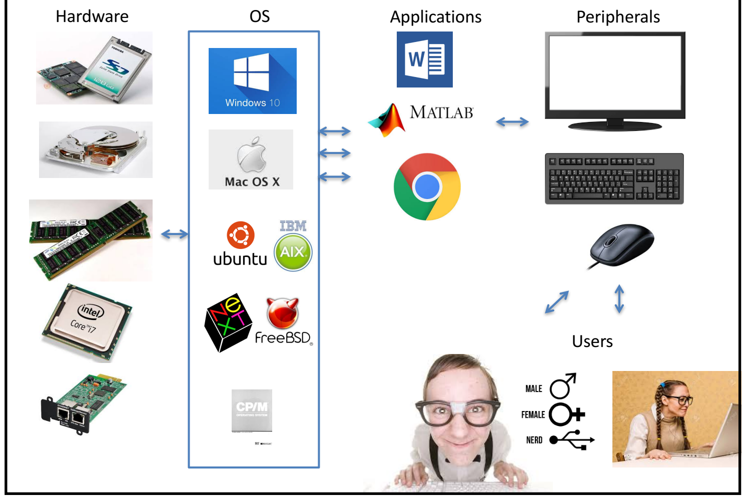
B16 Operating Systems

Introduction

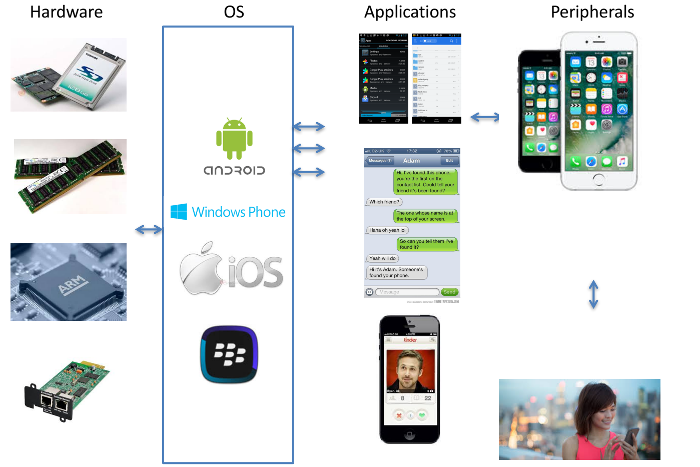
Computer Operating Systems As You Know Them

Hardware | OS | Applications | Peripherals



Mobile Phone Operating Systems As You Know Them

Hardware | OS | Applications | Peripherals



Learning Outcomes *

- Familiarity with general operating system concepts and how to use them in code
  - File
  - Process
  - Thread
  - Synchronisation
  - Memory
  - Paging
  - Socket
  - Port
- Datastructures / implementations and why they are important
  - Page table
  - Semaphore
  - Mutex
  - Socket

* Marks examinable material

Course Contents

- User perspective – how you use and interact with OS's *
  - We'll focus on Linux (posix compliant OS)
  - You should understand system calls (fork, wait, open, printf, etc.)
  - Be aware of and study using command line utilities (e.g. man <section>)
  - Read C-pseudo code that uses operating system calls.
- Operating system *implementation* perspective – how OS's are implemented
  - Will talk about an Unix/Linux-like "Simple-OS" abstraction
  - Discuss implementation of synchronization and threading in particular

* Marks examinable material

B16 Operating Systems

Lecture 1 : History and User Perspective

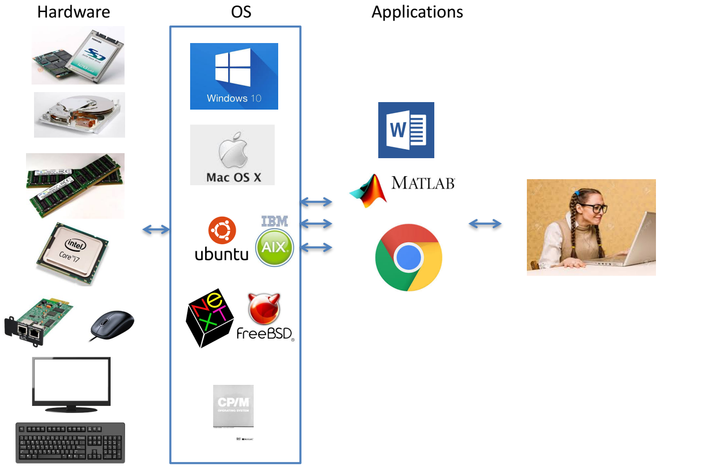
Material from
Operating Systems in Depth
(spec. Chapter 1)
by
Thomas Doeppner

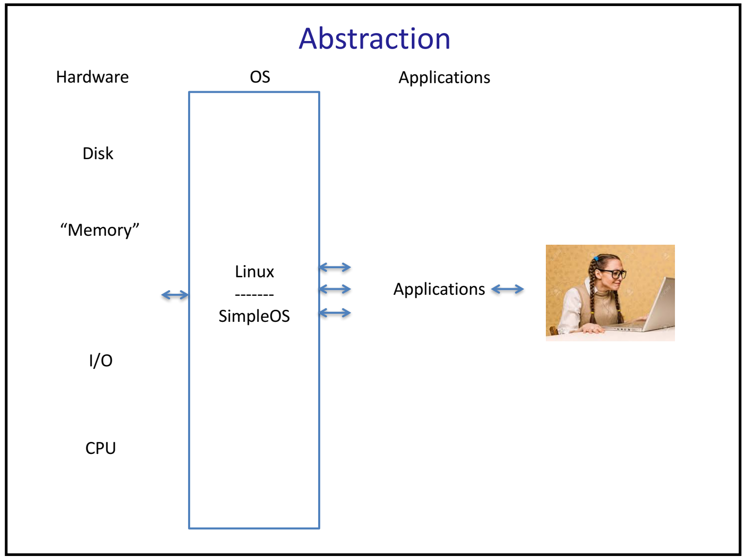
THIS BOOK HAS EVERYTHING IN IT!

What is an operating system?

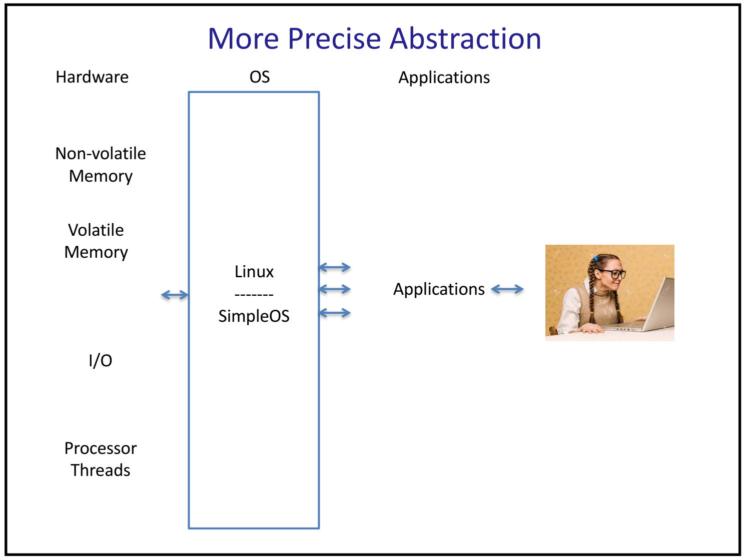
- Operating systems allow for *sharing*
- To do this operating systems provide software abstractions of
  - Processors
  - RAM (physical memory)
  - Disks (secondary storage)
  - Network interfaces
  - Display
  - Keyboards
  - Mice
  - …
- Some OS-provided abstractions are
  - Processes
  - Files
  - Sockets

Intuition

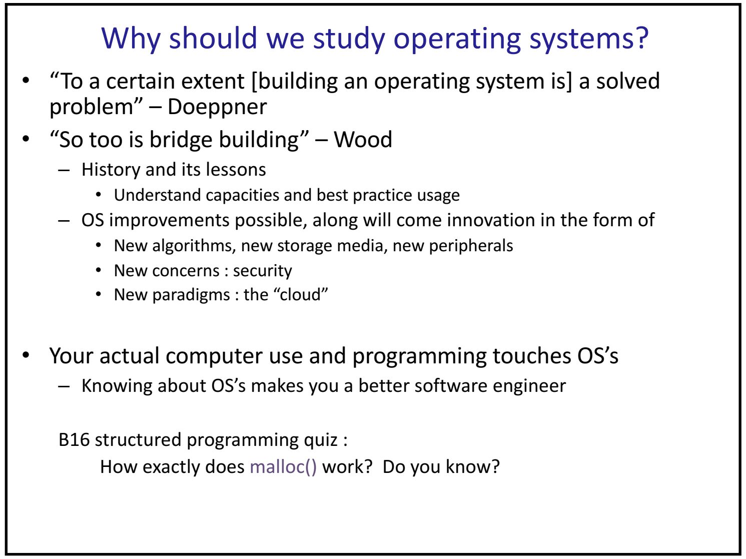




Abstraction



More Precise Abstraction

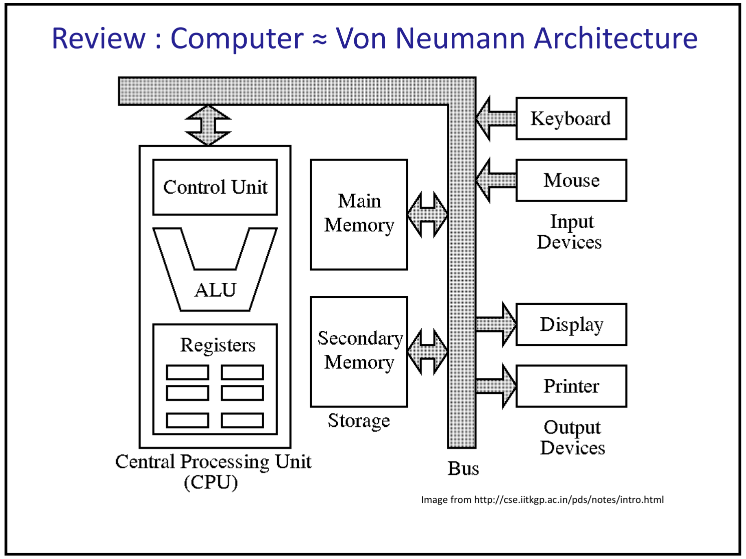


Why should we study operating systems?

- “To a certain extent [building an operating system is] a solved problem” – Doeppner
- “So too is bridge building” – Wood
  - History and its lessons
    - Understand capacities and best practice usage
  - OS improvements possible, along will come innovation in the form of
    - New algorithms, new storage media, new peripherals
    - New concerns : security
    - New paradigms : the “cloud”
- Your actual computer use and programming touches OS’s
  - Knowing about OS’s makes you a better software engineer

B16 structured programming quiz :

How exactly does malloc() work? Do you know?



Review : Computer ≈ Von Neumann Architecture

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

Review : Computer ≈ Von Neumann Architecture

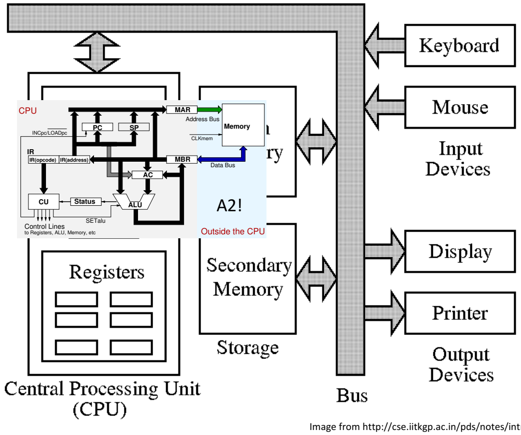


Image from http://cse.iitkgp.ac.in/pds/notes/intro.html

A2 Review : Computers Execute Machine Code

- Machine code : instructions that are 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)
```

B16 Struct. Prog. Review : 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: gcc, g++, etc.
- A linker takes one or more object files generated by a compiler and combines them into a single executable
  - Gathers libraries, resolving symbols as it goes
  - Arranges objects in a program's address space
    - B16 Structure programming quiz : what's a program's address space?
- Programs run on *shared* hardware using OS system calls, libraries, virtual memory, program address space specifications, etc.
  - Note: modern OS' provide dynamic linking, i.e. runtime resolution of referred-to but run-time unresolved symbols

B16 Operating Systems In a Nutshell : *Sharing*

- Loading a *single* compiled program into memory and executing it does *not* require an operating system (recall A2 L1-4)
- Operating systems are about *sharing* *
  - One computer has
    - Processor threads
    - Disk space
    - RAM
    - I/O
      - Including peripherals

which must be efficiently shared between different programs and users

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 Were

- Batch systems that
  - Facilitated running multiple jobs sequentially
- Suffered from I/O bottlenecks
  - Computation stopped to for I/O operations
- Led to interrupts being 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 thread 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 that got invented then
  - "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
- Also
  - 1993 : First release of Windows NT, subsequent Windows OS's based on NT
  - 1991 : Linus Torvalds ported Minix to x86 (LINUX!)

Using Operating Systems – The Basics

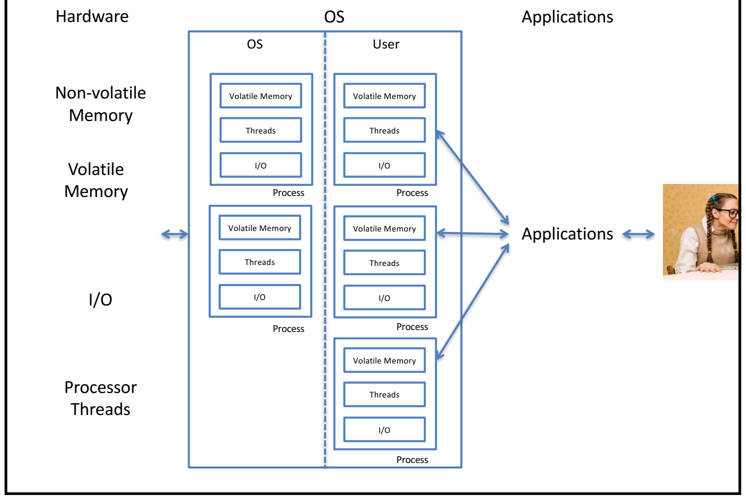
The Basics - Outline

- Processes
- Threads
- Virtual memory
- Address space
- Files
- Pipes

Key Concept : Processes *

- OS abstraction that includes
  - Address space (*virtual memory* *)
  - Processors (*threads* of control *)
- Processes are largely disjoint
  - Processes cannot directly access each others' memory
    - Exceptions exist for sharing memory; mmap etc.
- Running a program
  - Creates a "process"
  - Causes the operating system to load machine code from a file into the process's *address space*
  - The operating system creates a single *thread* of control that starts executing the first instruction at a predefined address in the program's *virtual memory*

Processes



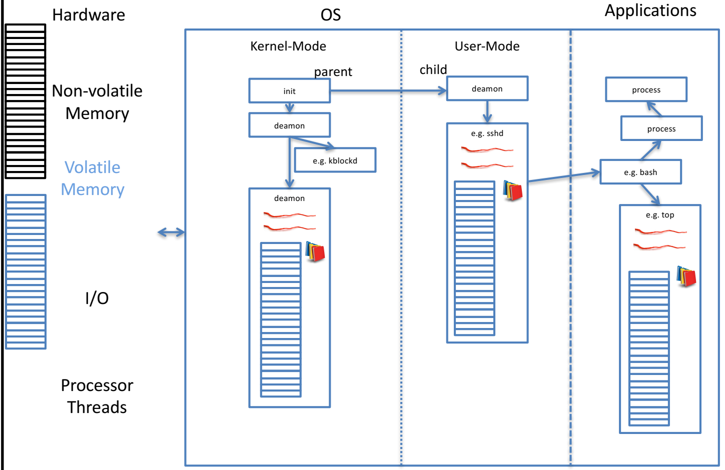
Processes and Threads **** (fork_example_1.c)

- Processes are created via a system call named fork()
  - fork() creates an exact copy of the calling process is made
    - Efficiently implemented via lazy, copy-on-write marking of pages
  - fork() returns *twice!*
    - Once in the child (return value 0)
    - Once in the parent (return value the *process id* (PID) of the child process)
    - Two "threads of control", the parent process thread and the child process thread.
- Processes report termination status via the system call exit(*ret_code*)
- Processes can wait() for the termination of child processes
- Example uses
  - Terminal / Windows
  - Apache cgi

```
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 */
}
```

NB: You are *required* to download, read, compile and play with the source code examples that can be found on the course website. The course website for this year is http://www.robots.ox.ac.uk/~fwood/teaching/B16_OS_hilary_2016_2017/index.html and fork_example_1.c can be viewed and downloaded from there.

Forking Makes a Tree : The Process Tree



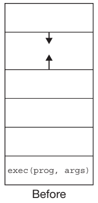
Starting Programs in Processes (fork_example_2.c)

- execl() system call is used to do this

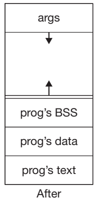
```
int pid;
if ((pid = fork()) == 0) {
   /* we'll soon discuss what might take place before exec
      is called */
   execl("/home/twd/bin/primes", "primes", "300", 0);
   exit(1);
}

/* parent continues here */

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



- execl() replaces the entire contents of the processes *address space* and
  - the stack is initialized with the passed arguments
  - a special start routine is called that itself calls main()
  - exec doesn't return except if there is an error!

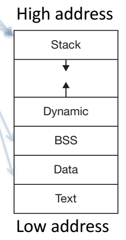


Virtual Memory = Address Space, i.e. 2^64 words

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

- Arrows indicate variable placement
- malloc() claims space in dynamic*

```
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);
}
```

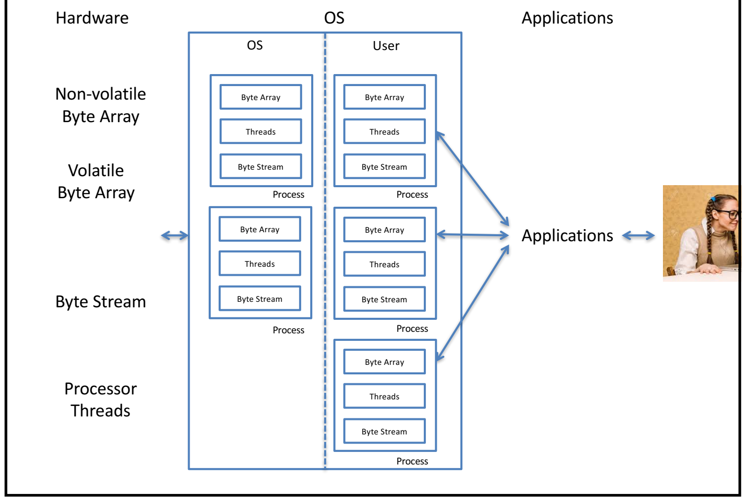


Should know this from B16 Structured Programming

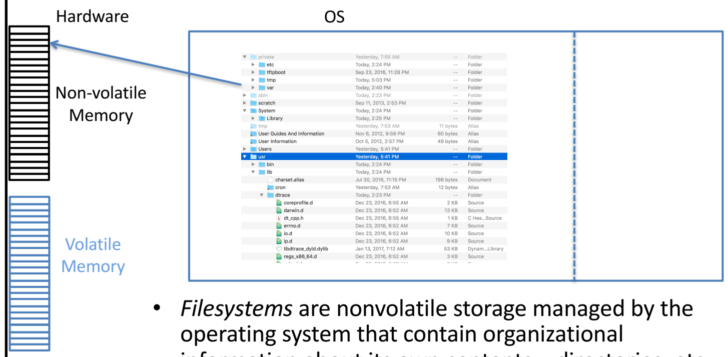
Files = Named Byte Arrays *

- Files are an operating systems' *primary abstraction* for ***everything***
  - Keyboard
  - Display
  - Non-volatile storage
  - Other processes
- Naming files
  - *Filesystems* generally are tree-structured directory systems
  - *Namespaces* are generally shared by all processes

Files = Byte Arrays, I/O = Byte Streams



Filesystems and Namespaces



```
▼ private                         Yesterday, 7:55 AM      --          Folder
   ▶ etc                          Today, 2:24 PM          --          Folder
   ▶ tftpboot                     Sep 23, 2016, 11:28 PM  --          Folder
   ▶ tmp                          Today, 5:03 PM          --          Folder
   ▶ var                          Today, 2:40 PM          --          Folder
▶ sbin                            Today, 2:23 PM          --          Folder
▶ scratch                         Sep 11, 2013, 2:53 PM   --          Folder
▼ System                          Today, 2:24 PM          --          Folder
   ▶ Library                      Today, 2:25 PM          --          Folder
  tmp                             Yesterday, 7:53 AM      11 bytes    Alias
  User Guides And Information     Nov 6, 2012, 9:58 PM    60 bytes    Alias
  User Information                Oct 5, 2012, 2:57 PM    49 bytes    Alias
▶ Users                           Yesterday, 5:41 PM      --          Folder
▼ usr                             Yesterday, 5:41 PM      --          Folder
   ▶ bin                          Today, 2:24 PM          --          Folder
   ▼ lib                          Today, 2:24 PM          --          Folder
        charset.alias             Jul 30, 2016, 11:15 PM  198 bytes   Document
        cron                      Yesterday, 7:53 AM      12 bytes    Alias
      ▼ dtrace                    Today, 2:23 PM          --          Folder
           coreprofile.d          Dec 23, 2016, 6:55 AM   2 KB        Source
           darwin.d               Dec 23, 2016, 6:52 AM   13 KB       Source
           dt_cpp.h               Dec 23, 2016, 6:55 AM   1 KB        C Hea...Source
           errno.d                Dec 23, 2016, 6:52 AM   7 KB        Source
           io.d                   Dec 23, 2016, 6:52 AM   10 KB       Source
           ip.d                   Dec 23, 2016, 6:52 AM   9 KB        Source
           libdtrace_dyld.dylib   Jan 13, 2017, 7:12 AM   53 KB       Dynam...Library
           regs_x86_64.d          Dec 23, 2016, 6:52 AM   3 KB        Source
```

- *Filesystems* are nonvolatile storage managed by the operating system that contain organizational information about its own contents – directories, etc.
- *Namespaces* are the convention for how to name things, in this case, files. Here is
- ssh namespace : fwood@donna.robots.ox.ac.uk:/usr/lib/strace/darwin.d
- smb namespace (general) : \\<DOMAIN.NAME>\<dfsroot>\<path>

Files = Named Byte Arrays *

- Files have two different interfaces
  - Sequential access and
  - Random-access
- Files are opened with open() and return a file descriptor
- read() sequentially takes the contents of the file and puts it into a buffer in volatile memory and advances the “read pointer” associated to the file descriptor.

```
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 (fork_example_2.c)

- 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()
- This is how applications launched by the operating system, whether through a window interface or from the command line, write their stdout to reasonable places.

```
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. The following code

```
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);
}
```

reverses a file with lseek repositioning the ”read pointer” to a given offset position.

NB. you should be able to read code like this easily, interpreting it as pseudo-C code. *

A Different Kind of File: A Pipe * (pipe_example.c)

- A pipe is a means for one process to send data to another directly
- pipe() returns two nameless file descriptors that can be written to and read from

```
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);
   }
}
```

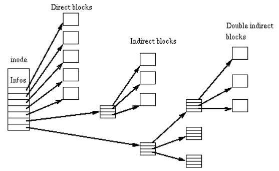
What is a Directory?

- A directory is a file
  - that is interpreted as containing references to other files by the OS *
- An example organization might include an array of (implementation dependent)
  - Component names and
  - inode numbers
    - an inode is a data structure on a filesystem on Linux and other Unix-like operating systems that stores all the information about a file except its name and its actual data.
    - inodes are files too

Component name	Inode number

Directory entry

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



Keeping Stuff Around After the Computer Turns Off : Creating Files

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

- “man 2 open” (man is a Unix command for viewing manual entries about commands – you should know about this in general)

```
OPEN(2)                    BSD System Calls Manual                    OPEN(2)

NAME
     open, openat -- open or create a file for reading or writing

SYNOPSIS
     #include <fcntl.h>

     int
     open(const char *path, int oflag, ...);

     int
     openat(int fd, const char *path, int oflag, ...);

DESCRIPTION
     The file name specified by path is opened for reading and/or writing, as specified by the argument oflag;
     the file descriptor is returned to the calling process.

     The oflag argument may indicate that the file is to be created if it does not exist (by specifying the
     O_CREAT flag).  In this case, open() and openat() require an additional argument mode_t mode; the file is
     created with mode mode as described in chmod(2) and modified by the process' umask value (see umask(2)).

     The openat() function is equivalent to the open() function except in the case where the path specifies a…
```

Review : User Perspective on OS *

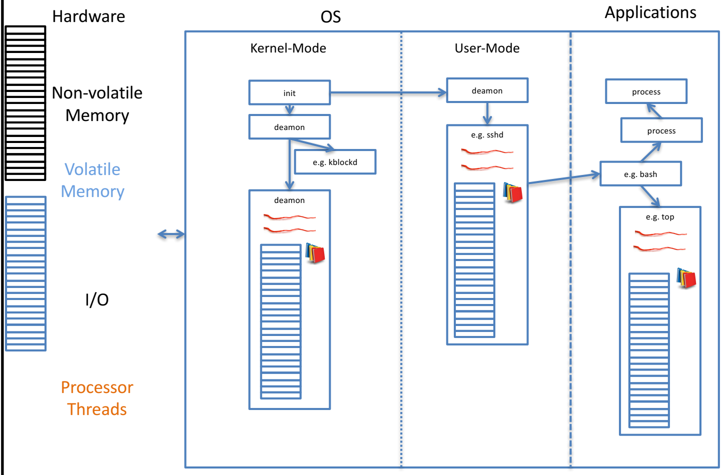
- Traps / system calls
  - exec()
  - fork()
  - open()
  - pipe()
  - exit()
  - close()
  - read()
  - write()
  - dup()
  - …
- The operating system provides this interface that abstracts the “computer” as a process and allows sharing.
- Next lecture : more user basics.
- Final two lectures : some key points in implementing an OS that provides these functionalities

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



Threads * (thread_example_1.c)

- What is a thread?
  - A thread of control
  - Abstraction of a processor thread within a process
- Property
  - All threads in a process share the same address space with all other threads in the process
- Why threads?
  - Mechanism for concurrency in user-level programs
  - Can dramatically simplify code, for instance writing code to
    - Concurrently handle multiple database requests
    - Run a server listening on a socket responding to simultaneous client requests
  - Requires care
    - Synchronization <- this is a major issue to provision

NB: Code examples will use POSIX ("portable operating system interface") specification of threads and synchronization primitives

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);
}
```

Passing Arguments to Threads

- Care must be taken with threads in general because of address space sharing and ability to address the calling stack

```
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);
}
```

i.e. problem with this code in and out are *local* variables thus leave scope when rlogind exits

Thread Termination (thread_example_2.c)

- Threads can modify any part of the shared address space, however
- A pointer to memory must be provided for thread to place explicit return values, here &result

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

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

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

Thread Attributes

- For instance threads must have their own stacks
- e.g. to specify the stack size for a thread one initializes an attributes datastructure thr_attr

```
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);
```

NB: “man pthread_attr_init”

Synchronization *** (thread_example_3.c)

- Threads share access to common data structures
  - Same address space
- We need something called *mutual exclusion,* a form of thread synchronization, to make sure two things don’t happen at once
- Example, two threads each doing

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

- Can result in 1 or 2; adversarially reordering the assembly code below shows what kind of bad things can happen

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

POSIX Mutexes ***

- OS's typically 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 " or is "held by" a single thread or belongs to no thread

POSIX Mutexes ***

- A mutex is created via pthread_mutex_init()
- A thread may "own" or "lock" a mutex by calling pthread_mutex_lock()
- A mutex may be unlocked by calling pthread_mutex_unlock()

```
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);
```

Mutual exclusion can result in DEADLOCK!

- Using threads in practice can be fraught with difficulty
- *Deadlock* means that one or more threads are trying to own a set of mutexes in a way that results in some subset of the threads can never proceed *
- In the following code example deadlock can occur, can you see it?

```
void proc1( ) {                      void proc2( ) {
   pthread_mutex_lock(&m1);             pthread_mutex_lock(&m2);
   /* use object 1 */                   /* use object 2 */
   pthread_mutex_lock(&m2);             pthread_mutex_lock(&m1);
   /* use objects 1 and 2 */            /* use objects 1 and 2 */
   pthread_mutex_unlock(&m2);           pthread_mutex_unlock(&m1);
   pthread_mutex_unlock(&m1);           pthread_mutex_unlock(&m2);
}                                    }
```

Deadlock is nasty, difficult to detect, and to be avoided at all cost

- One useful avoidance mechanism is pthread_mutex_trylock()

```
proc1( ) {                            proc2( ) {
 pthread_mutex_lock(&m1);              while (1) {
 /* use object 1 */                     pthread_mutex_lock(&m2);
 pthread_mutex_lock(&m2);               if (!pthread_mutex_trylock(&m1))
 /* use objects 1 and 2 */                break;
 pthread_mutex_unlock(&m2);             pthread_mutex_unlock(&m2);
 pthread_mutex_unlock(&m1);            }
}
                                        /* use objects 1 and 2 */

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

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
```

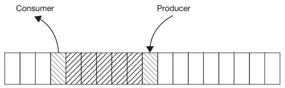
- Note : Mac's use a different naming scheme for semaphores, a Mach spec. named-semaphore via sem_open()

Why Do We Need Synchronization?

- Image that you have a multi-threaded webserver that handles parallel ticket-window selling of a finite supply of tickets with each web request handled by one of the threads in a thread pool.
- The number of tickets is critical, shared data that must be atomically decremented.

Canonical Data Sharing Problem : Producer-Consumer *

- A first peek at the kind of thing that has to happen inside the OS
- When you write to network socket there will be a kernel thread that intermediates between your requests and other requests
- Your producer thread write() will go into a buffer with a finite number of slots
- Two Threads
  - Producer (you) : puts things in the buffer
  - Consumer (os) : removes things from the buffer
- Producer must wait if buffer is full; consumer if buffer is empty



Semaphore sol’n to the producer-consumer problem

- Example sheet (don’t cheat by looking at this solution)

```
                  Semaphore empty = B;
                  Semaphore occupied  = 0;
                  int nextin = 0;
                  int nextout = 0;

void Produce(char item) {            char Consume( ) {
   P(empty);                            char item;
   buf[nextin] = item;                  P(occupied);
   nextin = nextin + 1;                 item = buf[nextout];
   if (nextin == B)                     nextout = nextout + 1;
      nextin = 0;                       if (nextout == B)
   V(occupied);                            nextout = 0;
}                                       V(empty);
                                        return(item);
                                     }
```

Deviations

- *Signals* *
  - Forces a user thread to put aside current activity
  - Call a pre-arranged handler
  - Go back to what it was doing
  - Similar to interrupt handling inside 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 ^c 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 (OS implementation perspective)

- Signals are processed by a single thread of execution
- Communication at right not problem-free because of asynchronous access to state
  - Displayed state could be incoherent
- 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);
}
```

Review

- New OS usages essentials
  - Threads
  - Mutexes
  - Semaphores
  - Signals
- From Lecture 1
  - Processes
  - Memory allocation
  - Files
  - Pipes
  - I/O
- Pretty much all the system calls you actually use except
  - Networking
  - Window-creating and UI

Lecture 3 : Inside the OS; Booting, System Calls, Interrupts, Threading, Processor & Memory Management (A very-high-level OS Implementation Perspective)

Material from

Operating Systems in Depth

(spec. Chapters 5 and 7)

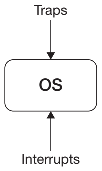
by

Thomas Doeppner

GET THIS BOOK AND READ IT!

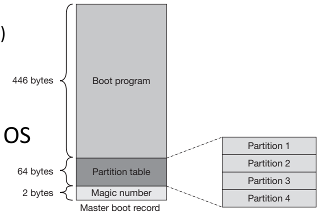
Some Definitions

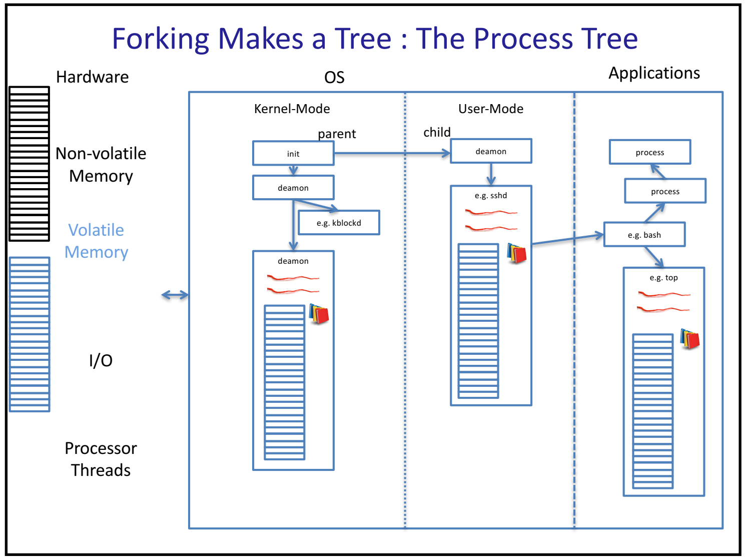
- Our terminology comes from from "Operating Systems in Depth" by Thomas Doeppner which describes the "Simple OS"
  - A hypothetical OS based-on Unix (6th edition) that is simplified to be
    - Monolithic
      - i.e. the OS is a single file loaded into memory at boot time
    - And with interfaces called
      - *Traps* which originate from user programs and
      - *Interrupts* that originate largely from hardware
    - and having two modes
      - User-mode in which parts of the OS run as if they were normal user programs and
      - Privileged / System-mode in which parts of the OS have access to everything
  - Terminology : Kernel *
    - The subset of an OS that runs in privileged mode



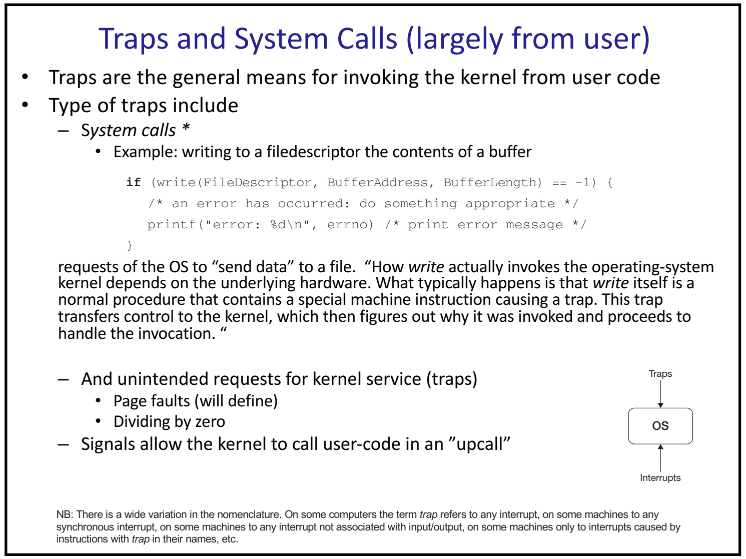
Booting

- 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





Forking Makes a Tree : The Process Tree



Traps and System Calls (largely from user)

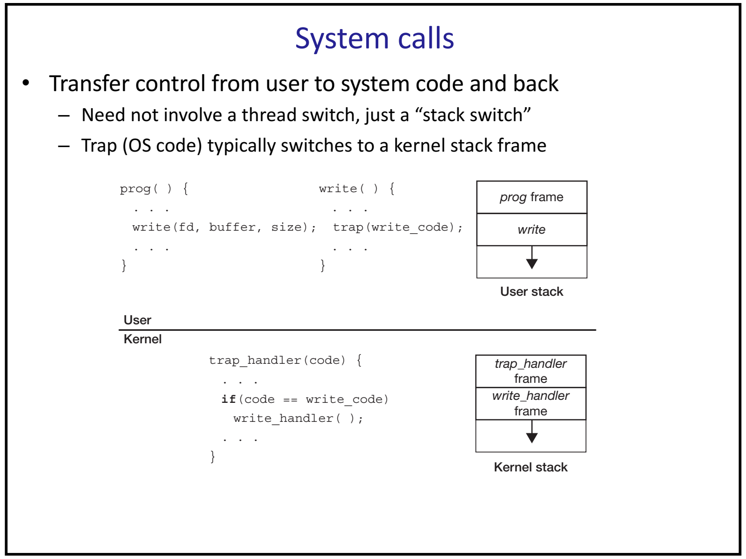
- Traps are the general means for invoking the kernel from user code
- Type of traps include
  - *System calls **
    - Example: writing to a filedescriptor the contents of a buffer

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

requests of the OS to "send data" to a file. "How *write* actually invokes the operating-system kernel depends on the underlying hardware. What typically happens is that *write* itself is a normal procedure that contains a special machine instruction causing a trap. This trap transfers control to the kernel, which then figures out why it was invoked and proceeds to handle the invocation. "

- And unintended requests for kernel service (traps)
  - Page faults (will define)
  - Dividing by zero
- Signals allow the kernel to call user-code in an "upcall"

NB: There is a wide variation in the nomenclature. On some computers the term *trap* refers to any interrupt, on some machines to any synchronous interrupt, on some machines to any interrupt not associated with input/output, on some machines only to interrupts caused by instructions with *trap* in their names, etc.



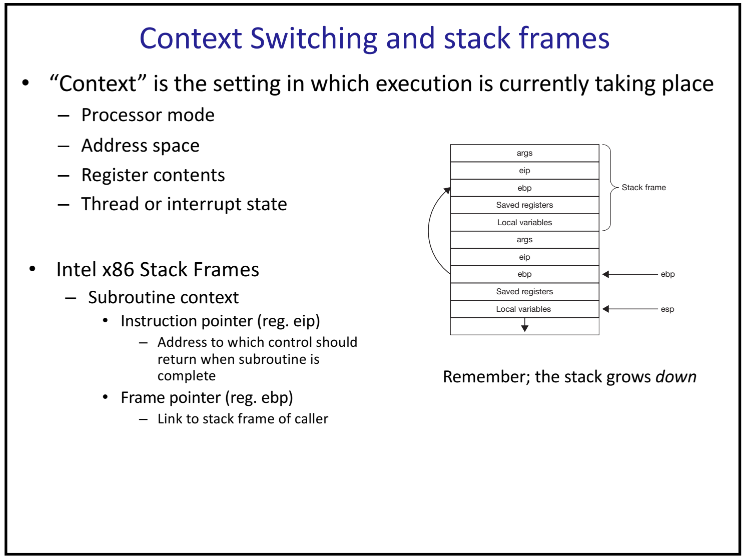
System calls

- Transfer control from user to system code and back
  - Need not involve a thread switch, just a "stack switch"
  - Trap (OS code) typically switches to a kernel stack frame

```
prog( ) {                  write( ) {
  . . .                      . . .
  write(fd, buffer, size);   trap(write_code);
  . . .                      . . .
}                          }
```

User
Kernel

```
trap_handler(code) {
  . . .
  if(code == write_code)
    write_handler( );
  . . .
}
```



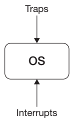
Context Switching and stack frames

- "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

Remember; the stack grows *down*

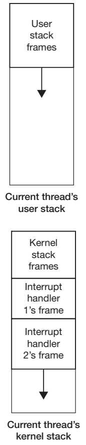
Interrupts (largely from hardware)

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



Interrupts

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


User stack frames
Current thread's user stack
Kernel stack frames
Interrupt handler 1's frame
Interrupt handler 2's frame
Current thread's kernel stack

Threads Implementations

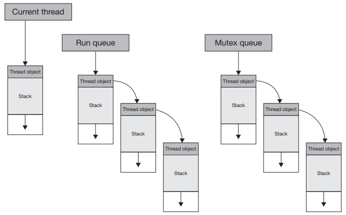
- OS goal is to transparently support user-level application programs running on a single set of shared hardware
- OS implementation and design issues largely related to how to switch running threads in support of traps and interrupts
- This brings up two major implementation concerns, how to implement
  - Scheduling
  - Synchronization
- And how to organize the operating system in terms of how to assign and organize actual processor threads to user-process threads through the OS
  - One-level model
  - Two-level model

OS Thread Implementation Strategies

- One-level model
  - Each user thread is mapped to a kernel thread
- Two-level model
  - Single kernel thread (e.g. 1-core CPU but not necessarily)
    - Each process gets one kernel thread
    - Threads multiplexed on this kernel thread
    - Synchronization primitives implemented via thread queue datastructures and yielding strategies
  - Multiple kernel threads (e.g. multi-core CPU but not necessarily)
    - Many kernel threads. User-level threads distributed across them
    - Avoids blocking problem of single-kernel thread model but introduced complications
- Other approaches exist …

One Hypothetical Thread and Synchronization Implementation

- User-level simple thread package “straight-threads” implementation
- Assume
  - One processor thread
  - No interrupts
- Assert
  - Thread object datastructure which has
    - Stack
    - etc.
  - Current thread pointer
  - Run queue datastructure
    - threads waiting to run
  - Mutex queue datastructure
    - threads waiting to own mutexes, one for every mutex



Yielding the Processor

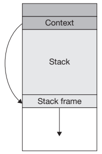
- Assume that threads must voluntarily yield by calling system call, e.g. imagine an OS that, for instance, requires explicit yielding if trying to own a mutex fails. Such a yield call would look like

```
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

}
```



- Here swapcontext, saves the caller’s register context in its thread object, then restores that of the target thread from its thread object

Implementing Mutexes

- Because our hypothetical OS does not have interrupts by assumption 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));
}
```

What about Multiple Processor Threads?

- thread_switch() now insufficient because mutex_lock() and mutex_unlock() could be called simultaneously
- Actual concurrent threads like this require actual thread synchronization
  - Synchronization implementation has big OS performance impact
- Types of actual implementation
  - 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 (actual synchronization) can be implemented
  - Note mutex with 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
     }
   }
}
```

Spin-Lock Implementation of Blocking Mutex In Hyp. OS

- Spin-locks consume processor resource and thus should be used sparingly
- blocking_lock works as before – threads waiting on mutex 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 subtle bug arising on true multiprocessor systems
  - Left for example sheet

Scheduling – Sharing in a Transparent Way

- 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 Sharing

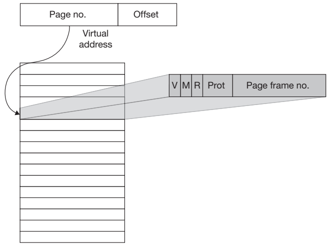
- Efficient implementations require deep understanding of hardware capabilities and software requirements
- Involves
  - Memory abstraction
  - Optimizing against available physical resources
    - High-speed cache
    - Moderate-speed primary storage
    - Low-speed secondary storage
- Security
  -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

Virtual Memory Provisioned via Per-Process Page Table *

- Operating system maintains a *page table* for each running process that maps virtual memory *pages* to real memory pages.
- Every memory access in a thread is intercepted by the OS/hardware and translated to a real memory address
- This is extremely important to high performance and relies upon large amounts of hardware acceleration

- Assume
  - 32-bit virtual address
  - Page size 4096 bytes
    - Implies
      - 12 bit offset (2^12 = 4096)
      - 20-bit page number (2^32 / 2^12)

- V = validity bit
  - If set, page frame no. is high-order bits of address in real memory
  - If not a page-fault occurs and the OS takes over to allocate or load a page
- 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.

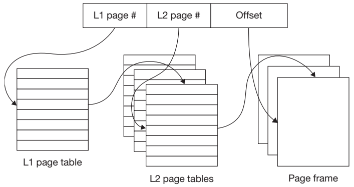


On a 32-bit arch. the page table is 2^20 * 4 bytes = 4MB.

What about a 64-bit architecture? *

Forward-Mapped (FM) Page Tables

- Page tables take a lot of memory, FM page tables off lower-overhead approach
  - 32 bit machine, 4MB *per process*
- e.g. solution: each virtual address divided into two 10-bit numbers
  - L1 page number
  - L2 page number
  - Offset



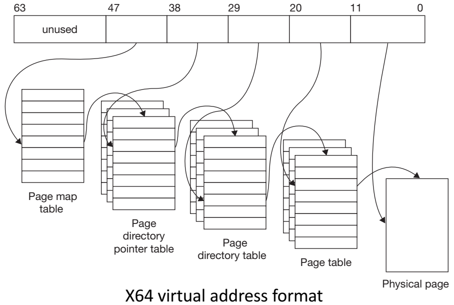
- Advantages
  - Lower overhead – e.g. not all L2 pages need be in memory at once
  - Pages can be lazily allocated on demand
- Disadvantages
  - More lookups

Note : memory access is slow; caching is imperative

- Hardware supports address translation via "translation lookaside buffers"
  - Fast processor-based memory containing some entries of address translation table
  - Automatic translation in hardware

64-Bit Issues

- Assume 8-Kb pages, how big is a page table for a 64-bit arch?
  - HUGE! (Example sheet)
  - One solution, don't use entire address range and use forward mapping aggressively



X64 virtual address format

Operating-System Memory Management

- 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?

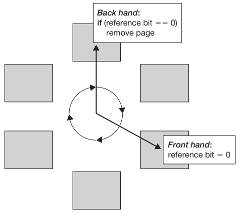
OS Response to a Page Fault *

- An interrupt that occurs when a program requests data that is not currently in real memory. The interrupt triggers the operating system to fetch the data from a virtual memory and load it into RAM. An invalid **page fault** or **page fault** error occurs when the operating system cannot find the data in virtual memory.

- Steps
  - Detect page fault
  - Find a free page frame (in real memory)
  - Write a page out to secondary storage if none free (*swapping*)
  - 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 Implementation 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
  - OS uses a dediciated page-out thread (e.g.linux: kswapd)
    - One hand sets reference bit to 0
    - Other hand triggers page flush if another thread hasn't set reference bit to 1



Efficient Fork via Copy-on-Write

- Can `fork()` be made less expensive to implement?
  - Remember `fork()` copies a process' entire memory space
- Lazy evaluation
  - Let copies share address space, in particular page tables
  - Mark all pages read only
  - On write() make copies unique to each process, update individual process page tables appropriately
  - OS bookkeeping requires care

Shared Memory and `mmap()` *** `(mmap_shared_memory_example.c)`

- `mmap()` maps files to contiguous virtual memory
- Files may be mapped to address space shared across processes!
  - Shared
    - Modifications seen by all forked processes (shared memory parallel processing!)
  - Private
    - Modifications remain private to each forked process (copy on write)

Review

- OS Sharing of processor and memory
  - Threads implementation
    - Entails multiplexing threads 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
    - Page tables entail translating between virtual and real memory addresses at runtime

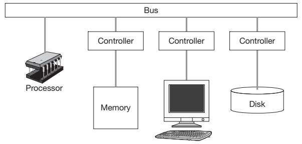
Lecture 4 : File Systems & Networking

Material from
Operating Systems in Depth
(spec. Chapters 6 and 9)
by
Thomas Doeppner

GET THIS BOOK AND READ IT!

I/O Architecture Types (Simplified Overview)

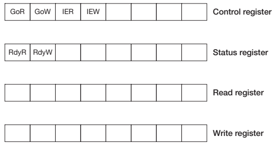
- Memory-mapped
  - Each device has a controller
  - Each controller has registers
  - Registers appear to processor as physical memory
  - Actually attached via a bus
- Categories of I/O devices
  - Programmed I/O (PIO)
    - One word per read/write
    - e.g. terminal
  - Direct memory access (DMA)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



If an OS were written in C++ a device driver would be a class with instances for each device.

PIO and DMA Example

PIO (terminal)



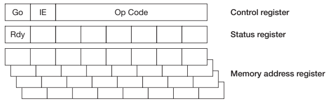
Legend:		
	GoR	Go read (start a read operation)
	GoW	Go write (start a write operation)
	IER	Enable read-completion interrupts
	IEW	Enable write-completion interrupts
	RdyR	Ready to read
	RdyW	Ready to write

Usage:

1. Store byte in write register
2. Set GoW bit in control register
3. Wait for RdyW in status register

4. Can request an interrupt

DMA (disk)



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

Usage:

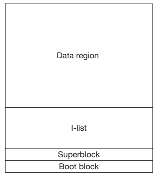
1. Set disk address in device register
2. Set memory address in memory address register
3. Set Op Code, Go, IE in control register

File Systems

- Purpose
  - Provide easy-to-use permanent (with respect to process lifetime) 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

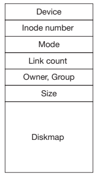
Basics

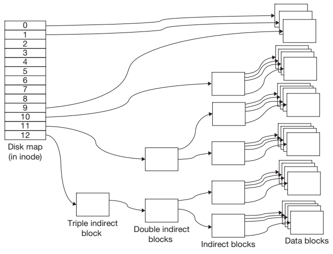
- Pedagogical review of Unix system 5 File System (S5FS)
- Revolutionary, simplifying Unix file abstraction
  - A file is an array of bytes, period.
- File system layout
  - Boot block
    - First-level boot program that reads OS into memory
  - Superblock
    - Describes layout of remaining filesystem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



Unix's S5FS

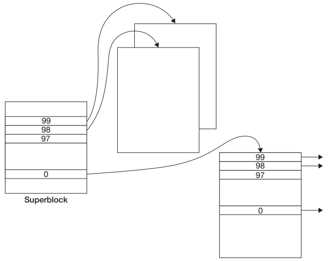
- Each file is described by an inode
- Directories are files containing names and inode numbers
- Diskmap
  - Maps logical blocks numbered relative to the beginning of the file to physical blocks numbered relative to the beginning of the file system
  - Assume
    - Block length = 1024 bytes
    - 13 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





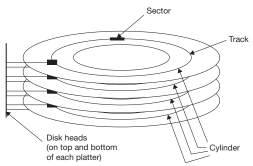
Organizing Free Storage on Disk

- 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



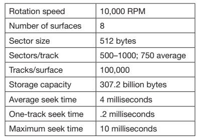
Disk Architecture

- 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

- 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

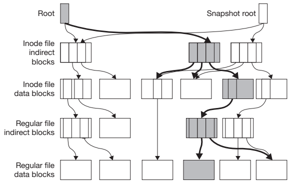
- S5FS inode table is a fixed array
  - Requires predicting number of files the system will have
  - Can't add more disk space to the file system
- Solution
  - Treat inode array as a file
  - Keep inode for the inode file at a fixed location on disk
    - Backup

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)

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

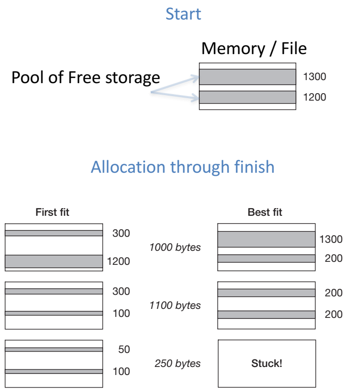


Directories and Naming

- 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!

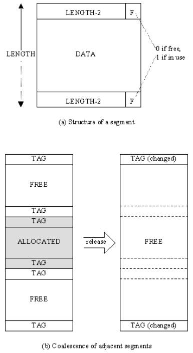
(Dynamic) Storage Allocation

- Storage allocation is very important in OS's
  - Disk
  - Memory
- Example
  - 1000, 1100, 250 bytes in order
- Competing approaches
  - First-fit
  - Best-fit
- Knuth simulations revealed (non-intuitively) first-fit was best
  - Intuition : best-fit leaves too many small gaps



Freeing Storage Is More Complex

- Knuth : ref; "boundary-tag" method and algorithm
  - Combines free segments greedily upon release
  - Requires datastructure that represents free or not-free
- Helps avoid "fragmentation"
  - External
    - Free spaces too small
  - Internal
    - Allocated memory unnecessarily too large (this situation arises in different, not-covered allocation approaches like the "slab" approach)



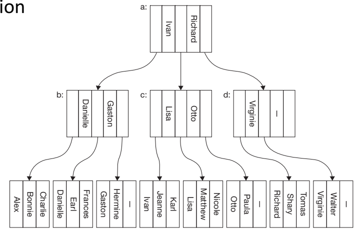
(a) Structure of a segment

(b) Coalescence of adjacent segments

FIGURE C-2. THE BOUNDARY TAG METHOD

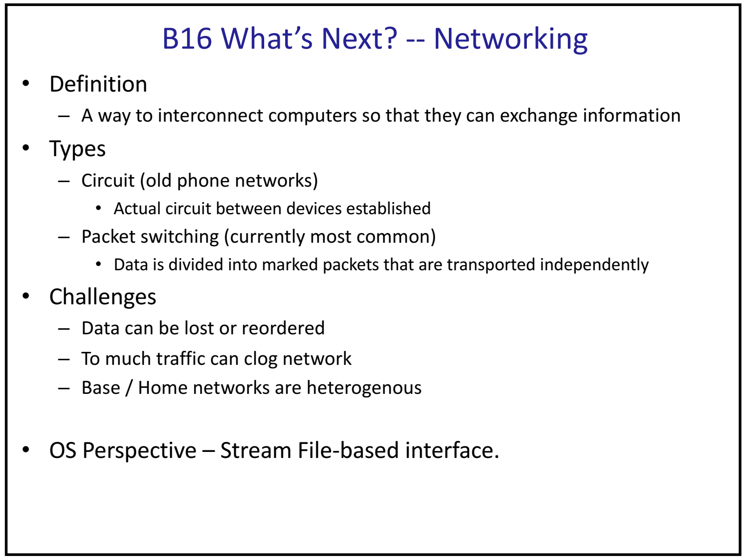
Data Structures for Nonvolatile Storage : B Trees

- 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



Wrap - Up

- Covered a lot of ground, major points include:
- Using an OS effectively
  - Process
  - Thread
    - Synchronization
  - Virtual memory
  - File
  - System calls
- Implementing an OS if you had to
  - Page tables
  - Filesystems
  - Thread switching
  - Synchronization



B16 What’s Next? -- 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 heterogenous

- OS Perspective – Stream File-based interface.