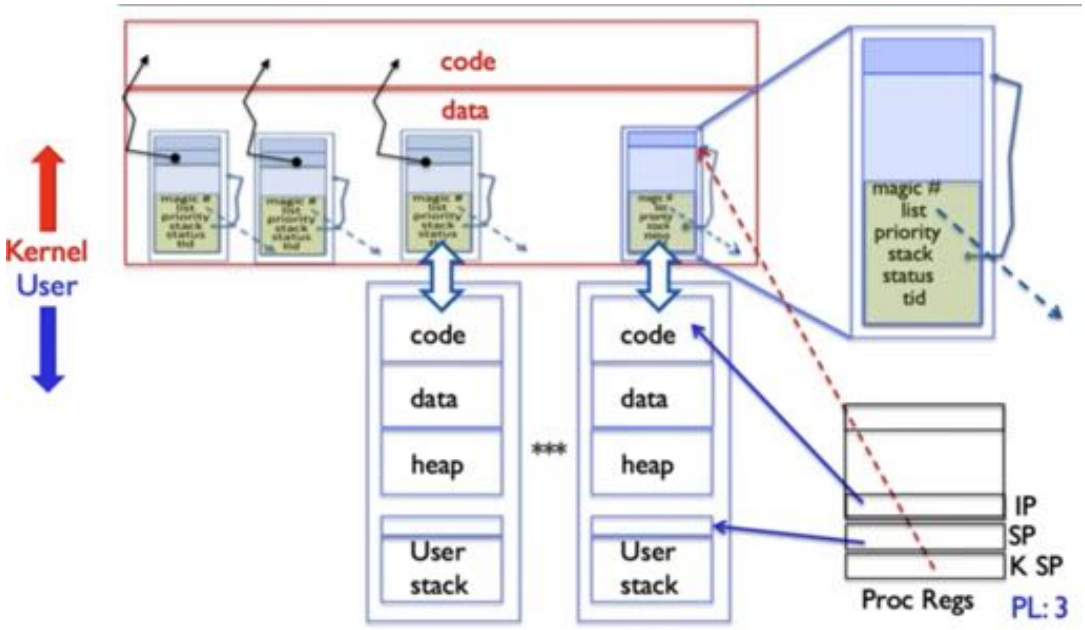
In Kernel Thread: No User Component

- User -> Kernel (exceptions, syscalls)
- Mechanism to resume k-thread goes through interrupt vector



User

- Each user process associated witha kernel thread, described by a 4KB page object containing TCB and kernel stack for the kernel stack
- Typically Kernel thread is “standing by”
- During iret function :
    - Restores user stack, IP, and PL



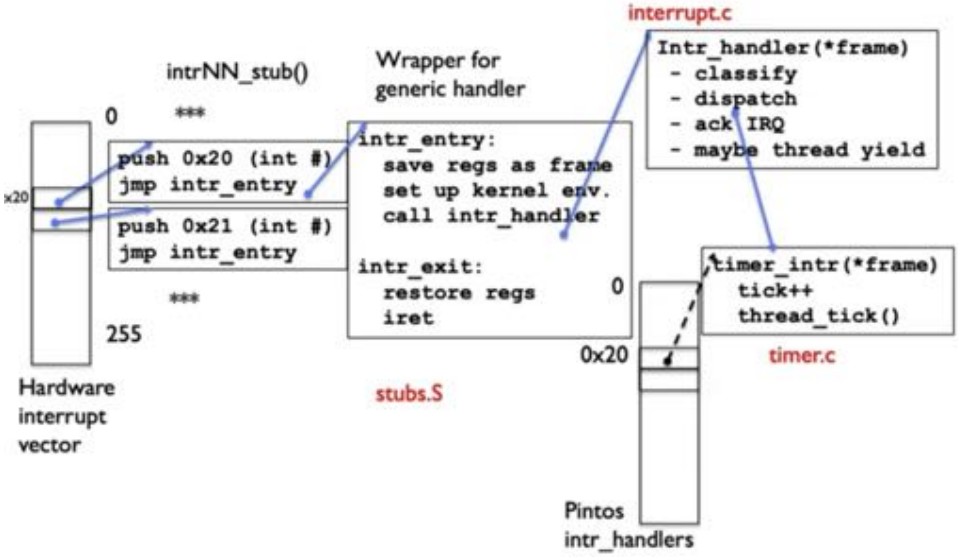
Pintos Interrupt Processing

- Pushes generic handler,
- Wrapper for generic handler

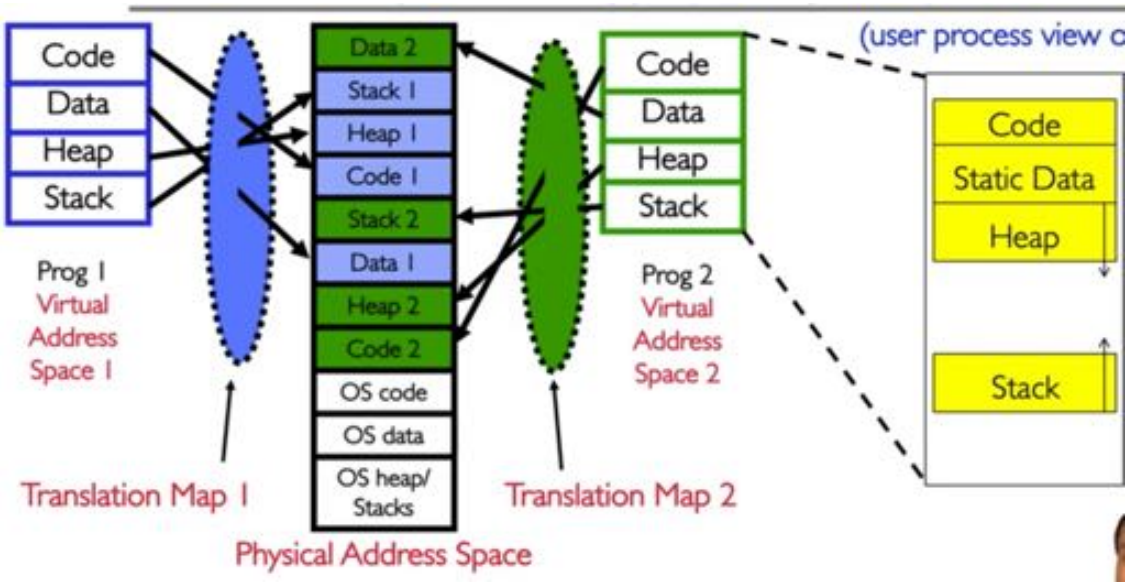
Scheduling

- Deciding which thread given access to resources from moment to moment

Address Space



- Page table is primary mechanism
- Privilege level determine which region can be accessed
- System can access all, User only part
- Each process has its own address space
- All system threads share the same address space and memory



Internal OS File Description

- Internal Data Structure describing everything about the file
- Pointer: struct file *file
- Struct file_operation *f_op

Why everything can look like a file

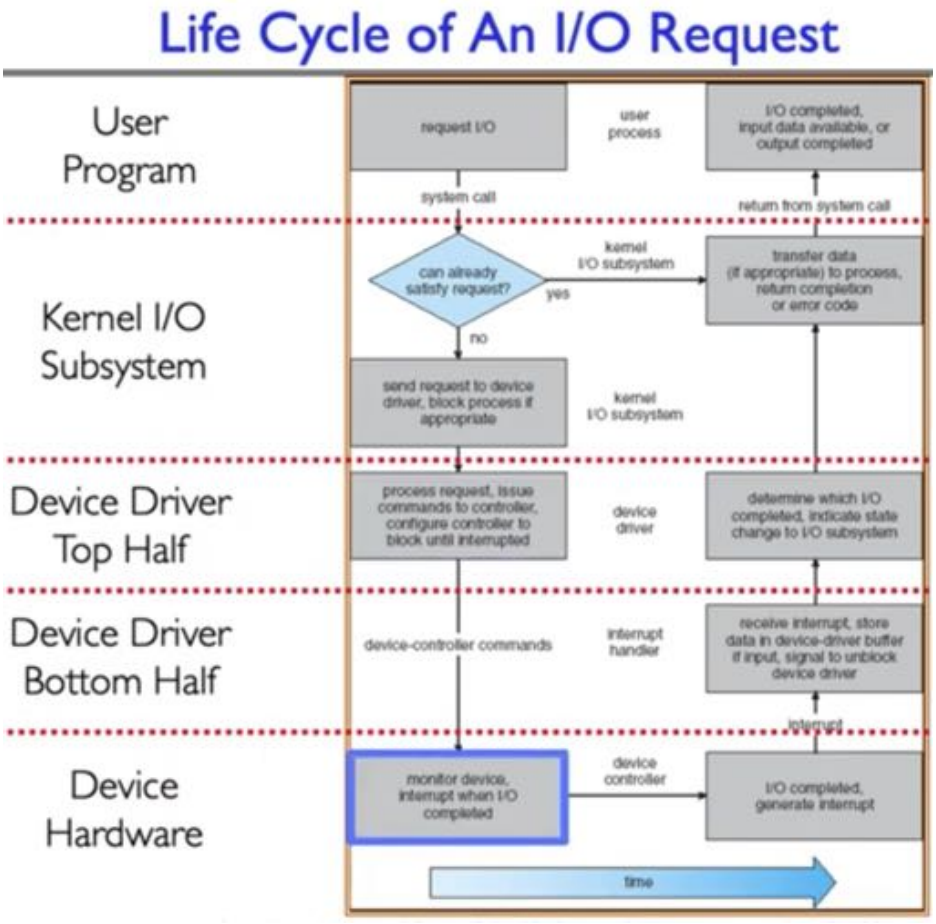
- Associated with particular hardware device or environment
- Registers / Unreigsters itself
- Handler function for each of the file operations

From syscall to driver

1. Ssize_t vsf_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
    a. Read up to count bytes from file from pos into buf Return error or bytes read
2. Make sure allowed to read
3. Check if fiie has read operations
4. Check if can write to buf
5. Check wether we read from a valid range in the file
6. If driver has read function, use i otherwise use sync read
7. Notify parent that file was read
8. Update the number of bytes read
9. Update number of read syscalls by current task

Device Driver

- Device specific code in the kernel that interacts directly with the device hardware
    - Supports a standard, internal interface
    - Special device specific configuration supported with the ioctl() system call
- Device drivers divided into two pieces
    - Top half: accessed i call path from system calls
        - Standard cross-device calls: open() close() read() write()
        - Kernels interface to device driver
    - Bottom half: run as interrupt routine
        - Get s input
        - May wake up sleeping threads



Conclusion

- Monitors: A lock plus one or more condition variables
    - Always acquire lock before accessing shared data
    - Use condition variables to wiat inside critical section
    - 3 Operations
        - Wait
        - Signal
        - Broadcast
    - Monitors represent the logic of the program
- Readers/Writers monitor example
- Kernel thread: Stack + State for independent execution in kernel
- Every user-level thread paried with kernel thread
- Device Driver: Device specific code in kernel that interacts directly with device hardward

Week 6: Lecture 10 Scheduling 1: Concepts and Classic Policies (2/23)

How the kernel decides what runs next on the CPU,
Scheduling: Deciding which threads are given access to resources from moment to moment
Scheduling Assumptions

- Many implicit assumptions for CPU scheduling
- One program per user
- One thread per program
- Programs are indpenent

CPU Bursts

- Execution model: programs alternate between bursts of CPU and I/O
- Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
- Each scheduling decision is about which job to give to the CPU for use in next CPU
- Timeslicing: thread forced to give up CPU before finishing current CPU

Scheduling Policy Goals/Criteria

1. Minimize Response Time
    a. Minimize elapsed time to do an operation
2. Maximize Throughput
    a. Maximize operations per second: minimizing response time will lead to more context switching
    b. Minimize overhead (context switching), efficient use of resources
3. Fairness
    a. Share CPU equitably, better avg response time by making system less fair

Waiting time for P: time before P got scheduled

Average waiting time: average of all processes wait time

Completion time: waiting time + running time

First Come, First Served (FCFS, FIFO)

- One program scheduled until done

Convoy effect: short process stuck behind long process

Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

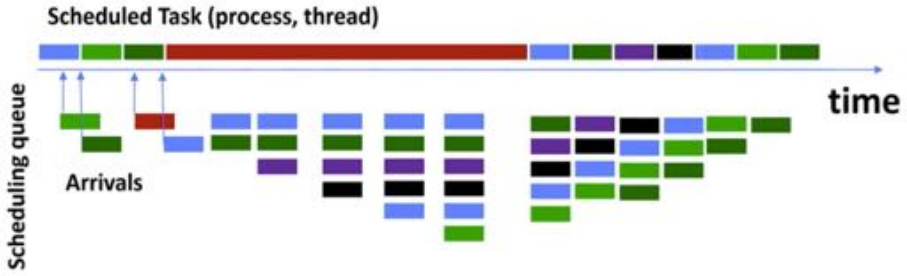
– Suppose processes arrive in the order: P_1 , P_2 , P_3
The Gantt Chart for the schedule is:

– Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
– Average waiting time: (0 + 24 + 27)/3 = 17
– Average Completion time: (24 + 27 + 30)/3 = 27

Convey effect:

- Convoys of small tasks tend to build up when a large one is running
- Sensitive to arrival of processes

- Completion time artificially lengthened



FIFO Pros and Cons

- Pro: simple
- Cons: short jobs get stuck behind long ones

Round Robin (RR) Scheduling

Each process gets a small unit of CPU time

After quantum expires, the process is preempted and added to end of the ready queue

N processes in ready queue and time quantum is q

- Each process gets 1/n of the CPU time
- At most q times
- No process waits more than (n-1) q time units

The magic numbers

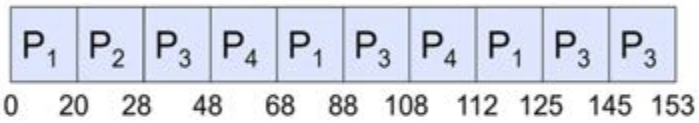
- Q large => FCFS, response time suffers
- Q small => Interleaved, throughput suffers
- Q large with respect to context switch, or else overhead is too high

Example of RR with Time Quantum = 20

Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

– The Gantt chart is:



– Waiting time for P_1= 0 + (68-20)+(112-88)=72
P_2=(20-0)=20
P_3=(28-0)+(88-48)+(125-108) + 0 =85
P_4=(48-0)+(108-68)=88

– Average waiting time = (72+20+85+88)/4=66¼

– Average completion time = (125+28+153+112)/4 = 104½

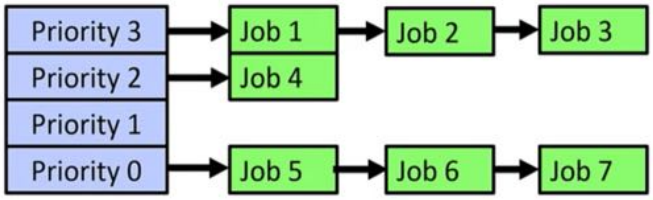
Pros: doesn't matter about arrival time , 1% used for context switching

Cons: Lost of context switching, High completion time
How to implement

- FIFO Queue as in FCFS, preempt job after quantum expires, send to back of the queue aftwards

Cache state must be shared between all jobs with RR, total time for RR longer even for zero cost switch
RR will never be the worst or best case
Handling differences in Importance: Strict Priority Scheduling
Execution Plan

- Always execute highest-priority runable jobs to completion



Starvation
- lower priority jobs don't get to run because higher priority jobs

- Deadlock: Priority Inversion
- Happens when low priority task has lock needed by high-priority taks

Scheduling Fairness

- Tradeoff: fairness gained by hurting avg response time

What if we knew the future

- Shortest Job FIrst (SJF): Run whwatever job has least amount of time, Shortest Time to Completion FIrst (STCF)
- Shortest Remaining Time First (SRTF)
    - Preemptive version of SJF, if job arrives and has a shorter time to completion than remaining currently, Shortest Remininm time to Completion first (SRTCF)
- Short jobs out, big effect on short jobs, small effect on long jobs

Discussion

- Provably optimal minimzing average response time
- SRTF: use it to compare, optimal,
- Con: Hard to predict the future

Predicting the Length of the Next CPU Burst

- Changing policy based on past behavior

Lottery Scheduling

- Give each job some number of lottery tickets
- On each time slice, randomly pick a winning ticket
- On average, CPU time is proportional to number of tickets given to job
- Assigning tickets
    - Short running jobs get more, long running jobs get fewer, every job gets at least one ticket
    - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesse

        - Cons: could choose long jobs

Multi-Level Feedback scheduling

- Multiple queues each with different priority
- Each queue has its own scheduling algorithm

Scheduling Details

- Result approximates SRTF: Short running I/O bound jobs stay near top
- Scheduling must be done between the queues
    - Fixed priority scheduling,
    - Time slice: each queue gets a certain amount of CPU time
- Countermeasure: user action that can foil intent of the OS designers
    - Put in a bunch of meaningless I/O to keep job's priority high

Multi-Core Scheudling

- Helpful to have per core scheduling dta structures
- Affinity Scheduling: once a thread is scheduled on a CPU, OS tries to reschedule in on the same CPU
    - Cache reuse

Mix of DIff types of Apps

- Consider mix of interactive and high throughput apps
- How to best schedule them
- How to recorgnize one from the other
- Is Burst Time useful to decide which application gets CPU time
- Short Bursts -> Interactivity -> High Priroity

How to Evaluate a Scheduling algorithm

- Deterministic modeling
    - Takes a predetermined workload and compute the performance of each algorithm for that workload
- Queueing models
    - Mathematical paproach for handling stochastic workloads
- Implementation/Simulation
    - Build system which allows actual algorithms
    - Most flexible/general

Schedules Threads

- Switch threads: save/restore resgisester
- Switch threads in different processes: switch address space

- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities and scheduling algorithms
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- Lottery Scheduling:
  - Give each thread a priority-dependent number of tokens (short tasks⇒more tokens)

Week 6: Lecture 11 Scheduling 2: Case Study, Real Time, Forward Progr (2/27)

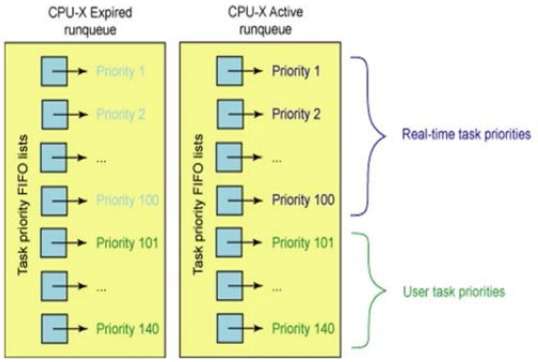
Case Study: Linux

Linux O(n) Scheduler

- At context switch:
  - Scan full list of processes in the ready queue
  - Compute relevant priorities
  - Select the best process to run
- Scalability issues
  - Context switch cost rose as processes increased

Linus: O(1) Scheduler

- Priority-based scheduling with 140 different priorities
  - Real-time kernel tasks assigned priorities 0 - 99 (0 is highest priority)
  - User tasks (interactive/batch) assigned priorities 100 - 139
- Scheduler User tasks
  - Two queues, Active queue and expired queue
    - Processes have not used up their time quanta
  - preemptive scheduler
    - ALL the active run queue finished, switch the next level
    - Prevents starvation
  - Split into timeslice granularity chunks -- round robin through priority



        - Heuristic is complicated when and how you move things between queues
            - Adjusted priority depending on higher sleep avg
            - Interactive Creidt: When tasks slept for a long time, spend when runs for a long time, special dispensation
    - Real Time tasks
        - No dynamic adjustment of priorities
        - SCHED_FIFO: preempts other tasks, no timeslice limit
        - SCHED_RR: preempts normal tasks, RR scheduling amongst tasks of same priority

CFS scheduler (Linux 2.6.22+)

<u>Real-Time Scheduling</u>

Goal: predictability of performance

- RTS, performance guarantees worst case response times for systems
- Need to be predictability in thes ame amount of time, meeting deadline like braking

Hard real-time

- For time-critical safety-oriented systems
- Meet all deadlines, determine in advance if this is possible
- Earliest Deadline FIrst (EDF), Least Laxity First (LLF), Rate monitonic scheduling (RMS), Deadline Monotonic Scheduling (DM)

Soft real-time: for multimedia

- Attempt to meet deadlines with high probability
- Constant Bandwidth Server (CBS)

Workload Characteristics

- Preemptable, independent with arbitrary arrival times
- Tasks have deadlines (D) and known computation times (C)
- RR fails, possible to miss deadlines
- Deadlines matter more than arrive

Earilest Deadline FIrst (EDF)

- Tasks periodic with period P and computatiaon C in each period (Pi, Ci) for each task i
- Preemptive priority based dynamic scheduling
- Assigned a priority based on how close the absolute deadline is
- Scheudler always schedules the active task with the closest absolute deadline
- Feasibility Testing
    - Exists if <= 1

computation time C and deadline D, a

$$\sum_{i=1}^{n} \left(\frac{C_i}{D_i} \right) \leq 1$$

<u>Starvation</u>

Starvation is solvable, Deadlock is not

- Starvation caused by
    - Scheduling policy never runs a particular thread on the CPU

        - Threads wait for each other or are spinning in a way that will never be resolved
    - CPU is never idle when there is work to do

Starvation of scheduling types

- FIFO time is bounded, LIFO easily results in starvation
- RR is bounded
- Priority queue can starve
- SRTF MLFQ starves long jobs in favor of short ones

Priority Inversion

- Thread with lower priority is blocking thread with higher priority, resource needed held by lower priority
- Priority Donation/ Inheritance: give the lower priority higher priority if it is blocking
- Pathfinder rover: had priority inversion and needed to turn on priority donation and led to random restarts

<u>Does priority always starve lower priority</u>

Proportional share scheduling

- Share proportional priority,
- Give each job a share of the CPU according to its priority
- Low -priority jobs get to run less often
- But all jobs can at least make progress

Lottery Scheduling

- Given set of jobs, provide with share of a resource

$$\frac{big\#W}{N_i}$$

Stride Scheduling

- Deterministic proportional fiar sharing
- Stride of each job is
- Larger your share of tickets, the smaller your stride
- Low stride jobs run more often,
    - Job with twice the tickets gets to run twice as often
- Each job has pass counter, scheduler: pick job with lowest pass, runs it, addits stride to its pass,

Linux Completely Fiar Scheduler

- Goal: Each process gets an equal share of CPU
- N threads simultaneously execute 1/n on the CPU
- Track CPU time per thread
- Chooses thread with less time than 1/n
- Use red black tree to add remove threads O(logN)
- Boost when sleeping

Responsiveness/Starvation Freedom

- Low response time and starvation freedom
- Constraint 1: Target latency
    - Quanta = target_latency /n

- Goal: Throughput
    - Avoid excessive overhead
    - Limits when too much interactivity

Prioirty in Unix : Being Nice

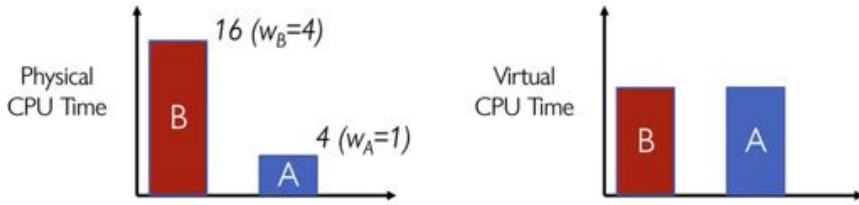
- Provided priority to enforce desired usage policies
- Nice values range from -20 to 19
- Negative values are "not nice"
- If you wanted friends get more time, nice up your job
- CFS: change rate of CPU cycles given to thread relative to priority

Proporitonal shares

- Key Idea: Assign a weight wi to each process I to compute the switching quanta Qi
- Basic equal share: Qi = Target latency 1/N
- Weighted share: Qi = (wi / sumpwp) target latency
- Reuse nice value to reflect share rather than priority
- Scale weights exponentially

Proprotional shares

- Track a thread's virtual runtime rather than its true physical runtime
    - Higher weight: virtual runtime increases more slowly
    - Lower weight
    - Scheduler's Decisions based on virtual CPU TIme
    - Sorted on virtual runtime variable



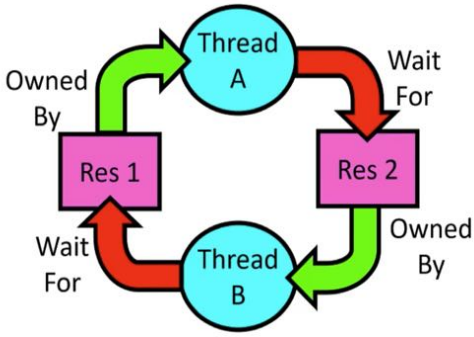
Each scheduling policy has their own pros and cons

Summary

- RR, Shortest Job first
- Realtime scheudlers (EDF)
- Lottery scheduling
- Linux CFS Fair fraction of CPU
- Stride scheduling
- Lottery scheduling

I Care About:	Then Choose:
CPU Throughput	FCFS
Avg. Response Time	SRTF Approximation
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Linux CFS
Fairness – Wait Time to Get CPU	Round Robin
Meeting Deadlines	EDF
Favoring Important Tasks	Priority

Week 7: Lecture 12 Scheduling 3: Deadlocks (3/2)

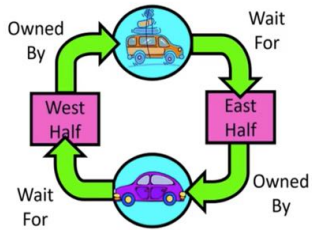


Deadlock: A Deadly type of Starvation

Cyclic waiting for resources

- Deadlock always leads to starvation
- Starvation does not mean deadlock

Bridge crossing Example

- 

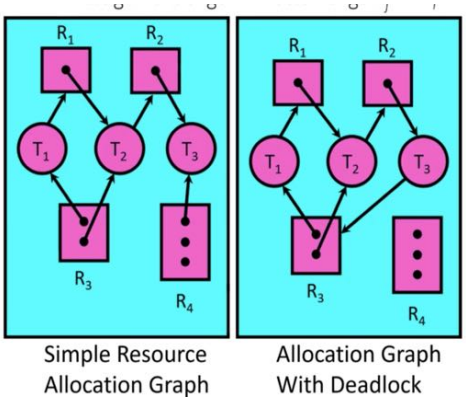
- Can solve with "external" intervention: killing thread,
- Deadlock with locks, Thread A owns Lock X, waits for Lock y, lock y owned by Thread d, wait for lock x

Lock pattern exhibits non-deterministic deadlock

- Could give each thread enough resources
- Make everyone "give up" after a while
- Make everyone do it atomically

4 requirements for occurrence of Deadlock

1. Mutual exclusion and bounded resources
    a. Only one thread at a time can use a resource
2. Hold and wait
    a. Thread holding at least oen resources is waiting while additional resources held by other threads
3. No preemption
    a. Released only voluntarily by thread holding resource, after thread is finished
4. Circular wait
    a. Exists T1 .. Tn waiting where T1 depends on T2....



Simple Resource Allocation Graph | Allocation Graph With Deadlock

Detecting Deadlock

- Resource Allocation Graph
- Uses thread with request() use() release()

Deadlock Detection Algorithm

- X represent an m-ary vector of non-negative integers
- FreeResources: Current free resources each type
- Request: current requests from thread X

- Alloc: Current resources held by thread X

See if tasks can eventually terminate on their own

```
[Avail] = [FreeResources]
Add all threads to UNFINISHED
do {
   done = true
   Foreach thread in UNFINISHED {
      if ([Request_node] <= [Avail]) {
         remove thread from UNFINISHED
         [Avail] = [Avail] + [Alloc_node]
         done = false
      }
   }
} until(done)
```

-

How should a system deal with deadlock

1. Deadlock prevention: write your code in a way that isn't prone to deadlock
2. Deadlock recovery: let deadlock happen, and then figure out how to recover from it
3. Deadlock avoidance: dynamically delay resource requests so deadlock doesn't happen
4. Deadlock denial: ignore the possibility of deadlock
    a. Make sure system isn't involved in deadlock, applications can be

Deadlock prevention

1. Mutla exclusion and bounded resources
    - Provide sufficient resources
    - Virtual memory allows unlimited
2. Hold and wait
    - Abort request or acquire requests atomically
    - Can get more expensive if acquire unnecessary resources
3. No preemption
    - Tell them to fail if have been waiting too long
    - FOrce thread to give up resource, database aborts: all actions are undone, and transaction must be retried
4. Circular wait
    - Order resources and usage in the same order
    - Force threads to request resources in a particular order preventing any cyclic use of resources
    - Very common

Thread A:	Thread B:
`x.Acquire();`	`y.Acquire();`
`y.Acquire();`	`x.Acquire();`
`…`	`…`
`y.Release();`	`x.Release();`
`x.Release();`	`y.Release();`

Techniques of Deadlock Avoidance

- Safe State
- System can delay resource acquisition to prevent deadlock

Thread A:	Thread B:
`x.Acquire();`	`x.Acquire();`
`y.Acquire();`	`y.Acquire();`
`…`	`…`
`y.Release();`	`x.Release();`
`x.Release();`	`y.Release();`

- Unsafe state
    - No deadlock yet
    - But threads can request resouces in a patter that unavoidably leads to deadlock
- Deadlock state
    - There exists a deadlock in the system
    - Also unsafe

Deadlock avoidance: prevent system from reaching an unsafe state

Banker's Aglorithm for Avoiding Deadlock

- Towrad right idea: state maximum resource needs in advance
- Allow pawrticular thread to proceed if
    - Available resources - #requested >= max

Banker's algorithm

- Allocate resources dynamically
    - Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - Requests all remaining resources, finishing then check if safe
    - Run algorithm on every allocation

Week 7: Lecture 13 Memory 1: Address Translation and Virtual Memory (3/4)

Virtualizing Resources

Address Space & Dual mode operation /Protection

- 2^k "things"
- Set of accessible addresses and the state associated with them

Important Aspects of Memory Multiplexing

- Protection
    - Prevent access to private memory of other processes
    - Different pages of memory can be given special behavior
    - Kernel data protected from User programs
    - Programs protected from themselves
- Translation
    - Ability to translate accesses from one address space (virtual) to a different one
    - When translation exists, processor uses virtual addresses
- Controlled overlap
    - Separate state of thread should not collide in physical memory
    - Would like the ability to overlap when desired

Interposing On Process Behavior

- OS itnerposes on process' I/O operations
- OS interposes on process' CPU usage
- Questions: How can the OS interpose on process' memory accesses

Binding of Instructions and Data to Memory

Addresses can be bound to final values anywhere in this path

- Depends on hardware

Uniprogramming (no Translation or Portection)

- Application always runs at same place in physical memory since only one application at a time
- Application can access any physical memory

Multiprogramming without Tranlsation or Proection

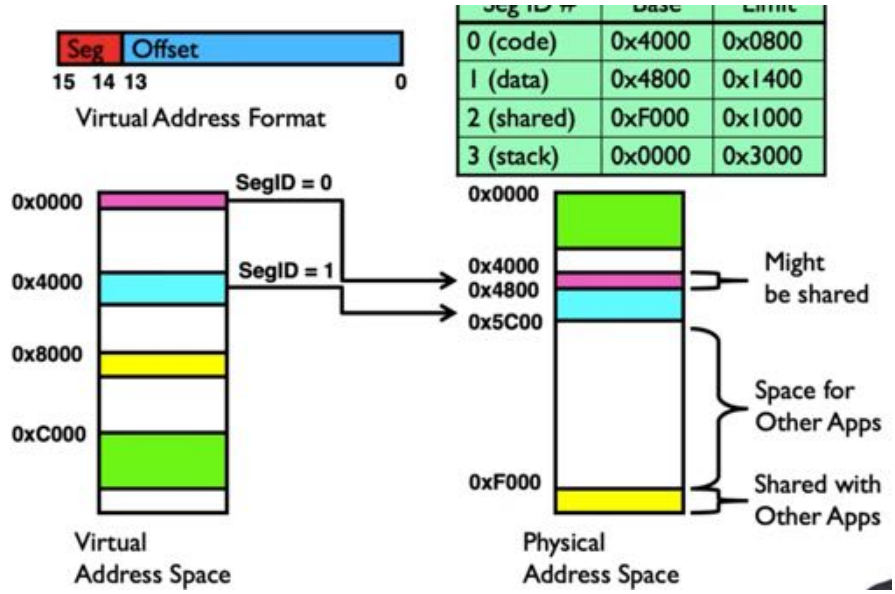
- Use Loader/Linker: Adjust addresses while progrram loaded into memory
- No protection

Multiprogrammming with Protection

- Two additional registers: base and bound
- Issues:
  - Fragmentation problem over time
  - Not every process is same size -> memory becomes pfragmented over time

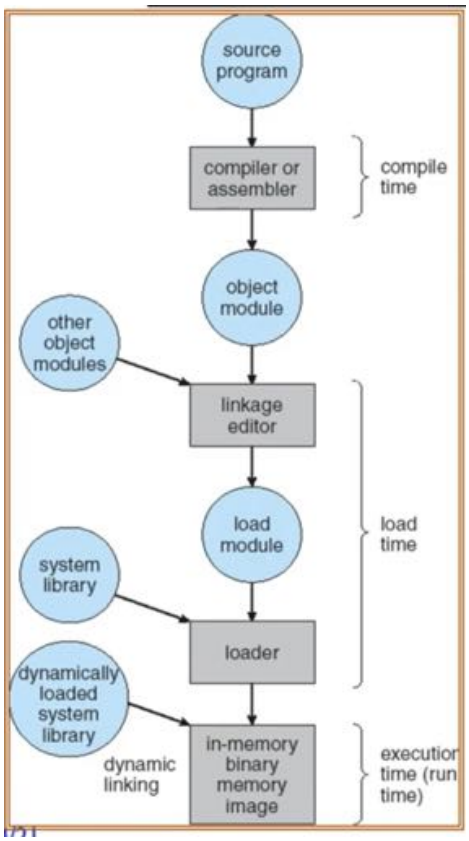
More flexible Segmentataion

- Logical view: multiple separate segments
- Segment map resides in processor
- Chunks of physical memory as entries
-



Observations about Sementatiaons

- Translation on every instruction fetch, load, or store
- Virtual address space has holes
- OK to address outside valid range
- Stock and heap allowed to grow



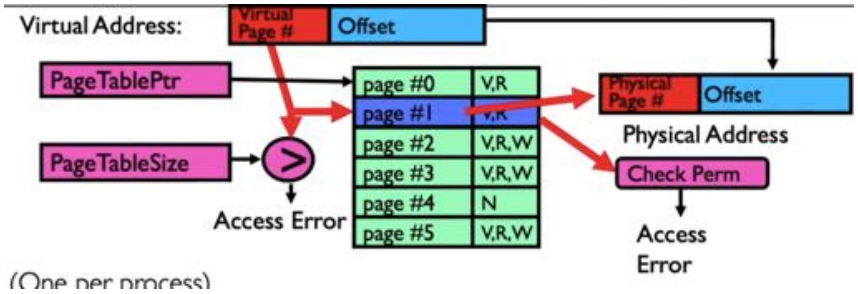
- Need protection mode in segment table
- Segment table stored in CPU

What if not all segments fit?

- Swapping, some or all of previous process is moved to disk
- Some way to keep only active portions of a process in memory at any one time

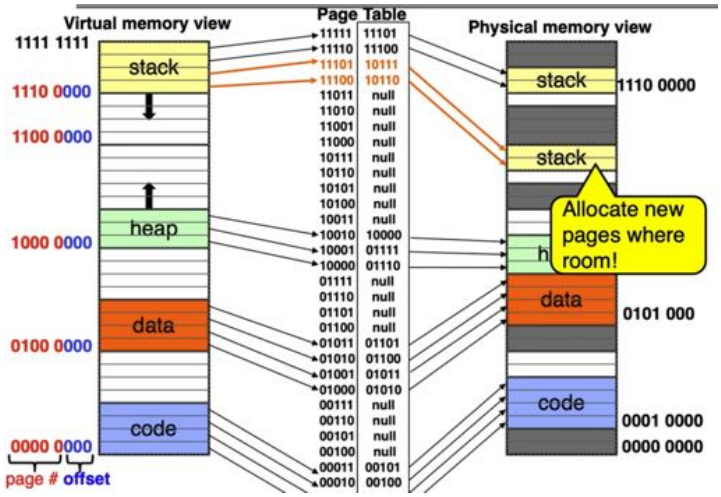
Paging; Physical Memory in Fixed Size Chunks

- Solution to fragmentation from segments
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
- Pages are very small



Simple paging

- Page table (one per process)
- Virtual address mapping
- Offset from virtual address copied to physical address
- Vitual page # is remaining
- Check bounds and permissions



Kernel region of every process has the same page table tnreis

- Process cannot access it at user level
- Different processes running smae binary
- Different processes running same binary
- Suer level system libraries (execute only)

Allows memory to be assigned anywhere and don't have to worry about reallocating memory

Represent large page tables entries or else will take too much space

Page Table Discussion

- Context switch switches
  - Page table pointer and limit
- Protection
  - Translation (per process) and dual mode
  - Can't let process alter its own page table
- Analysis
  - Pros
    - Simpel memory allocation
    - Easy to share
  - Con: what if address space is space
- Simpel Page table is way too big

Summary

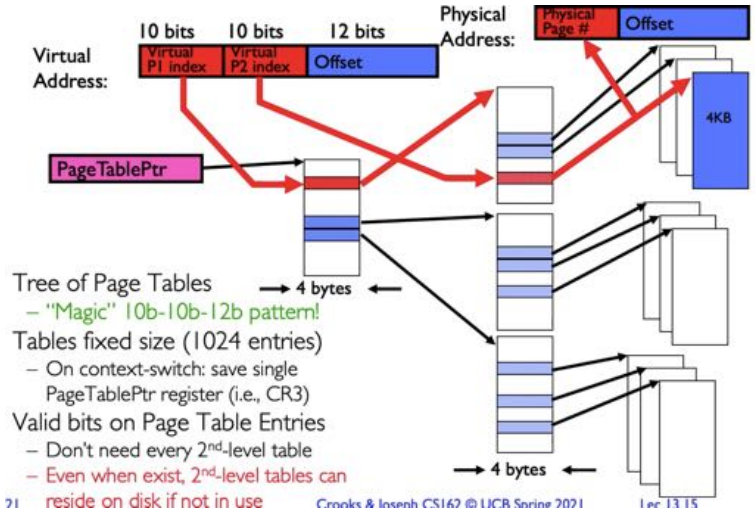
- Segment Mapping
    - Segment registers within processor
    - Segment ID associated with each access
    - fEach segment contains base and limit info
- Page tables
    - Memory divided into fixed-sized chunks of memory
    - Virtual page number from virtual address mapping
    - Offset of virtual address same as physical address
    - Large page tables can be placed into virtual memory

Week 8: Lecture 14 Memory 2: Virtual Memory, Caching and TLBs (3/9)

How to Structure a Page Table

- Page Table is a map from VPN to PPN
- Simple page table corresponds to a very large lookup table

Two level page table

- Tree of Page tables
    - Magic 10b-10b-12b pattern
    - 

    - Tope level page table called a "Page Directory"
        - With "Page Directory Entries"
    - CR3 provides physical address of the page directory

Page Table Entry (PTE)

- Pointer to next level page table or to actual page
- Permission bits: valid read-only, read-write, write only
- Present (valid), WRiteable, User accessible, Page cache, Accessed, Dirty, Page size

How to use PTE

- Demand Paging
    - Keep only active pages in memory
- Copy on Write

        - Copy page table entries, point to read only physical space
- Zero Fill on Demand
    - New data pages must carry no information
    - Creates zeroed pages in background

Multi-level Translation: segments + pages

- Lowest level page table -> memory still allocated with bitmap
- Higher levels often segmented
- Pros:
    - Only need to allocate as many page table entries as we need for applicaigton
    - Easy memory allocation
    - Easy sharing
- Cons:
    - One pointer per page
    - Page tables needs to be contiguous
    - Two lookups per reference

Larger page sizes supported as well, memory is now cheap

- Issue is internal fragmentation

Alternative: Inverted Page Table

- Use a hash table for virtual page to physical page map
- Cons:
    - Complexity of managing hash chains: often in hardware
    - Poor cache locality of page table

	Advantages	Disadvantages
Simple Segmentation	Fast context switching (segment map maintained by CPU)	External fragmentation
Paging (Single-Level)	No external fragmentation Fast and easy allocation	Large table size (~ virtual memory) Internal fragmentation
Paged Segmentation	Table size ~ # of pages in virtual memory Fast and easy allocation	Multiple memory references per page access
Multi-Level Paging		
Inverted Page Table	Table size ~ # of pages in physical memory	Hash function more complex No cache locality of page table

<u>How is Translation Accomplished?</u>

Memory Management Unit

- Translates virtual address to physical address
    - Instruction fetch, load, store,
- 1 level page table
    - Read PTE from memory
- 2 level page table
    - Read and check first level

        - Read, check, and update PTE
    - N-level page table

MMU does Page Table Tree Traversal to translate each address

MMU

- Reads multiple levels of page table entires to get physical frame or FAULT
- Through the caches to the memory
- Read/write the physical location

Cache

- Repo for copies that can be accessed more quickly than the original
- Make frequent case fast
- Average Memory Access Time (AMAT) = (Hit rate * hit Time) + (miss rate * miss time)

Translation lookaside buffer (TLB)

- Memory hierarchy
    - Take advantage of the principle of locality
- Record recent virtual page # to Physical Frame # translation

<u>What kind of Cache for TLB</u>

Sources of Cache Misses

- Compulsory
    - First access to a block
- Capacity
    - Cannot contain all blocks access by the program
    - Solution: increase cache size
- Conflict (collision)
    - Multiple memory locations mapped to the same cache location
    - Solution: increase cache size
    - Increase Associaitivie
- Coherence:
    - Other process updates memory

Block found in a cache

- Block is minimum quanum of caching
    - Data select field used to select data within block
- Index Used to lookup candidates in cache
- Tag used to identify actual copy

Direct Mapped Cache

- Can only be mapped to specific

Fully Associative

- Can go anywhere in set

Which block should be replaced on a miss?

- Easy for Direct Mapped: only one possibility
- Set Associative or Fully associative

    - Random
    - Least Recently Used (LRU)

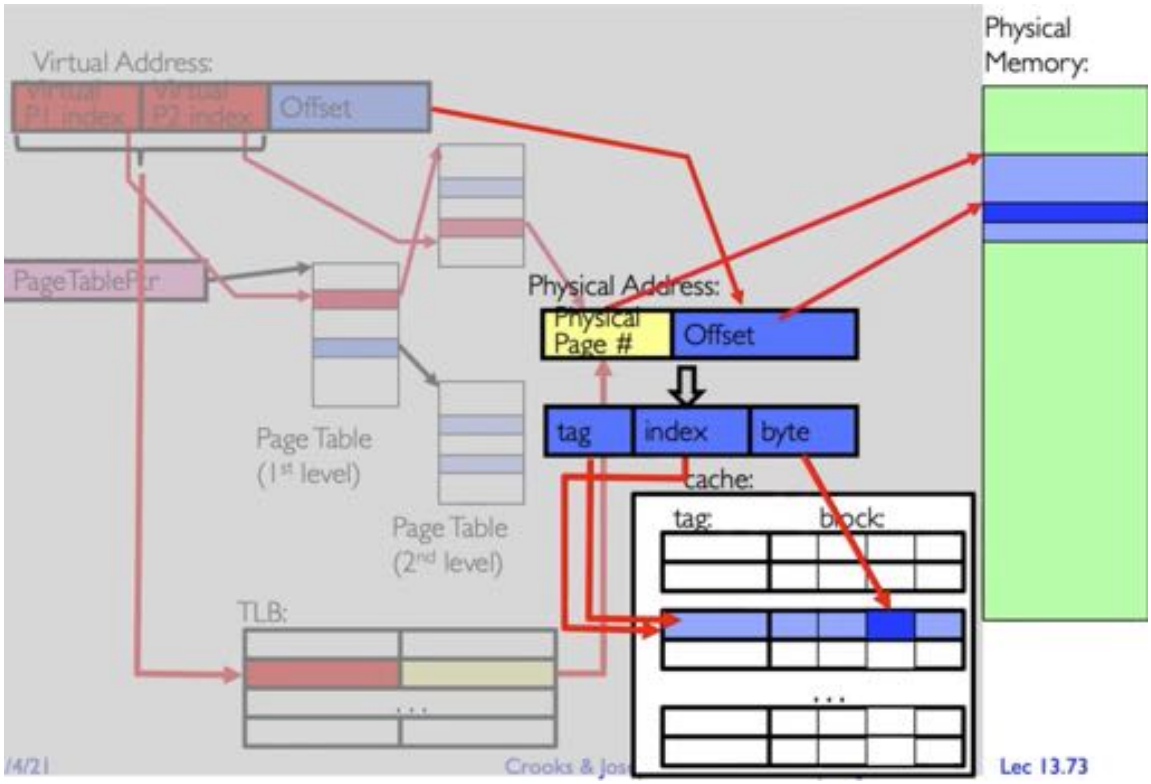
What happens on write

- Write through: info written to both the block in cache and to the memory
- Write back: written only to the block in the cache, block clean or dirty, written to main memory when it is replaced
    - Pro: read misses cannot result in writes
    - Con: Processor
        - More complex

TLB organization

- Thrashing: continuous conflicts between accesses
- What if use low order bits of page as index into TLB
- Usually small
- Small TLBs as fully asasociative cache
- What if fully associative is too slow
    - TLB Slice: Put direct mapped cache in front
- TLB Stages

What happens on Context Switch?



Summary

- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical pa number
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory
- Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- Inverted Page Table
  - Use of hash-table to hold translation entries
  - Size of page table ~ size of physical memory rather than size of virtual memory

- Principle of Locality:
- Program likely to access a relatively small portion of the address space
  - Temporal Locality: Locality in Time
  - Spatial locality: Locality in Space
- 3 major cache misses
  - Compulsory, conflict, capacity misses, coherence misses
  - Direct mapped, set associative, fully associative

TLB Small number of PTE and optional process IDs

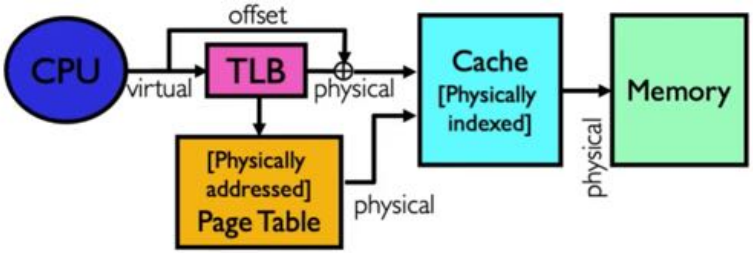
- Fully associative
- On TLB miss, page table must be traversed and if located invalid
- TLB logically in front of cache

Week 8: Lecture 15 Memory 3: Caching and TLBs, Demand Paging (3/11)

Physically-indexed vs virtually indexed caches

Physically-Indexed Caches (Typical)

- Address handed to cache after translation
- Page table holds physical addresses
- Pros:
  - Every piece of data has single place in cache
  - Cache can stay unchanged on context switch
- Con:
  - TLB is in critical path of CPU



Virtually Indexed Caches

- Address handed to cache before translation
- Page Table holds virtual addresses
- Pro
    - TLB not in critical path of lookup, faster
- Cons:
    - Same data can be mapped in multiple places of cache

Reducing translation time for physically-indexed caches

- TLB lookup is in serial with cache lookup
- Speed of TLB can impact speed of access to cache
- Overlap TLB lookup with acache access

Page Fault

- May occur on instruction fetch or adat access

Demand Paging

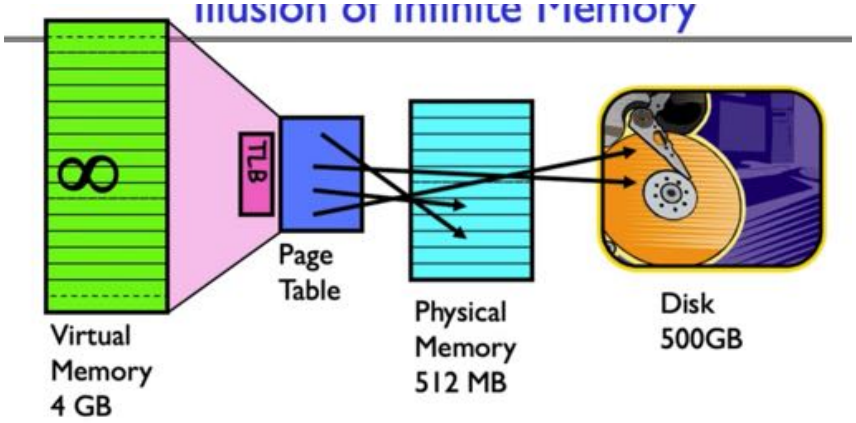
- Programs spend 90% of their time in 10% of their code
- Use main memory as cache for disk, secondary storage

Demand Paging as Caching

- Fully associative
- FIrst check TLB, then page table traversal
- Write back

Illusion of Infinite Memory

- Combined memory of running processes much larger than physical memory



Transparent Level of Indirection (page table)

- Supports flexible placemnt of physical data
- Variable location of data transparent to user program

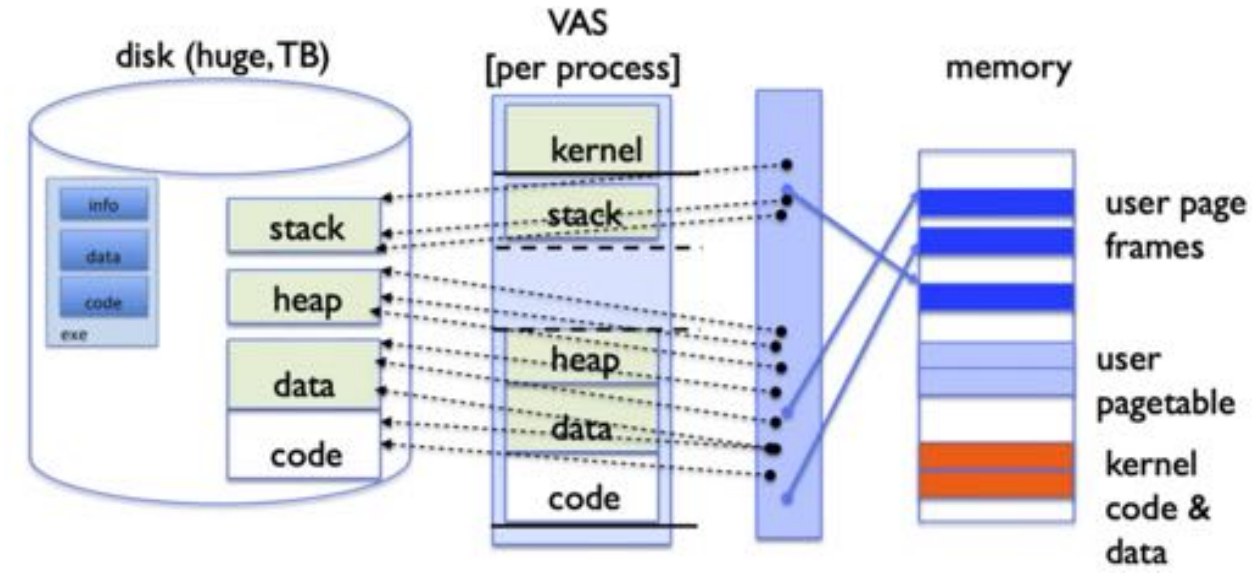
PTE makes demand paging implementable

- Valid -> Page in memory, PTE points at physical page
- Not Valid -> page not in memory, use info in PTE to find it on disk when necessary

<u>Executable</u>

Brings from DIsk into Memory and maps Virtual Address space to memory

For all other pages, OS must record where to find them on disk

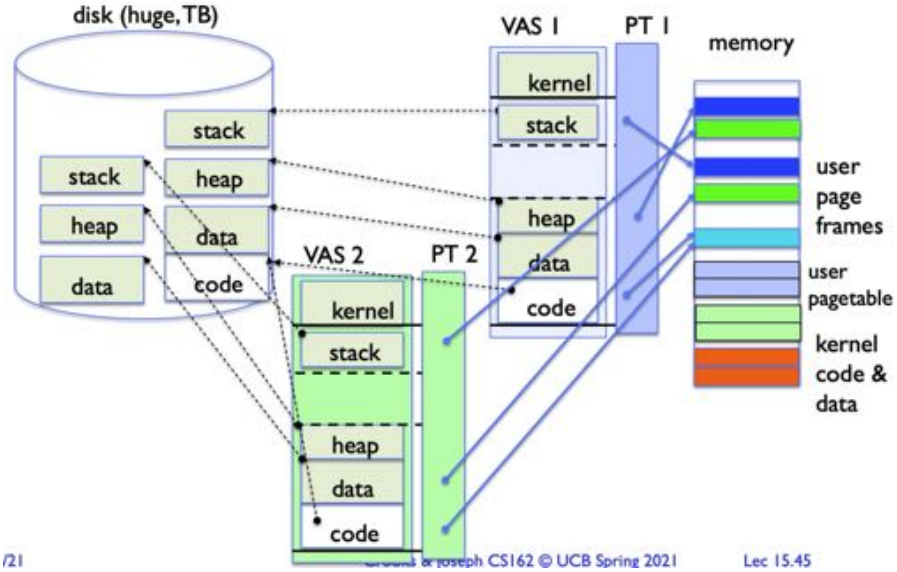


What Data Structure Maps Non-Resident Pages to Disk?

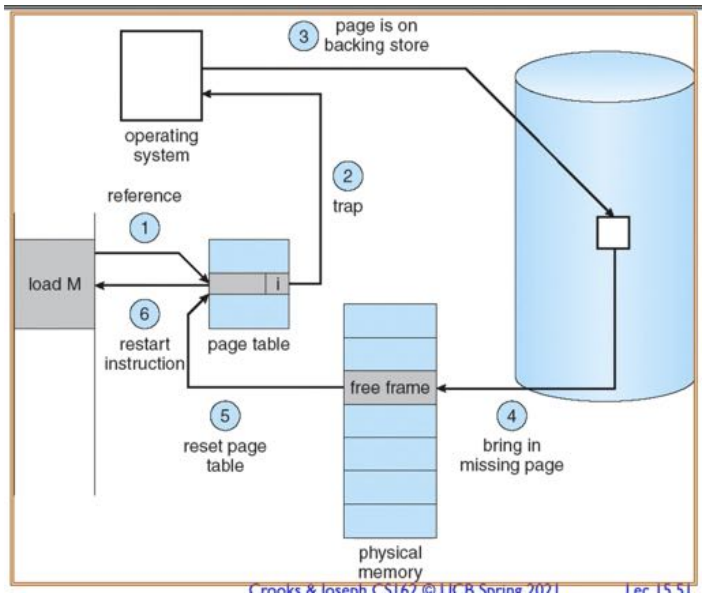
- Find block with the rest of the pages on disk
- FindBlock method to find -> disk_block
- May map code segment directly to on-disk image
    - Saves copy of code to swap file, locks the code
- May share code segment with multiple instances of the program

Uses of Vritual Memory and “demand Paging”

- Extend the stack, heap
- Process Fork, Exec, MMAP



Steps in Handling a Page Fault

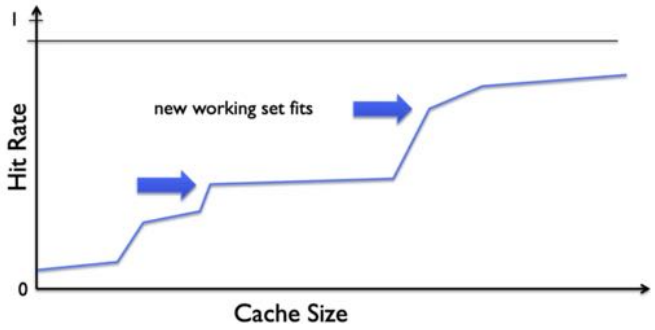


During a page fault, where does the OS get a free frame?

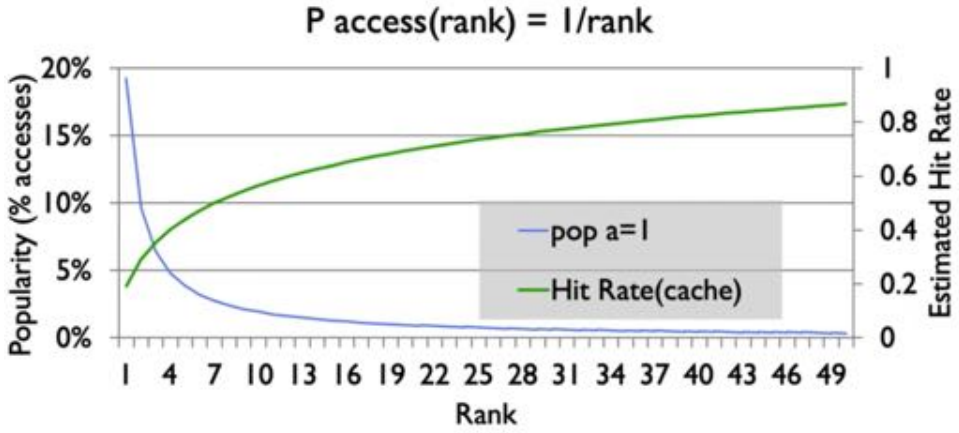
- Keep a free list
- Unix runs a "reaper" if memory gets too full
- Evict a dirty page first

Organize mechanism

- Cache size increases as hit rate increases



Zipf: Model of Locality



Effective Access Time

- EAT = Hit Rate * Hit Time + Miss Rate * Miss Time
- = Hit Time + MIss Rate * Miss Penalty

Page Replacement Policies

- Replacement is an issue with any cache
- Particularly important with pages

1. FIFO (First In, First Out)
    a. Throw out oldest page
    b. Bad: throws out heavily used pages instead of infrequently used
2. Random
    a. Pick random page for every replacement

    b. Typical solution for TLBs,
    c. Bad: Unpredictable
3. Min
    a. Replace page that won't be used for the longest time
    b. Great but can't know future
4. Least Recently Used (LRU)

Summary

- TLB:small number of PTEs and optional process IDs
- Demand Paging: treating DRAM as cache on disk
- Replacemnt policies

Week 9: Lecture 16 Memory 4: Demand Paging Policies (3/15)

Demand Paging Mechanisms

- Valid -> Page in memory, PTE points at physical page
- Not Valid -> Page not in memory, use info in PTE to find it on disk when necessary

Factors Lead to Misses in Page Cache

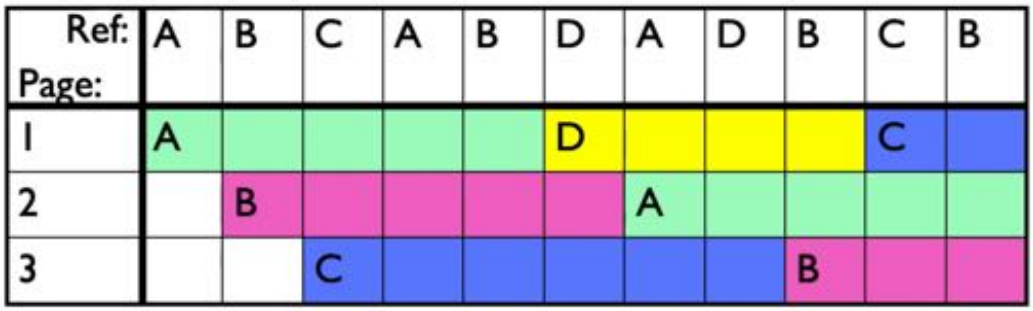
- Policy Misses
    - Caused when pages were in memory, but kicked out due to policy

How to implement Least Recently Used (LRU)

- Each use, remove page and place at head

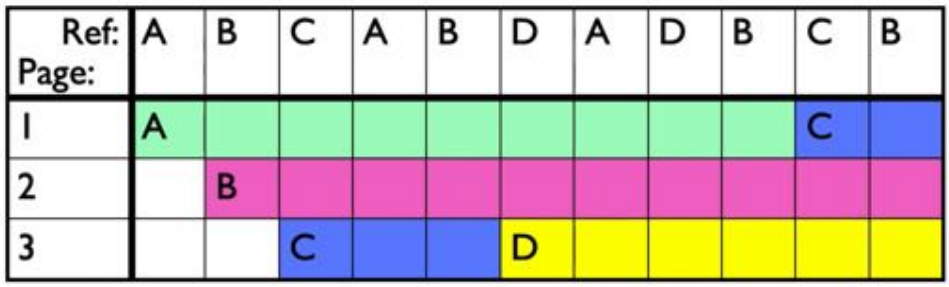
FIFO

- 7 Faults



MIN/LRU

- 5 faults
- Look for best frame to replace



Cases where MIN can do better than LRU, not necessarily optimal

Approximating LRU: Clock Algorithm

- Arrange physical pages in circle with a single clock hand
- Approximate LRU
- Replace an old page, not the oldest page

Hardware "use" bit per physical page

- Hardware sets use bit on each reference
- On page fault
    - Advance clock hand
        - Check use bit
            - 1: Used recently; clear and leave alone
            - 0: selected candidate for replacement

Nth Chance version of Clock Algorithm

- Nth chance algorithm: Give page N chances
- OS keeps counter per page: # sweeps
- ON page fault, OS checks use bit
    - 1 -> Clear use and also clear counter
    - 0 -> increment counter; if count == N, replace page

Meaning of PTE Bits

- Want to know "Present" bit
- Emulate modified bit by "read-only" bit

Clock Algoirthm Variations

- Mark all pages as read only W-> 0
- Writes cause page fault, set modified bit -> 1 marks as writable W-> 1
- Whenever page written back to disk clear modified bit -> 0, mark read only
- Do we need "use" bit
    - No emulate it similar to above,
    - Clear emulated "use" bits -> 0
    - Read or write to invalid page traps to OS to tell use page has been used
    - OS sets "use" bit -> 1 in software to indicate that page has been "used"
        - 1) If read, mark page as read only W->0
        - If write, set modified bit -> 1, marke page as writable W-> 1
        - When clock hand passes, reset emulated "use" bit -> 0

Second-Chance List Algoirthm (VAX/VMS)

- Split memory in two: Active list (RW), SC list (Invalid)
- Pages in active list at full speed, move overflow page to front of SC list, cost is trap
- How many pages on second chance list
    - Pro: few disk accesses
    - Con: increased overhead trapping to OS
- Page translation can adapt to any kind of access the program makes

Free List

- Single clock hand: advances as needed to keep freelist full
- Keep set of free pages ready for use in demand paging
- If page needed before reused, just return to active set
- Advantage: faster for page fault
    - Can always use page immediately on fault

Coremap

- How to know which PTEs to invalidate
    - Hard in the presence of shared pages
- Reverse mapping mechanism must be very fast
- Link together memory region descriptors

Allocation of Page Frames

- Each process needs minimum number of pages
- All processes are loaded into memory can make forward progress

Fixed/Priority ALlocation

- Equal allocation (Fixed Scheme)
    - Every process gets same amount of memory
- Proportional allocation (Fixed)
    - Allocate according to the size of process

s_i = size of process p_i and $S = \sum s_i$

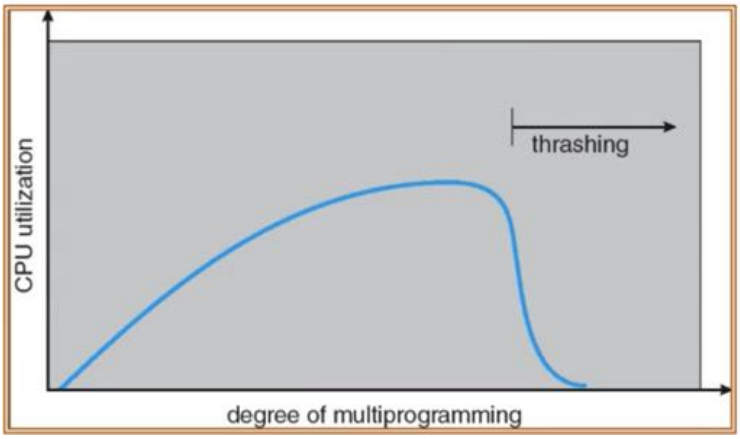
m = total number of physical frames in the system

a_i = (allocation for p_i) $= \frac{s_i}{S} \times m$

- Priority Allocation
    - Proportional scheme using priorities rather than size

Dynamically changing the number of pages/application

- Establish acceptable page-fault rate, upper bound lower bound
- Without enough pages, thrashing can happen: page fault rate is very high, process is busy swapping pages in and out with little or no actual progress



Locality in a Memory-Reference Patter

- Program Memory access patterns have temporal
- Working set model
    - WS = total set of pages referenced in most recent delta
    - Delta = Working set window
    - If delta too small will not encompass Δ
    - Δtoo larger will compans several localities
    - Total demand frames $\sum |WS_i|$

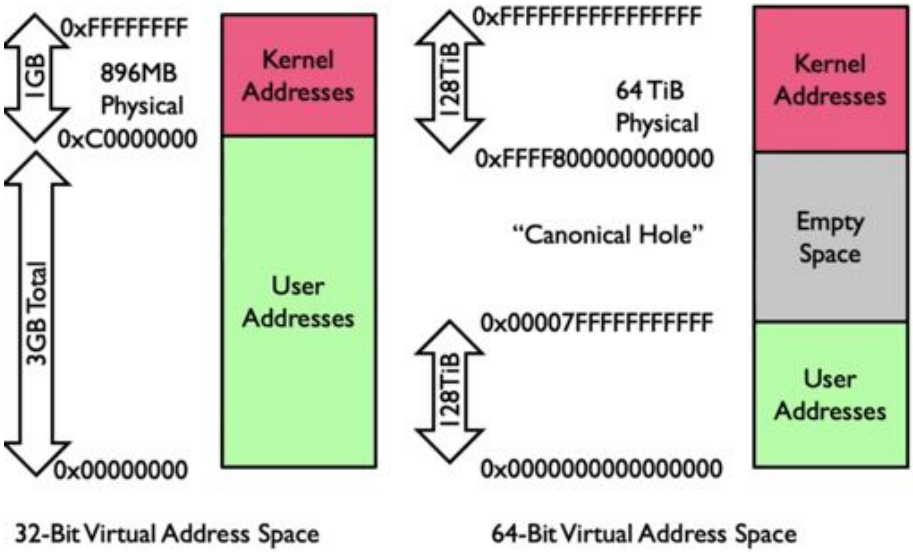
Reduce Compulsory Misses

- Clustering
    - Bring in multiple pages "around the faulting pages"
- Working Set tracking
    - Use algorithm to try to track working set of application

<u>Linux Memory Detals</u>

Memory Zones: physical memory categories

- ZONE_DMA: < 16 MB memory, DMA-able on ISA bus
- ZONE_NORMAL: 16 MB -> 896 MB
- ZONE_HIGHMEM: Everything else (> 896 MB)



Post Meltdown Memory Map

- Read value of specific kernel address can figure out contents by figuring out what is cached and what is not

Summary

- Replacment policies
    - FIFO, MIN, LRU
- CLock Algorithm: Approximation to LRU
    - Arrange all pages in circular list

    - Sweep through them, marking as not “in sue”
    - If page not “in sue” for one pass, then can replace
- Nth chance clock algorithm: give pages multiple passes of clock hand before replacing
- Second-CHance List algorith: divide into two groups, true LRU and managed on page faults
- Working Set: Set of pages touched by pcprocess recently

Week 11: Lecture 17 General I/O, Storage Devices (3/30)

CPU

- Input/Output is the mechanism through which the computer communicates with the outside world

Standard Interfaces to Devices

1. Block Devices: disk drives, tape drives, DVD-ROM
    a. Access blocks of data,
    b. open(), read(), write(), and seek()
    c. Raw I/O or file-system access
2. Character Devices
    a. keyboards , mice, serial ports
    b. Single characters at a time
    c. get(), put()
3. Network Devices
    a. Ethernet, wireless, bluetooth
    b. Differenent enough from blocks character to have own interface
    c. Unix and Windowsn have socket interface
    d. Pipes, FIFOs, streams, queues

IO Subsystem: abstraction,

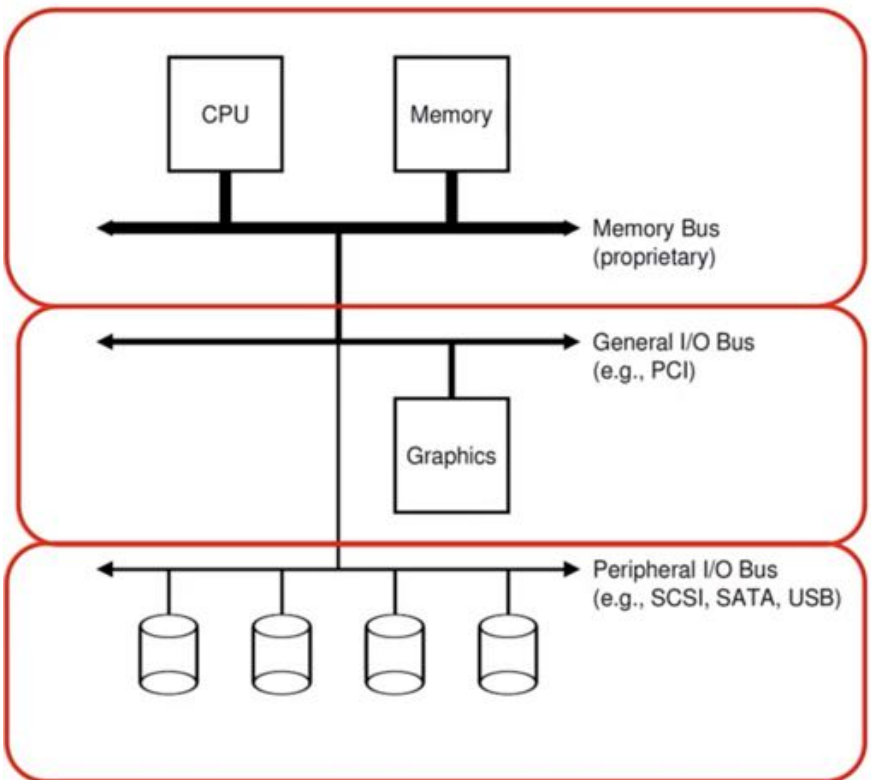
- IO abstracts away
- Want to write code that controls devices with a standard interface

Requirements of I/O layer

- How can we standardize the interfaces to these devices
- Devices unreliable: media failures and transmission errors
- Devices unpredictable and/or slow

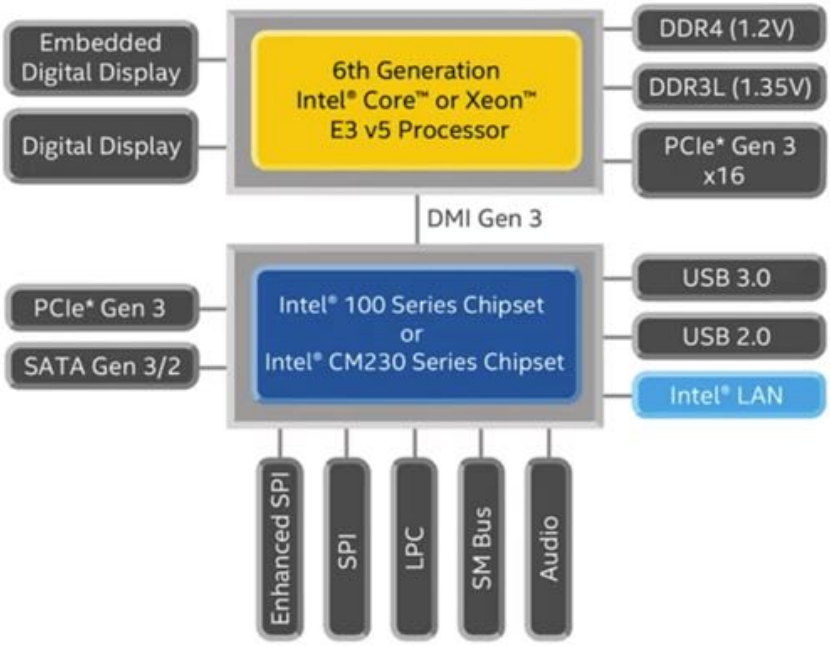
Simplified IO architecture

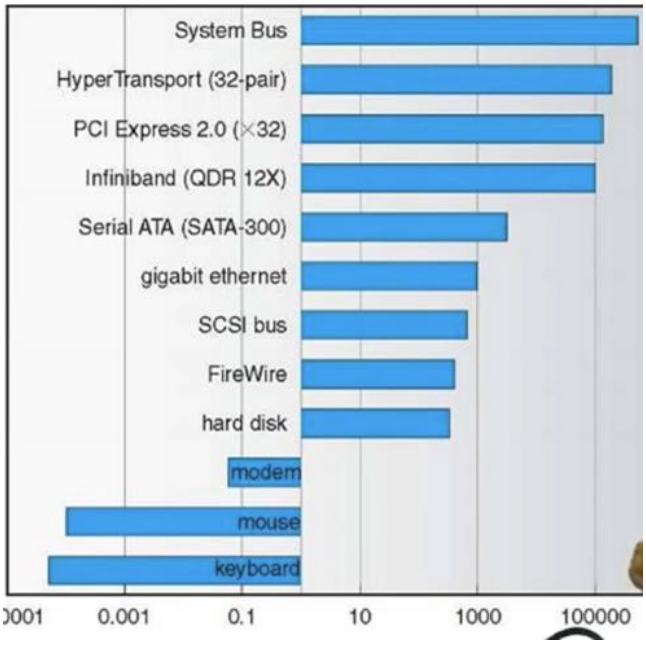
-



Intel Sky Lake I/O
Platform Controller Hub

- Connected to processor with proprietary hub
- Different devices have different speeds





Bus

- Common set of wires for communication among hardware devices and protocols for carrying out data transfer transactions
- Parts
    1. Data bus
    2. Address bus: where sent
    3. Control bus: commands
- Protocol: initiator requests, arbitration to grant,

Why a bus?

- Connect n devices over single set of wries, connections and protocols
- Downside:
    - One transaction at a time
    - Rest must wait
        - Limited to speed of slowest device
        - Peed is set to that of the slowest device

PCI Express "Bus"

- Graphics cards
- No longer a parallel bus
- Collection of fast serial channels or langes
- Devices can use as many as they need to achieve a desired bandwidth
- SLow devices don't have to share with fast ones
- Succcesses of device abstraction was able to migrate from PCI to PCI express without any changes in code

Process talks to devices

- User just sees list of files
- Hardware interface device presents to OS
- Internals (what needed to implement the abstraction)

Hardware interface

- 3 registers:
    1. Status: read by OS, status of device
    2. Command: written by OS, command to device
    3. Data: read/write data

Port-Mapped I/O

- Privileged in/out instructions
    - 0x21.AL
- Memory-mapped I/O: load/stores instructions
    - Appear in physical memory instructions

A simple protocol

- Status == BUSY
    - Wait, polling
- Write data to data registers
- Write command to command register
    - Starts the device and execute the command
- While status busy
    - Wait until device done with request
- CPU is responsible for moving data

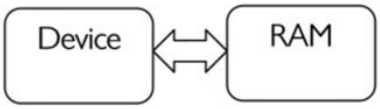
Polling vs Interrupt-driven IO

- Allows CPU to process another taks. Will get notified when task is done
- Interrupt handler will read data & error code
- Hybrid: Poll for short amount of time and schedules interrupt after a few
- Interrupt coleasing: batch interrupt together and do them
- From programmed IO to direct memory access
- CPU issues read request

Direct Memory Access (DMA)

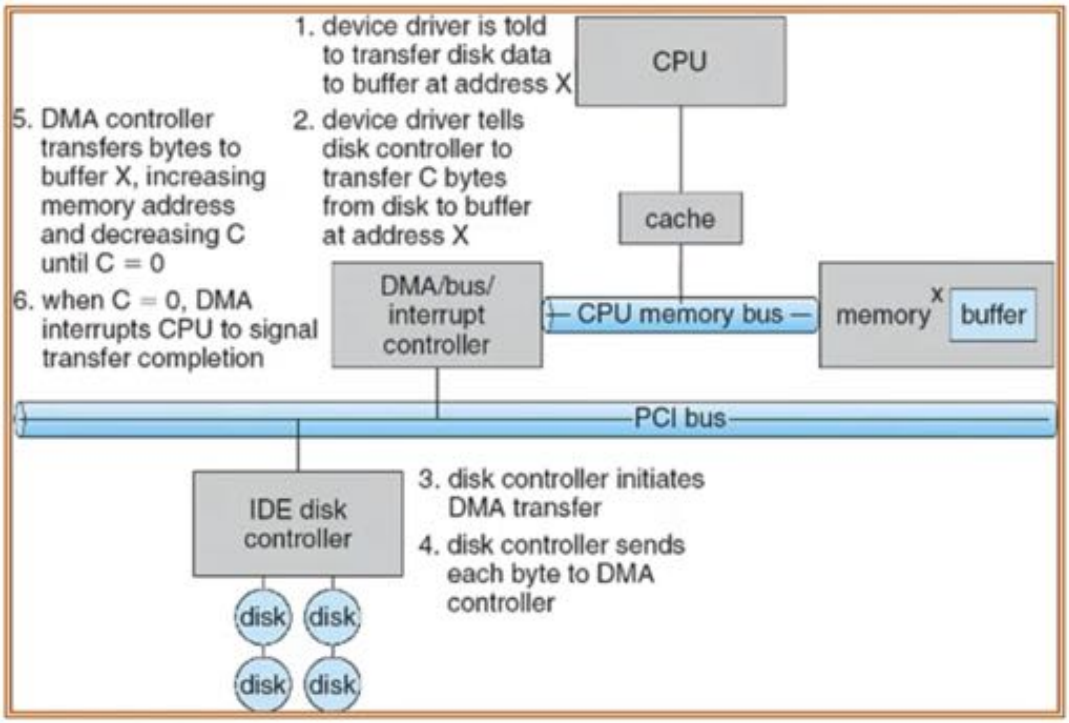
- CPU sets up DMA request, giving controller access to memory bus

- Device puts data on bus & RAM accepts it
- Device interrupts CPU when done



Sidesteps the CPU

DMA in more detail



How can the OS handle all devices

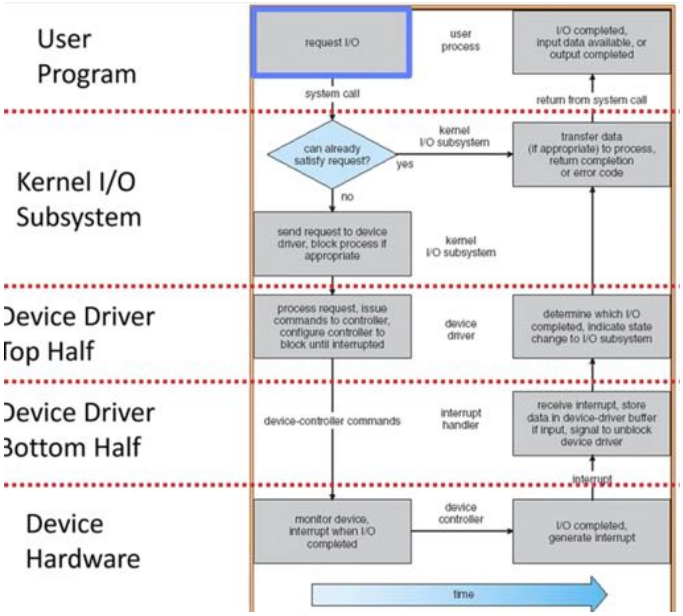
- Build a device netral OS and hide details of devices from most of OS
- Device drivers encapsulate all specifics of device interaction, Part of OS , software
- Implement device

Device Driver:

- Device specific code in the kernel that interacts directly with the device hardware
- Supports a standard, internal interface
- Special device specific configuration suproted with ioctl() system call

Divided into

1. Top half: accessed in call path from system calls
    a. Set of standard, cross-device calls like open(), close(), read(), write(), ioctl(), strategy()
    b. Interface tto the device driver
    c. Top half starts IO to device, may put thread to sleep until finsihsed
2. Bottom half: run as interrupt routine
    a. Gets input or transfers next block of output
    b. May wake sleeping thread if IO now complete



Conclusion

- IO devices Types:
    - Different speeds
    - Different access patterns
        - Block devices, character devices, network devices
- IO controllers: hardware that controls actual device
    - Processor accesses through IO instructions, load/store to special physical memory
- Notification mechanism
    - Interrupts, polling
- Device drivers interface to IO devices
    - Provide clean w/r interface

Week 11: Lecture 18 Storage Devices, Performance, Queuing Theory (4/3)

Ways of Measuring Performance: Times (s) and Rates (op/s)

- Response Time or Latency
    - Time to complete a task
- Throughput or Bandwidth
    - Measured in units of things per unit time (ops/s)
- Start up or Overhead
    - Time to initiate an operation

Storage Devices

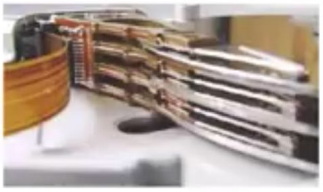
Magnetic disks

- Storage that rarely is corrupted, large capacity at low cost, block level random access
- Slow performance for random access

- Better performance for sequential access

Hard Disk Drive (HDDs)

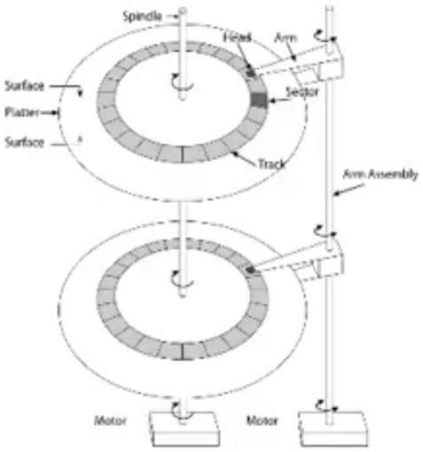
- Head moves around the platters
- Change the content using magnetic waves

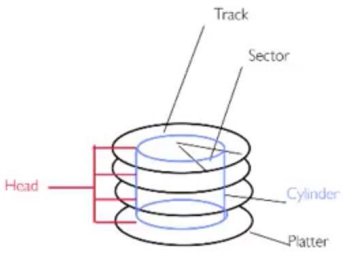


Read/Write Head Side View

The Amazing Magnetic Disk

- Store data manetically on thin metallic film bonded to rotating disk of glass, ceramic, or aluminum
- Track: concentric circle on surface
- Sector: slice of track
    - Smallest addressable unit, units of transfer
- Cylinder: all the tracks under the head at a given point on all surfaces
- Track lengths vary across disk: outside tracks have more sectors
- Organized into regions of tracks with the sam enumber of sectors
- read/write data is a 3 stage process:
    1. Seek time: position the head/arm over the proper track
    2. Rotational latency: wait for desired sector to rotate under r/w head
    3. Transfer time: transfer a block of bits under r/w head
- Request Time = Queueing Time + Controller Time + Seek + rotational + transfer
-





Parameter	Info/Range
Space/Density	Space: 18TB (Seagate), 9 platters, in 3½ inch form factor! Areal Density: **≥ 1 Terabit/square inch! (PMR, Helium, …)**
Average Seek Time	Typically 4-6 milliseconds
Average Rotational Latency	Most laptop/desktop disks rotate at 3600-7200 RPM (16-8 ms/rotation). Server disks up to 15K RPM. Average latency is halfway around disk so 4-8 milliseconds
Controller Time	Depends on controller hardware
Transfer Time	Typically 50 to 250 MB/s. Depends on: • Transfer size (usually a sector): 512B – 1KB per sector • Rotation speed: 3600 RPM to 15000 RPM • Recording density: bits per inch on a track • Diameter: ranges from 1 in to 5.25 in
Cost	Used to drop by a factor of two every 1.5 years (or faster), now slowing down

Disk Performance Example

- Key to using disk effectively is to minimize seek and rotational delays
- Reading random block
    - Seek (5ms) + rot Delay (4ms) + transfer (0.082 ms) = 9 ms

Lost of Intelligence in the controller

- Sectors contain sophisticated error correcting codes
- Sector sparing
    - Remap bad sectors to spare sectors

Solid State Drivers

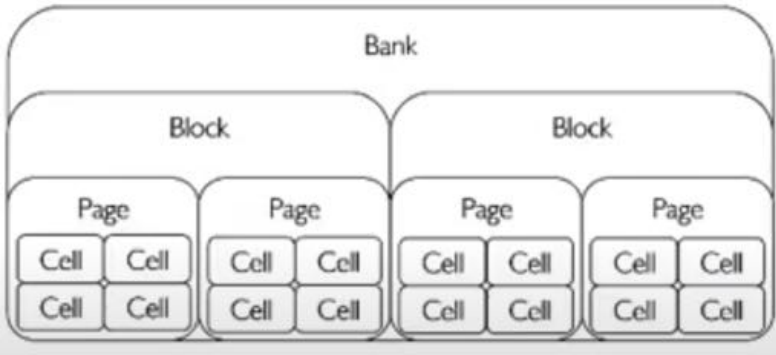
1995 - battery backed DRAM

2009 - use flash memory

- Trapped Electrons ditingusih between 1 and 0
- No moving parts (no rotate/seek)
    - Eliminatese seek and rotation
    - Limited write cycles

The Flash Cell

- Encode bit by trapping electrons into a cell
- Single-level cell (SLC)
    - Single bit is stored within a transistor
    - Faster more lasting
- Multi Level cell (MLC)
    - Multi bits stored



Flash chips

- Organized in banks
    - Banks can be accessed in parallel
- Blocks : 128 KB
- Pages: Few KB
- Cells: 1 to 4 bits

Low level flash operations

- Chip supports reading pages
- Independent of the previously read pages

Writing?

- Must first erase the block, quite expensive
- Once erased, can then program a page
    - Change 1s to 0s within a page
    - 100s of microseconds
- Blocks can only be erased a limited number of times

SSD Architecture

SSD uses low level flash operations to provide same interface as HDD read and write chunks at a time

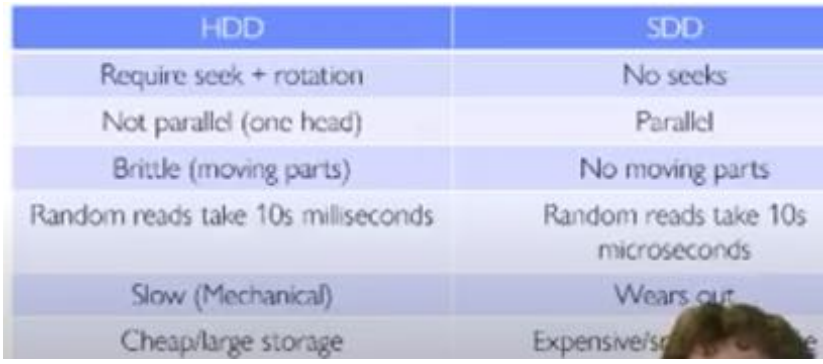
Flash Translation layer (FTL)

- Add a layer of indirection which translates request for logical blocks to low level flash blocks and pages
- Goal: performance and reliability
- Reduce write amplification
    - Ratio of total write traffic in bytes issues by the flash chip by the FTL divided by the total write traffic issued by the OS to the device
- Avoid wear out

- Single block should not be erased too often

FTL -- Two Systems Principles

- Uses indirection and copy-on-write
- Maintains mapping tables in DRAM
  - Map virtual block numbers to physical page numbers
  - Can now freely relocate data w/o OS knowing
- Copy on Write/Log-structured FTL
  - Don't overwrite a page when OS updates its data
  - Write a new version in a free page
  - Update FTL mapping to point to new location



HDD	SDD
Require seek + rotation	No seeks
Not parallel (one head)	Parallel
Brittle (moving parts)	No moving parts
Random reads take 10s milliseconds	Random reads take 10s microseconds
Slow (Mechanical)	Wears out
Cheap/large storage	Expensive/s

Overall Performance for I/O Path

- Sequential Server performance
- Single Pipelined Server k stages for tasks length L (L/k per stage)
- Latency (L): time per op
- How long does it take to flow through the system
- Bandwidth (B); rate , Op/s

Little's Law (B -> λ)

- Average arrival rate = Average departure rate
- N (jobs) = λ(jobs/s) * L (s)
- Utilization : $\rho = \frac{\lambda}{\mu_{max}}$

Bottleneck Analysis

- Each stage has own queue and max service rate
- Bottleneck stage dictates the max service rate μ_{max}

Queuing

- If Request rate (λ) exceeds max service rate,
- Short bursts can be absorbed by queue
- Prolonged queue with greater than service rate will grow without bounds
- Memoryless: likelihood of an event occuring is independent of how long we've been waiting

Steady State Queuing Theory

- Assumptions: system in equilibrium, no lmimit to the queue, time between successive arrivals is random and memoryless

- Memoryless service distribution ($C = 1$)—an "M/M/1 queue":

$$T_Q = \frac{\rho}{1-\rho} \cdot T_S$$

- General service distribution (no restrictions)—an "M/G/1 queue":

$$T_Q = \frac{1+C}{2} \cdot \frac{\rho}{1-\rho} \cdot T_S$$

-

Conclusion

- Performance
    - Bottleneck & queuing delay
    - Model arrival/departure rate as probability distributions

Week 12: Lecture 19 Filesystems 1: Filesystem Design, Case Studies (4/6)

Overall performance = queue + I/O device service time

Disk Scheduling

- Order to choose to do queued requests, can only do one at a time

1. FIFO Order
    a. Fair among requesers, btu order of arrival may be random sports on the disk
2. Shortest seek time first (SSTF)
    a. Pick the request that's closest on the disk
    b. Include rotational delay
    c. Con: may lead to starvation
3. Elevator Algorithm, take the closest request in the direction of travel (SCAN)
    a. No starvation, but retains flavor of SSTF
4. Circular Scan (C-SCAN)
    a. Only goes in one direction, skips requests on the way back
    b. Fairer than SCAN not biases towards pages in middle

Blocking Interface: "Wait"

- Request, put process to sleep until data read

Nonblocking interface: Don't wait

- Returns quickerly with count of bytes successfully transferred

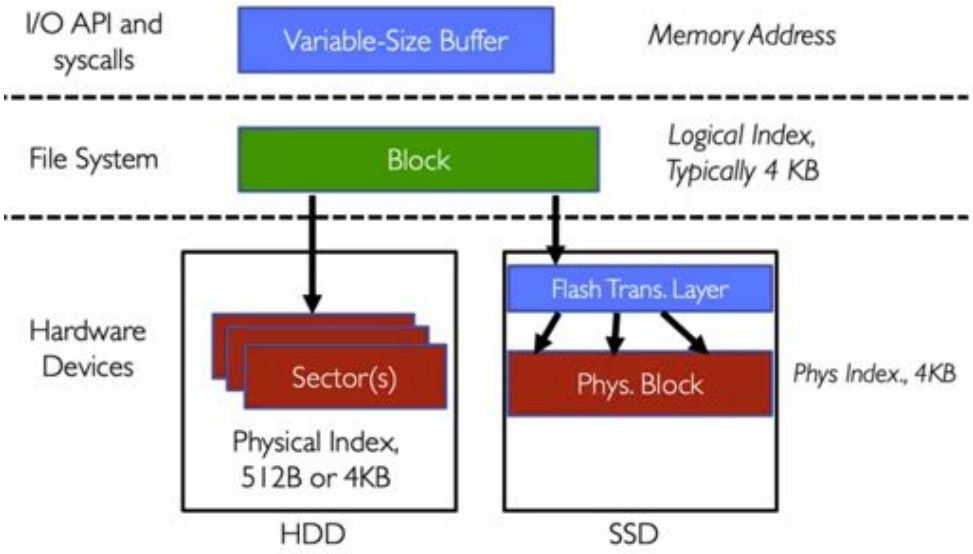
Asynchronous Interface: Tell me later

- Take pointer to user's buffer, fills buffer

Buiilding a file system

- File System: Layer of OS that transforms block interface of disks into Files, Directories
- Classic OS situation: Take limited hardware interface and provide a more convient/useful interface

- Naming, organization



User vs System View of File

- User's view
  - Durable Data Structures
- System's view
  - Collection of bytes
- Systems view (inside OS)
  - Collection of blocks

Disk Management

- Basic entities on a disk
- File: user-visible group of blocks
- DIrectory: user-visible index mapping names to files

Disk accessed as linear array of sectors

- Logical Block Addressing (LBA)
  - Sector has integer address
  - Controller translates from address -> physical position
  - Shields OS from structure of disk

FIle System Need

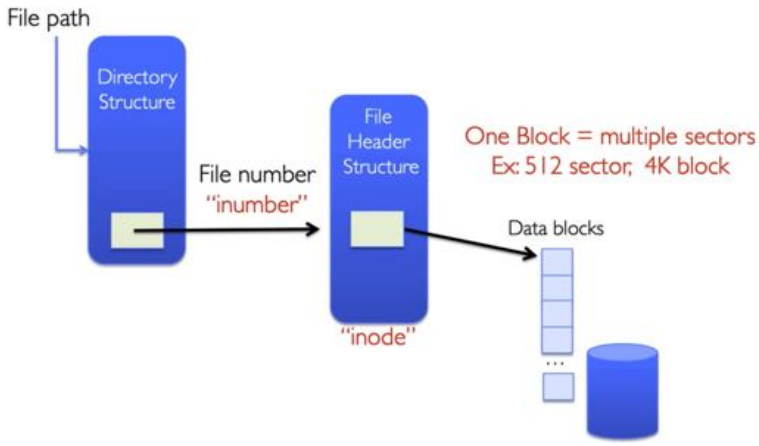
- Track free disk blocks
- Track which blocks contain data for which files
- Track files in a directory
- Where to maintain all data?

Data Structures on Disk

- Bit different than data structures in memory
- Access a block at a time, sequential access patterns
- Durability
  - Meaningful state upon shutdown

Critical Factors in File System Design

- Disks Performance:
  - Max sequential access, minimize seeks
- Open before Read/write
  - Can perform protection checks and look up where the actual file resource are, in advance
- Size is determined as they are used
  - Can expand the file
- Organized into directories
  - What data structure on disk for that
- Carefully allocate / free blocks
  - Access remains efficient



Open file desciritopn better described as inumber (file number)

Open performs Name Resolution

- Translates path name into file number

Read and Write operate on teh file number

- Use file number as index to locate the blocks

4 components

- Directory, index structure, storage blocks, free space map

How to Get the File Numbers

- Look up in directory structure
- Directory is file name : file number mapping
- Process isn't allowed to read the raw bytes of a directory
  - Read function doesn't work on directory
  - Readdir, iterates over map without revealing the raw bytes

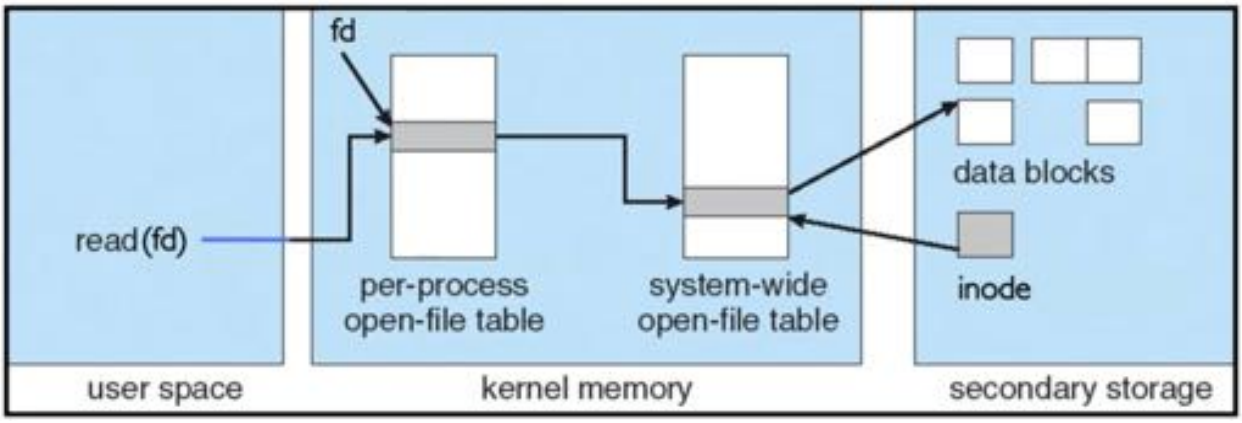
Idrectory Strucutre

- How many to resolve "/my/book/count"
- Read in file header for root
- Read in first data block for root
- Read in file header for "my"
- Read in first data block for "my" search for "book"
- Read in file header for "book"
- Read in first data block for "book" search for "count"
- Read in file header for "count"

Current working directory: Per address space pointer to a directory used for resolving file names

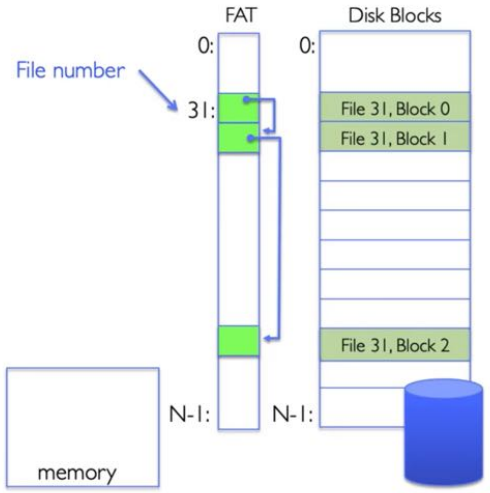
- Specify relative filename instead of absolute

- Reduce number of reads



In Memory file system structures

Most files on system are small but most space taken up on file system are large files



<u>Case Study</u>

File Allocation Table (FAT): MS-DOS 1977

File Allocation table (FAT)

- Have a way to translate a path to a file number
- Disk Storage is a collection of blocks
- File is a collection of disk blo ks
- FAT is a linked list 1-1 with blocks
- File number is index of root of block list for the file
- File offset: block number and offset within block
- Follow list to get get block number
- Unused blocks marked as free

FAT

- On disk
- Format a disk
    - Zero the blocks, mark FAT entries “free”
    - Haow to quick format a disk
        - Mark FAT entries “free”
- Simple but not efficient

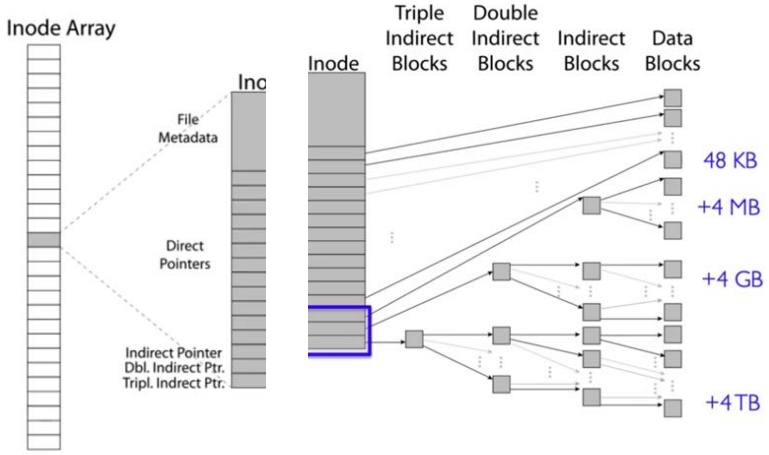
Unix File System (Berkeley Fast File System)

- File Number is index into set of inode arrays
- Index structures is an array of inodes
    - File number is an index into array of inodes
    - Each inode corresponds to a file and contains its metadata
- Inode maints a multi level tree structure to find storage blocks for files

Direct pointers point to smaller files lots of files

Indirect pointers

- Point to a disk block
    - Better for larger files that are less



Critical Factors in File System Design

- Disk Performance
    - Max sequential access

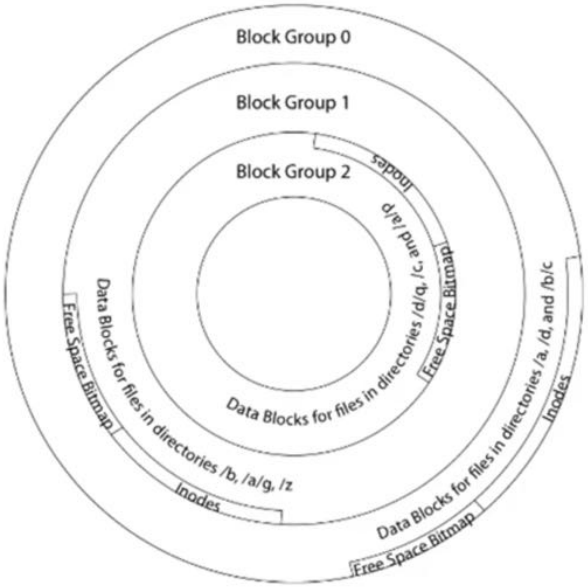
Fast File System

Optimized for Performance and Reliability

- Distribute inodes among different tracks to be closer to data
- Uses bitmap allocation in place of freelist
- Attempt to allocate files contiguous
- 10% reserved disk space
- Skip-sector positioning

FFS Changes in Inode Placemnt: movement

- Block Groups
    - Distributed header info closer to the data blocks
    - Inode for file stored in same "cylinder group"
    - Makes "ls" of that directory run very fast
    - Reserve space in block group



Summary: FFS Inode Layout Pros

- Small directories can fit all data, file headers, in same cylinder, no frees
- File headers much smaller tha whole block, multiple headers fetch at the same time
- Reliabile even if directores disconnected

Problem 3: missing blocks due to rotational delay

- Issue: read one block, do processing, read next block, dis missed nextblock while turning.
- Solution 1: skip sector position, place blocks from one file on every other block of a track give time for proeccesing to overlap rotation
- Solution 2: read ahread: read next block right after first, even if application hasn't asked for it yet
    - Can be done by OS or disk
    - Disk controller has internal RAM to read a complete track
- Track Buffer holds complete track
- Modern disks + controllers do many things under the covers
- Track buffers, elevator algorithms bad block filtering

Pros: Efficient storage for both small and large files

- Localifty for both small and large files, metadata

Cons:

- Inefficient for tiny files
- Inefficient encoding when a file is mostly contiguous on disk
- Need to reserve 10-20% of free space to prevent fregamentation

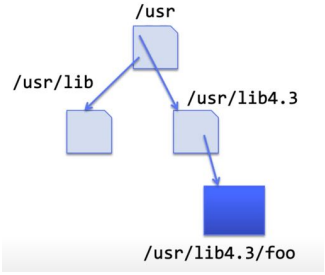
Conclusion

- Systems designed to optimize performance and reliability
- File Systems:
    - Transforms blocks into files and directories
    - Optimize for access and usage patterns
    - Maximize sequential access, allow efficient random access
- Defined by inode
- Naming: translating from user-visible names to actual system resources
- File ALlocaiton Table Scheme
    - Linked list approach
- Look at actual file access patterns

Week 12: Lecture 20 Filesystems 2: Filesystem Case Studies, Buffering (4/8)

Linux Example

- Disk divided into block groups
- Provides locality



Hard Link

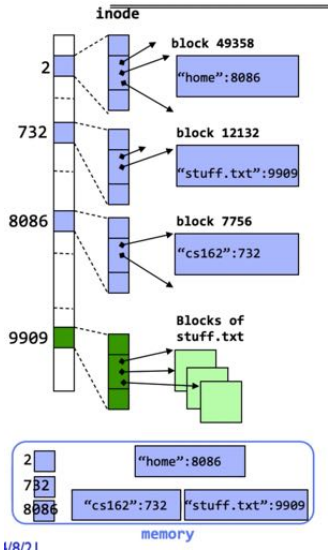
- Mapping from name to file number in directory structure

Soft link (symbolic link or shortcut)

- Directory entry contains the path and name of the file

Directory Traversal

- /home/cs162/stuff.txt
- Get root directory, scan for home to get inumber



Large DIrectories: B-Trees (dirhash)

- FreeBSD, NetBSD, OpenBSD

Case Study: New Technology File System (NTFS)

Instead of FAT or inode array: Master File Table

- Database
- Flexible 1 KB entries for metadata/data
- Variable-sized attribute records (data or metadata)
- Extend with variable depth tree

Memory Mapped Files

- Map the file directly into an empty region of our address space
    - Page it in when we read it, write itand eventually page it out
- mmap() system call
- Map specific region or let system find

Can share through mapped files

- Point to the same memory

Buffer cache

Kernel must copy disk blocks to main memory to access their contents and write them back if modified

- Cache disk data in memory
- Locality by caching disk data in memory
- Name translations: mapping form paths -> inodes
- Disk blocks: mapping from balck address -> disk content

Buffer Cache

- Memory used to cache kernel resources, including disk blocks and name translations
    - Contains dirty blocks

File System Buffer Cache

1. Directory lookup as needed, load block of directory, search for map
2. Create reference via open file descriptor

Implmented entriely in OS software

- Unlike memory caches and TLB
- Blocks go through transitional states between free and in-use
    - Being read from disk, being written to disk
- Blocks used for variety of purposes: inodes,, data for dirs and files
- Termination: open read, write
- Replacement? LRU, can afford overhead full LRU implementation
    - Works very well for name translation
    - Works well in general as long as memory is big enough to accommodate host's working set of files
- Disadvantages
    - Fails when scans through file system, flushing cache with data

Cache Size: How muc memory should OS allocate to buffer cache vs virtual memory

- Too muc memory to file system cache
- Too little memory to file system cache -> many applications may run slowly
- Solution: adjust boundary dynamically so paging and file access are balanced

File System Prefetching

- Read Ahead Prefetching: fetch sequential blocks early
    - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks haed of current read request
    - Too much prefdetching imposes delays on requests buy other applications

Delayed Writes

- Buffer cache is a writeback cache
    - write() copies data from user space to kernel buffer cache
    - read() is fulfilled by cache so reads see the results of writes

        - Some files never actually amke it all the way to disk
            - Short lived files

Buffer Caching vs Demand Paging

- Replacemnt Policy
    - Demand paging: LRU is infeasible, use approx, runnign at memory speeds
    - Buffer cache: lru is ok, since running at disk speeds
- Eviction policy
    - Demand Paging: evict not recentlu sed when memory close to full
    - Buffer cache: periodically write back: minimize data loss

Can still crash with dirty blocks in cache

File Systems need recovery mechanisms

File System Summary

- File System
    - Transforms blocks into files and directories
    - Optimize for size, access and usage patterns
    - Max sequential access, allow efficient random access
    - Projects the OS protection and security regime
- File defined by head, inode
- Naming: translate user visible name to actual sys resources
- Multilevel indexed scheme
- NTFS: variable size extends, not fixed blocks, tiny files data is in header
- File layout driven by freespace management
    - Optimizations for sequential access: start new files in open ranges of fre blocks
    - Integrate freespace, indoe table, file blocks and idrs into block group
- Deep interationc between meme management, file syste, sharing
    - mmap(): map file or anon segment to memory
- Buffer cache: memory used to cache kernel resources

Week 13: Lecture 21 Filesystems 3: Reliability and Transactions (4/13)

Ext2/3

- Has journaling
- NTFS: Master FIle Table
    - Extents : variable length contingouous regions

Important “ilities”

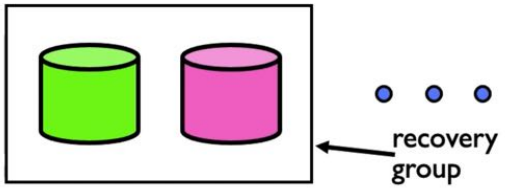
- Availability: probability that the system can accept and process requests
    - Measured in “nines” of probability
    - Key idea here is independence of failures
- Durability: the ability of a system to recover data despite faults
    - Idea is fault tolerance applied to data
    - Durable but could not be accessed

- Reliability: Ability of a system to perform its required functions under stated conditions for a specified period of time
    - Must be working correctly
    - Includes availability, security, fault tolerance/durability
    - Data survives system crashes, disk crashes

How to make file systems more durable

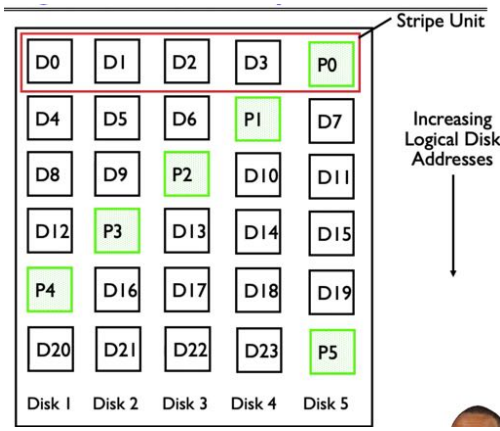
- Disk blocks contain Reed-Solomon error correcting codes to deal with small defects in disk drive
- Make sure writes survive in short term
- Either abandon delayed writes or Use special battery-backed RAM for dirty blocks in buffer cache
    - Non-voltile RAM
- Make sur data survives in long term
    - Replicate, more than one copy of data

RAID 1: Disk Mirroring/Shadowing



- Each disk fully duplicated onto its shadow
- For high I/O , high availability environments
- Logical write = two physical writes
- Can have two independent reads to smae data

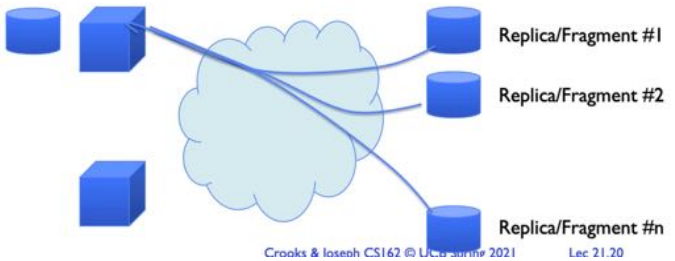
RAID 5+: High I/O Rate Parity



- Data stripped across multiple disks
    - Succcessive blocks stored on successive disks
    - Increased bandwidth over single disk
- Parity block in green by XORing data blocks in stripe
- Reonstruct by
- Can spread info widely across internet for durability
    - RAID algorithms work over geographic scale

RAID 6:

- Raid X is an erasure code
- Ability to know which disks are bad, treating missing disk as an erasure
- Time to repair disk is soo long, another disk might fail in process
- "RAID" : allow 2 disks in replication strip to fail
    - Requires more complex erasure code EVENODD code
- More general option for general erasure code: Reed-Solomon codes
    - M data points can tolerate n-m failures tolerated
- Erausre codes not just for disk arrays.



    - Split data into m = 4 chunks, generate n = 16 fragments and separate out data

Raid 6: Highly Durable, Highly avaialbal,

File System Reliability

- Difference from block level reliability
- What happens if disk loses power or software crashes
- Having RAID doesn't necessarily protect against all such failures
    - No proteciton against writing bad state
- File system needs durability
    - Data previously stored can be retrieved, regardless of failure

Storage reliability problem

- Single logical file operation can involve updates to multiple physical disk blocks
    - Inode, indirect block
- At physical level, operations complete one at atime

Threats to Reliability

- Interrupted Operation
    - Leave stored data in an inconsistent state
- Loss of stored data
    - Failure of non-volatile storage media

Two Reliability Approaches

1. Careful Ordering and Recovery
    a. FAT & FFS
    b. Each step builds structure
    c. Last step links it into res of FS
    d. Recover sacnas structure looking for incomplete actions
2. Versioning and Copy on Write
    a. ZFS
    b. Version files at some granularity
    c. Create new structure linking back to unchanged parts of old
    d. Declare that the new version is ready

Careful Ordering and Recover

- Sequence operations in specific oder
- Post-crash recovery
    - Read data structures to see if there were any operations in progress
    - Clean up/finisha s needed
- Many app level recovery schemes (Word)

Berkeley FFS: Create a file

Normal operation:

- Allocate data block
- Write data block
- Allocate inode
- Write inode block
- Update bitmap of free blocks and inodes
- Update directory with file name → inode number
- Update modify time for directory

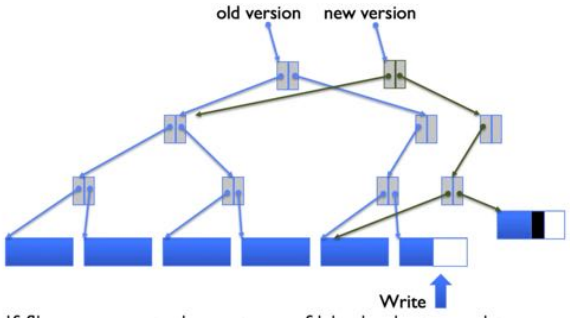
Recovery:

- Scan inode table
- If any unlinked files (not in any directory), delete or put in lost & found dir
- Compare free block bitmap against inode trees
- Scan directories for missing update/access times

Time proportional to disk size

Reliability Approach #2: Copy on Write File Layout

- Instead of verwriting existing data blocks and updating the index structure
  - Create new version of file with updated data
  - Reuse blocks that don't change much of what is already in place
  - Copy on Write (COW)
- Updates can be batched
  - All disk writes in parallel
- Network file structure



Transactions

- More general reliability Solutions
  - Use transactions for atomic updates
  - Internally update filesystem structuers and metadata
- Provide redundancy for media failures
  - ECC, replication, above that

Transactions

- Atomic update from memory to stable storage
- Atomically update multiple persistent data structure s
- Applications user temporary files and rename

Key Concept: Transaction

- transaction : atomic sequence of reads and writes that takes system from consistent state to another

Typical Strucutre

- Begin transaction
- Do a bunhc of updates
    - If any fail, roll-back
    - Any conflicts, roll back
- Commit

Concept of a log

- Transactions to seal commitment to whole series of actions

<u>Transactional Filesystems</u>

Transaction FileSystems

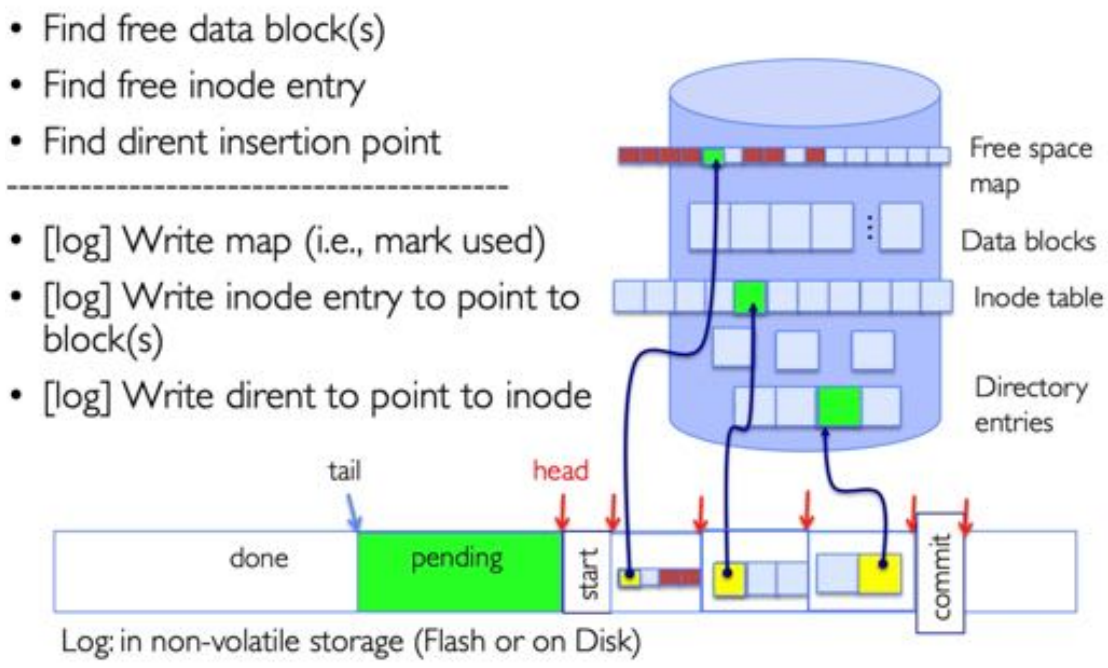
- Better reliability through use of log
- Changes are treated as transactions
- A transaction is committed once it is written to the log
- File system may not be updated immediately, data preserved in the log

Difference btween Log Structured and Journaled

- Ini a log structured file system, data stays in log form

Journaliing FIle System

- Don't modify data structures on disk directly
- Write each update as a transaction recorded in a log
- Once changes are in log, safely applied to file system
- Garbage collection: once a change is iapplied, removed its entry from the log
- Linux took original FFS-like file system and added a journal to get ext3
    - Some options: whether or not to write all data to journal or just metadata



-

- Scan log, find start, find matching commit, redo it as usual
- Updates atomic, even if we crash, update either gets fully applied or discarded

Log structured file system (LFS)

- The LOG IS the storage
- LFS writes everything sequentially

Log Structure

- Index and directories are written into the log too
- Large important portion of the log is cached in memory
- Each segment has summary of all operations within the segment
- Free space as continual cleaning process of segments

Flash Filesystems

- Cannot overwrite pages
    - Move contents to an erased page
- Program/Erase (PE) Wear
    - Permanedent damage
- Flash Translation Layer (FTL)
    - Translates between logical blocka dresses and physical flash page addresses
- Management process
    - Keep freelist full, manage mapping

LFS: F2F2: A flash file system

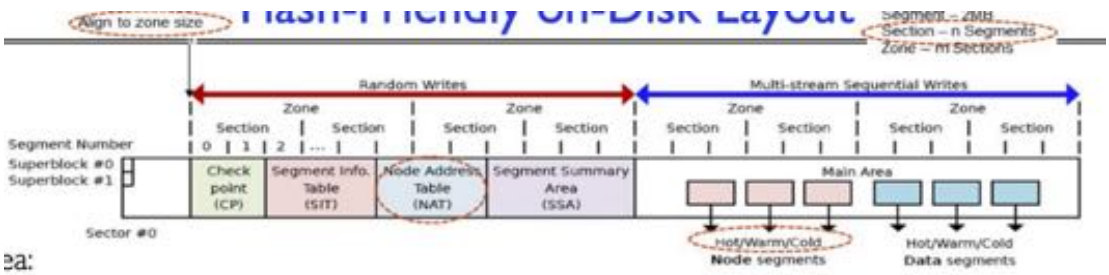
- Used on my mobile devices
- Assumes standard SSD interface
- With built=in flash translation layer
- Random reads are as fast as sequential reads
- Random writes are bad for flash storage

Minimize Wrties/updates and otherwise keep writes “sequential”

- Start with log structure file system
- Keep writes as sequential as possible
- Node Translation Table (NAT) for logical to physical translation

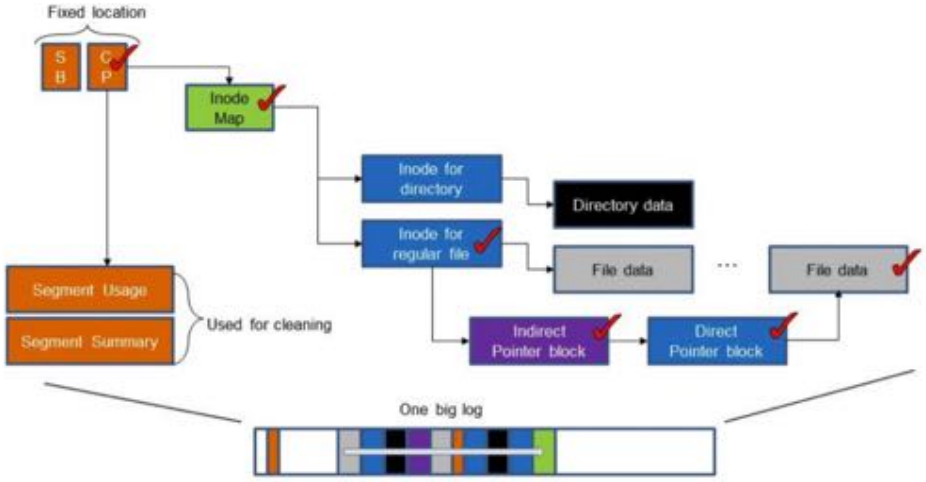
Flash-Friendly on-Disk Layout

- Main Area: divided into segmentts
- Node Address Table: Independent of FTL
- Updates to data sorted by predicted write frequency (Hot/Warm/Cold) to optimize FLASH management
- Checkpoint (CP) : keep the file system status
- Segment information Table (SIT)
    - Per segment info, used for garbage collection



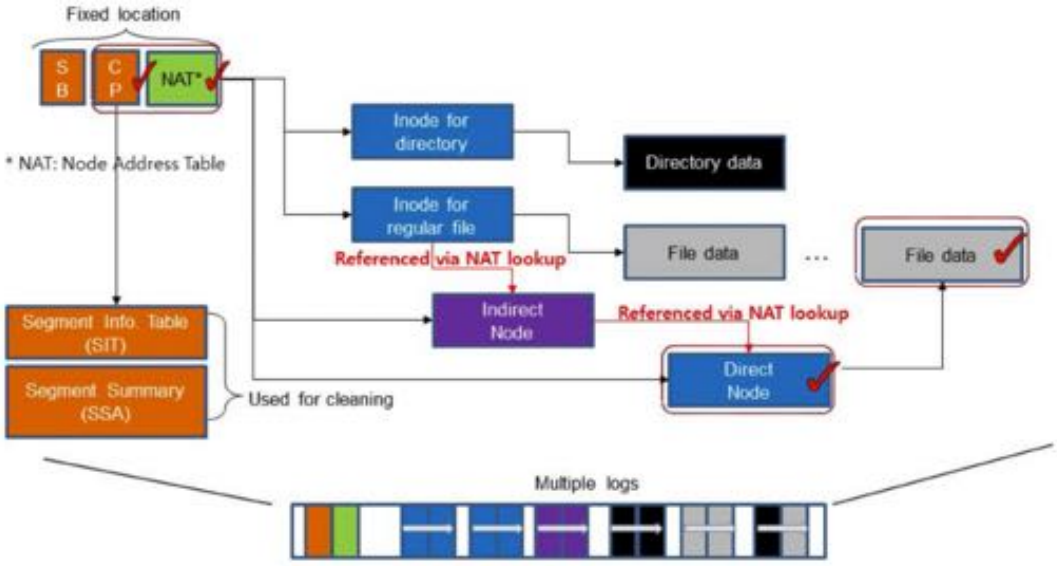
Normal LFS Index Structure: Forces Cascading Updates when Updating Data

- Update propagation issue: wandering tree



-

F2FS Index Structure: Indirection and Multi-Headc Logs Optimize Updates



Summary

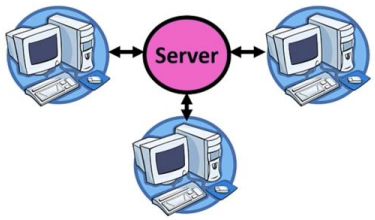
- File system operations have multiple updates to blocks on disk
    - Creash may occur in the midst of the sequence
- Traditional file system perform check and recovery on boot
- Copy-on-write provides richer function with much simpler recovery
- Transactions over a log provide a general solution
    - Commit and update disk
    - Log precedence over disk
    - Replay committed transactions, discard partials
    - Important system props

        - Availability: how often is resource available
        - Durability: how well is data preserved against faults
        - Relaibilty: how often is resource performing correctly
- RAID: Redundant Arrrays of Inexpensive Disks
    - Parity block
- Use of Log improves reliability
- Transacitons over a log provide general solution

Week 13: Lecture 22 End-to-End Args, Distributed Decision Making (4/15)

Centralised vs Distributed Systems

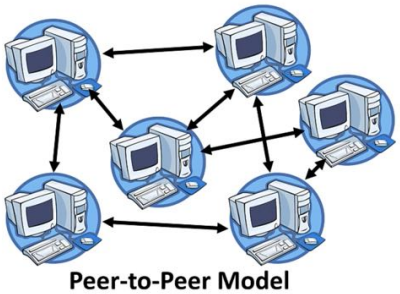
- The world is a large distributed system, microprocessors in everything



Client/Server Model

Two types of distirbuted systems

1. Client/Server Model
    a. Clients make remote prodceudre calls to server
    b. Server serves requests from clients
    c. Hierarchical relationship
2. Peer-to-Peer Model
    a. Each computer acts as a peer
    b. No Heirarchy or central point of coordination
    c. All way communications between peers, gossiping


Peer-to-Peer Model

How do i store all my data

- Server that does routing
- Contact router (load balancer) to find which machine it is
- Store data on multiple servers, still located on machine that is alive
- Have multiple routers
- Sharding: split onto multiple servers
- Reptions: multiple servers
- Load balancer: multiple copies to scale

Promise of Distributed Systems

- Availability: When it is in functioning state
    - Proportion of time system is in functioning condition
    - One machine goes down, use another
- Fault-tolerance
    - Defined behavior when fault occurs
    - Store data in multiple locations
- Scalability
    - Ability to add resources to system to support more work
    - Add machines when need more storage/processing power

Requirements of distributed Systems

- Transparency
    - Ability of system to mask complexity behind a simple interface
        - Location transparency: can't tell where resources are located
        - Migration: resources may move without the user knowing
        - Replication: can't tell how many copies of resource exist
        - Concurrency: can't tell how many users there are
        - Parallelism: system may speed up large jobs by splitting them into smaller pieces
        - Fault Tolerance: system hide various things that go wrong

Challenges of distributed systems

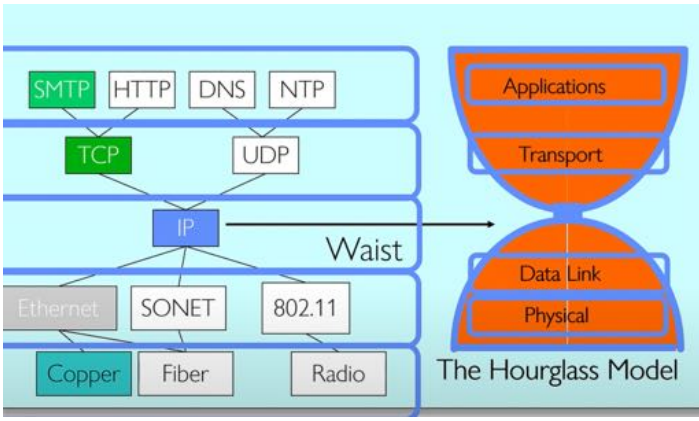
- How to get machines to communicate
- How to get to coordinate
- How do you deal with failures
- How do you deal with security?

How do entities communicate?

- A protocol exchange
- Agreement on how to communicate
    - Syntax: how a communication is specified & structured
        - Format, order messages are sent and received
    - Semantics; What a communication means
        - Actions taken when transmitting, receiving, or when a timer expires
    - Formally described by state machine

Case Study: The Internet

- Largest distributed system that exists
- Many different applications
- Layering & end-to-end principle



Internet: Layers, Layers, Layers

- Intermediate layer
- TCP: reliable, guarante evnaully get across
- UDP: might get dropped
- Applicatoins: http, dns, ntp, smtp
- Narrow waist with interoperability

Implications of Hourglass

- Single Internet-layer module (IP)
    - Allows arbitrary networks to interoperate
    - Allows applications to function on all networks
    - Supports simultaneous innovations above and below IP

Drawbacks of Layering

- Layer N may duplicate layer N-1 functionality
    - Error recovery to retransmit lost data

        - Error recovery to retransmit
    - Layers may need same information
        - Time stamps, maximum transmission unit size
    - Layering can hurt performance
        - Hiding details about what is really going on
    - Some layers not always cleanly separated
        - Inter-layer dependencies

End-to-End Argument

- "End-to-End Arguments in System Design"
- "Sacred Text" of the internet
    - Endless disputes about what it means
    - Everyone cites it as supporting their position
- Simple Message: Some types of network functionality can only be correctly implemented end-to-end
- Hosts cannot reply on network to meet so must implement themselves

Reliable File Transfer

- Solution 1: Make each step reliable, then concatenate
- Solution 2: end-to-end check and try again if necessary
    - Full functionality can be entirely implemented at application layer with no need for reliability from lower layers
- Any need to implement reliability at lower layers
    - No benefit sematically checking in the middle

End-to-End Principle

- Implementing complex functionality in the network
- Doesn't always reduce host implementation complexity
- Does increase network complexity
- Imposes delay and overhead on all applications, even if they don't mnbeed functionaliy

Conservative Interpretation of E2E

- Don't implement a function at the lower leves of the system unless it can be completely implemented at the level
    - Unless you can relieve burden from hosts, don't bother

Moderate Interpretataion

- Think twice between implementing functionality int he network
- If hosts can implement functionality correctly, implement it in a lower layer only as a performance enhancement
- Do so only if does not impose burden on applications that do not require that functionality

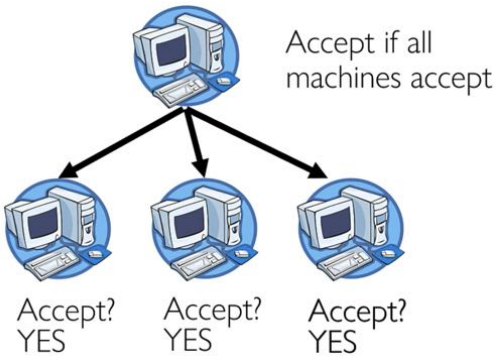
Coordination: making distributed decisions

- Functionality is spread across machines, required coordination to reach distributed decision

- Functionality is spread across machines, requires coordination to reach distributed decision

Coordination is hard!

- When machines can fail, slow, unreliable, machines receive conflicting proposals



General's Paradox

- Can message over unreliable network be used to guarantee two entities something simultaneously
- Problem: Two generals, separate mountains, communicate via messengers, can be caputred
- Problem: need to coordinate attack, at different times, they all die
- Impossible to achieve simulatous acknowlekdgement

Eventual Agreement: Two-Phase Commit

- Distirbuted transaction: two or more machines agree to do something, atomically
- No constraints on time just eventually happen

Two-Phase Commit protocol: Developed by Turing award Jim Gray

- Used in most modern distbruted systems

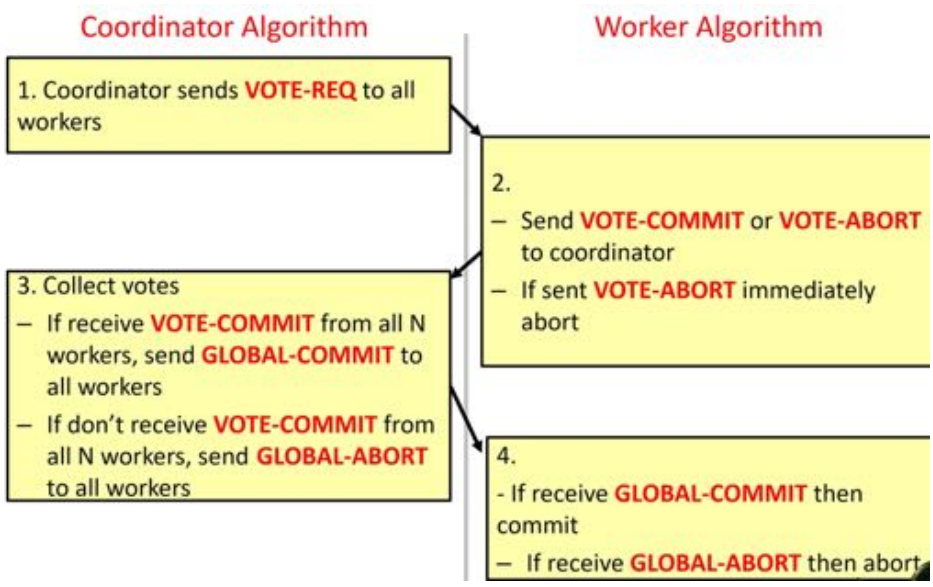
TPC: Determine wheter should commit or abort a transaction

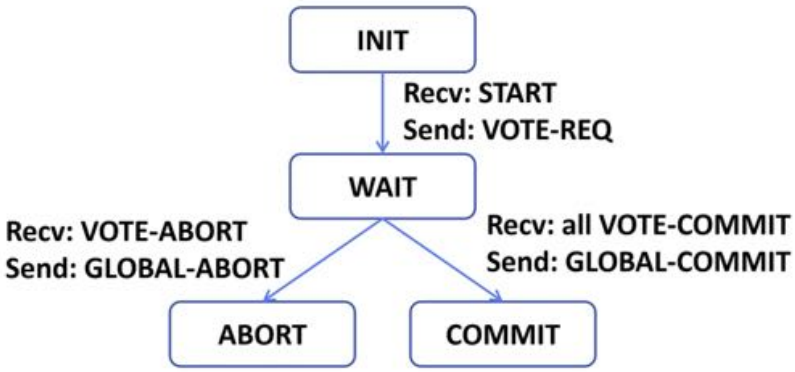
- All processes that reach a decisions reach the same one (Agreement)
- A process cannot reverse its decisions after it has reached one (finality)
- If there are no failures, every process votes yes, decision will be commit (Consistency)
- If all failures are reqparied and there are no more failures, then all processes will eventually decide commit/abort (Termiantion)

2PC Terminology

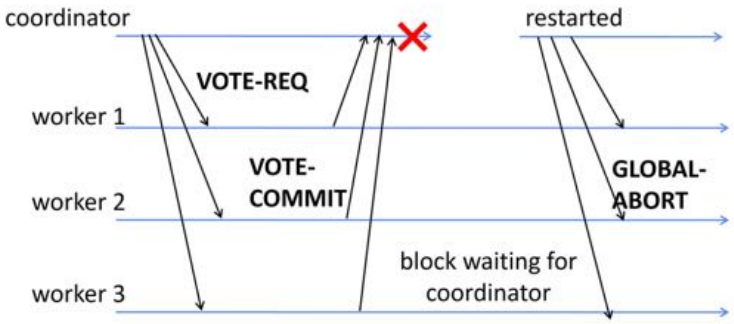
- Setup:
  - One coordinator
  - A set of participants
- Each process has access to a persistent log:
  - Recorded information on the log will persist after crashes
- Coordinator asks all processes to vote
- Each participant can vote either YES or NO
  - If all vote YES, COmmit

State machine of coordinator





1) What happens when waiting for message never comes
    a) Step 2: Worker waiting from VOTE-REQ from coordinator
        i) Abort and halt
    b) Step 3: Coordinator is waiting for vote from participants
        i) Votes abort and sends GLOBAL Abort
    c) Step 4: Worker who voted YES is waiting for decision
        i) Worker must run termination protocol



Termination protocol

1. Option 1: Simply wait fro coordinato ro recover
2. Ask a friendly participate p
    a. P decided commit/abort, forwards decision to initialor
    b. P not decided , votes ABORT sends abort,
    c. IF P voted COMMIT, P is also stuck and can't help initiator

If every pariticaple voted COMMIT, must wait for coordinator to get back

Machine recovery

- All nodes use stable storage to store current state (backed by disk/SSD)
- Upon recovery, nodes can restore stature and resume
- When coordinator sends VOTE-REQ, writes START-2PC to log
    - Reads, log if sees VOTE-REQ but no decision, ABORT
- Before voting, particpatn write VOTE-* to stable log and sends vote
    - Reads log, if doesn't see record sends, VOTE-ABORT, if VOTE COMMIT, contacts friend
- Before sending decision, coordinator writes GLOBAL-* to stable log, then sends decision
    - Coordinator reads log, if sees GLOBAL-(, resends decision

- After reciinv GLOBAL-*, participate writes commit/abort to stable log
    - Participants read log, 2PC instance already been terminated

2PC Summary

- Why not subject to generals' paradox
    - DOn't come to asame decisions t the same time
    - Allowing us to reboot and continue allows time to collecting and collating decisions
- Biggest downside of 2PC: blocking
    - Failed node can prevent the system from making progress
    - Still very popular

3 Phase commit: one more phase, allows nodes to fail or block and still make progress

- Extra cost not worth

PAXOS: alt used by Google no 2PC blocking

- Leslie Lamport
- No fixed leader, can choose new leader on fly, deal with failure

What happens if one or more of nodes is malicious

- Malicious: attempting to compromise the decision making
- Byzantime agreement and blockchains

Summary

- Protocol: Agreement between two parties as to how info is transmitted
- E2E argument encourages us to keep Internet Communication simple
- Two-phase commit: distributed decision making

Week 14: Lecture 23 Networking and TCP/IP (4/21)

Internet : Goals

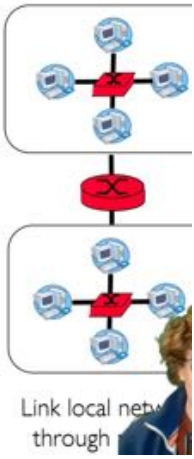
- Robust to failure
- Support multiply types of delivery services (copper , optic, wireless)
    - Should be independent
- Accommodate a variety of networks
- Allow distributed management
    - Not managed by anyway, self regulate
- Easy Host attachment
- Cost effective

Internet through Graphs

- Link to connect two machines
- Share network links
- Router links networks together

Point-to-point links

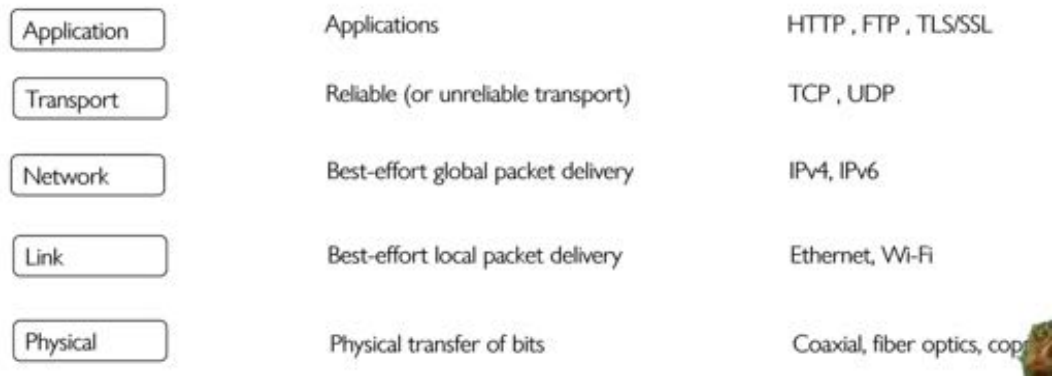
Share network links switches



Link local net through

- Many different disjoint networks routed through routers

Layers



Internet Entities

- Hosts
    - Implements all layers
    - Bits arrive on wire, must make it up to the application
- Switches
    - Implement physical and data layers
    - Transfer dat within a small network
- Routers
    - Implement physical and data layers and the network layer
    - Route packets across networks

Internet Protocol (IP)

- IP : Internet's network layer
- "best-Effort" packet delivery
    - Tries its best to deliver packet to its destination

Whats an IP (v4) address

- IP address: a 32 bit integer used as destination of IP packet, 4 dot separated integers 1-256
- Host has one or more IP addresses used for routing
- Subnet: network connecting hosts with related IP addresses
    - Subnet is identified by 32 bit value with the bits
    - Network of networks can be viewed as network of subnets
- Routers: forward each packet received on incoming link to outgoing link
    - Forwarding table: mapping between IP address and output link

Setting up Routing Tables

- Internet has no centralized state
    - No single machine knows entire topology
    - Need dynamic algorithm that acquired routing tables
        - One entry per subnet or portion of address

        - Exchange routing information with neighbouring peers
            - Inform peers of best route it knows to particular subnet

Setting up routing tables

- Sends out

Naming in the Internet

How to map human readable names to IP addresses

- www.berkeley.ed = 128.32.139.48
- IP addresses hard to memorize
- IP addresses change
    - Server 1 crashes ge

Domain Name System (DNS)

- hierarchical mechanism for naming
- Allows to retreive given hostname IP address
- Go to 13 root servers
- Top level (com, edu)
- Edu -> berkeley.edu -> eecs
- Then finds the ip address
- Queries DNS

How to implemment abstraction of communication channels from host to host

Transport Layer

- Service
    - Provide e2e communication between processes
    - Demultiplexing of communication between hosts
    - Reliability, timing, rate adaption
- Interface: send message to “specific process”
    - Named by port numbers

Internet Transport Protocols

- Datagram service (UDP) : IP Protocol 17
    - Multiplexing/demulitplexing among processes
    - No reliability, no flow control, no congestion control
- Relibaile in order deliver (TCP): IP Protocol
    - Connection and tear down
    - Discarding corrupted packets,
    - congestion control
    - Retransmission os lost packets

Reliable message Delivery: the problem

- All physical networks can garble and/or drop packets
    - Physical media: packet not transmitted/recieved
    - Congested: no place to put incoming packet
- Reliable message delievery on top of unreliable packets

TCP

- Application examples: file transfer, chat, http

TCP Service

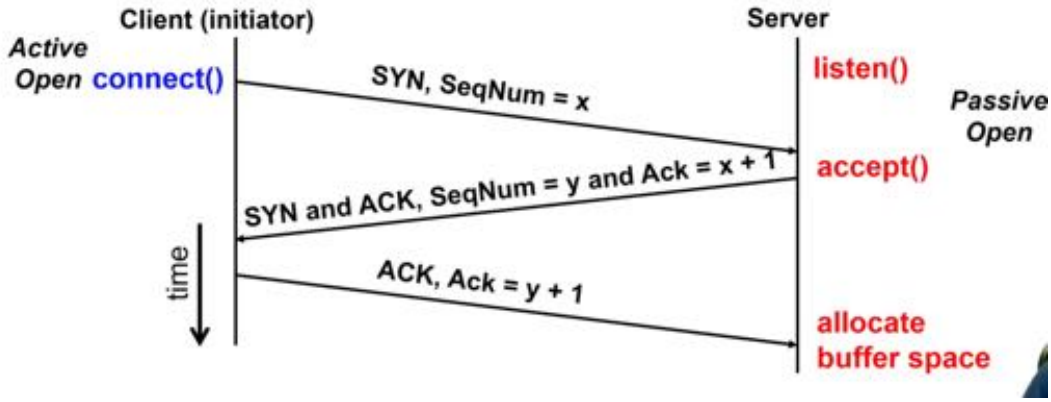
1) Open conection: 3-way handshaking
2) Reliable byte stream transfer from (IPa, TCP_port1) to (IPb, TCP_Port2)
3) Close connection

Socekt creation and conneciotn

- Provide a means for processes to communicate to other process
- Form 2 way pipes between processes

Open Connection: 3 way handshaking

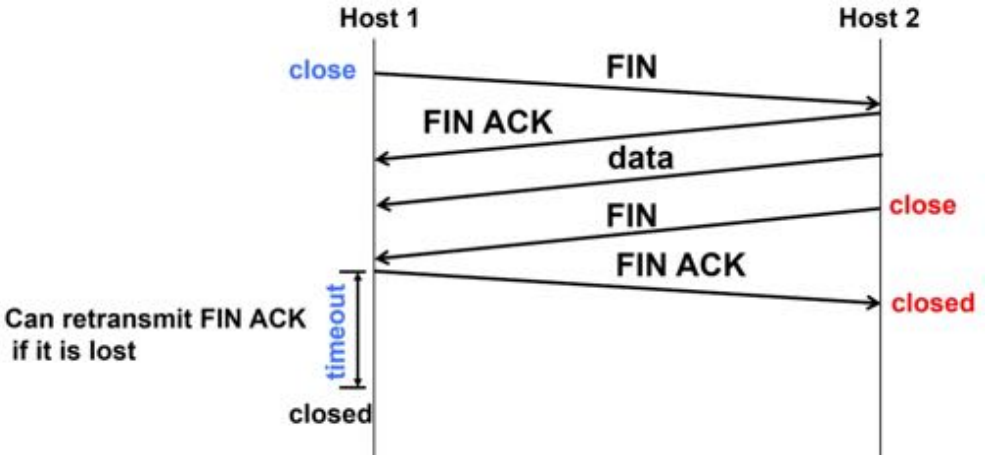
- Server waits for new connection calling listen()
- Sender call connect() passing socket which contains server's IP address and port numbers
- If enough resources, server calls accept to accep connection and sends back SYN ACK



- Why?
  - Congestion control SYN acts as cheap probe
  - Protects against delayed packets from other connection

Close connection

- Similar process

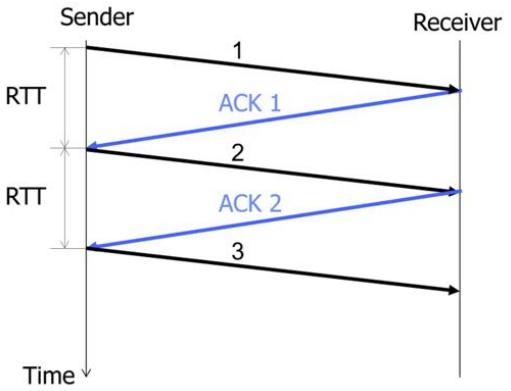


Components of a solution for reliable transports

- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgements
    - cumulative/selective
- Sequence numbers (duplicates, windows)
- Sliding windows
    - Go Back N (GBN) / Selective Replay (SR)

Detecting Packet Loss

- Timeouts
- Missing ACKs
    - Receiver ACKs each packet
- NACK: Negative ACK
    - Sends a nack specifying a packet it is missin g

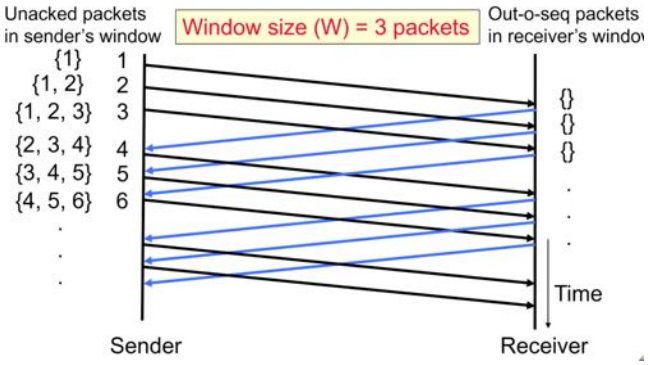


Stop & Wait w/o Errors

- Round Trip Time (RTT) : time it takes for packet to travel from sender to receiver and back
- 1 packet / RTT

Sliding Window

- Windo = set of adjacent sequence numbers
- Size of set is the window size
- Assume window size is n
- A be last ACK'd packet of sender without gap
- Sender can send packets in its window
- Reciever can accept out of sequence if in window
- Messages in quick succession
- Sliding window to cover



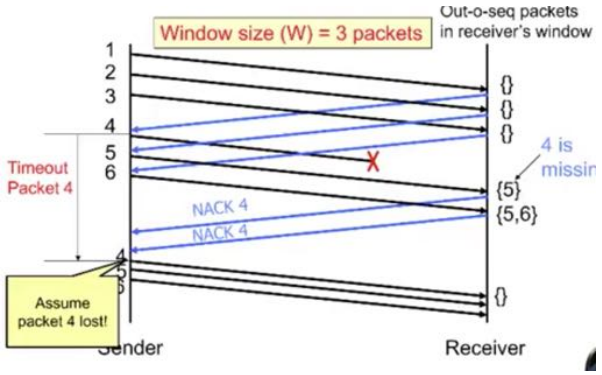
W = C* RTT/packet_size

Sliding Window with Errors

- Differe ways to deal with errors

Go-Back-n (GBN)

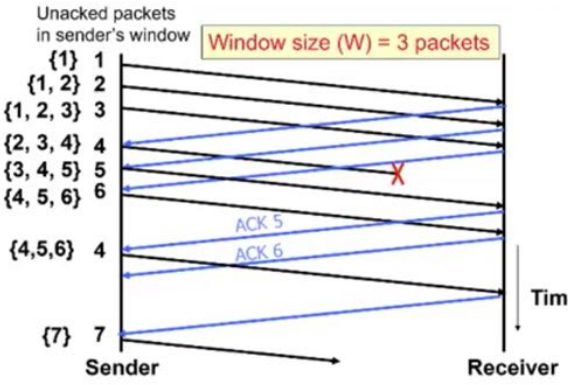
- Sender transmits up to n unacknowledged packets
- Reeiver only accpets packets in order
- Receiver uses cumulative acknolwedmgnets
- Sender sets timer for 1st outstanding ack
- If timeout, retransmit
- Discare everything if dropped



Selective Repeat (SR)

- Sendering transmits up to n unacknowledged packets
- Assume packet k is lost

- Receiver : indicates packet k + 1 correctly received
- Sender: retrasnmit only packet k on timeout
- Efficient in retransmissions but complex book-keeping



TCP

- Sequence numbers are byte offsets
- Sender and receiver maintain a sliding window
- Receiver sends cumulative acknowledgments (GBN)
- Sender mainstays a single retry timer
- Do not drop out of sequence packets
- Fast retransmit: optimization that uses duplicate ACKs to trigger early retries
- Introdu ces timeout estimation algorithms

Congestion Control

- Too much data trying to flow through some part of the newtork
- IP's solution: Drop Packets
- What ahppens to TCP Connections
- Lots of retransmissions and waiting for timeouts

How to detect congestion

- Packet delays
    - Noisy signal (delay often varies considerably)
- Router tell end hosts they're congested
- Packet loss
    - Failsail signal that TCP already has to detect
    - Non congestive loss (che ksum errors)
- Indicator of packet loss
    - No ACK after certain time interval: timeout
    - Multipley duplicate ACKs

Not All Losses the Same-

- Duplicate ACKs: isolated loss
    - Still getting packets
- Timeout more serious

How does sender adjust its sending rate

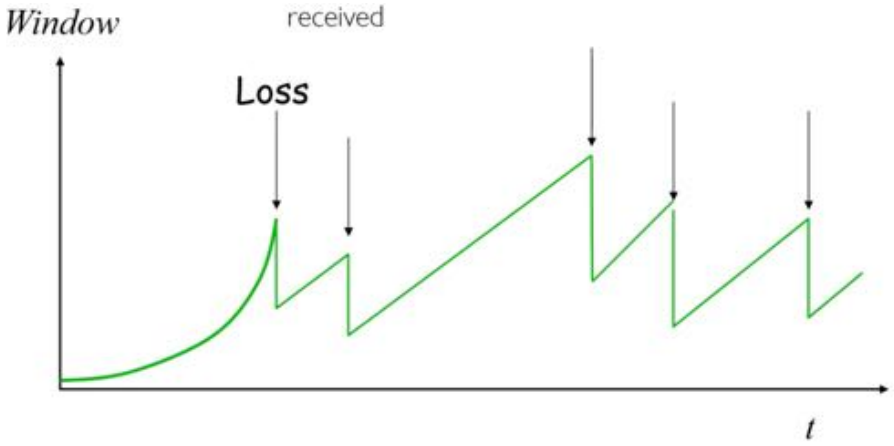
- Finding available bottlenecks
- Adjusting to bandwidth, sharing bandwidth

Rate Adjustment

- Basic Structure
    - Receipt of ACK: increase rate
    - Detection of loss: decrease rate
- Discovering available bottleneck bandwidth vs adjusting to bandwidth variations

Additive Increase, Multiplicative Decrease

- When packet dropped, cut window size in half
- If no timeouts, increase window size by C for each acnolwedgmeent received



Summary

- Internet 5 layers: Application Transport, Network, LInk, Physical
- IP layer: hourglass, routers to route packets from one end to internet through the other DNS helps resolve
- TCP in transport layer: sliding windows and acks to implement reliable delivery, Uses congestion control to reate-limit protocol

Week 14: Lecture 24 RPC, Distributed File Systems (4/23)

Distributed Applications Build with Messages

- One Abstraction: Send/receive messages

Data Representatiaon

- Object in memory has machine-specific binary representation
- Threads within a single process have the same view of wha'ts in ememory
- In absence of shared memory, turn object to sequential sequence of bytes
- Serialization/Marshalling: Express an object as a sequence of bytes
- Deserialization/Unmarshalling: reconstruct from sequence of bytes

Simple Data types

- Write x to a file, open file

1. fprintf(f, "%lu", x)
2. fwrite(&x, sizeof(uint32_t), 1, f);

- Difference in string vs bytes

Endianness

- Byte-address machine, which end of a mchine-recognized object odees its byte-address refer to
- Big Endian: address is the most-significant bits

- Little Engidan: address is the least significant bit

What Endian is the Internet

- Big Endian
    - Newtork byte order vs host type order

Dealing with Endianness

- Decide on an "on-write" endianness
- Convert from native endianness to "on-write" endinness before sending out data
- htonl, htons, ntohl, ntohs

What about recher objects

- How to write list as binary object
- Must write the content of the data structure

Data Serialization Formats

- Google Protobuffers, JSON, XML commonly used in web applications

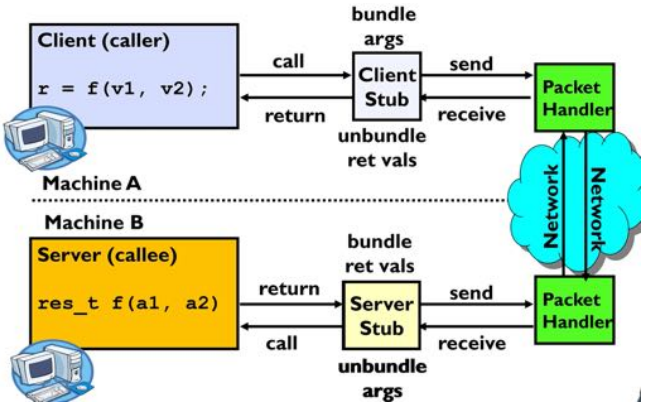
<u>Remote Procedure Call (RPC)</u>

RPC

- Raw messaging is a bit too low level
- Must wrap information into message at source
- Deal with machine representatiaon by hand

Remote Procedure Call (RPC)

- Calls a procedure on a remote machine
- Make communication look like ordinary call
- Automate all complexity of translating between represetnataions
- Client calls
    - remoteFileSystem -> Read
- Server calls
    - fileSystem -> read
- Ex: Java RMI



RPC Implmentation

- Stub provides glue on client/server
- Marshalling involves
    - Converting values to canonical form, serializing objects, copying args passed by reference

RPC Details

- Look like regular procedure call
    - Parameters <-> Request Message
    - Results <-> Reply message
    - Name of Procedure: passed in request message
    - Return Address: mbox2
- Sub generator: compiler that gneeratees stubs

        - Input: interface definitions oa interface definition language (IDL)
        - Output sub code in appropriate language

RPC Details

- Cross-platform issues
    - What if different architectures/languages
    - Convert everything to/from some canonical form
- How does client known which mbox (destinagtion quee) to send to ?
    - Translate name of remote service into network endpoint
    - Binding: process of converting a user-visible name into a network endpoint
        - Another word for naming at network level
        - Static: fixed at compile time
        - Dyanmic: done at runtime

RPC Details

- Dynamic Binding
    - Use dynamic binding via name serve
    - Access control: check whoo is permittedc to access service
    - Fail-over : if server fails, use a different one
- What if multiple servers
    - Flexibility at binding time
    - Provide smae router level redirect
        - Choose unloaded server
- Multiple clients
    - Pass pointer to client specific return mbox in request

Porblems with RPC: non-Atomic Failures

- Different failure modes
    - User level bug causes address space to crash
    - Machine failure, kernel bug causes all process on asmae machine to fail
    - Some machine is comprised by malicious party
- Before RPC: whole system would crash/die
- After RPC: one machine crases while others keep working
- Can lead to inconsistent view of the world

Problems: Performance

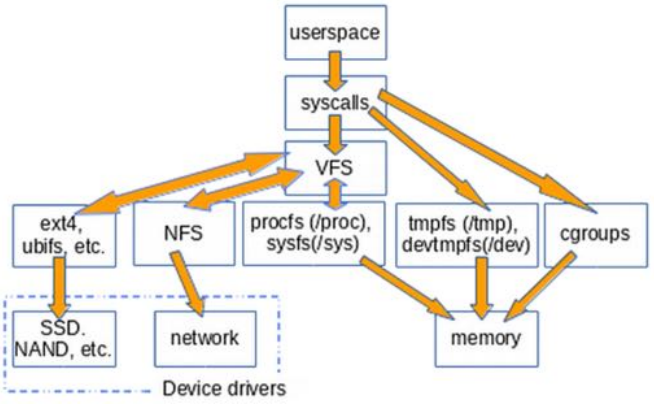
- RPC is not performance transparent
    - Cost of procedure call < same machine RPC < network RPC
    - Overheads: Marshalling, Stubs, Kernel-crossing, communication
    - Programmers aware RPC

Distirbuted FIle system

- Read and write to files not on own laptop
- Transparent access to files stored on remote disk
- Mount remote files into your local file system

- Naming choices
    - Hostname, localname: filename includes server
    - Global name space: filename unique in world

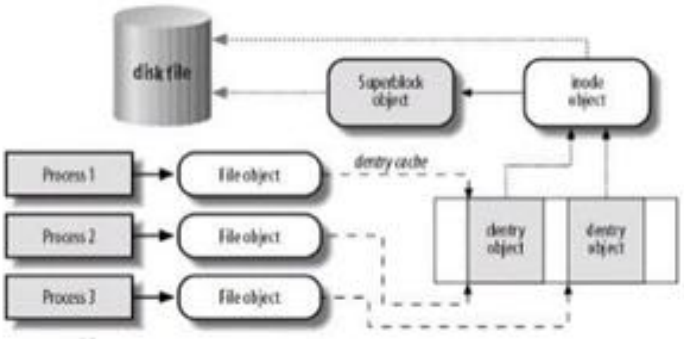
Virtual FIlesystem Switch

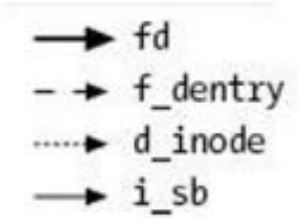


- Fully independent
- VFS: Virtual abstraction of file system
    - Virtual superblocks, indies
- Allows same system call interface (API) used for different types of file systems
    - API to VFS systems
- Pass through the VFS when making syscalls
- Abstraction for different file systems to coexist

VFS Common FIle MOdel in Linux

- Four Primary object types for VFS
    - Superblock object: specific mounted filesystem
    - Indoe object: specific file
    - Dentry object: directory entry
    - File object: open file with process
- May need to fit the model by faking it





Simple Distributed File System

- Remote Disk: Reads and writes forwarded to server
    - Use remote Procedure Calls (RPC) to trnaslate file system calls into remote requests
    - Advantage: server had consisten view of file system to multiple clients
- Going to network slower than local memory
- Server can be a bottleneck

Use caching to reduce network load

- Idea; Use caching to reduce network load
    - Use buffer cache at source and destination
- Advantage: If can be done locally, odn't need to do network traffic .. fast
- Failure:

- Client caches have data not committed at server
- Client caches not consistent with server/each other

Dealing with Failures

- What if server crashes?
  - Can client wait until it comes back and make requests
    - Changes in server's cache but not in disk are lost
- What if shared state across RPC?
  - Client opens file, server crashes
- What if client removes a file but server crashese before acknolwedgement

Stateless protocol:

- A protocol which all info required to service a request is included with the request
- Idempotent OPerations: repeating an operation multiple times is same as executing it just once
- Client: timeout expires without reply, just run the operation again
- HTTP: stateless
  - Include cookies with request to simulate a session

<u>Network FIle System (NFS)</u>

Three Layers for NFS system,

- UNIX file-system interface: open , read, write, close
- VFS layer: distinguishes local from remote files
- NFS service layer: bottom layer of architecture

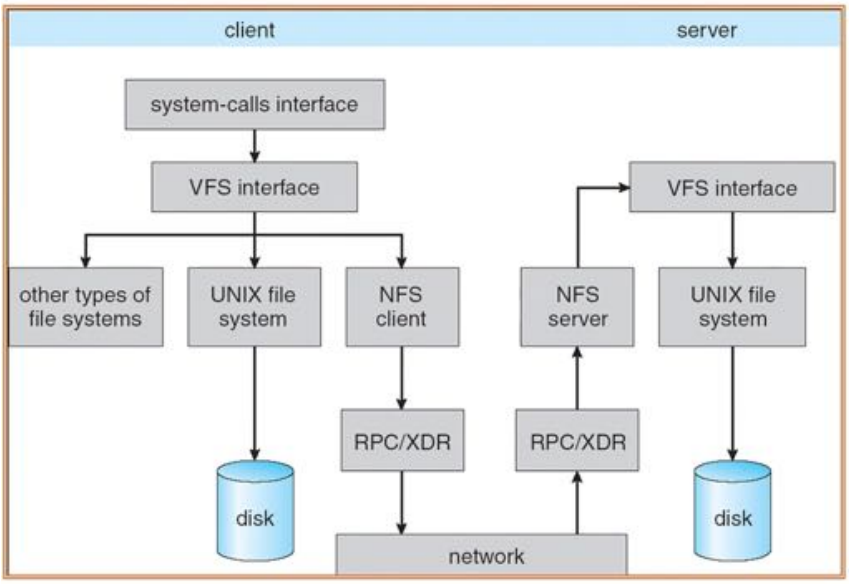
NFS Protocol : RPC for file operations on server

Write through caching: modified data committed to server's disk before results are returned to client

- Lost some advantages of caching
- Time to perform write can be long
- Mechanism to reaqders to eventually notice changes

NFS

- NFS servers are stateless: each request provides all arguments require for execution
  - Include information fo rentire operation
- Idempotent: performing requests multiple times has same effect as performing them exactly once
- Failure Model: transp arent to client system
  - Is good idea? NFS provides
    - Hang until server comes back up
    - Return an error
- Program defensively



NFS Cache Consistency

- NFS protocol: weak consistency
- Client polls server periodically to check for changes
- What if multiple clients write to smae file
    - NFS, can get either version

Sequential Ordering Constraints

- Behave as if running on centralized system
- Actions of sequential program correct
    - If read finishes before write states, get old copy
    - If read starts after write finishes, get new copy
    - Otherwise, get either new or old copy
- Read starts more than 30 secs after write, get new copy; otherwise could get

NFS Pros:

- Simple, Highly Portable

NFS Cons:

- Sometimes inconsistent!
- Doesn't scale to large # clients
    - Must keep checking to see if caches out of date
    - Server becomes bottleneck due to polling traffic

Summary

- Message passing and challenges of serialization/seserialization
- Remote procedure Calls: Abstraction of local computation on remote machiens
- Distributed File Systems using VFS
    - NFS; weak consistency but efficient
    -