CSC 415: Lecture 2

Homework

snprintf() and MSVC

- Visual Studio 2014 / 2015 have snprintf()
- Older versions have _snprintf()
- _snprintf() != snprintf()
- snprintf() always NUL-terminates, even on truncation
- _snprintf() does not terminate on truncation

snprintf() Example

```
char buf[3];

snprintf(buf, sizeof(buf), “foo”);
```

- `buf[]` will contain ‘f’, ‘o’, ‘\0’
- If you use `_snprintf()`, `buf[]` will contain ‘f’, ‘o’, ‘o’ with no nul terminator

snprintf() Workaround

- If you are using Visual Studio < 2014, use _snprintf() and explicitly nul terminate:

```
char buf[4];

_snprintf(buf, sizeof(buf), “foo”);
buf[sizeof(buf) – 1] = ‘\0’;
```

Topics

- Review from Chapter 1
- System Calls
- Operating System Structures
- Operating System Debugging
- C Programming

Iterative Design

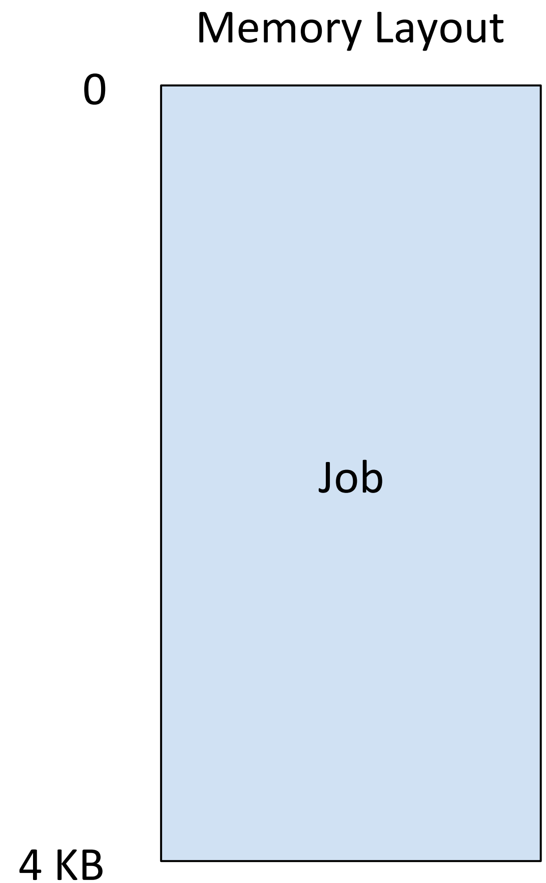
- Experiences from Existing Systems Inform Design of Future Systems
- New Systems Built on Old Systems
  - Don't Build From Scratch Each Time
- Affects Hardware & Software
- History Sometimes Explains Quirks / Complexity

Evolution of Process Scheduling

- Dedicated Machine
- Multiprogramming
- Multitasking

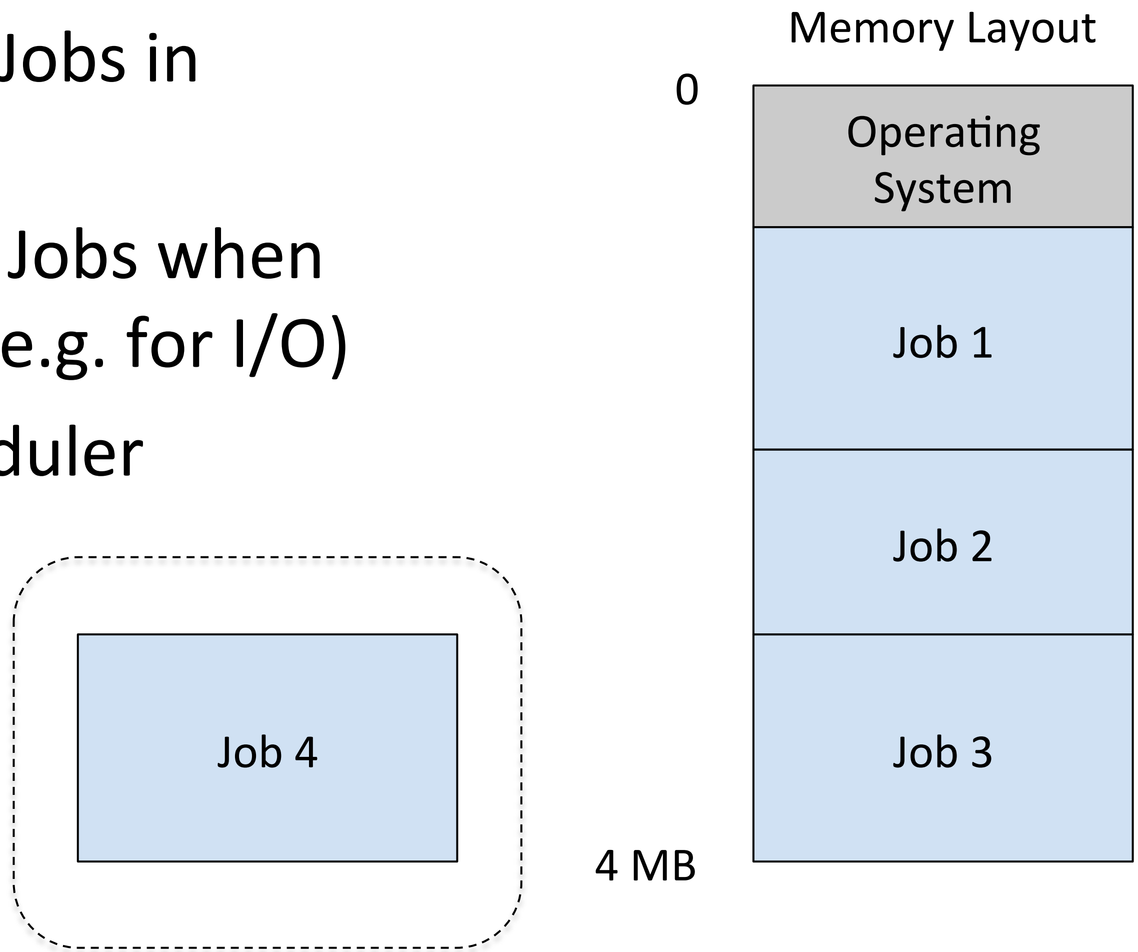
Dedicated Machine

- Single Job
- No Need to Arbitrate Resources
- CPU Idle While Waiting



Multiprogramming

- Multiple Jobs in Memory
- Switches Jobs when Waiting (e.g. for I/O)
- Job Scheduler

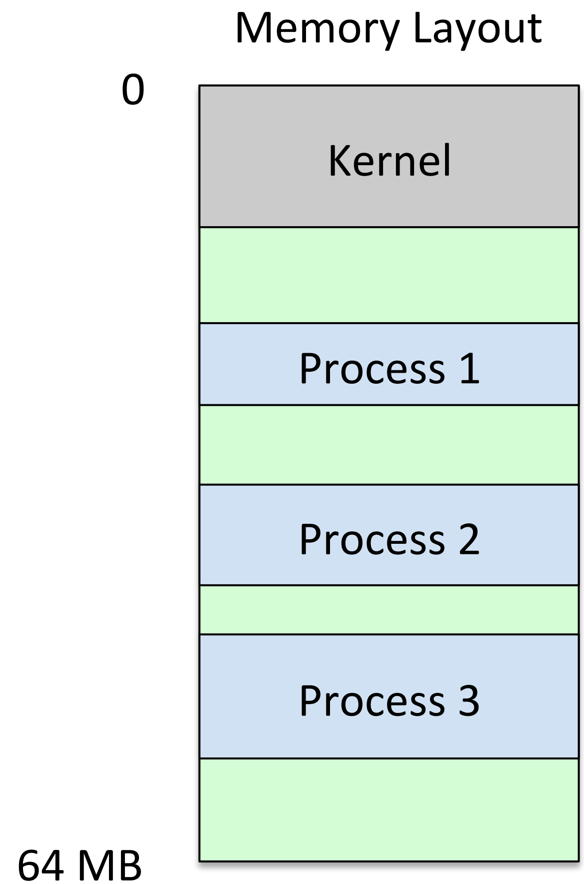


Context Switch

- CPU switches from one process to another
- Steps
  - Set status of old process to Suspended
  - Save state (registers) from old process in a Process Control Block (PCB)
  - Load registers with values from new process' PCB
  - Stack Pointer is a register!
    - Swapping stack pointers swaps stacks
  - Set status of new process to Running

Multitasking

- Schedules Process Switches Frequently
  - Timer Provides Event
  - Many Each Second
  - Appearance of Concurrency
- Often Paired with Virtual Memory



Sequence

- Dedicated Machine
  – Only One Job at a Time (Wasted CPU Cycles)
- Multiprogramming
  – Multiple Jobs
    - Memory Sharing & Context Switches
  – Not Interactive, Batch Only
- Multitasking
  – Interactive Processes

Event Driven

- Operating System Responds to Events
  - Events have associated Handlers
- Operating System Borrows Context
  - Event Handlers Run in Kernel Mode of Existing Process
    - Does Not Switch to Some “Kernel” Process
  - May Defer Some Work to Kernel-only Processes

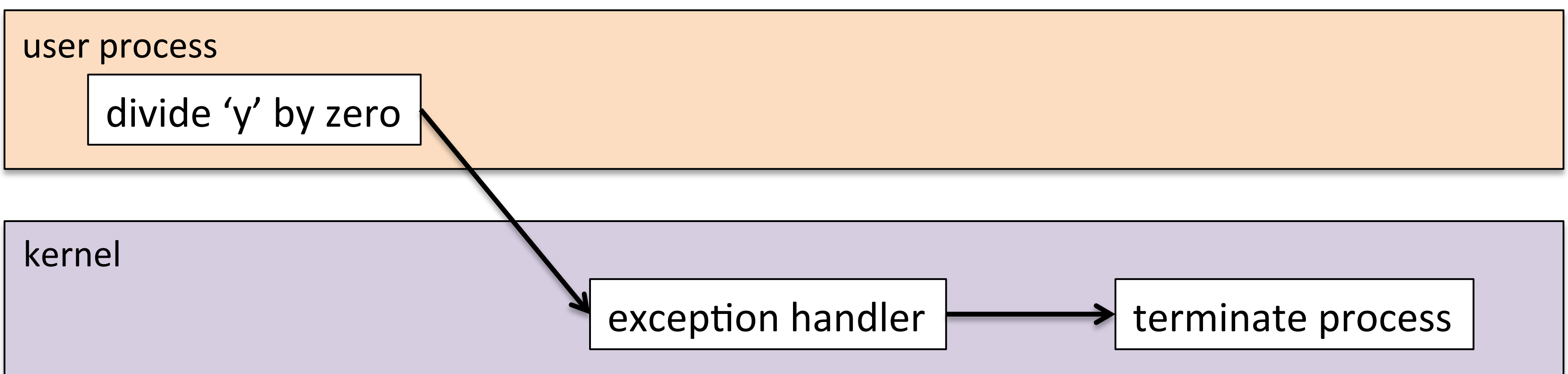
Classes of Events

- Exception
  - Direct result of a CPU instruction
  - Errors (e.g. divide by zero, use NULL pointer)
  - Requests (e.g. System Calls)
- Interrupt
  - External event not result of current instruction
  - I/O Device (e.g. Key pressed on keyboard)

Exception Timeline

- User Code:

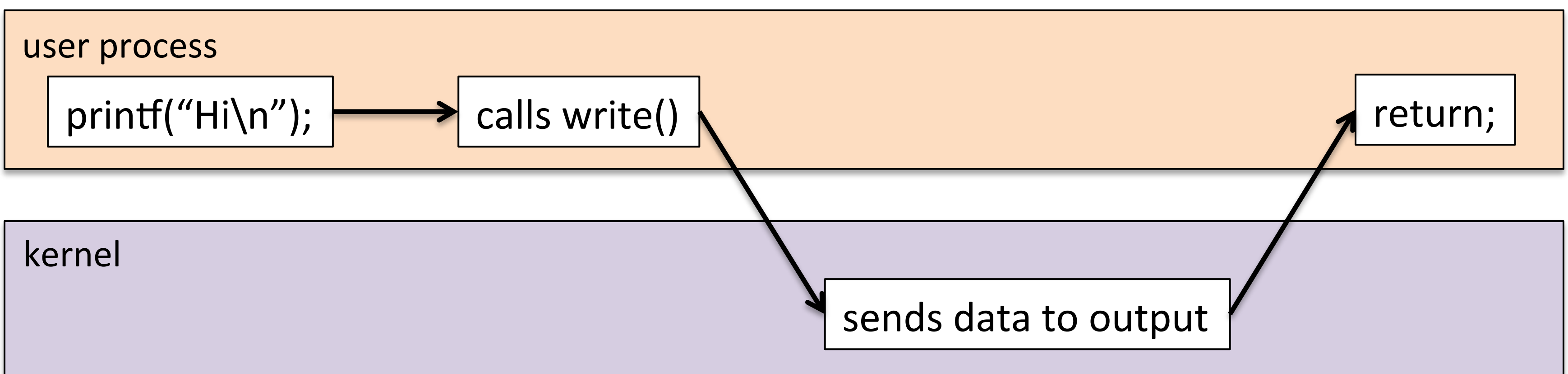
```
x = y / 0;
```



System Call Timeline

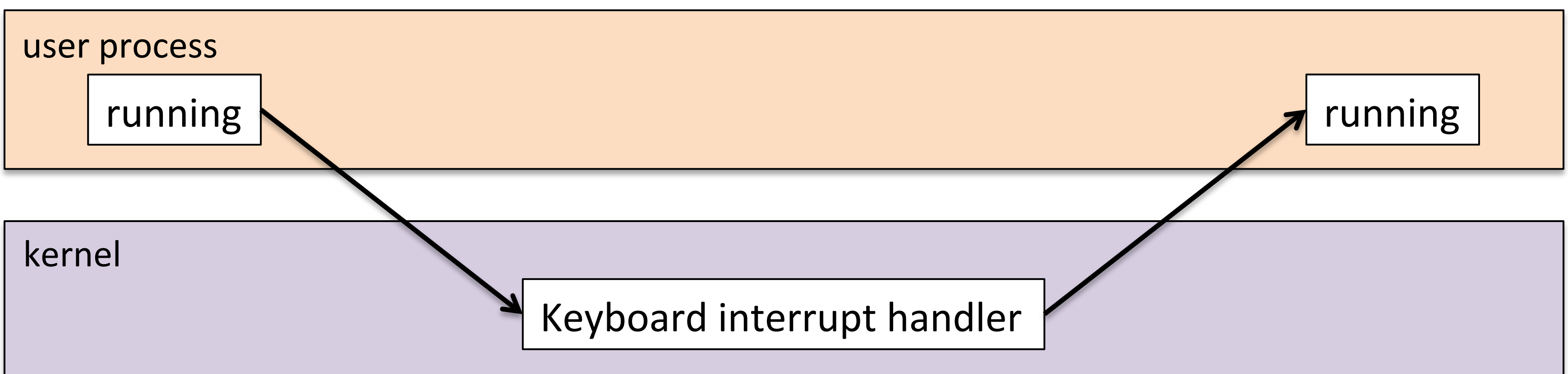
- User Code:

```
printf("Hi\n");
return;
```



Device Interrupt Timeline

- User Presses Key on Keyboard
- Handler Saves Character in Buffer



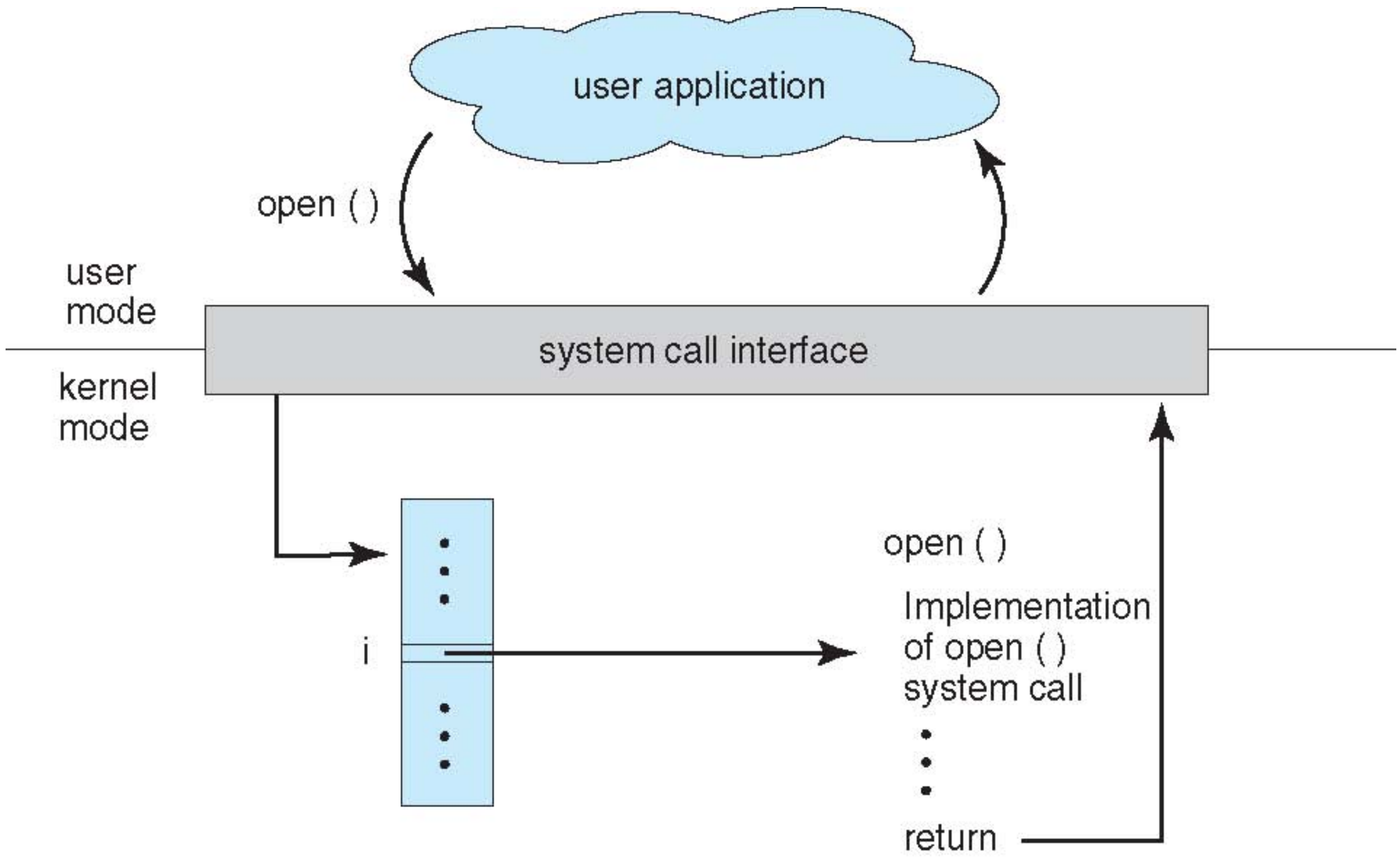
Topics from Chapter 2

- System Calls
- Operating System Structure
- Operating System Debugging

System Calls

- Interface for applications to request services from the system
- Typically have wrappers in runtime libraries
  - libc on UNIX-like systems
- Special calling convention
  - Software trap ("syscall" instruction on 64-bit x86)
  - Request identified by one of the arguments

API – System Call – OS Relationship



Argument Passing

- All in registers
  - Linux on 32-bit and 64-bit x86
  - FreeBSD on 64-bit x86 (can spill to stack)
- Arguments pushed on stack
  - FreeBSD on 32-bit x86 (all but request ID)
- Single register points to parameter block
- One argument selects call to make
  - %eax / %rax for x86 on Linux and FreeBSD

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

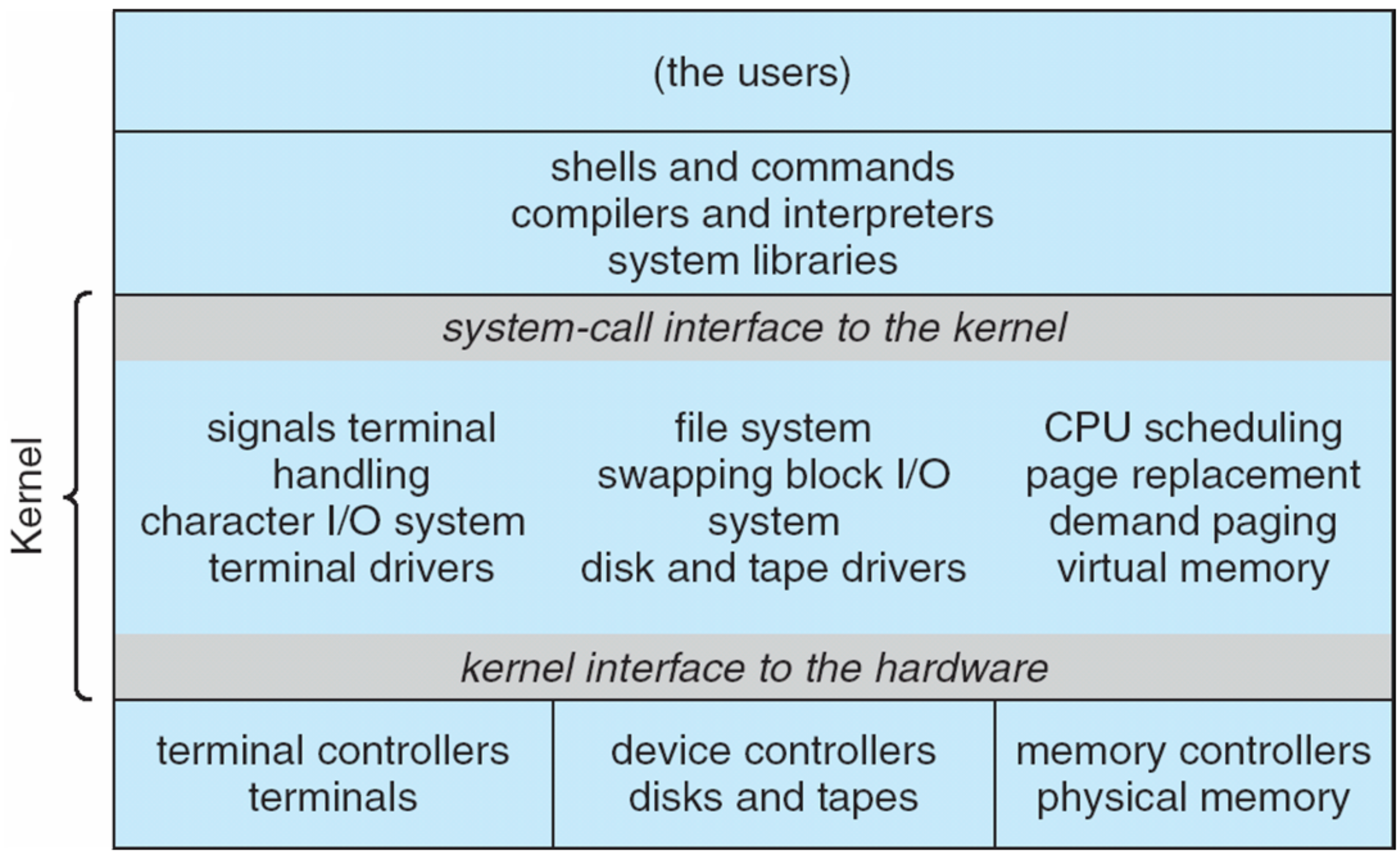
Operating System Structures

- “Just a big C program”
  - Large programs need structure / organization
- OS Structure Types
  - Monolithic
  - Microkernel
  - Modular

Monolithic Kernel

- Single, self-contained program
- Shared address space
  - Fast and easy to share data
  - Bug in one part can corrupt memory used by another part

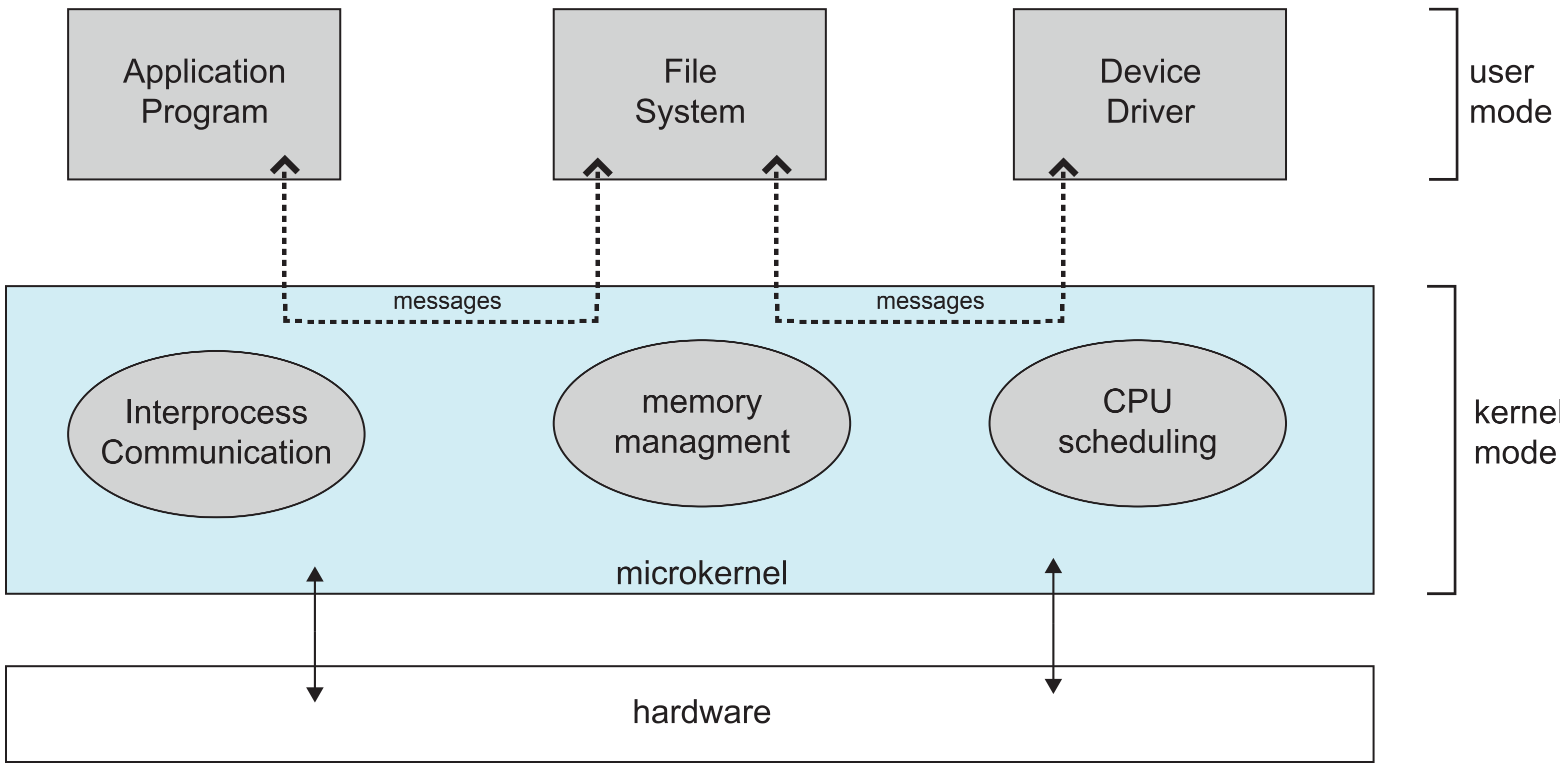
Traditional UNIX System Structure



Microkernel

- Moves parts of kernel into user processes
- “Core” kernel remains, but smaller
- Message Passing
- More reliable
- Significant overhead from copying messages

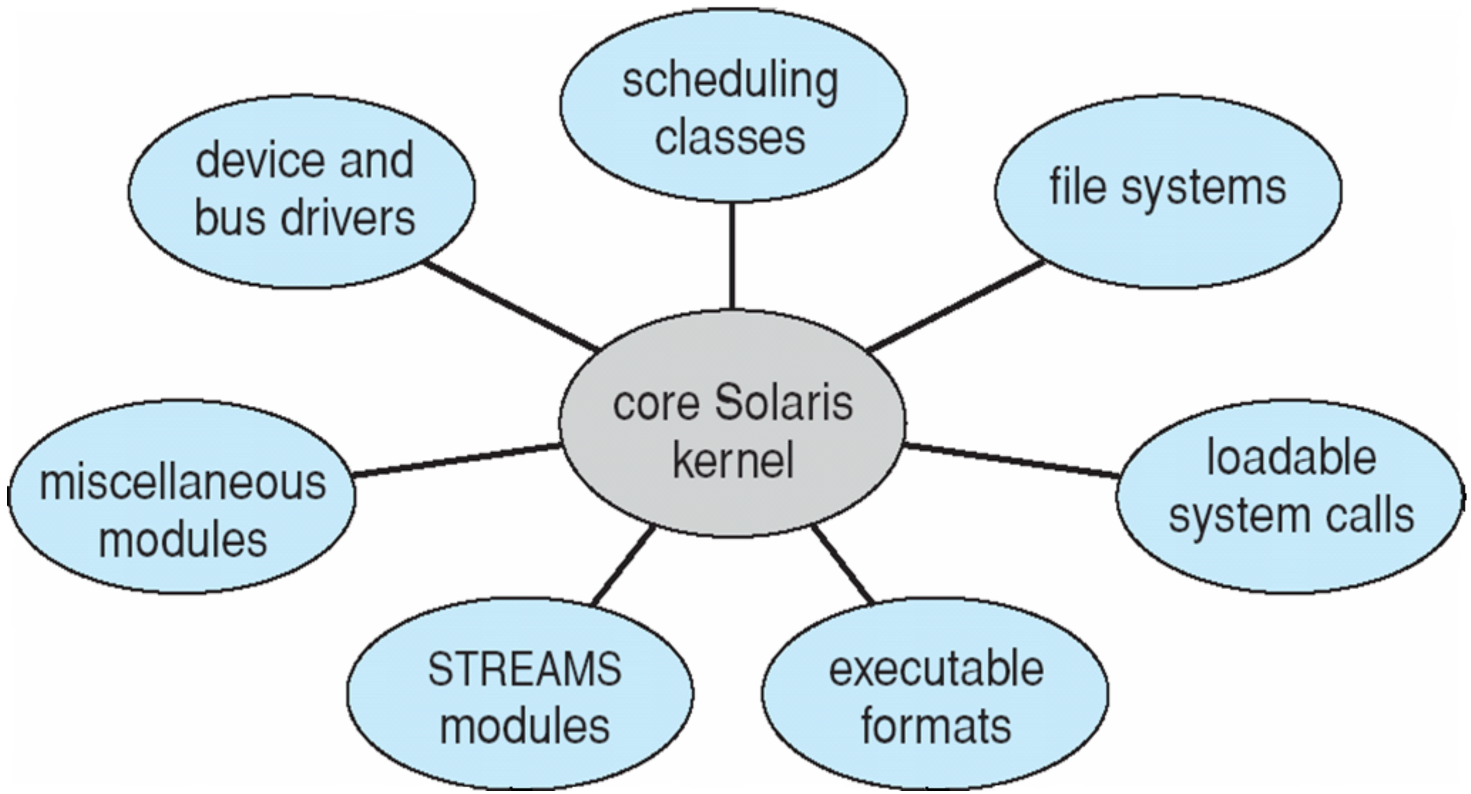
Microkernel System Structure



Modular Kernel

- Splits parts of kernel into separate modules
- Modules use functional interfaces to communicate
  - Similar to OOP
- Can load some modules at runtime
- Better suited to certain parts of the kernel
  - Device drivers are well-suited
- Usually shared address space like monolithic

Solaris Modular Approach



Operating System Debugging

- System-wide Performance Analysis
- Inspecting Process Interactions
  - Kernel Responds to Events
  - Often Need Information about the Events
- Kernel Debugging
  - Undesired Outputs
  - Crashes

Hardware Counters

- CPU Performance Counters
  - Per-CPU counters of events
  - Can apply to process or system
- I/O Device Statistics
  - Packet and byte counts on a network interface

Counting CPU Instructions

- How many CPU instructions to run a process

```
# pmcstat -p INSTR_RETIRED_ANY echo Hi
Hi
# p/INSTR_RETIRED_ANY
        3632693
```

Software Counters

- Kernel counts software events
  - Packets passing through network stack
  - Disk I/O requests
  - Interrupts handled

Network Statistics

- ‘netstat 1’ on OS X / FreeBSD shows per-second counts of network events

> netstat 1

input		(Total)	output			
packets	errs	bytes	packets	errs	bytes	colls
42	0	28733	37	0	5562	0
17	0	1725	12	0	818	0
1	0	219	2	0	289	0

Process State

- Determine which processes are running with tools like ps(1) or top(1)

```
% ps
  PID TT  STAT     TIME COMMAND
  868  0  Is    0:00.03 -tcsh (tcsh)
59621  1  Ss    0:00.04 -tcsh (tcsh)
59677  1  R+    0:00.00 ps
```

Tracing

- Logging events as they occur
- Tracing system calls invoked by a process:
  - strace (Linux)
  - truss (BSDs)
  - dtruss (OS X)

Kernel Debugging

- System Crashes similar to Process Core
  - Often a modified debugger (e.g. kgdb)
- Sadly, much printf() / printk()
  - Debugging services for userland hard to provide in kernel mode
  - Buffer of recently traced events
- Virtual Machines
  - Can suspend guest OS without cooperation

C Programming

- Pointers and memory addresses
- Dynamic allocation
- Function pointers

Introduction

Should know basics of C/C++

Topics to review:

- External declarations
- Pointers and pointer arithmetic
- Data structures using pointers
- Function pointers

Extern Declarations

Suppose we have 3 files:
my_incl.h, prog1.c, prog2.c

my_incl.h

```c
extern int my_var;
```

prog1.c

```c
#include "my_incl.h"
void f()
{
        my_var = 1;
}
```

prog2.c

```c
#include "my_incl.h"
int my_var;
void g()
{
        my_var = 0;
}
```

Compiler looks at my_incl.h:

- `my_var` is an external identifier (it's an int)
- No memory is allocated (yet)
- But other programs can now use `my_var`!

`my_var` must be properly declared (without `extern`) in some file.

In prog2.c, `my_var` is declared properly; memory is allocated for it.

Linker *resolves* all references to `my_var`; it's the variable declared in prog2.c.

Memory allocation and pointers

Compiler allocates memory for each variable.

Pointers contain addresses of variables (of the correct type);
&x means *address of x*

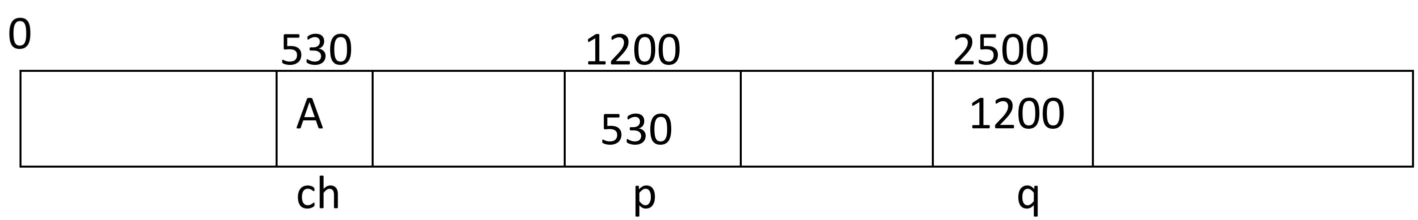
```
int main() {
  int x = 1;
  int y = 200;
  int *ptr;

  ptr = &x;
}
```

	Code for main()
x: 1000	1
y: 1004	200
ptr: 1008	1000

Dereferencing pointers

What is printed?

```
void main()
{
    char ch = 'A';
    char *p = &ch;
    char **q = &p;
    cout << ch << endl;
    cout << (int) p << endl;
    cout << (int) q << endl;
    cout << *p << endl;
    cout << (int) *q << endl;
    cout << **q << endl;
}
```



Casting

Consider assignment: `x = y;`

- x and y must have the same type, or
- *cast* y to be the same type as x

Example:

```
char *c;
c = 1000; // error: c is pointer, 1000 is int
```

This is ok (tell compiler to assume 1000 is address):

```
c = (char *) 1000; // no error
```

Using pointers and addresses

```
void kernel_main()
{
      char* screen_base = (char *) 0xB8000;
      *screen_base = 'A';
}
```

screen_base 0xb8000

0xb8000 'A'

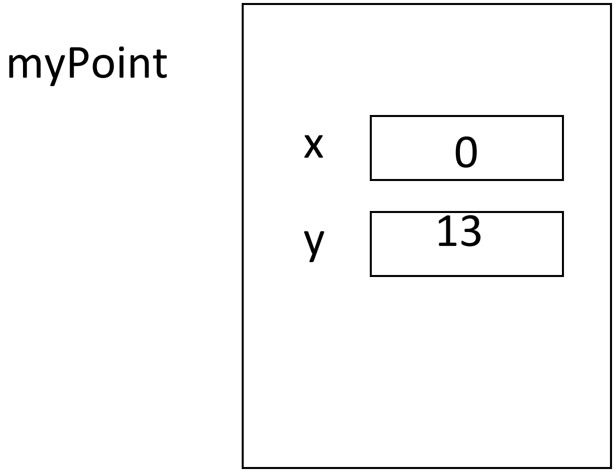
- Normally we do not explicitly place numeric addresses into pointers! (Will probably crash in Unix, Windows, because of memory protection.)
- But OS kernel code may have control over specific addresses; this may be ok.

C struct (quick review)

C struct's are like primitive classes (without code!)

Define a struct:

```
typedef struct _Point
{
      int x, y;
} Point;
```



Declare a Point:

```
Point myPoint;
```

Access fields of myPoint:

```
myPoint.x = 0; // set coordinates to (0, 13)
myPoint.y = 13;
```

Access myPoint through a pointer:

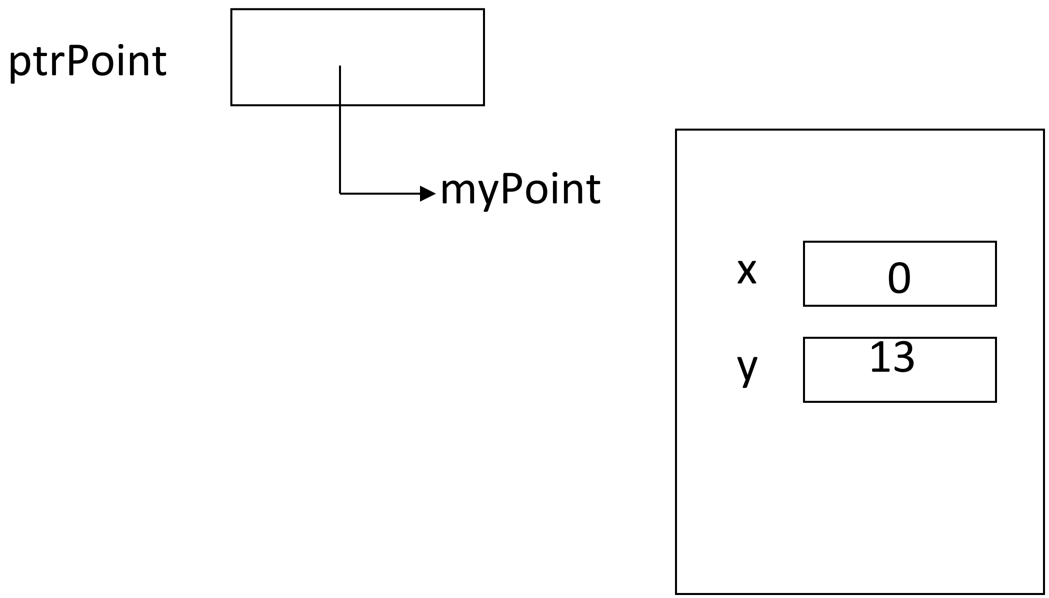
```
Point *ptrPoint;
ptrPoint = &myPoint;   // set ptrPoint to point to myPoint
```

Access fields of myPoint through ptrPoint:

```
(*ptrPoint).x = 0; // set coordinates to (0, 13)
(*ptrPoint).y = 13;
```

Or:

```
ptrPoint->x = 0;
ptrPoint->y = 13;
```



Dynamic Data Structures

Static allocation: `int x[100];`

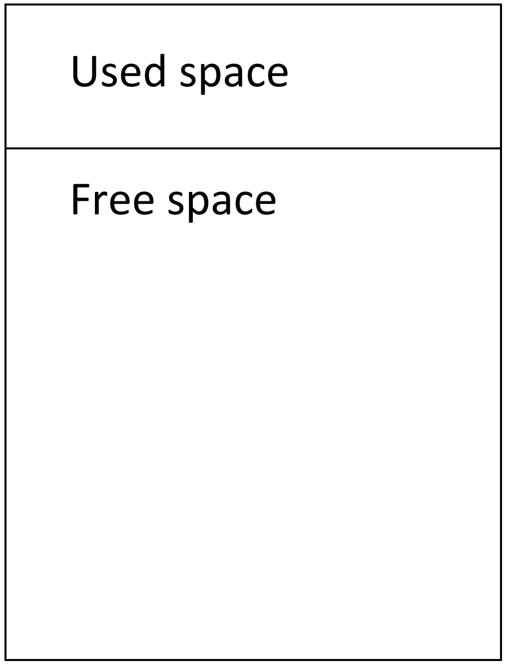
- We know we need 100 int's at compile time

Dynamic allocation:

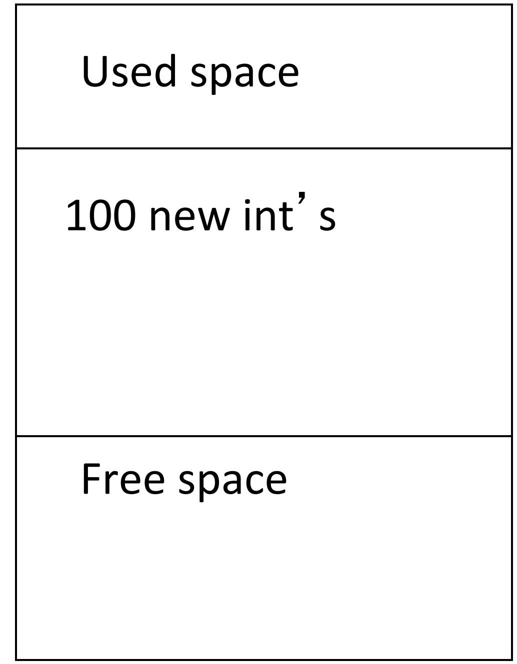
- Size of array or data structures determined at runtime
- C++: **`new`** to allocate, **`delete`** to deallocate
- C: **`malloc()`** to allocate, **`free()`** to deallocate

Example:

```
int *ptr;
ptr = (int *) malloc(100*sizeof(int));
// allocate 100 int's, starting at
address in ptr
```



Memory: before malloc()



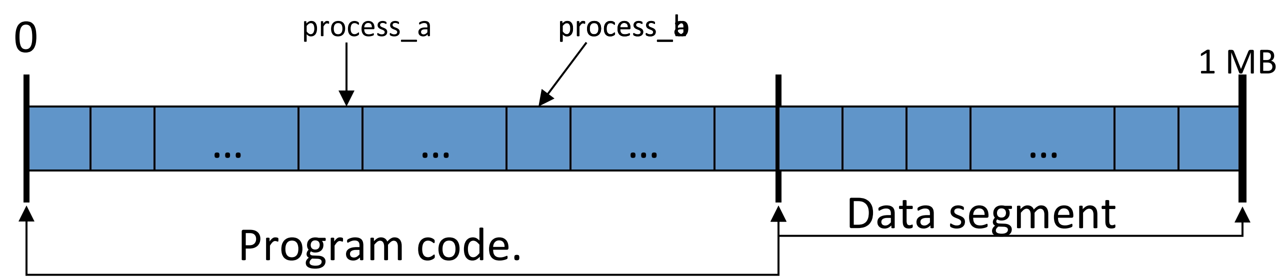
Memory: after malloc()

Pointers to Functions

```
void process_a (int x)
{
    printf (“Process a got %d\n”, x);
}

void process_b (int x)
{
    printf (“Process b got %d\n”, x);
}

void call (void (*func)  (int), int arg)
{
     (*func) (arg);
}
void main()
{
    call (process_a, 10);
    call (process_b, 20);
}
```

Produces output:

```
Process a got 10
Process b got 20
```



Function pointer syntax details

```
void call (void (*func)  (int), int arg)
```

- Call() is a function with no return values
- Call() has 2 arguments:

1) `void (*func)  (int)`

   pointer to a function func(), with return type void and one int argument

2) `int arg`

In code for call(), this line calls process_a or process_b:

```
(*func) (arg);
```

Example: `call (process_a, 10);`

Call() is called, with 2 arguments:

1) Address of process_a

2) 10

The line

```
(*func) (arg);
```

… calls the function whose address was passed (i.e., process_a), with argument 10.

Hence, same effect as

```
process_a(10);
```