B16 Operating Systems

Introduction

Learning Outcomes (Examinable Material *)

- Familiarity with operating system concepts
  - File
  - Process
  - Thread
  - Synchronisation
  - Memory
  - Paging
  - Socket
  - Port

- Datastructures / implementations
  - Page table
  - Semaphore
  - Mutex
  - Socket

Perspective

- User perspective *
  - Linux (posix compliant OS)
  - System calls (fork, wait, open, printf)
  - Command line utilities (man <section>)
  - C programs

- Operating system *implementation* perspective
  - “Simple-OS”

B16 Operating Systems

Lecture 1 : History and User Perspective

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
  - Improvement possible
    - New algorithms, new storage media, new peripherals
    - New concerns : security
    - New paradigms : the "cloud"

Review : Computer ≈ Von Neumann Architecture

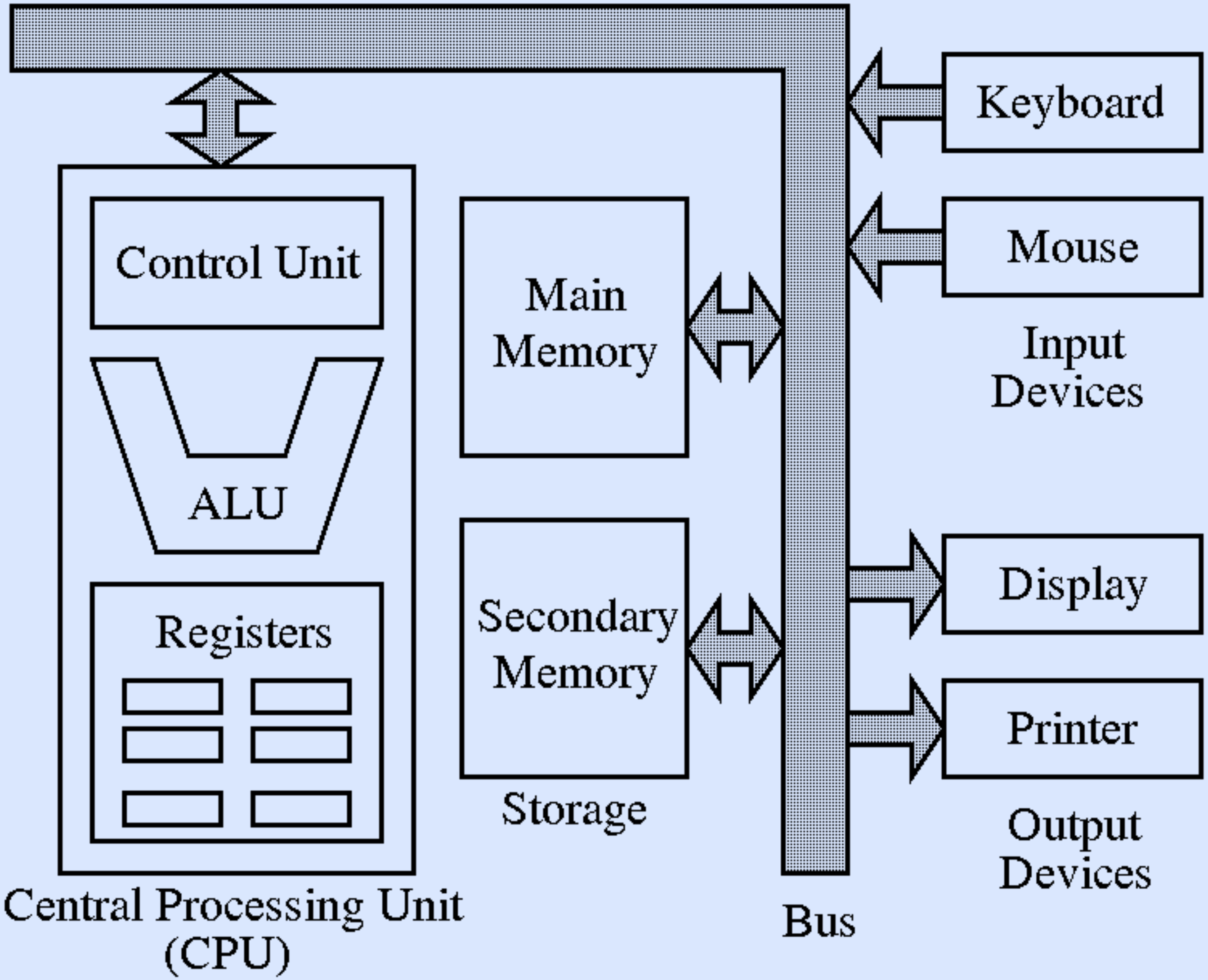


Image from http://cse.iitkgp.ac.in/pds/notes/intro.html

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)

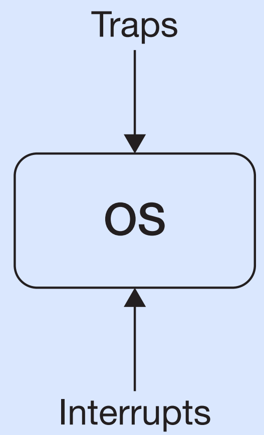
- 1970's : CP/M
  - One application at a time – no protection from application
  - Three components
    - Console command process (CCP)
    - Basic disk operating system (BDOS)
    - Basic input/output system (BIOS)
- Apple DOS (after CP/M)
  - 1978 Apple DOS 3.1 ≈ CP/M
- Microsoft
  - 1975 : Basic interpreter
  - 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

Implementation Perspective : “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 (largely from user)

- *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

- Unintended requests for kernel service
  - Using a bad address
  - Dividing by zero

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

Processes *

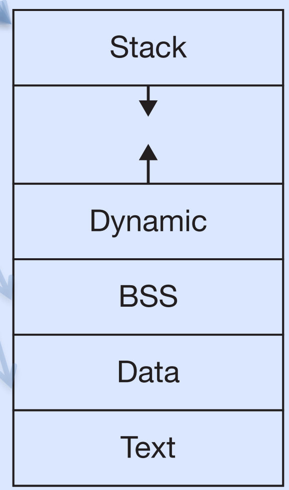
- Abstraction that includes
  - Address space (*virtual memory* *)
  - Processors (*threads* of control *)
- Usually disjoint
  - Processes usually cannot directly access each other's memory
    - Parallel processing via pipes, shared memory, etc.
- Running a program from the shell
  - Creates a "process"
  - Program is loaded from a file into the process's address space
  - Process's single thread of control then executes the program's compiled executable code

Memory = Address Space = e.g. 2^32 words, etc.

- 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

- 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);
}
```



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

- Processes are created via the system call fork()
  - Any exact copy of the calling process is made
    - Efficient – copy on write
  - fork() returns *twice!*
    - Once in the child (return value 0)
    - Once in the parent (return value the PID of the child process)
- 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 */
}
```

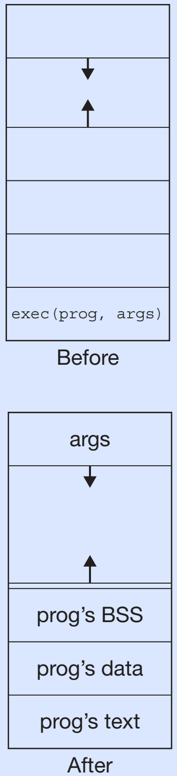
Loading Programs into Processes (fork_example_2.c)

- execl() 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 */
   ;
```



- execl() replaces the entire contents of the processes address space
  - the stack is initialized 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 ***everything***
  - Keyboard
  - Display
  - Other processes
- Naming files
  - Filesystems generally are tree-structured directory systems
  - Namespaces 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 (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()

```
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);
}
```

- Reverses a file

Pipes * (pipe_example.c)

- 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

- "man 2 open" :

```
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 Simple OS

- Rough idea of what goes inside an OS
- Traps / system calls
  - exec()
  - fork()
  - open()
  - pipe()
  - exit()
  - close()
  - read()
  - write()
  - dup()
  - …
- Next lecture : more user basics.
- Final two lectures : OS implementation 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 * (thread_example_1.c)

- What is a thread?
  - Mechanism for concurrency in user-level programs
  - "Lightweight process"
  - Processor(s) 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 Arguments to Threads

- Care must be taken with threads in general
- Problem with this code
  - In and out are local variables thus leave scope when rlogind exits

```
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);
}
```

Thread Termination (thread_example_2.c)

- 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

- "`man pthread_attr_init`"
- e.g. 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 *** (thread_example_3.c)

- 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 Mutexes ***

- 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);
```

Mutual exclusion can result in DEADLOCK!

- 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 avoidance 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);
}
```

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

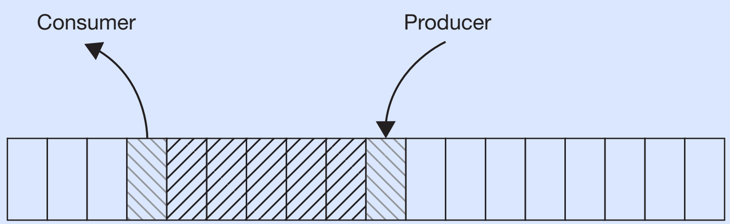
```
sem_t semaphore;
int err;

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()

OS Implementation Problem : Producer-Consumer *

- 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



Semaphore sol'n to the producer-consumer problem

- Example sheet

Deviations

- Signals
  - Force 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
- 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);
}
```

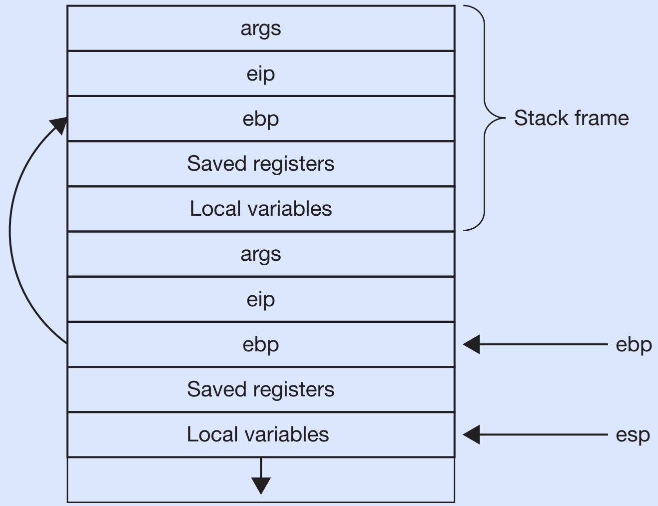
Other Basic OS Concepts

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

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



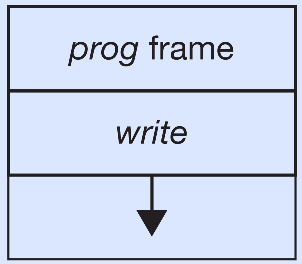
Remember; the stack grows *down*

System calls

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

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

```
write( ) {
  . . .
  trap(write_code);
  . . .
}
```

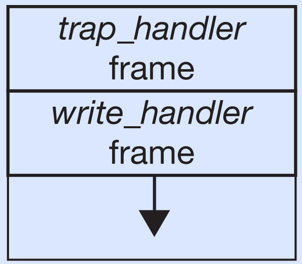


User stack

User

Kernel

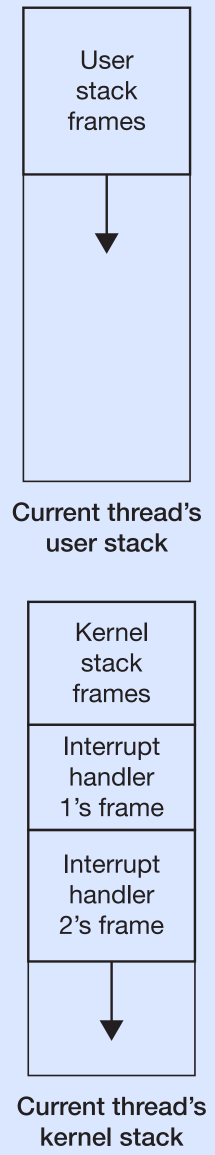
```
trap_handler(code) {
  . . .
  if(code == write_code)
    write_handler( );
  . . .
}
```



Kernel stack

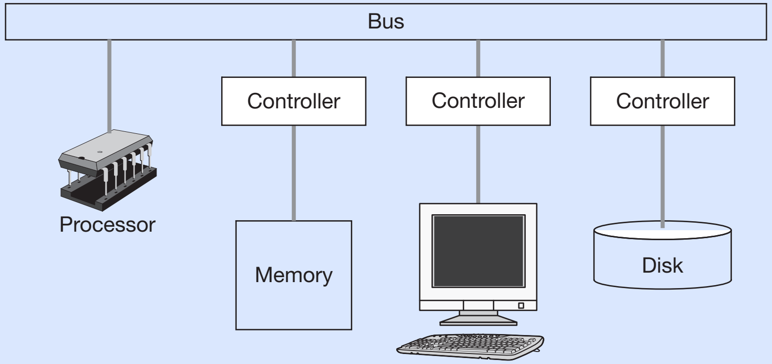
Interrupts

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



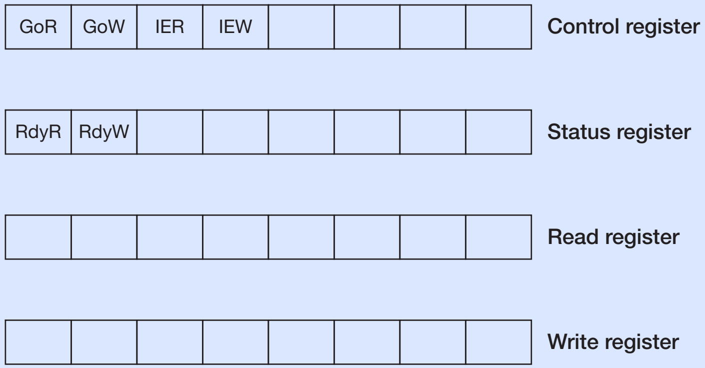
I/O Architecture Types (Simplified)

- 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

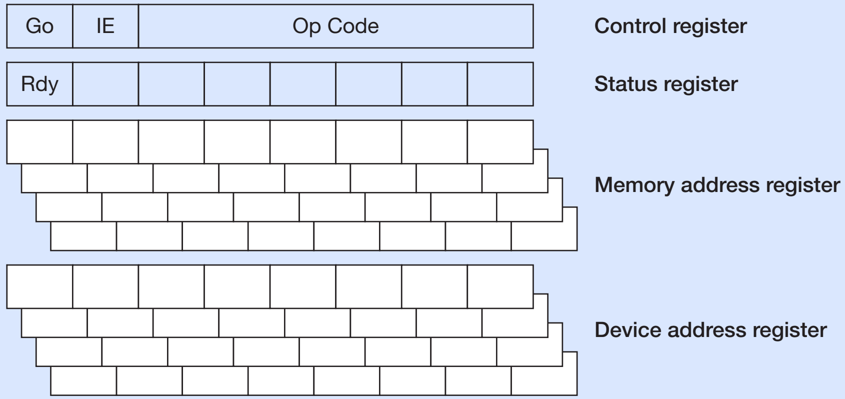
PIO



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



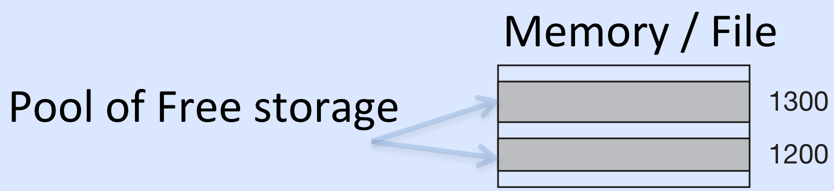
Legend:

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

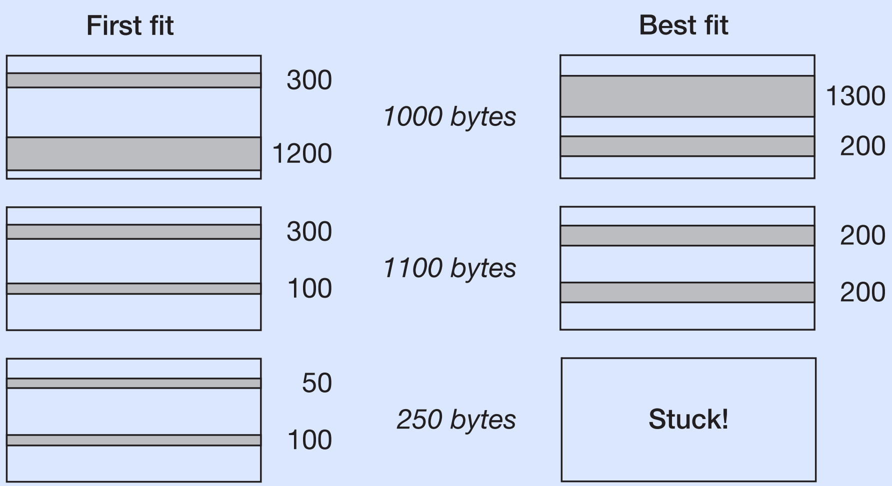
(Dynamic) Storage Allocation

- Storage allocation is very important in OS's
  - Disk
  - Memory
- 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

Start



Allocation through finish



Freeing Storage Is More Complex

- Knuth : “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)

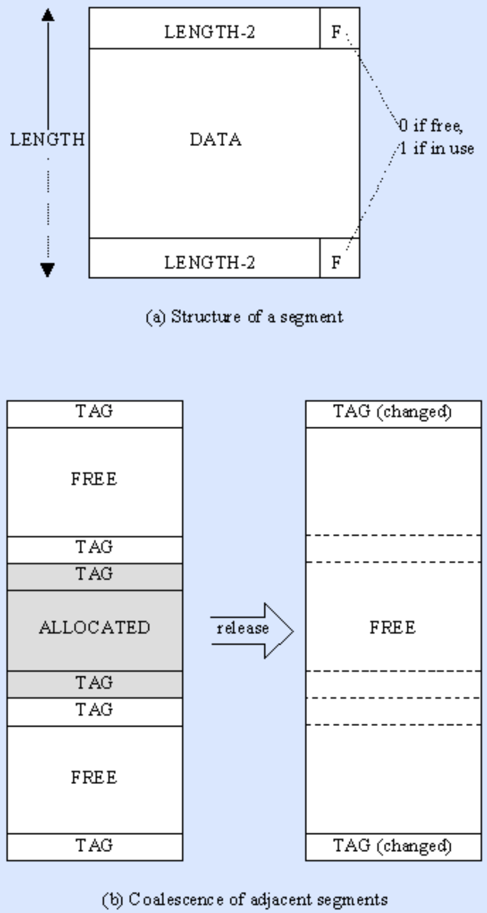


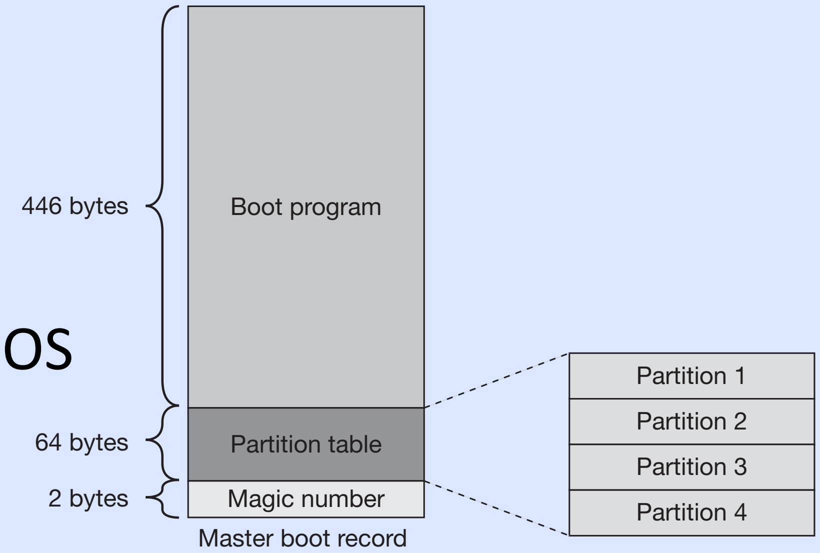
FIGURE C-2. THE BOUNDARY TAG METHOD

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



Review

- OS essentials
  - Threads
  - Context switching for management of processors
  - I/O for file systems
  - Dynamic storage allocation