B16 Operating Systems

Lecture 1 : History and Overview

Material from

Operating Systems in Depth

(spec. Chapter 1)

by

Thomas Doeppner

GET THIS BOOK AND READ IT!

What is an operating system?

- Operating systems provide software abstracts of
  - Processors
  - RAM (physical memory)
  - Disks (secondary storage)
  - Network interfaces
  - Display
  - Keyboards
  - Mice
- Operating systems allow for sharing
- Operating systems typically provide abstractions for
  - Processes
  - Files
  - Sockets

Why should we study operating systems?

- "To a certain extent [building an operating system is] a solved problem" – Doeppner
- "So too is bridge building" – Wood
  - History and its lessons
    - Capacity and correct usage
      - Takoma Narrows equivalent?
  - Improvement possible
    - New algorithms, new storage media, new peripherals
    - New concerns : security
    - New paradigms : the "cloud"

Review : Computer ≈ Von Neumann Architecture

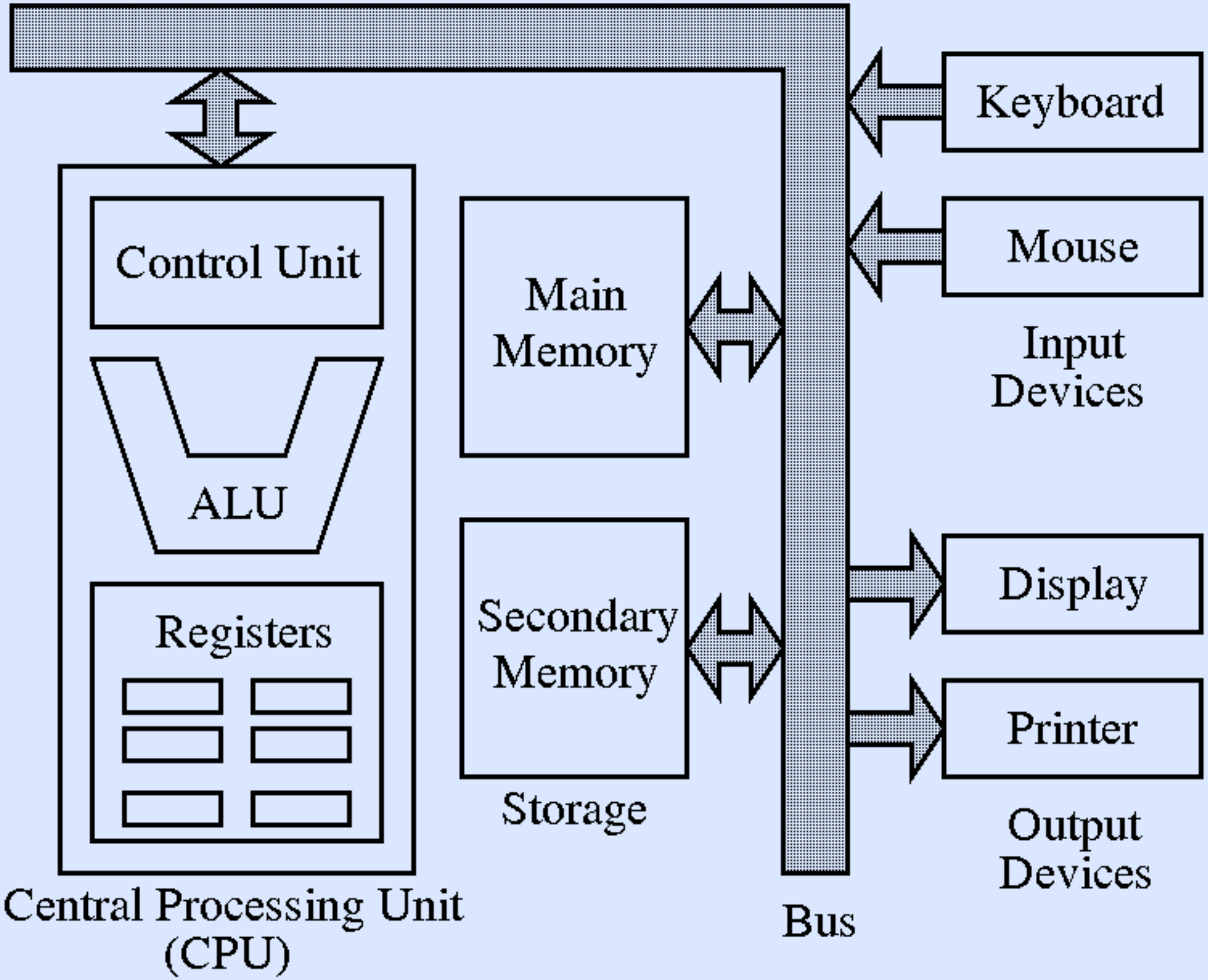


Image from http://cse.iitkgp.ac.in/pds/notes/intro.html

Review : Machine Instructions and Assembly Code

- Machine code : instructions directly executed by the CPU
  - From Wikipedia :
    - "the instruction below tells an x86/IA-32 processor to move an immediate 8-bit value into a register. The binary code for this instruction is 10110 followed by a 3-bit identifier for which register to use. The identifier for the AL register is 000, so the following machine code loads the AL register with the data 01100001."

```
10110000 01100001
```

- Assembly language : one-to-one mapping to machine code (nearly)
  - Mnemonics map directly to instructions (MOV AL = 10110 000)
  - From Wikipedia :
    - "Move a copy of the following value into AL, and 61 is a hexadecimal representation of the value 01100001"

```
MOV AL, 61h       ; Load AL with 97 decimal (61 hex)
```

Compilation and Linking

- A compiler is a computer program that transforms source code written in a programming language into another computer language
  - Examples : GNU compiler collection
- A linker takes one or more object files generated by a compiler and combines them into a single executable program
  - Gathers libraries, resolving symbols as it goes
  - Arranges objects in a program's address space

- Touches OS through libraries, virtual memory, program address space definitions, etc.
  - Modern OS' provide dynamic linking; runtime resolution of unresolved symbols

History : 1950's

- Earliest computers had no operating systems
- 1954 : OS for MIT's "Whirlwind" computer
  - Manage reading of paper tapes avoiding human intervention
- 1956 : OS General Motors
  - Automated tape loading for an IBM 701 for sharing computer in 15 minute time allocations
- 1959 : "Time Sharing in Large Fast Computers"
  - Described multi-programming
- 1959 : McCarthy MIT-internal memo described "time-share" usage of IBM 7090
  - Modern : interactive computing by multiple concurrent users

Early OS Designs

- Batch systems
  - Facilitated running multiple jobs sequentially
- I/O bottlenecks
  - Computation stopped to for I/O operations
- Interrupts invented
  - Allows notification of an asynchronous operation completion
  - First machine with interrupts : DYSEAC 1954, standard soon thereafter
- Multi-programming followed
  - With interrupts, computation can take place concurrently with I/O
  - When one program does I/O another can be computing
  - Second generation OS's were batch systems that supported multi-programming

History : 1960's, the golden age of OS R&D

- Terminology
  - "Core" memory refers to magnetic cores each holding one bit (primary)
  - Disks and drums (secondary)
- 1962 : Atlas computer (Manchester)
  - "virtual memory" : programs were written as if machine had lots of primary storage and the OS shuffled data to and from secondary
- 1962 : Compatible time-sharing system (CTSS, MIT)
  - Helped prove sensibility of time-sharing (3 concurrent users)
- 1964 : Multics (GE, MIT, Bell labs; 1970 Honeywell)
  - Stated desiderata
    - Convenient remote terminal access
    - Continuous operation
    - Reliable storage (file system)
    - Selective sharing of information (access control / security)
    - Support for heterogeneous programming and user environments
  - Key conceptual breakthrough : unification of file and virtual memory via *everything is a file*

History : 1960's and 1970's

- IBM Mainframes OS/360
- DEC PDP-8/11
  - Small, purchasable for research
- 1969 : UNIX
  - Ken Thompson and Dennis Ritchie; Multics effort drop-outs
  - Written in C
  - 1975 : 6th edition released to universities very inexpensively
  - 1988 System V Release 4
- 1996 : BSD (Berkeley software distribution) v4.4
  - Born from UNIX via DEC VAX-11/780 and virtual memory

1980's : Rise of the Personal Computer (PC)

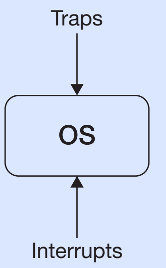
- 1970's : CP/M
  - One application at a time – no protection from application
  - Three components
    - Console command process (CCP)
    - Basic disk operating system (BDOS)
    - Basic input/output system (BIOS)
- Apple DOS (after CP/M)
  - 1978 Apple DOS 3.1 ≈ CP/M
- Microsoft
  - 1975 : Basic interpreter
  - 1979 : Licensed 7-th edition Unix from AT&T, named it Xenix
  - 1980 : Microsoft sells OS to IBM and buys QDOS (no Unix royalties) to fulfill
    - QDOS = "Quick and dirty OS"
    - Called PC-DOS for IBM, MS-DOS licensed by Microsoft

1980's 'til now.

- Early 80's state of affairs
  - Minicomputer OS's
    - Virtual memory
    - Multi-tasking
    - Access control for file-systems
  - PC OS's
    - None of the above (roughly speaking)
- Workstations
  - Sun (SunOS, Bill Joy, Berkeley 4.2 BSD)
    - 1984 : Network file system (NFS)
- 1985 : Microsoft Windows
  - 1.0 : application in MS-DOS
    - Allowed cooperative multi-tasking, where applications explicitly yield the processor to each other
- 1995 : Windows '95 to ME
  - Preemptive multi-tasking (time-slicing), virtual memory (-ish), unprotected OS-space
- 1993 : First release of Windows NT, subsequent Windows OS's based on NT
- 1991 : Linus Torvalds ported Minix to x86

Teaching example : Simple OS

- Based on Unix (6th edition)
  - Monolithic
    - The OS is a single file loaded into memory at boot time
  - Interfaces
    - *Traps* originate from user programs
    - *Interrupts* originate from external devices
  - Modes
    - User
    - Privileged / System
  - Kernel
    - A subset of the OS that runs in privileged mode
    - Or a subset of this subset



Traps and System Calls

- Unintended requests for kernel service
  - Using a bad address
  - Dividing by zero
- *System calls*
  - Example

```
if (write(FileDescriptor, BufferAddress, BufferLength) == -1) {
   /* an error has occurred: do something appropriate */
   printf("error: %d\n", errno) /* print error message */
}
```

requests the OS to send data to a file

Interrupts

- Request from an external device for a response from the processor
  - Handled independently of any program
- Examples
  - Keyboard input
  - Data available

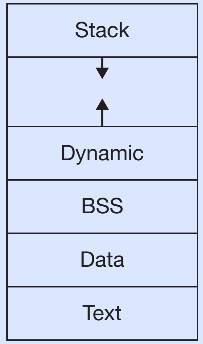
Processes

- Abstraction that includes
  - Address space (memory)
  - Processors (threads of control)
- Usually disjoint
  - Processes usually cannot directly access each other's memory
    - Parallel processing?
- Running a program
  - Creates a "process"
  - Program is loaded from a file into the process's address space
  - Process's single thread of control then executes the program's code

Address Space

- Text
  - Program code
- Data
  - Initialized global variables
- BSS (block started by symbol)
  - Uninitialized global variables
- Dynamic (Heap)
  - Dynamically allocated storage
- Stack (grows “downward”)
  - Local variables

- Which variables go where?
- Where does malloc() claim space?




```
const int nprimes = 100;
int prime[nprimes];
int main() {
  int i;
  int current = 2;
  prime[0] = current;
  for (i=1; i<nprimes; i++) {
    int j;
  NewCandidate:
    current++;
    for (j=0; prime[j]*prime[j] <= current; j++) {
     if (current % prime[j] == 0)
      goto NewCandidate;
    }
    prime[i] = current;
  }
  return(0);
}
```

Processes and Threads

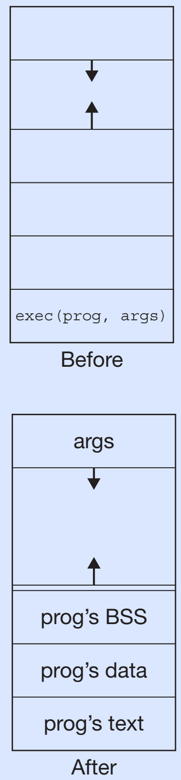
- Processes are created via the system call fork()
  - Any exact copy of the calling process is made (Is this efficient?)
  - fork() returns twice
    - Once in the child (return value 0)
    - Once in the parent (return value the PID of the child process)
- Processes terminate via the system call exit(*ret_code*)
- Processes can wait() for the termination of child processes

```
short pid;
if ((pid = fork()) == 0) {
   /* some code is here for the child to execute */
   exit(n);
} else {
   int ReturnCode;
   while(pid != wait(&ReturnCode))
      ;
   /* the child has terminated with ReturnCode as its
      return code */
}
```

Loading Programs into Processes

- exec() system call used to do this

```
int pid;
if ((pid = fork()) == 0) {
   /* we'll soon discuss what might take place before exec
      is called */
   execl("/home/twd/bin/primes", "primes", "300", 0);
   exit(1);
}

/* parent continues here */

while(pid != wait(0))  /* ignore the return code */
   ;
```



- exec() replaces the entire contents of the processes address space
  - the stack is intialized with the passed program args.
  - a special start routine is called that itself calls main()
  - exec doesn't return except if there is an error!

Files

- Files are Unix's *primary abstraction* for places outside processes
  - Keyboard
  - Display
  - Other processes
- Naming files
  - Filesystems generally are tree-structured directory systems
  - Name spaces are generally shared by all processes
- Accessing files
  - The directory-system name-space is outside the process
    - open(name) returns a file *handle*, read(args)
    - OS checks permissions along path

```
int fd;
char buffer[1024];
int count;
if ((fd = open("/home/twd/file", O_RDWR) == -1) {
   /* the file couldn't be opened */
   perror("/home/twd/file");
   exit(1);
}
```

```
if ((count = read(fd, buffer, 1024)) == -1) {
   /* the read failed */
   perror("read");
   exit(1);
}
/* buffer now contains count bytes read from the file */
```

Using File Descriptors

- File descriptors survive exec()'s
- Default file descriptors
  - 0 read (keyboard)
  - 1 write (primary, display)
  - 2 error (display)
- Different associations can be established before fork()

```
if (fork() == 0) {
   /* set up file descriptor 1 in the child process */
   close(1);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      perror("/home/twd/Output");
      exit(1);
}
   execl("/home/twd/bin/primes", "primes", "300", 0);
   exit(1);
}

/* parent continues here */

while(pid != wait(0))  /* ignore the return code */
   ;
```

File Random Access

- lseek() provides non-sequential access to files

```
fd = open("textfile", O_RDONLY);
/* go to last char in file */
fptr = lseek(fd, (off_t)-1, SEEK_END);
while (fptr != -1) {
   read(fd, buf, 1);
   write(1, buf, 1);
   fptr = lseek(fd, (off_t)-2, SEEK_CUR);
}
```

Pipes

- A pipe is a means for one process to send data to another directly
- pipe() returns two nameless file descriptors

```
int p[2];    /* array to hold pipe’s file descriptors */
pipe(p);     /* create a pipe; assume no errors */
   /* p[0] refers to the output end of the pipe */
   /* p[1] refers to the input end of the pipe */
if (fork() == 0) {
   char buf[80];
   close(p[1]);     /* not needed by the child */
   while (read(p[0], buf, 80) > 0) {
      /* use data obtained from parent */
      …
   }
} else {
   char buf[80];
   close(p[0]);        /* not needed by the parent */
   for (;;) {
      /* prepare data for child */
      …
      write(p[1], buf, 80);
   }
}
```

Directories

- A directory is a file that is interpreted as containing references to other files by the OS
- Consists of an array of
  - Component name
  - inode number
    - an inode is a datastructure maintained by the OS to represent a file

Component name	Inode number

Directory entry

.	1
..	1
unix	117
etc	4
home	18
pro	36
dev	93

Creating Files

- creat() and open() (with flags) are used to create files

Review : Simple OS

- Rough idea of what goes inside an OS
- Traps / system calls
  - exec()
  - fork()
  - open()
  - pipe()
  - exit()
  - close()
  - read()
  - write()
  - dup()
  - …
- Next lecture : basics. Last two lectures : OS specific issues

Lecture 2 : Basics; Processes, Threads, …

Material from

Operating Systems in Depth

(spec. Chapters 2&3)

by

Thomas Doeppner

GET THIS BOOK AND READ IT!

Threads

- What is a thread?
  - Mechanism for concurrency in user-level programs
  - "Lightweight process"
  - Process within a process
  - Share process memory with other threads

- Why threads?
  - Can dramatically simplify code
    - Multi-threaded database concurrently handling requests
    - Server listening on a socket responding to client requests
  - Requires care
    - Synchronization
- POSIX ("portable operating system interface") specification

Thread Creation

```
void start_servers( ) {
   pthread_t thread;
   int i;

   for (i=0; i<nr_of_server_threads; i++)
      pthread_create(
               &thread,         // thread ID
               0,               // default attributes
               server,          // start routine
               argument);       // argument
}

void *server(void *arg) {
   // perform service
   return (0);
}
```

- Alternative specifications exist; all conceptually similar

Passing Multiple Arguments to Threads

- Care must be taken with threads
- Consider

```
typedef struct {
  int first, second;
} two_ints_t;


void rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;
  two_ints_t in={r_in, l_out}, out={l_in, r_out};
  pthread_create(&in_thread,
      0,
      incoming,
      &in);
  pthread_create(&out_thread,
      0,
      outgoing,
      &out);
}
```

- What could go wrong here?
  - Hint : in and out are local variables
  - Solution?

Thread Termination

- Space from caller must be provided for thread to place return values

```
pthread_create(&createe, 0, CreateeProc, 0);
…
pthread_join(create, &result);
…
```

- pthread_exit() terminates thread, exit() terminates process

```
void *CreateeProc(void *arg) {
   …
   if (should_terminate_now)
      pthread_exit((void *)1);
   …
   return((void *)2);
}
```

Thread Attributes

- To specify the stack size for a thread one initializes an attributes datastructure

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 20*1024*1024);

...

pthread_create(&thread, &thr_attr, startroutine, arg);
```

Synchronization

- Remember: threads share access to common data structures
- Mutual exclusion is a form of thread synchronization
  - Makes sure two things don't happen at once
  - Example, two threads each doing

```
x = x+1;
```

Can result in 1 or 2; reordering the assembly code shows why

```
ld      r1,x
add     r1,1
st      r1,x
```

POSIX Thread Solution

- OS must support thread synchronization mechanisms
- POSIX defines a data type called a *mutex* (from "mutual exclusion")
- Mutexes can ensure
  - Only one thread is executing a block of code (code locking)
  - Only one thread is accessing a particular data structure (data locking)
- A mutex either belongs to a single thread or no thread
- A thread may "lock" a mutex by calling pthread_mutex_lock()
- A mutex may be unlocked by calling pthread_mutex_unlock()
- A mutex datastructure can be initialized via pthread_mutex_init()

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
            // shared by both threads
int x;      // ditto

   pthread_mutex_lock(&m);

   x = x+1;

   pthread_mutex_unlock(&m);
```

Mutually exclusive access to multiple datastructures

- In the following, "deadlock" can occur

```
void proc1( ) {
   pthread_mutex_lock(&m1);
   /* use object 1 */
   pthread_mutex_lock(&m2);
   /* use objects 1 and 2 */
   pthread_mutex_unlock(&m2);
   pthread_mutex_unlock(&m1);
}
```

```
void proc2( ) {
   pthread_mutex_lock(&m2);
   /* use object 2 */
   pthread_mutex_lock(&m1);
   /* use objects 1 and 2 */
   pthread_mutex_unlock(&m1);
   pthread_mutex_unlock(&m2);
}
```

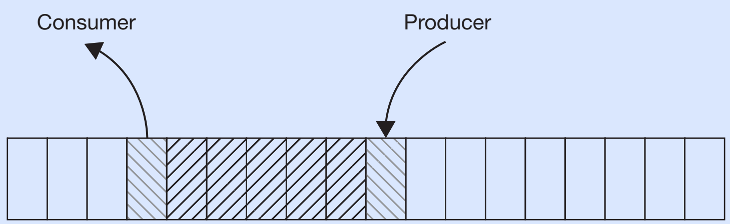
- Deadlock is nasty, difficult to detect, and to be avoided at all cost
  - One useful mechanism is pthread_mutex_trylock()

```
proc1( ) {
 pthread_mutex_lock(&m1);
 /* use object 1 */
 pthread_mutex_lock(&m2);
 /* use objects 1 and 2 */
 pthread_mutex_unlock(&m2);
 pthread_mutex_unlock(&m1);
}
```

```
proc2( ) {
 while (1) {
  pthread_mutex_lock(&m2);
  if (!pthread_mutex_trylock(&m1))
   break;
  pthread_mutex_unlock(&m2);
 }

  /* use objects 1 and 2 */

 pthread_mutex_unlock(&m1);
 pthread_mutex_unlock(&m2);
}
```

Producer-Consumer Problem

- Buffer with a finite number of slots
- Threads
  - Producer : puts things in the buffer
  - Consumer : removes things from the buffer
- Producer must wait if buffer is full; consumer if buffer is empty



Semaphores

- A semaphore is a nonnegative integer with two atomic operations
  - P (try to decrease) : thread waits until semaphore is positive then subtracts 1
    - []'s are notation for guards; that which happens between them is atomic, instantaneous, and no other operation that might take interfere with it can take place while it is executing

```
when (semaphore > 0) [
   semaphore = semaphore – 1;
]
```

  - V (increase)

```
[semaphore = semaphore + 1]
```

- Mutexes can be implemented as semaphores

```
semaphore S = 1;
void OneAtATime( ) {
   P(S);
   …
   /* code executed mutually exclusively */
   …
   V(S);
}
```

POSIX Semaphores

- Interface

```
sem_t semaphore;
int err;

err = sem_init(&semaphore, pshared, init);
err = sem_destroy(&semaphore);
err = sem_wait(&semaphore);            // P operation
err = sem_trywait(&semaphore);         // conditional P operation
err = sem_post(&semaphore);            // V operation
```

- Note : Mac’s use Mach spec. named-semaphore via sem_open()

Deviations

- Signals
  - Force a thread to put aside current activity
  - Call a pre-arranged handler
  - Go back to what it was doing
  - Similar to interrupt handling in the OS
- Examples
  - Typing special characters on the keyboard (^c)
  - Signals sent by other threads (kill)
  - Program exceptions (divide by zero, addressing exceptions)
- Background
  - Graceful termination via ^c and SIGINT

Signals and Handled by Handlers

- Setting up a handler to be invoked upon receipt of a signal

```
int main( ) {
   void handler(int);
   sigset(SIGINT, handler);

   /* long-running buggy code */
   …
}

void handler(int sig) {
   /* perform some cleanup actions */
   …
   exit(1);
}
```

- Signals can be used to communicate with a process

Async-signal safe routines

- Signals are processed by the single thread of execution
- Communication at right not problem-free because of asynchronous access to state
- Mutex use will result in deadlock
- Making routines async-signal safe requires making them so that the controlling thread cannot be interrupted by a signal at certain times (i.e. in update_state)
  - Signal handling turned on and off by
    - sigemptyset()
    - sigaddset()
    - Sigprocmask()
- POSIX compliant OS's implement 60+ async-signal safe routines

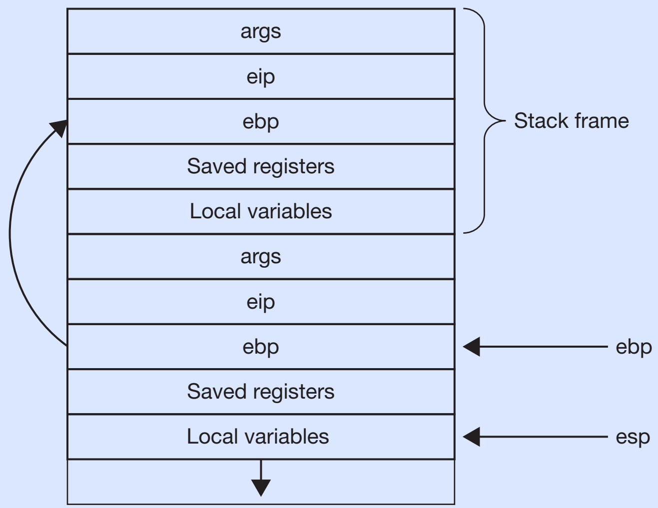
```
computation_state_t state;

int main( ) {
   void handler(int);

   sigset(SIGINT, handler);

   long_running_procedure( );
}

long_running_procedure( ) {
   while (a_long_time) {
      update_state(&state);
      compute_more( );
   }
}

void handler(int sig) {
   display(&state);
}
```

Other Basics

- Context switching
  - Stack frames
  - System calls
  - Interrupts
- I/O
- Dynamic Storage Allocation
  - Best-fit, first-fit
- Linking and loading
- Booting

Stack frames

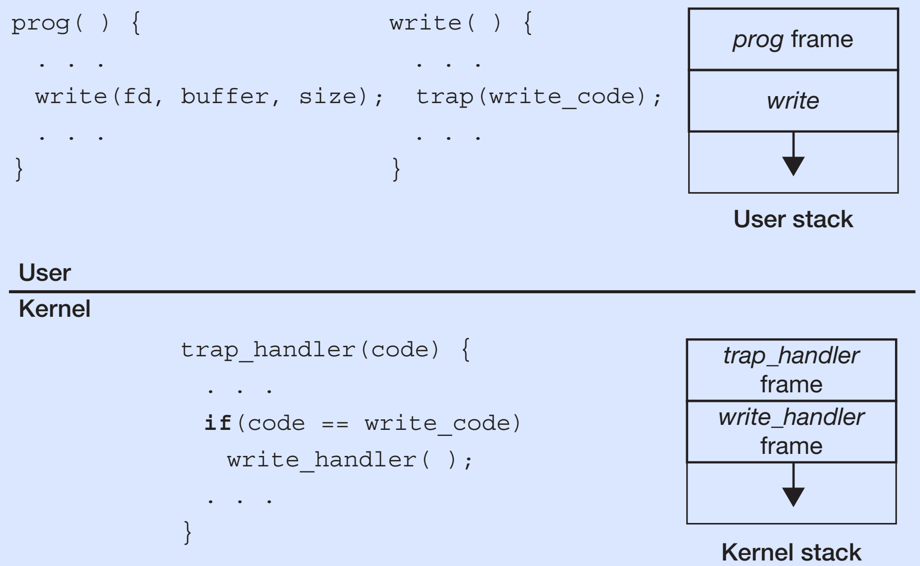
- "Context" is the setting in which execution is currently taking place
  - Processor mode
  - Address space
  - Register contents
  - Thread or interrupt state



- Intel x86 Stack Frames
  - Subroutine context
    - Instruction pointer (reg. eip)
      - Address to which control should return when subroutine is complete
    - Frame pointer (reg. ebp)
      - Link to stack frame of caller

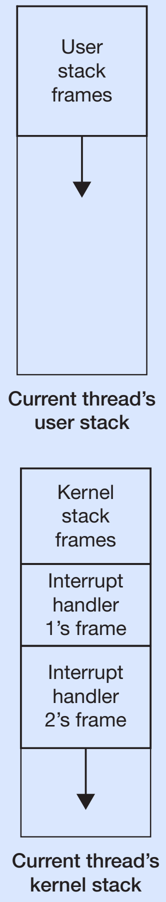
System calls

- Transfer control from user to system code and back
  - Does not involve switch in thread
  - Typically uses a kernel stack frame



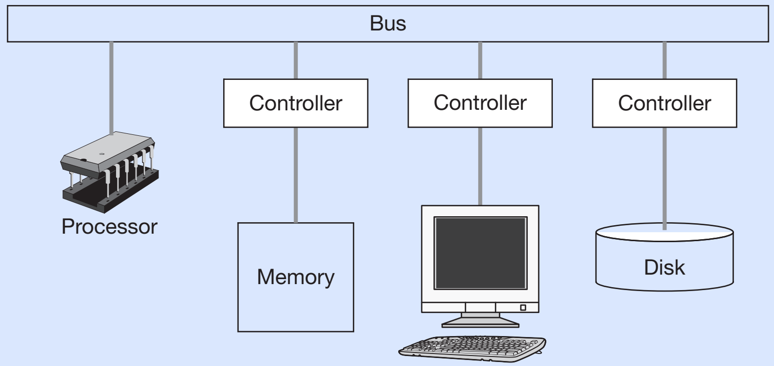
Interrupts

- On interrupt
  - Processor
    - Puts aside current context
    - Switches to interrupt context
- Interrupts require stacks
  - OS's differ
  - Common choice : kernel stack



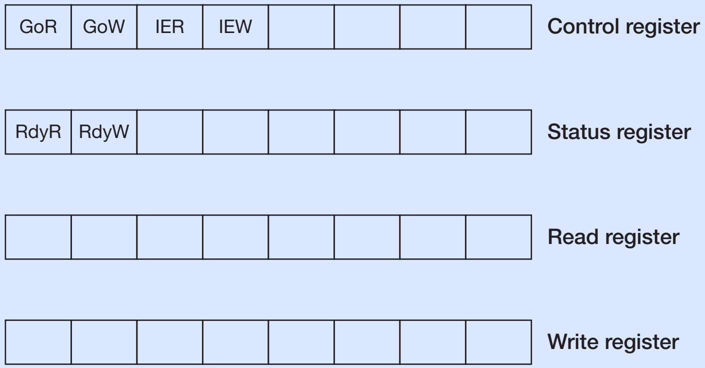
I/O Architecture (Simplified)

- Memory-mapped
  - Each device has a controller
  - Each controller has registers
  - Registers appear to processor as physical memory
  - Actually attached via a bus
- Categories of I/O devices
  - Programmed I/O (PIO)
    - One word per read/write
    - e.g. terminal
  - Direct memory access (DMA)
    - Controller directly manipulates physical memory in location specified by processor
    - e.g. disk



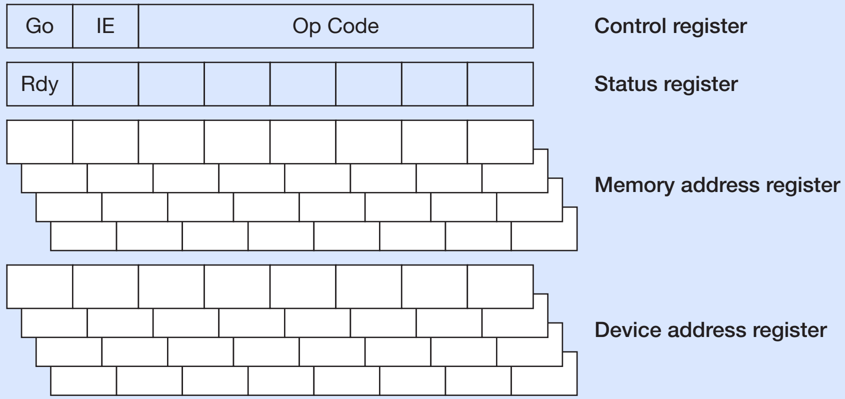
PIO and DMA Example

PIO



Legend:		
	GoR	Go read (start a read operation)
	GoW	Go write (start a write operation)
	IER	Enable read-completion interrupts
	IEW	Enable write-completion interrupts
	RdyR	Ready to read
	RdyW	Ready to write

DMA

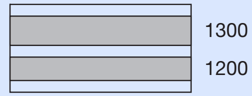


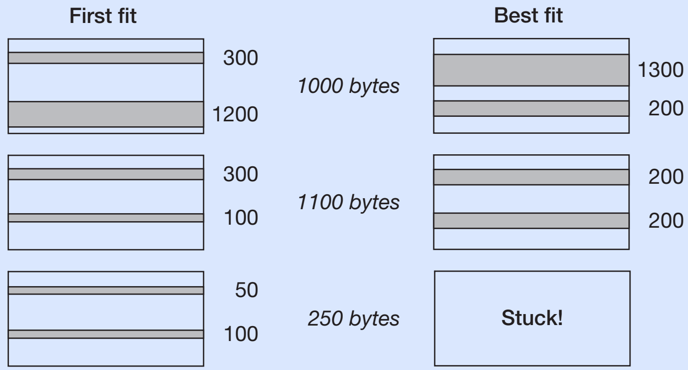
Legend:		
	Go	Start an operation
	Op Code	Operation code (identifies the operation)
	IE	Enable interrupts
	Rdy	Controller is ready

Dynamic Storage Allocation

- Storage allocation is very important in OS's
  - Disk
  - Memory
- Approaches
  - First-fit
  - Best-fit
  - Knuth simulations revealed first-fit was best
    - Intuition : best-fit leaves too many small gaps
- Others
  - Buddy
  - Slab

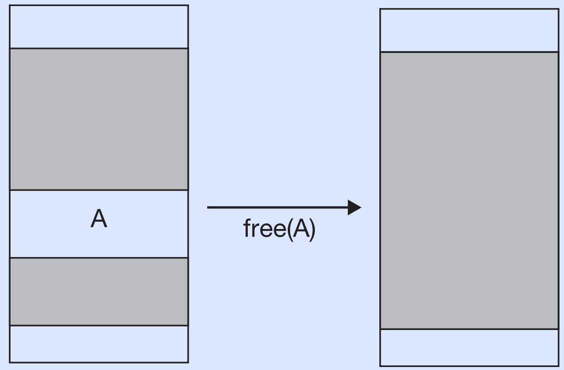
Pool of Free storage





Freeing Storage Is More Complex

- Would like to combine free segments
- Requires datastructure that represents free or not-free
- Knuth provided boundary-tag method and algorithm
- Fragmentation
  - External
    - Free spaces too small
  - Internal
    - Allocated memory unnecessarily too large (different allocation approaches)

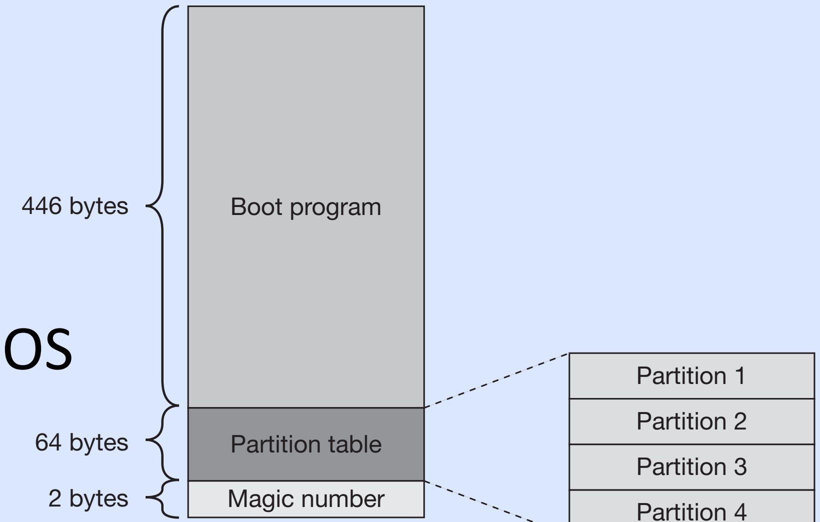


Linking and loading

- ld links and relocates code by resolving addresses of variables and procedures
- Shared libraries require mechanisms that delay linking until run-time
- Loading requires setting up address space then calling main

Booting

- Thought to be derived from “to pull yourself up by your bootstraps”
- Modern computers boot from BIOS read only memory (ROM)
  - Last 64K of the first MB of address space
- When the computer is powered on it starts executing instructions at 0xffff0
- Looks for a boot device
  - Loads a master boot record (MBR)
    - Cylinder 0, head 0, sector 1 (hard disc)
- Loads boot program
- Transfers control to boot program
- Boot progam (lilo, grub, etc.) loads OS
- Transfers control



Master boot record

Review

- OS building blocks
  - Threads and how they are implemented
  - Multi-threaded programming within OS's
  - Context switching for management of processors
  - I/O for file systems
  - Dynamic storage allocation

Lecture 3 : Processor & Memory Management

Material from

Operating Systems in Depth

(spec. Chapters 5 and 7)

by

Thomas Doeppner

GET THIS BOOK AND READ IT!

Threads Implementations

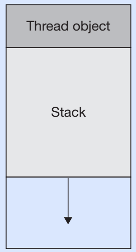
- OS goal is to support user-level application programs
- Design issues related to thread support
  - Scheduling
  - Synchronization
- In kernel or out of kernel?
  - One-level model
  - Two-level model

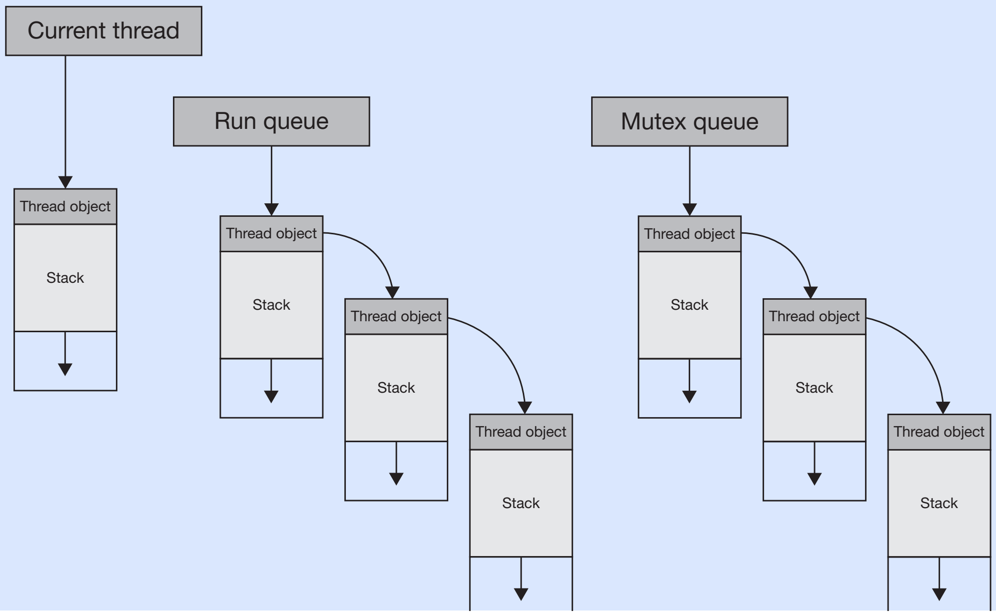
Strategies

- One-level model
  - Each user thread is mapped to a kernel thread
- Two-level model
  - Single kernel thread
    - Each process gets one kernel thread
    - Threads multiplexed on this kernel thread
    - Synchronization via thread queues
    - Disadvantage : if any thread calls blocking system call (i.e. read()) all threads stop
  - Multiple kernel threads
    - Many kernel threads. User-level threads distributed across them
    - Avoids blocking problem of single-kernel thread model
- Other approaches exist (scheduler activations,…)

Example, Simple Threads Implementation

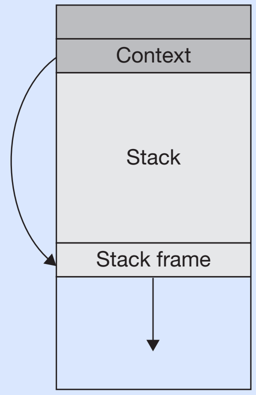
- User-level thread package “straight-threads implementation”
  - One processor
  - No interrupts
- Need
  - Thread object
  - Current thread (global var)
  - Run queue (threads waiting to run)
  - Mutex queue of threads waiting to lock (one for every mutex)





Yielding the Processor

- Straight-threads voluntarily yield
- Switching contexts requires
  - Context
  - Stack frame (thread's current register state, …)




```
void thread_switch( ) {
   thread_t NextThread, OldCurrent;

   NextThread = dequeue(RunQueue);
   OldCurrent = CurrentThread;
   CurrentThread = NextThread;
   swapcontext(&OldCurrent->context, &NextThread->context);

   // We're now in the new thread's context

}
```

Implementing Mutexes

- Because the simple straight-threads system does not have interrupts and all threads run until voluntarily yielding, mutex_lock doesn't need to do anything special to make its action atomic

```
void mutex_lock(mutex_t *m) {
  if (m->locked) {
    enqueue(m->queue, CurrentThread);
    thread_switch();
  } else
    m->locked = 1;
}

void mutex_unlock(mutex_t *m) {
  if (queue_empty(m->queue))
    m->locked = 0;
  else
    enqueue(runqueue, dequeue(m->queue));
}
```

Multiple Processors

- thread_switch() insufficient
- Simple approach : special idle threads, one for each processor

```
void idle_thread() {
  while(1)
    thread_switch();
}
```

- Actual concurrent threads require actual synchronization
  - more complicated
  - has big OS performance impact
- Types
  - Spin lock (hardware supported)
  - Futexes

Spin-locks

- Operation provided by some processors (e.g. x86) with hardware guaranteed atomicity (compare and swap)

```
int CAS(int *ptr, int old, int new) {
   int tmp = *ptr;
   if (*ptr == old)
      *ptr = new
   return tmp;
}
```

- With CAS spin-locks can be implemented
  - Zero value means unlocked

```
void spin_lock(int *mutex) {
   while(!CAS(mutex, 0, 1))
      ;
}

void spin_unlock(int *mutex) {
   *mutex = 0;
}
```

Faster Spinlock

- Providing atomicity guarantees slows down processors
- Unsafe checks result in overall speedup

```
void spin_lock(int *mutex) {
  while (1) {
    if (*mutex == 0) {
      // the mutex was at least momentarily unlocked
      if (!CAS(mutex, 0, 1)
        break; // we have locked the mutex
      // some other thread beat us to it, so try again
    }
  }
}
```

Blocking Lock

- Spin-locks consume processor resource
- Blocking locks work as before – threads waiting for lock queue

```
void blocking_lock(mutex_t *mut) {
  spin_lock(mut->spinlock);
  if (mut->holder != 0)
    enqueue(mut->wait_queue, CurrentThread);
    spin_unlock(mut->spinlock);
    thread_switch();
  } else {
    mut->holder = CurrentThread;
    spin_unlock(mut->spinlock);
  }
}
```

```
void blocking_unlock(mutex_t *mut) {
  spin_lock(mut->spinlock);
  if (queue_empty(mut->wait_queue)) {
      mut->holder = 0;
  } else {
      mut->holder = dequeue(mut->wait_queue);
      enqueue(RunQueue, mut->holder);
  }
  spin_unlock(mut->spinlock);
}
```

- Use of spin-lock prevents collisions on mut->holder
  - e.g. holder unlocking at exact instance empty queue is being joined
- There is still a *very* subtle problem (exercise)

Interrupts

- Processors usually run in thread contexts
- Interrupts are handled in interrupt contexts
- Interrupts typically (varies from one arch and OS to another) borrow stacks
  - Note; x86 hardware saves registers
- Signal handlers are similar to interrupts
- Interrupts preempt the execution of normal threads
  - Interrupts are used for scheduling
- Interrupts can have priorities
- Interrupts can be masked
  - Interrupt processing can prohibit interruption from other interrupts

Synchronization and Interrupts

- Access to kernel datastructures must be carefully synchronized between thread and interrupt processing

```
int X = 0;
SpinLock_t L = UNLOCKED;

void AccessXThread() {
    DisablePreemption();
    MaskInterrupts();
    SpinLock(&L);
    X = X+1;
    SpinUnlock(&L);
    UnMaskInterrupts();
    EnablePreemption();
}

void AccessXInterrupt() {
    …
    SpinLock(&L);
    X = X+1;
    SpinUnlock(&L);
    …
}
```

- Disabling preemption prevents deadlock scenario due to scheduling switch to different thread
- Masking interrupts prevents deadlock scenario due to interrupt
- Locks ensure consistency

Signals

- Threads check for pending signals on return to user mode
- Unix signal handlers are user-mode equivalents of interrupt handlers
- Threads behave as if a procedure call to the signal handler was made at the point at which the thread received the call
  - Almost : register state must be handled differently

Scheduling

- OS's manage resources
  - Processor time is apportioned to threads
  - Primary memory is apportioned to processes
  - Disk space is apportioned to users
  - I/O bandwidth may be apportioned to processes
- Scheduling concerns the sharing of processors
  - Dynamic scheduling is the task
  - Objectives
    - Good response to interactive threads
    - Deterministic response to real-time threads
    - Maximize process completions per hour
    - All of the above?

Approaches to Scheduling

- Simple batch systems
  - One job at a time
- Multi-programmed batch systems
  - Multiple jobs concurrent
  - Scheduling decisions
    - How many jobs?
    - How to apportion the processor between them?
- Time-sharing systems
  - How to apportion processor to threads ready to execute
  - Optimization criteria : time between job submission and completion
- Shared servers
  - Single computer, many clients, all wanting "fair" share
- Real-time systems

Time-Sharing Systems

- Primary scheduling concern is the appearance of responsiveness to interactive users
- Threads assigned user-level priority “importance” (UNIX nice())
- OS assigned thread priority rises and falls based on
  - Length of bursts of computation (before yielding)
  - Length of time between bursts
- Sensible strategy
  - Decay priority while thread is running
  - Increase priority while thread is waiting

Real-Time Systems

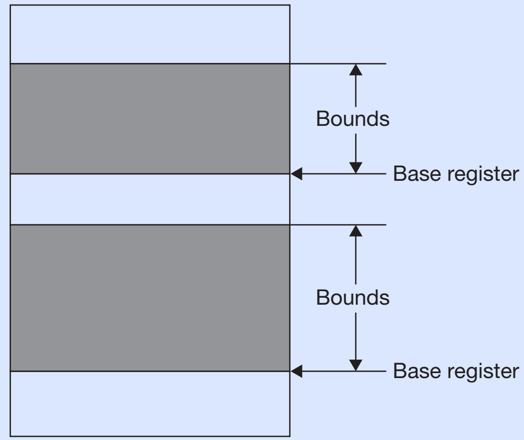
- Real-time system scheduling must be dependable
  - Music
  - Video
  - Nuclear power plant data processing
- Approximate real-time by adding very-high real-time priorities
  - Interrupt processing still preempts threads
  - Synchronized access to kernel resources can cause priority-inversion
    - Low-priority threads locks a resource a real-time thread needs

Memory Management

- Requires deep understanding of hardware capabilities and software requirements
- Involves
  - Memory abstraction
  - Optimizing against available physical resources
    - High-speed cache
    - Moderate-speed primary storage
    - Low-speed secondary storage
- Security
  - Protect OS from user processes
  - Keep user processes apart
- Scalability
  - Fit processes into available physical memory

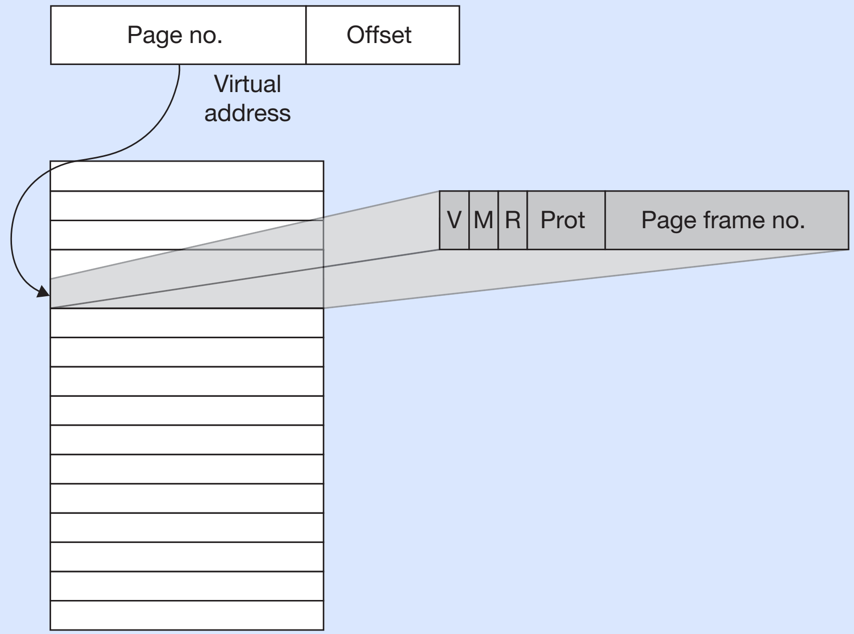
Virtual memory

- Virtual memory is the support of an address space that is independent of the size of primary storage
- Hardware support for base and bounds checking
- Basic approaches
  - Fixed-size *pages*
  - Variable-size *segments*
- Paging common today
  - Suffers from internal fragmentation



Virtual Memory Implementation

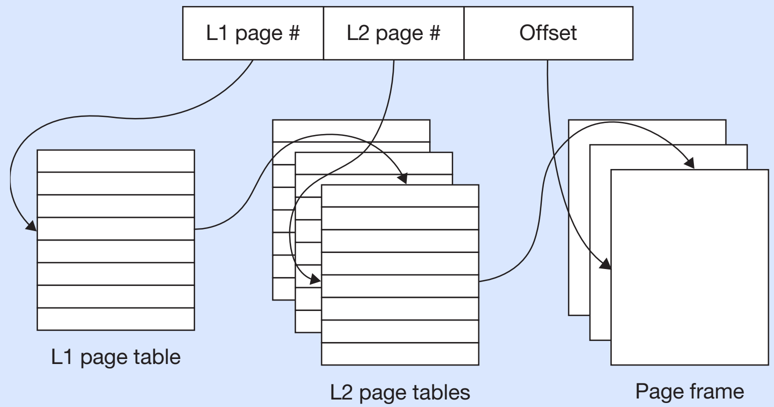
- Page table
  - Assume 32-bit virtual address
  - Page size 4096 bytes
    - Implies
      - 12 bit offset
      - 20-bit page number
  - V = validity bit
    - If set Page frame no. is high-order bits of address in real memory
  - R = referenced bit
    - If page is referenced by a thread
  - M = modified bit
    - set if page is modified
  - Prot. = page-protection bits
    - user, os., exec, data, etc.



On a 32-bit arch. how big is the page table?

Forward-mapped Page Tables

- Lower-overhead approach
- Each virtual address divided into two 10-bit numbers
  - L1 page number
  - L2 page number
  - Offset



- Advantages
  - Lower overhead
- Disadvantages
  - More lookups

Other Approaches to Address Translation

- Linear Page Tables
  - Use small number of bits to divide memory into spaces
  - Allocate registers for addresses of space page tables (eliminates read)
- Hashed Page Tables
  - Use hashing to index into page tables
- Note : memory access is slow; caching is imperative
  - Hardware support via "translation lookaside buffers"
    - Fast processor-based memory containing some entries of address translation table

Operating-System Issues

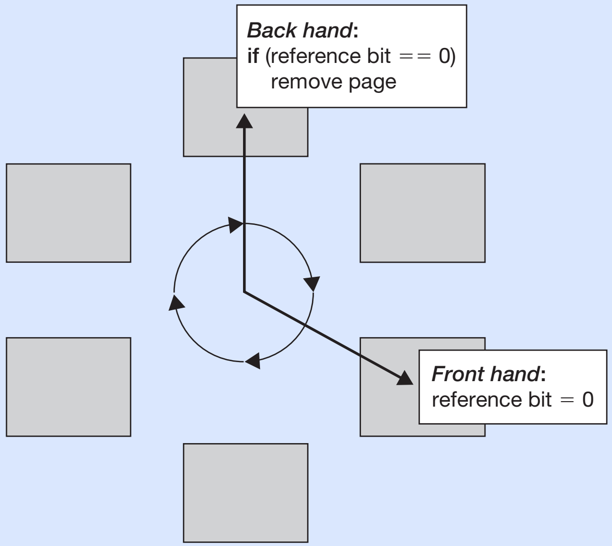
- OS responsible for ensuring programs execute at reasonable speed
- OS must determine which pages should be in primary memory
- OS virtual memory policy decisions
  - Fetch
  - Placement
  - Replacement
- Simple approaches
  - Demand paging
    - Fetch only when a thread references something in that page
  - Placement
    - Anywhere
  - Replacement
    - When full, eject page in memory longest (FIFO)
      - Problem?

OS Response to a Page Fault

- Steps
  - Detect page fault
  - Find a free page frame
  - Write a page out to secondary storage if none free
  - Fetch desired page from secondary storage
  - Return from trap
- Latter steps very costly
  - Read in extra pages
    - Prepaging – How? Why?
  - Write-out pages preemptively
    - Dedicate a page-out thread

Page Caching Strategy

- Optimal replacement strategies are impractical
- Least-recently-used (LRU) good in practice
  - Except counting references is impractical
- Two-handed clock algorithm used in practice
  - One hand sets reference bit to 0
  - Other hand triggers page flush



Copy-on-Write and Fork

- Can fork() be made less expensive to implement?
  - Remember fork() copies a process' entire memory space
- Lazy evaluation
  - Let copies share address space
  - Mark all pages read only
  - On write make copies
  - OS bookkeeping requires care
- Note files may be mapped to shared process address space (mmap())
  - Shared
    - Modifications seen by all forked processes
  - Private
    - Modifications remain private to each forked process (copy on write)

Review

- Process management
  - Entails multiplexing threads, interrupts, and system calls to available processors
- Memory management
  - Virtual memory allows large programs to run on systems with small amounts of primary storage
  - Virtual memory allows co-existence of multiple programs

Lecture 4 : File Systems & Networking

Material from

Operating Systems in Depth

(spec. Chapters 6 and 9)

by

Thomas Doeppner

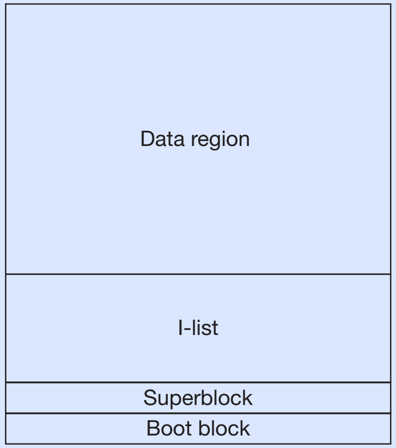
GET THIS BOOK AND READ IT!

File Systems

- Purpose
  - Provide easy-to-use permanent storage with modest functionality
  - Performance of file system critical to system performance
  - Crash tolerance a function of file system capabilities
  - Security a major concern
- Criteria
  - Easy
    - File abstraction should be easy to use
  - High performance
    - No waste of space, maximum utilization of resource
  - Permanence
    - Dependable
  - Security
    - Access control should be strict

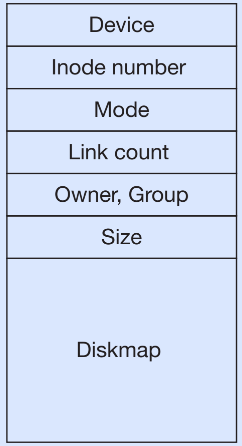
Basics

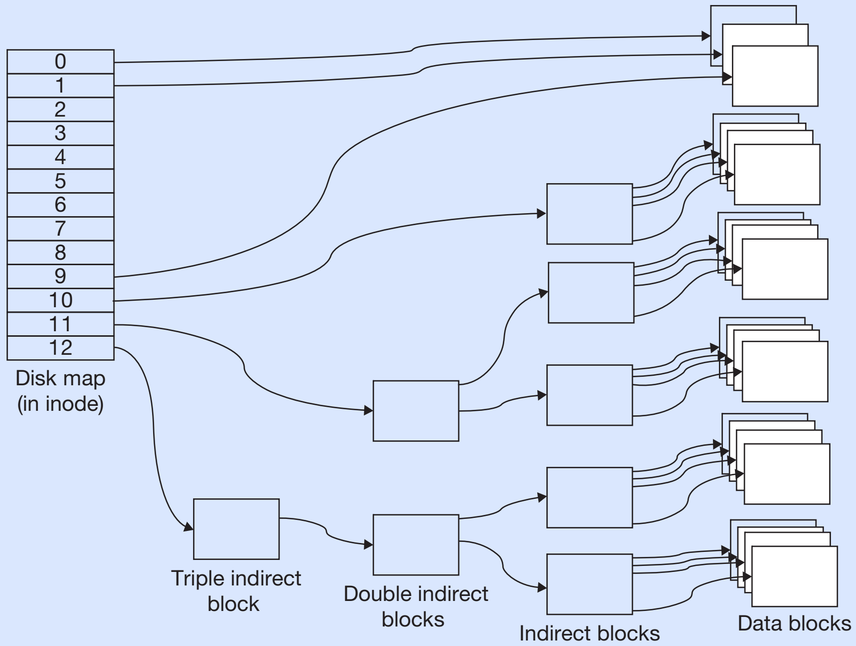
- Pedagogical review of obsolete Unix system 5 File System (S5FS)
- Revolutionary, simplifying Unix file abstraction
  - A file is an array of bytes
- File system layout
  - Boot block
    - First-level boot program that reads OS into memory
  - Superblock
    - Describes layout of remaining filesystem
  - i-list
    - Array of index nodes (inodes)
  - Data region
    - Disk blocks holding file contents



Unix's S5FS

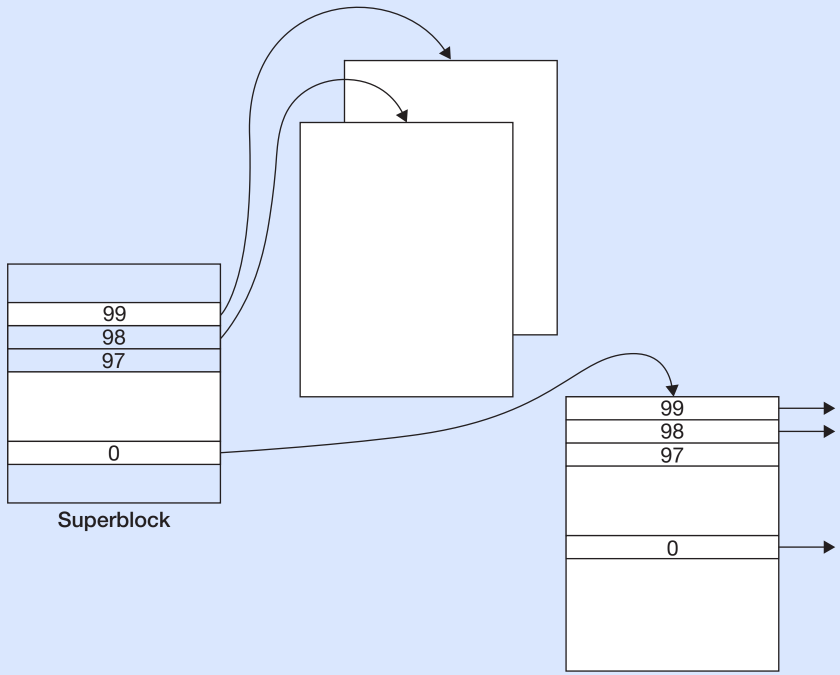
- Each file is described by an inode
- Directories are files containing names and inode numbers
- Diskmap
  - Maps logical blocks numbered relative to the beginning of the file to physical blocks numbered relative to the beginning of the file system
  - Assume
    - Block length = 1024 bytes
    - 13 pointers
      - First 10 point directly to disk blocks
      - Next singly indirect
      - Doubly
      - Triply
  - 0 pointer counts as block of all zeros
    - Efficient for sparse files





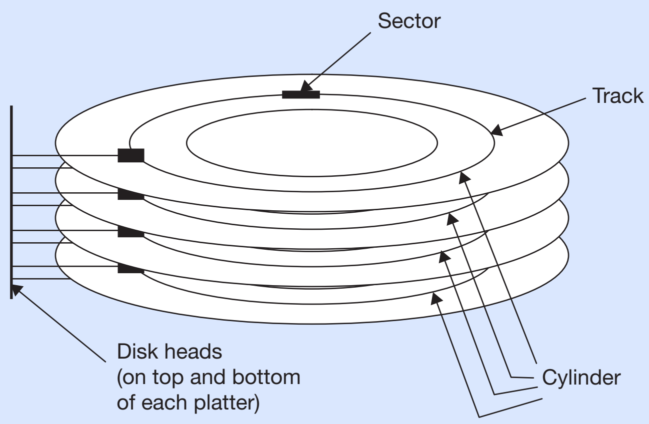
Organizing Free Storage on Disk

- Free disk blocks are represented as a linked list
- Superblock
  - Contains addresses of up to 100 free disk blocks
  - Last pointer points to another block containing free disk blocks
  - Contains cache of indices of free inodes
- Inodes
  - Simply marked as free or not on disk
  - Disk writes required for allocation and frees
    - Aids crash tolerance – inode updates are immediate



Disk Architecture

- File systems optimize performance by being aware of disk architecture
- Architecture
  - Many platters (top and bottom)
  - Many tracks per platter
  - Tracks divided into equal length sectors
  - Read a write heads per surface
  - One head active at a time
  - Set of tracks selected by heads at one moment calls a cylinder
- Nomenclature
  - Seek time : time to position the heads over the correct cylinder
  - Rotational latency : time 'til desired sector is underneath head
  - Transfer time : time for sector to pass under head



2013 Disk Performance

- "Rhinopias Disk Drive"
- Tricks of the trade
  - Maximizing throughput
    - Head skewing
      - Sectors offset on each head by some number of sectors to account for head switch time
    - Cylinder skewing
      - Sectors offset by some amount to account for one track seek time

Rotation speed	10,000 RPM
Number of surfaces	8
Sector size	512 bytes
Sectors/track	500–1000; 750 average
Tracks/surface	100,000
Storage capacity	307.2 billion bytes
Average seek time	4 milliseconds
One-track seek time	.2 milliseconds
Maximum seek time	10 milliseconds

S5FS Problems and Improvements

- File allocation strategy results in slow file access
- Small block size results in slow file access
- Lack of resilience in the face of crashes is a killer

- Possible improvements
  - Increase block size
    - Fragmentation becomes an issue
  - Rearrange disk layout to optimize performance

Dynamic Inodes

- S5FS inode table is a fixed array
  - Requires predicting number of files the system will have
  - Can't add more disk space to the file system
- Solution
  - Treat inode array as a file
  - Keep inode for the inode file at a fixed location on disk
    - Backup

Crash Resiliency

- To recover from a crash means to bring the file system's metadata into a consistent state
- Some operations (rename()) require many steps, requiring multiple writes
- Approaches
  - Consistency preserving
  - Transactional

- Transaction support common in databases
  - Journaling
    - New value – modification steps are recorded in a journal first, then applied
    - Old value – old blocks are recorded in a journal, then filesystem updated
  - Shadow-paging
    - Original versions of modified items retained
    - New versions not integrated into the file system until the transaction is committed (single write)

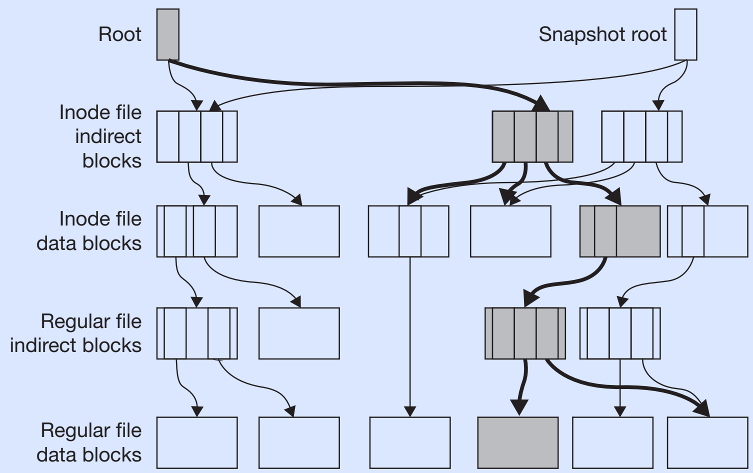
Journaled File Systems

- Many file systems use journaling for crash tolerance
- Journaling may be used to protect
  - Metadata
  - User data
  - Both

- Ext3 example
  - Updates grouped into time-delimited transactions
  - Separate commit thread copies from file-system block cache to a journal
  - Back-links are maintained to cache that allowing freeing journal space upon final commit
  - Upon crash any journaled updates are processed

Shadow-Paged File Systems

- Also called copy-on-write file systems
  - e.g. WAFL and ZFS
- Filesystem updates result in entirely new inode indirect reference tree
- Snapshot root always allows recovery of a consistent filesystem



Directories and Naming

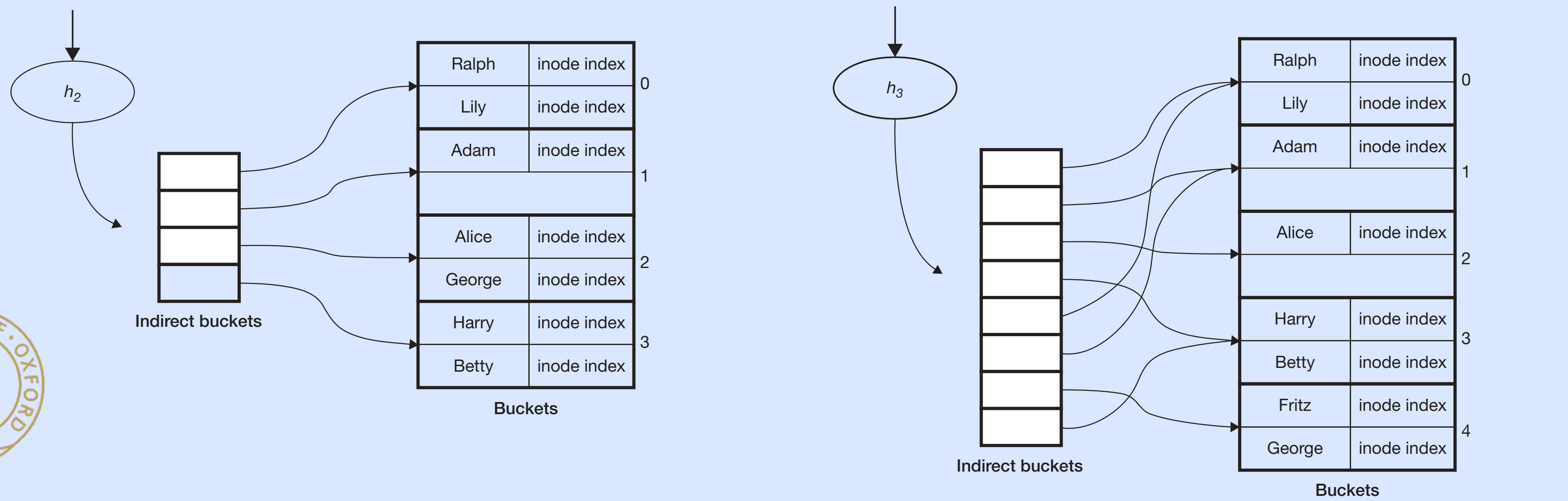
- Opening a file requires
  - Following its pathname
  - Opening directory files
- Creating a file
  - Verifying pathname
  - Inserting component in last

- S5FS
  - Linear sequence of fixed length names and inode numbers
  - Deleting entries involved marking slots as free
  - No directory space ever given back to filesystem!
  - Sequential search!
- Subsequent generation directory structure
  - Variable length names
  - First fit replacement

- Directory operations were a major bottleneck!

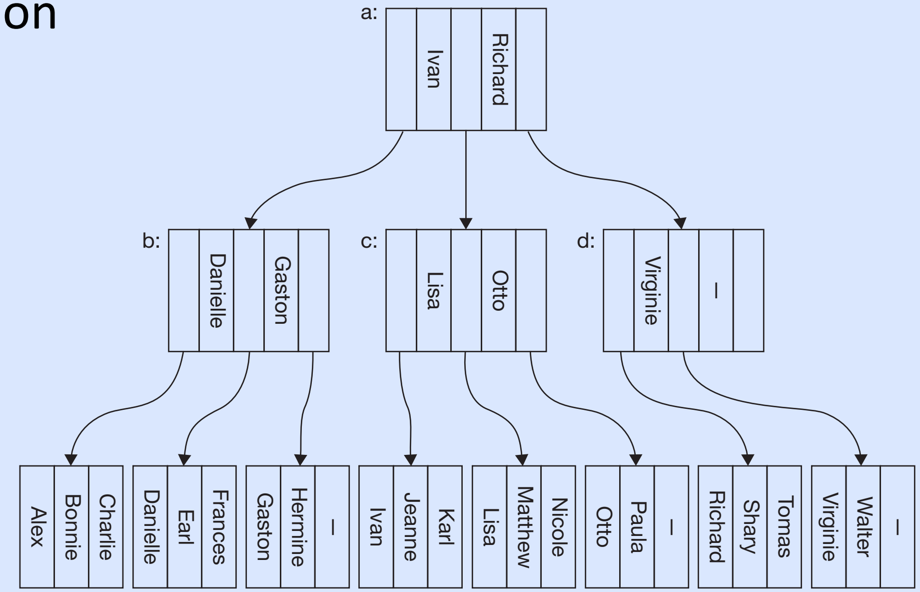
Hashing

- Extensible hashing
  - Sequence of hash functions
    - h_0, h_1, h_2, …, h_i,
  - Low order bits of h_i are enforced to be the same as h_{i-1}
  - h_i hashes to 2^i buckets
- Example : Adding Fritz (hashed to bucket 2)
  - Indirect buckets used to efficiently and compactly implement rehashing by replicating non-split bucket pointers



B+ Trees

- Balanced tree
  - Node-degree requirement : each node fits in a block
  - Node-size requirement : each block must be at least half full
  - Leaves are linked together
- Example tree with block size 3
  - Consider inserting Lucy
  - Consider deletion

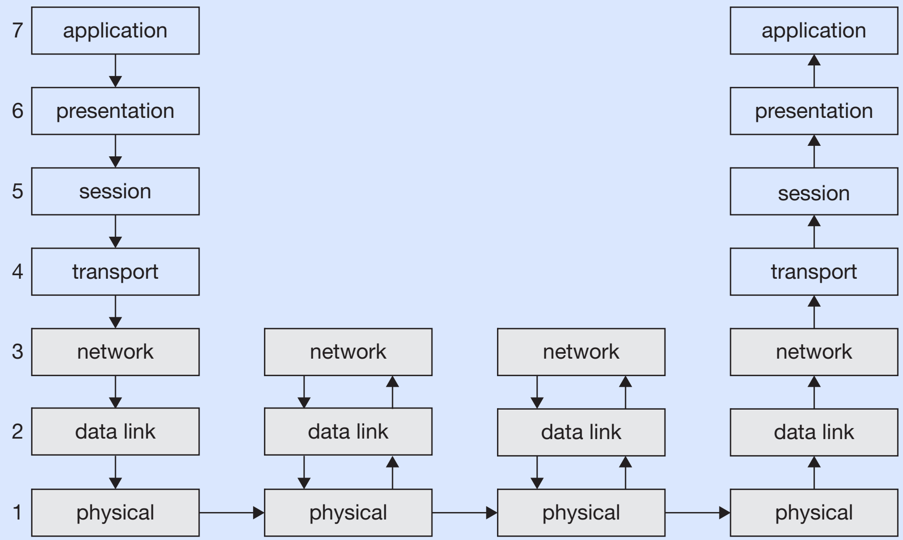


Introduction to Networking

- Definition
  - A way to interconnect computers so that they can exchange information
- Types
  - Circuit (old phone networks)
    - Actual circuit between devices established
  - Packet switching (currently most common)
    - Data is divided into marked packets that are transported independently
- Challenges
  - Data can be lost or reordered
  - To much traffic can clog network
  - Base / Home networks are heterogeneous

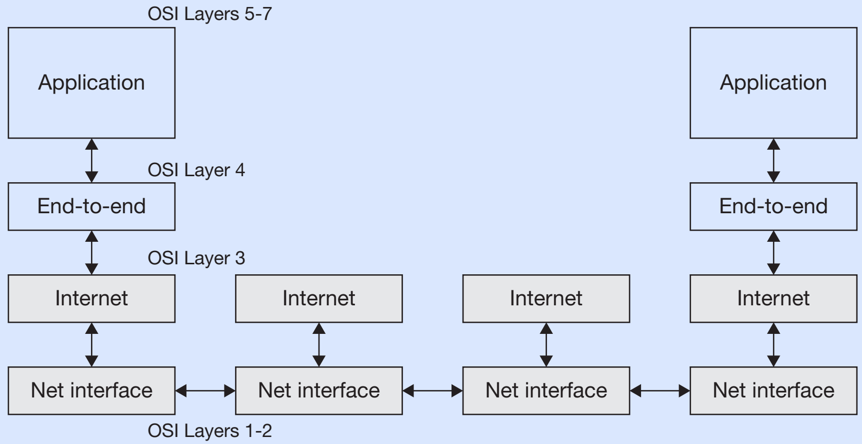
Standardization

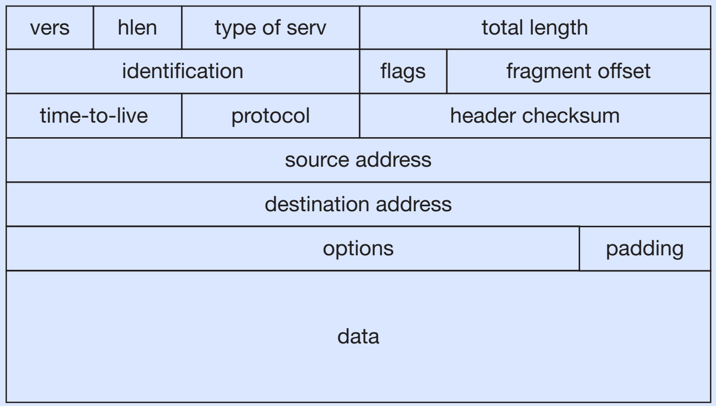
- International Standards Organization (ISO) Open Systems Interconnect (OSI) 7-layer network model
- Layers
  1. Physical layer (the wire, EM, etc.)
  2. Data link layer (e.g. ethernet)
     - Means for moving data on and off wire
     - Info. representation scheme in EM waves
       - Sequences of bits known as *frames*
     - Sharing mechanisms
     - Medium access control (MAC) addresses
       - Used to decide who should get what
  3. Network layer
     - Addressing, delivery, packets
  4. Transport layer
     - Ensures communication is reliable
  5. Session layer
     - Dialogue control (who talks when), synchronization (error recovery), etc.
  6. Presentation layer
     - Deals with transforming datastructures (endianness, floating point numbers, …)
  7. Application layer (e.g. http)
     - High-level application (support) software



Internet Protocols

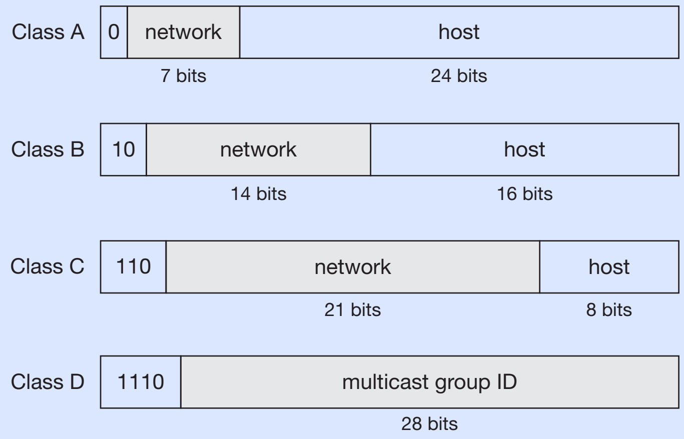
- Distinctions in top three layers ignored
- Base network combines OSI 1&2
  - Called internet protocol (IP)
  - Protocol data unit (packet)
- IP Packet
  - Header (addresses)
  - Data (PDU of higher layer)
    - Called a segment
- Packaging
  - Normally a header is added to a segment
  - If a segment is too large it is split
    - e.g. ethernet's maximum transfer unit (MTU) is 1500 bytes
- Routing
  - Controlled externally
  - Picked from routing tables





IPv4 Addresses

- Structured 32-bit numbers
- Allows
  - 2113658 networks
  - 3189604356 hosts
- Issues
  - Too many (large routing tables)
  - Too few (not enough hosts)
- IPv6 = 128 bit addresses



Internet Transport Protocols

- User datagram protocol (UDP)
  - Not reliable
  - Provides checksum only
  - Allows one to implement own reliability scheme
- Transmission control protocol (TCP)
  - Reliable communication
  - Copes with network congestion
  - 32 bit sequence number transmitted from sender to receiver indicating how many bytes have been transmitted
  - Response returned indicating successful receipt of the whole sequence numbered less than the returned value
  - Sequence numbers must be reused
    - Gives rise to a maximum segment lifetime (set “by fiat” to 2 min.)
  - Starting sequence number decided in handshake
- Both augment IP addresses with 16 bit *port* numbers