

Week 14 – Monday

COMP 3400

Last time

- What did we talk about last time?
- Reliable storage and location
 - GFS
 - Distributed hash tables
- Consensus in distributed systems
 - Byzantine generals problem
- Blockchain

Questions?

Assignment 8

Review

Final exam format

- **Final exam will be in this room:**
 - **Wednesday, April 30, 2025**
 - **8:00 – 10:00 a.m.**
 - **50% longer than previous exams, but you have 100% more time**
- Mostly short answer questions
- One or two matching questions
- A couple of debugging questions
- A couple of programming questions

Linux/UNIX commands you should know

- `cat`
- `cd`
- `chmod`
- `cp`
- `grep`
- `kill`
- `less`
- `ls`
- `make`
- `man`
- `mkdir`
- `mv`
- `ps`
- `pwd`

Fixed-Width Types

Fixed width types

- Although it's a bit ugly, C99 specifies types with fixed sizes
- To use them, **#include <stdint.h>**
- Then, you're guaranteed the following:
 - **int8_t** 1 byte (8 bits), signed
 - **int16_t** 2 bytes (16 bits), signed
 - **int32_t** 4 bytes (32 bits), signed
 - **uint8_t** 1 byte (8 bits), unsigned
 - **uint16_t** 2 bytes (16 bits), unsigned
 - **uint32_t** 4 bytes (32 bits), unsigned
- And you probably get **int64_t** and **uint64_t** as well

What about printing those things?

- If you want to print an `int`, you use `%d`
- If you want to print an `int32_t`, what do you do?
- There are some (ugly) macros used:
 - `PRId8`
 - `PRId16`
 - `PRId32`
 - `PRId64`
- You can use these macros for octal or hex by changing `d` to `o` or `x`, e.g. `PRIx32`

Using the print macros

- To use these macros, `#include <inttypes.h>`
 - Note that `inttypes.h` includes `stdint.h`, so you can kill two birds with one stone
- These macros are special strings
- There's an obscure rule in C that treats consecutive strings literals like a single string literal:
 - `"goats" "boats" "moats"` is the same to the compiler as `"goatsboatsmoats"`
- To use a macro, it has to "float" in between the rest of a formatting string

```
int a = 7;
int32_t b = 7;
printf ("Value: %d\n", a); // int version
printf ("Value: %" PRId32 "\n", b); // int32_t version
```

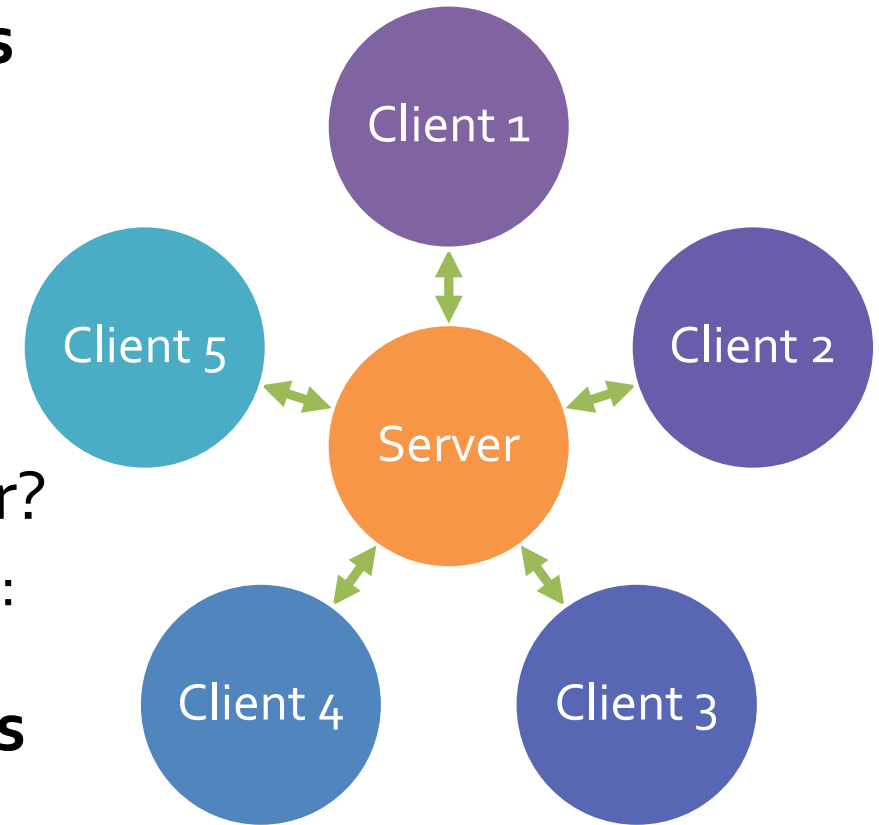
System Architectures

System architectures

- System architectures are models of systems that describe:
 - Relationships between entities in the system
 - Ways the entities communicate
- Different architectural styles have pros and cons
- Using a certain style can have big impacts on system performance
- Common styles:
 - Client/server
 - Peer-to-peer (P2P)
 - Layered
 - Pipe-and-filter
 - Event-driven
 - Hybrid

Client/server architectures

- This book considers **client/server architectures** from the perspective of a many clients connecting to a single server
 - If you recall, the Software Engineering book describes client/server as a system with many servers, each of which offer a single service
- How does a client know how to reach the server?
 - Uniform resource identifier (URI) is a common way: www.goats.net/image.jpg
- Client/server architectures depend on **protocols** to define how clients can request services and understand the response



Client/server advantages and disadvantages

ADVANTAGES

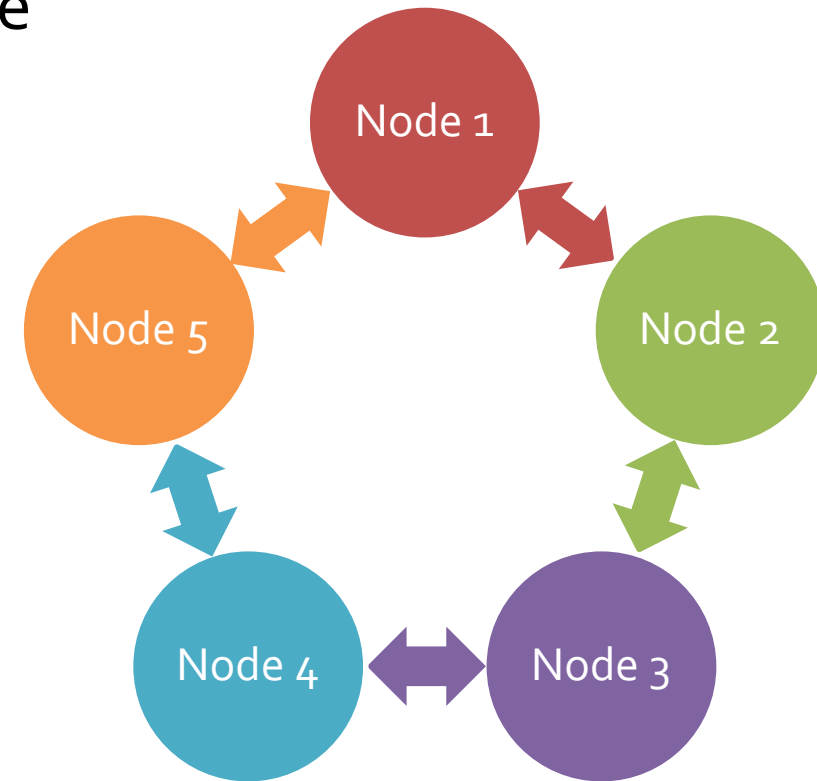
- Updates are simple, because only the server needs to be updated
- Only the server needs to be checked for security problems or data corruption
- To reduce the single point of failure problem, it's common to have multiple servers that offer the same services or files
- To work, these servers must coordinate with each other when one is updated

DISADVANTAGES

- Single point of failure

Peer-to-peer (P2P) architectures

- If more and more servers are used, the architecture begins to look like a **P2P architecture**
 - BitTorrent
 - DNS
- In P2P, there is usually no distinction between clients and servers, since most entities act as both
- Advantages:
 - Service scales, staying the same or improving as the number of users goes up
- Disadvantages:
 - Security: A corrupted node can be hard to detect
 - Administration: Propagating changes can be difficult



Layered architectures

- **Layered architectures** divide systems into a strict hierarchy of components
- Each layer can only communicate with the layer above and below it
- Advantages:
 - As long as a new layer knows how to talk to the layer above and below, it can be swapped out with an old layer
 - New layers can be added on top
- Disadvantages:
 - It's hard to divide systems into hierarchical layers
 - It can be inefficient to prevent one layer from talking directly to one much lower or higher
 - Some services at each layer are redundant

Presentation Layer

Business Layer

Services Layer

Persistence Layer

Pipe-and-filter architectures

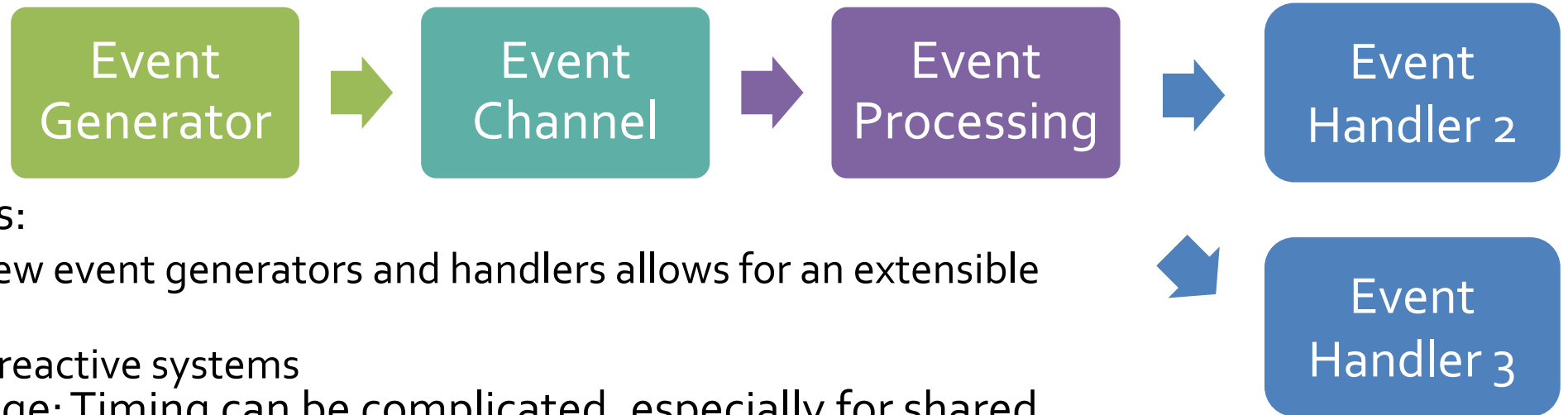
- **Pipe-and-filter architectures** send data in one direction through a series of components
- The output of one stage is the input of the next
- Each stage transforms the data in some way
- Examples:
 - Linux command-line piping

```
sort foo.txt | grep -i error | head -n 10 > out.txt
```

- Java stream filtering
 - Stages of a compiler
- Advantages:
 - Good for serial data processing
 - Modular components that have the same input and output can be reused in different sequences
- Disadvantage: No error recovery if something breaks in the middle

Event-driven architectures

- **Event-driven architectures** react to events, changes in the state of the system
 - GUIs are a common example of event-driven architectures
- Event generator create events
- Event channels send the event to the appropriate event handlers



- Advantages:
 - Adding new event generators and handlers allows for an extensible system
 - Good for reactive systems
- Disadvantage: Timing can be complicated, especially for shared resources

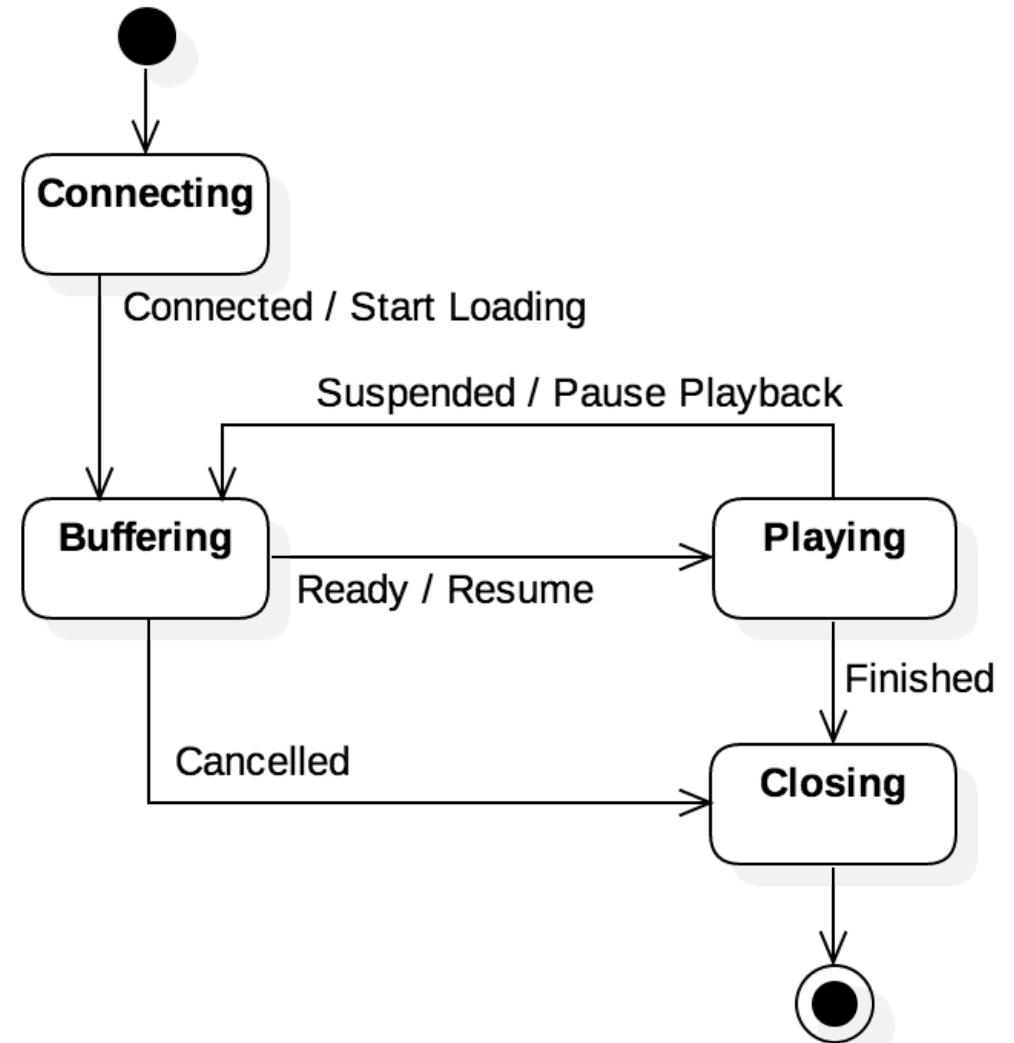
Hybrid architectures

- We talk about the previous architectures because they're models that have been successful in the past
- Most real systems are a mix of different architectures
 - The whole system could be one architecture, but its components have their own
 - A system is mostly one architecture, but it breaks a couple of rules
 - There can be different ways of looking at the same system
- Example: OS kernel
 - Event-driven because it has interrupt handlers to respond to signals from the hardware
 - Client/server because applications that make system calls are making requests
 - Layered because file systems and networking operate with layers from the generic operation down to the requirements of particular hardware

State Machines

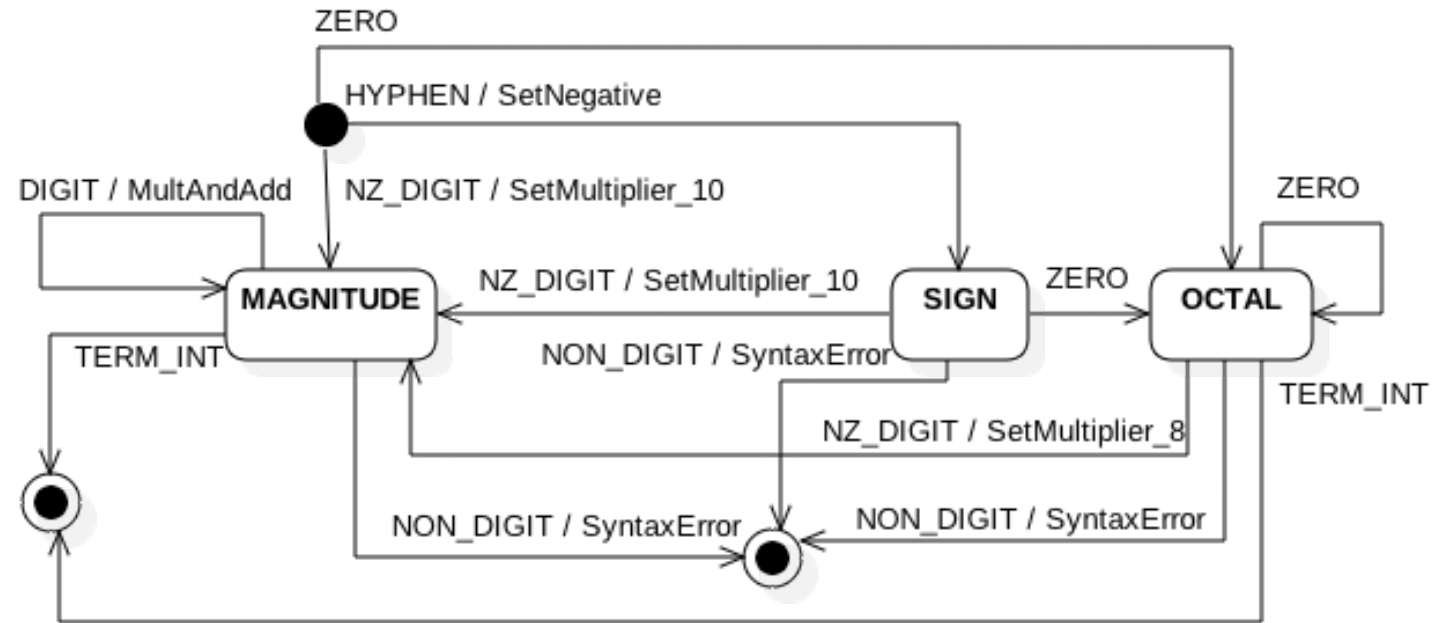
UML state models

- As discussed in COMP 3100, UML standardizes **state models** as a way to visualize states and transitions
 - States are shown as rounded rectangles
 - A solid circle shows the initial state
 - A solid circle in a circle shows the final state
 - Transitions are shown as labeled arrows
 - Effects (if any) are written after a slash after the transition label



State machines as recognizers

- State machines are often used to recognize strings as being legal or illegal
- Consider a state machine from Project 1 designed to recognize integer values (formatted in either decimal or octal)
- In addition to recognizing integers as legal or illegal, the machine builds the integer based on the effects

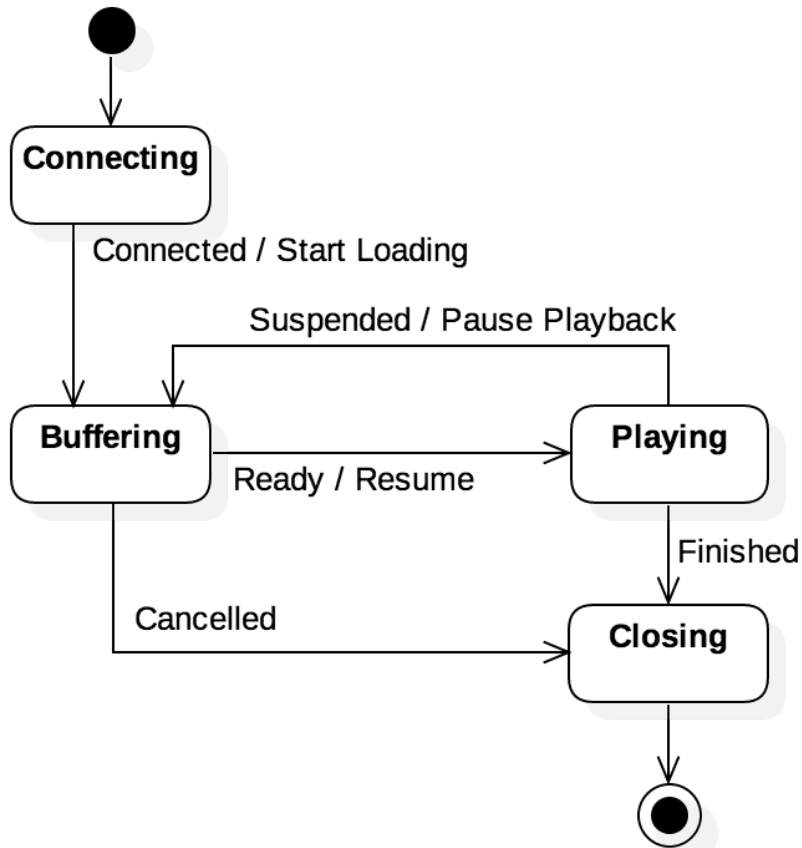


Implementing state machines

- There are algorithms to convert between regular expressions and state machines
- Most regular expression libraries build a state machine as a way to see if strings match the regular expression
- One way to implement state machines is with a 2D array
 - One row for every state
 - One column for every event, saying which state a given state will transition to
- If there are effects, a second 2D array can show which effects happen on those transitions
- If an action happens whenever a state is entered, a 1D array can hold that information

Example transition table

- The state model on the left has a transition table on the right



| States | Events | | | | |
|------------|-----------|-----------|---------|---------|---------|
| | Connect | Suspend | Ready | Finish | Cancel |
| Connecting | Buffering | | | | |
| Buffering | | | Playing | | Closing |
| Playing | | Buffering | | Closing | |
| Closing | | | | | |

Example table in code

- Two **enums** are used to list the states and the events
- A 2D array stores the transitions

```
typedef enum { CONN, BUFF, PLAY, CLOS, NST } ms_t;
typedef enum { Connect, Suspend, Ready, Finish, Cancel } event_t;
#define NUM_STATES (NST+1)
#define NUM_EVENTS (Cancel+1)
static ms_t const _transition[NUM_STATES][NUM_EVENTS] =
{
    // Connect  Suspend  Ready    Finish  Cancel
    { BUFF,    NST,     NST,     NST,    NST }, // Connecting
    { NST,     NST,     PLAY,    NST,    CLOS }, // Buffering
    { NST,     BUFF,    NST,     CLOS,   NST }, // Playing
    { NST,     NST,     NST,     NST,    NST }  // Closing
};
```

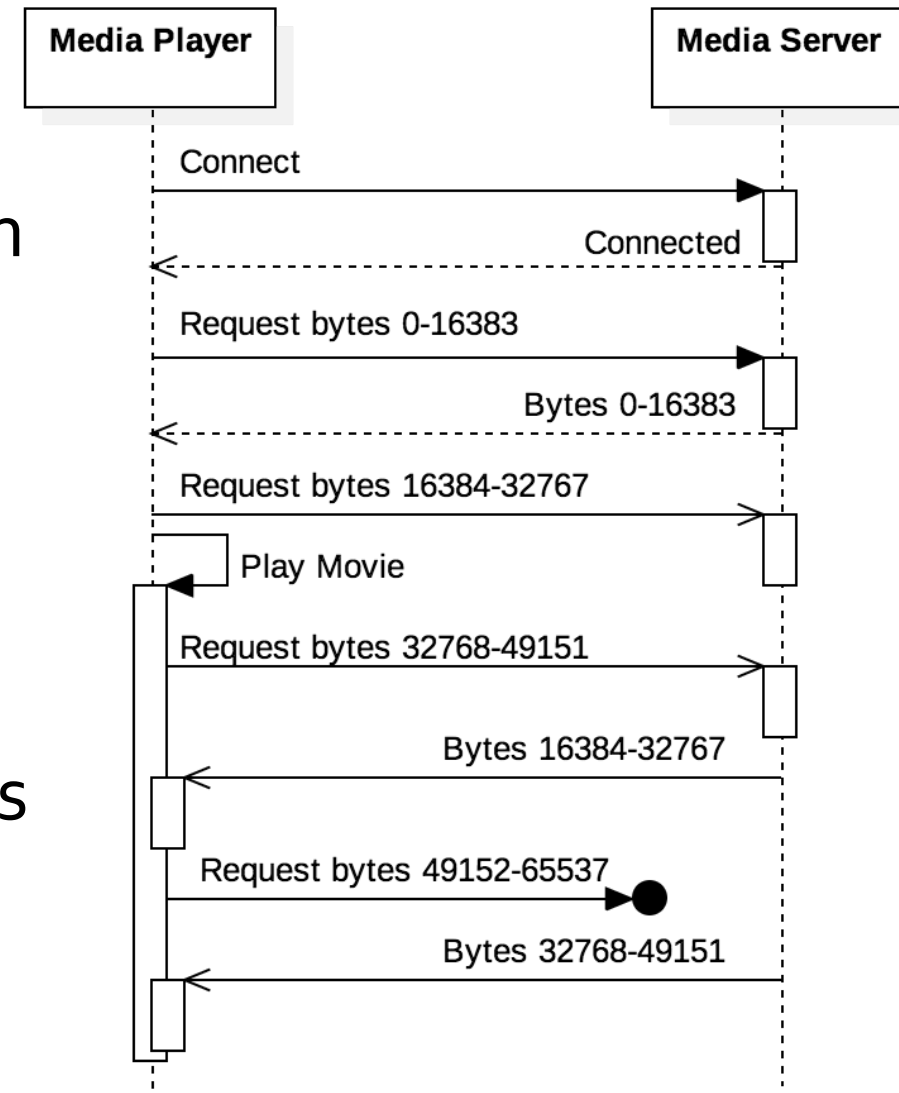
Effects

- A table filled with function pointers can be used for effects

```
static action_t const _effect[NUM_STATES][NUM_EVENTS] = {
    // Connect      Suspend      Ready      Finish      Cancel
    { start_load,  NULL,          NULL,       NULL,       NULL }, // Connecting
    { NULL,        NULL,          resume,     NULL,       NULL }, // Buffering
    { NULL,        pause_play,   NULL,       NULL,       NULL }, // Playing
    { NULL,        NULL,         NULL,       NULL,       NULL } // Closing
};
```

Sequence models

- State models don't have any timing or sequence information
- **Sequence models** show the order in which messages are sent from one entity to another
 - Solid arrows show synchronous messages
 - Open arrows show asynchronous messages
 - Dotted lines show responses
 - Messages that end in circles are lost
- The order of messages in sequence models is logical, not scaled by time



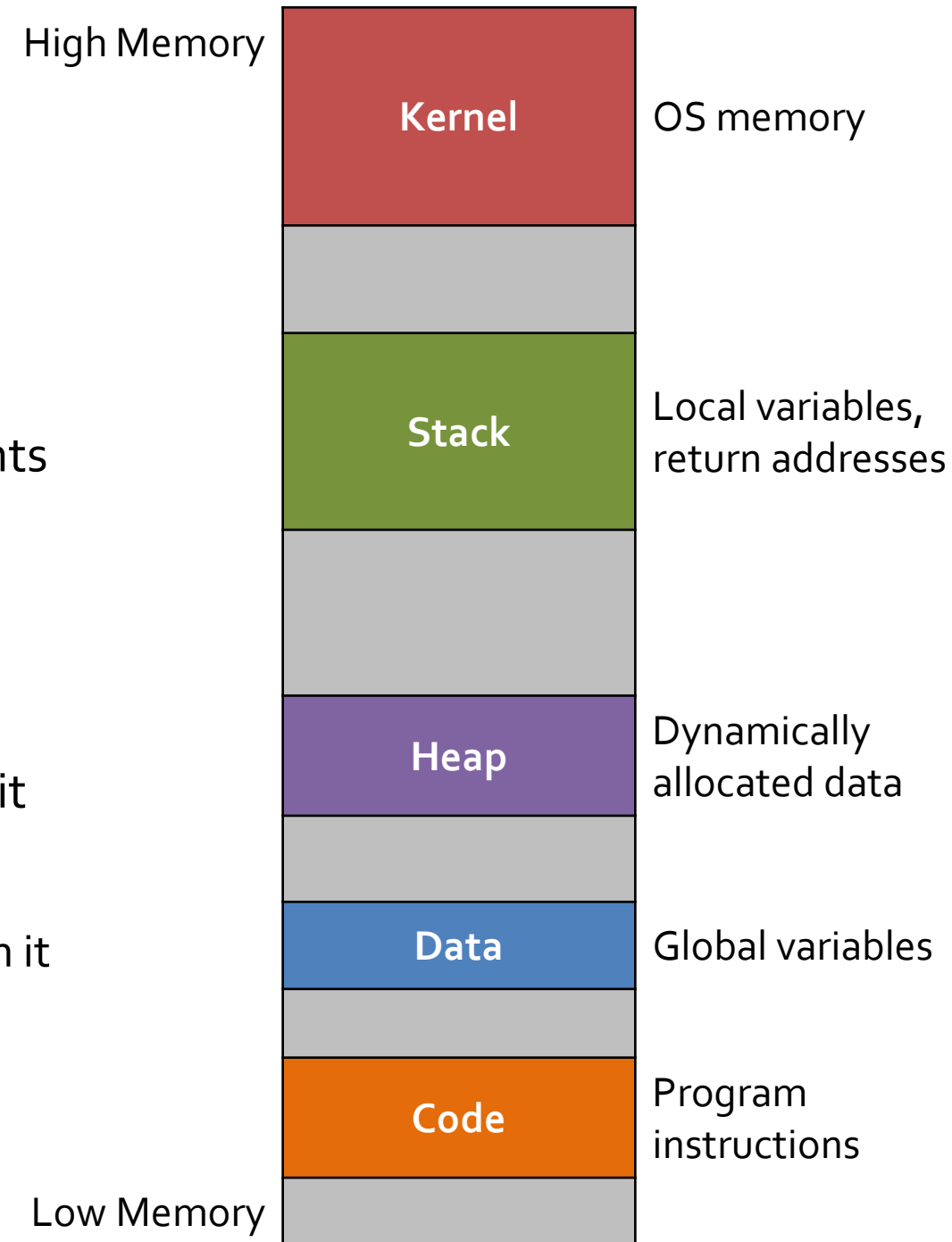
Processes

Processes

- A **program** is an implementation of an algorithm in a programming language
 - A list of instructions for the computer
- A **process** is program being executed
 - Usually, processes are different programs
 - But it's not unusual to have several processes running at the same time that are the same program
- Running a program creates a new process

Virtual memory

- Every process has its own virtual memory
 - Addresses from 0 up to 2^{32} or 2^{64} bytes
- Each instance of virtual memory is organized into segments
 - Code
 - Data
 - Heap
 - Stack
 - Kernel
- Each segment has certain kinds of operations allowed on it
- Do illegal operations, and you get a segmentation fault
- As functions get called, the stack grows downward
 - Call too many functions, and you'll get a stack overflow when it gets too big
- Depending on the system, the heap can grow too
 - `malloc()` returns NULL when you run out of heap space



Why is it *virtual* memory?

- Addresses in one process have nothing to do with addresses in another
- The OS maps the virtual addresses to physical addresses
 - Transparently!
 - Each process has no idea what the location of, for example, its virtual address **0x0432A8F8** is in physical memory
- Benefits:
 - **Security:** One process cannot (normally) interfere with the memory inside another process
 - **Bookkeeping:** The OS only gives each process what it needs and can temporarily store parts of a process's memory on disk to make more space

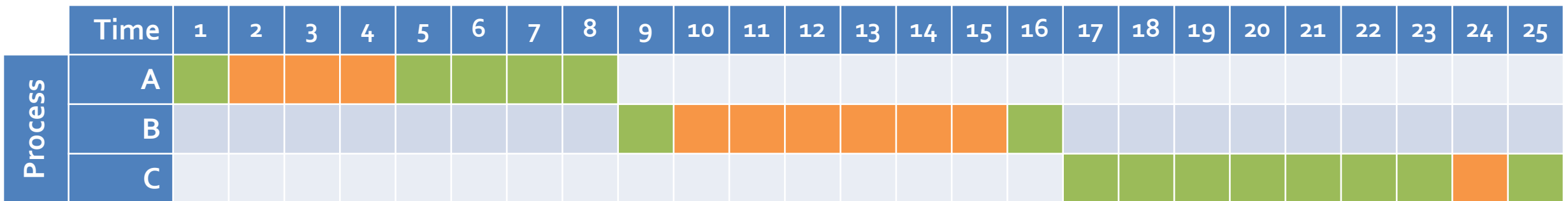
Operating systems

- OS sometimes means the entire operating system, including utilities, window managers, and lots of other stuff
- Sometimes OS means just the kernel
- The kernel is the part of the OS that does deep stuff:
 - Scheduling processes
 - Accessing devices
 - Managing memory
- Some operations can only be done in **kernel mode**, the mode that the kernel runs in
- Normal programs run in **user mode**

Multiprogramming

Problems with naïve batch processing

- One approach to batch processing is running Process A until it's done, then Process B, then Process C
- The problem is that programs do I/O
 - I/O is slow
 - The CPU isn't in use while waiting for I/O
- Consider the following example:
 - Green is computation
 - Orange is I/O
- Nothing is getting done during I/O!



Multiprogramming

- With true multiprogramming, you have more than one process loaded into memory
- Then, when one process is waiting on I/O, we can start running another
- Using multiprogramming, we could run Processes A, B, and C as follows:

| | Time | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---------|------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
| Process | A | █ | █ | █ | █ | █ | █ | █ | █ | | | | | | | | |
| | B | | █ | █ | █ | █ | █ | █ | █ | █ | | | | | | | |
| | C | | | █ | █ | | | | | | █ | █ | █ | █ | █ | █ | █ |

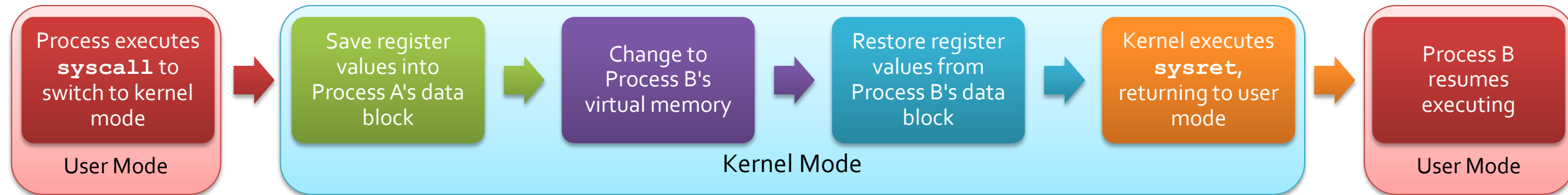
- Doing so gives us a CPU utilization of $15/16 = 93.75\%$ and only 16 time units to finish the work

Types of multiprogramming

- **Preemptive multitasking:**
 - Processes get a maximum amount of time to run called a **quantum**
 - If the process starts doing I/O, the OS switches to another process
 - Otherwise, the OS switches when the process runs out of time
 - There's research about the ideal length of a quantum
- **Cooperative multitasking:**
 - Processes run until they do some I/O or voluntarily give up control
- Cooperative is good because it's simple and can have lower overhead
- Unfortunately, the problem of processes that don't give up control means that most modern systems use preemptive multitasking

Context switches

- A context switch happens when the running process changes
 - The virtual memory of one process changes to another
 - The kernel memory stays the same
- The **scheduler** in the OS decided which process runs next



- Because memory has to get saved and restored, cache is invalidated, and there's a switch from user mode to kernel mode and back, context switches have **overhead** that slows things down

Kernel

Kernel

- The kernel runs with full access privileges to everything
- The kernel controls:
 - Physical memory
 - File system
 - I/O devices
- It handles power disruption and people attaching USB devices
- Jobs of the kernel
 - Resource manager: Giving access to hardware when needed
 - Control program: Handling errors and access violations
- Because it has to work consistently, the kernel doesn't change much over the years

x86 operating mode

- The **current privilege level (CPL)** is a 2-bit value set in x86 CPUs
 - Also called a **ring**
 - Ring 3 is user mode
 - Ring 0 is kernel mode
 - The other two rings aren't used
- When in kernel mode:
 - All memory addresses can be accessed
 - Some special CPU instructions like halting the CPU or invalidating the cache can be executed
 - Some normal CPU instructions work differently

Kernel invocation

- The kernel can be invoked in two different ways
- System call:
 - A user mode program wants to do something (like open a file) that requires OS involvement
 - Somewhere in the library, a special trap instruction will ask the kernel to do something
- Interrupt or exception:
 - Interrupts are hardware events that cause the kernel to react, like clicking a mouse
 - Exceptions are software events that notify the kernel of a problem, like a segmentation fault
 - This kind of exception isn't the same as an exception in Java, although the Java exception can be triggered by an OS exception

System Calls

System calls

- User-mode processes can do normal CPU operations
 - Add, subtract, multiply, divide
 - Test for equality
- They can't do anything outside the CPU on their own
 - Read or write hard drive data
 - Send messages over the network
- To do these things, processes make **system calls**, asking the kernel to do the operation

How system calls work

- In assembly, a special trap instruction triggers a mode switch so that the kernel will start doing stuff
 - The x86 trap instruction is **syscall**
- The kernel checks to make sure that the process has all the necessary privileges to do the operation first
- After the system call, the kernel runs the **sysret** instruction, returning to user mode
- Many system calls are referred to by the C functions that are called to run them, even though those functions just do set up before running the real system call
 - For example: **write()**

Common system calls

- The 64-bit Linux kernel has more than 300 system calls
- These are just a few common ones:

| System Call | Number | Purpose |
|---------------------------|--------|--|
| <code>read</code> | 0 | Read from a file descriptor |
| <code>write</code> | 1 | Write to a file descriptor |
| <code>nanosleep</code> | 35 | High-resolution sleep (units in seconds and nanoseconds) |
| <code>exit</code> | 60 | Terminate the current process |
| <code>kill</code> | 62 | Send a signal to a process |
| <code>uname</code> | 63 | Get information about the current kernel |
| <code>gettimeofday</code> | 96 | Get the system time in seconds since midnight, January 1, 1970 |
| <code>sysinfo</code> | 99 | Get information about memory usage and CPU load average |
| <code>ptrace</code> | 101 | Trace another process's execution |

Process Life Cycle

Creating processes in code

- Processes are, of course, created when you run a program from the command line
- However, you can also create processes from within a program, using calls to special functions
- The **fork ()** function creates a new processes that's exactly the same as the current process
- The **exec ()** function allows you to replace the current process with another program
- Each process has a unique ID, its process ID or PID
 - **getpid ()** returns the PID of the current process
 - **getppid ()** returns the PID of the current process's parent process

Using fork()

- The **fork ()** function is pretty crazy!
 - When you call it, the process you're inside of keeps running
 - And another process spawns at exactly the same point in code
 - Both processes have *exactly* the same memory layout
 - The only difference is that **fork ()** returns the child PID for the original process and 0 if you're the process that just got forked

```
pid_t child_pid = fork ();

if (child_pid < 0)
    printf ("ERROR: No child process created\n");
else if (child_pid == 0)
    printf ("Hi, I'm the child!\n");
else
    printf ("Parent just gave birth to child %d\n", child_pid);
```

Fork bombing

- If you call `fork ()` in a loop, you will quickly create too many processes and slow/crash your computer
- Each `fork ()` creates a new process, but the old process keeps running
- The following code will have four prints:

```
pid_t first_fork = fork ();  
  
// Original parent and child create new processes  
pid_t second_fork = fork ();  
  
// This line prints four times  
printf ("Hello from %d!\n", getpid ());
```

Running another program

- Sometimes it's useful to fork a clone of yourself
- Other times, you want to run another program
- In those situations, you first fork yourself and then have your child call something from the **exec ()** family of functions:

| Function | Description |
|--|--|
| <code>execl(char *path, char *arg0, ..., NULL)</code> | Executes the program with the given path |
| <code>execle(char *path, char *arg0, ..., NULL, char* envp[])</code> | Executes the program with the given path and environment variables |
| <code>execlp(char *file, char *arg0, ..., NULL)</code> | Executes the program by looking it up in the current PATH |
| <code>execv(char *path, char *argv[])</code> | Like execl () but command-line arguments are in an array |
| <code>execve(char *path, char *argv[], char *envp[])</code> | Like execle () but command-line arguments are in an array |
| <code>execvp(char *file, char *argv[])</code> | Like execlp () but command-line arguments are in an array |
| <code>fexecve(int fd, char *argv[], char *envp[])</code> | Executes the program stored in the file descriptor fd |

Example with `exec()`

- The following programs runs `ls`, listing the contents of the current directory:

```
pid_t child_pid = fork ();
if (child_pid < 0)
    exit (1); // exit if fork() failed

if (child_pid == 0) // child process
{
    int rc = execlp ("ls", "ls", "-l", NULL);
    exit (1); // only reached if exec() failed
}
```

Waiting for a child to finish

- Once you've forked or spawned a process, it will be scheduled to run
- There are no guarantees about when a parent or a child will be scheduled relative to each other
- It can be useful for a parent process to wait until its child processes have terminated
- There are two functions for this:
 - `wait(int *stat_loc)`
 - Waits for all children
 - `waitpid(pid_t pid, int *stat_loc, int options)`
 - Waits only on child process with PID

Example with `wait()`

- Here's the `ls` example from earlier, except that the parent process waits for `ls` to finish
- More code isn't shown, but the parent could continue doing other things

```
pid_t child_pid = fork ();
if (child_pid < 0)
    exit (1); // exit if fork() failed

if (child_pid == 0) // child process
{
    int rc = execlp ("ls", "ls", "-l", NULL);
    exit (1); // only reached if exec() failed
}

wait (NULL); // waits for ls to finish
```

Files

Sharing resources

- Although physical memory is shared between processes, the virtual memory system means that processes don't share memory directly
- Other things must be shared by processes:
 - Network cards
 - Hard drives and SSDs
 - User input and output devices
- A uniform way to work with most shared resources is to *treat them all like files*
- This file abstraction makes many libraries similar and simpler

UNIX file abstraction

- The UNIX file abstraction uses two key ideas:
 - A file is a sequence of bytes
 - Everything is a file
- This abstraction is different from the traditional idea of files in a few ways:
 - Moving backwards and forwards within a file isn't always possible
 - Files don't always have names or live in a particular place
 - Files don't always have a set structure
- Even so, creating, deleting, opening, closing, reading, and writing can be treated the same

Opening files

- To open a file for reading or writing, use the **open ()** function
- The **open ()** function takes the file name, an **int** for mode, and an (optional) **mode_t** for permissions
- The name refers to an entity somewhere in the directory structure that might or might not be a normal file
- It returns a file descriptor as an **int**

```
int fd = open("input.dat", O_RDONLY);
```

Constants

- A number of constants specify whether the opening is for reading or writing
- The optional permissions value has other constants to set the permissions of the file when creating a new one
- Both sets of constants can be bitwise ORed together to make complicated values

| Access | Meaning |
|-------------------------|---|
| <code>O_RDONLY</code> | Open for reading only |
| <code>O_WRONLY</code> | Open for writing only |
| <code>O_RDWR</code> | Open for reading and writing |
| <code>O_NONBLOCK</code> | Do not block on opening while waiting for data |
| <code>O_CREAT</code> | Create the file if it does not exist, requires <code>mode_t</code> argument |
| <code>O_TRUNC</code> | Truncate to size 0 |
| <code>O_EXCL</code> | Error if <code>O_CREAT</code> and the file exists |

| Name | Description |
|----------------------|-----------------|
| <code>S_IRUSR</code> | Read (user) |
| <code>S_IWUSR</code> | Write (user) |
| <code>S_IXUSR</code> | Execute (user) |
| <code>S_IRGRP</code> | Read (group) |
| <code>S_IWGRP</code> | Write (group) |
| <code>S_IXGRP</code> | Execute (group) |
| <code>S_IROTH</code> | Read (other) |
| <code>S_IWOTH</code> | Write (other) |
| <code>S_IXOTH</code> | Execute (other) |

Example with other constants

- The following example shows how to open a file
 - For writing
 - By creating it
 - Truncating its size to 0 if there's already something in the file
 - Making it readable and writable to the user and readable to others

```
int fd = open("output.dat", O_CREAT | O_TRUNC |  
O_WRONLY, S_IRUSR | S_IWUSR | S_IROTH);
```

- It's also common to use numbers in octal for permissions, where the 64's place is permission for the user, the 8's place is permission for the group, and the 1's place is permission for others
 - `S_IRUSR | S_IWUSR | S_IROTH = 110 000 100 = 0604`

Reading from files

- Opening the file is actually the hardest part
- Reading is straightforward with the **read()** function
- Its arguments are
 - The file descriptor
 - A pointer to the memory to read into
 - The number of bytes to read
- Its return value is the number of bytes successfully read

```
int fd = open("input.dat", O_RDONLY);  
int buffer[100];  
// Fill with something  
read( fd, buffer, sizeof(int)*100 );
```

Closing files

- To close a file descriptor, call the `close()` function
- Close files when you're done with them

```
int fd = open("output.dat", O_WRONLY | O_CREAT | O_TRUNC,  
0644);  
// Write some stuff  
close( fd );
```

Special files

- Linux provides some "special" files
 - **/dev/full**
 - A file that says the device is full if you try to write to it, gives unlimited zeroes if you try to read from it
 - **/dev/null**
 - A file you can write to forever but simply discards the data (while saying that the write succeeded)
 - **/dev/random**
 - A file you can read a stream of random bytes from
 - **/dev/zero**
 - A file you can read an unlimited stream of zero bytes from
- They're not actually files, but you can treat them as if they are
- They can be useful for testing and sometimes even for the operation of program

Writing to files

- Writing to a file is almost the same as reading
- Arguments to the **write()** function are
 - The file descriptor
 - A pointer to the memory to write from
 - The number of bytes to write
- Its return value is the number of bytes successfully written

```
int fd = open("output.dat", O_WRONLY | O_CREAT | O_TRUNC, 0644);
int buffer[100];
int i = 0;
for (i = 0; i < 100; ++i)
    buffer[i] = i + 1;
write( fd, buffer, sizeof(int)*100 );
```


Seeking to locations

- It's possible to move the current location within the file using the `lseek()` function
- Its arguments are
 - The file descriptor
 - The offset (positive or negative)
 - Location to seek from:
 - `SEEK_SET` (beginning of file)
 - `SEEK_CUR` (current location)
 - `SEEK_END` (end of file)
- Seeking is more common when reading, but you can seek while writing too

```
int fd = open("input.dat", O_RDONLY);  
lseek( fd, 100, SEEK_SET );
```

Example getting file metadata

- The following code finds out how big a file (stored with file descriptor **fd**) is in bytes:

```
struct stat metadata;  
fstat (fd, &metadata);  
printf ("File size: %lld bytes\n",  
        (long long)metadata.st_size);
```

Interpreting metadata

- The following shows some fields in **struct stat**
- The **st_mode** field is a bitwise OR of permissions and other information from the table on the right

```
struct stat {  
    dev_t      st_dev;      // device of inode  
    ino_t      st_ino;     // inode number  
    mode_t     st_mode;    // protection mode  
    nlink_t    st_nlink;   // hard links to file  
    uid_t      st_uid;     // user ID of owner  
    gid_t      st_gid;     // group ID of owner  
    dev_t      st_rdev;    // device type  
    off_t      st_size;    // file size in bytes  
    // Other fields depending on OS ...  
};
```

| Name | Description |
|----------|-----------------------------|
| S_IFIFO | Named pipe (IPC) |
| S_IFCHR | Character device (terminal) |
| S_IFDIR | Directory file type |
| S_IFBLK | Block device (disk drive) |
| S_IFREG | Regular file type |
| S_IFLNK | Symbolic link |
| S_IFSOCK | Socket (IPC, networks) |

Events and Signals

Command line signals

- You can send signals to processes from the command line
 - **Ctrl-C: SIGINT** (interrupt)
 - **Ctrl-Z: SIGTSTP** (terminal stop, usually suspends)
- Signals often result in the process being killed
- Perhaps for that reason, the **kill** command is used to send arbitrary signals (not just killing ones)
 - Flag gives the kind of signal
 - Then specify the PID of the process

```
> kill -KILL 8382
```

Common signals

- When using the kill command, the flag can either be the name of the signal (**-KILL**) or its number (**-9**)
- Here are some common signals:

| Name | Number | Description |
|----------------|--------|---|
| SIGINT | 2 | Interrupts the process, generally killing it. Sent with Ctrl-C . |
| SIGKILL | 9 | Kills the process. Cannot be ignored or overwritten. |
| SIGSEGV | 11 | Sent to a process when it has a segmentation fault. |
| SIGCHLD | 18 | Sent to a parent when a child process finishes. Used by wait() . |
| SIGSTOP | 23 | Suspends the process. Cannot be ignored or overwritten. |
| SIGTSTP | 24 | Suspends the process. Sent with Ctrl-Z . |
| SIGCONT | 25 | Resumes a suspended process. |

Sending signals in a program

- Just as you can use the `kill` command from the command line, you can also call the `kill()` function to send a signal to another process
- The function takes two parameters:
 - PID of the process to kill
 - `int` value giving the signal, usually a named constant

```
kill (pid, SIGSTOP); // Suspends process with pid
```

- You can usually only kill processes that you own
 - Unless you're a superuser (like root)

Example of `kill ()` function

- Below, a parent forks a child
- The child goes into an infinite loop
- Then, the parent kills the child

```
pid_t child_pid = fork ();  
if (child_pid < 0)  
    exit (1); // exit if fork failed  
  
if (child_pid == 0)  
    while (1) ; // child loops  
  
sleep (1); // parent sleeps for 1 second  
kill (child_pid, SIGKILL); // parent kills the child
```


Custom signal handlers

- Although signals have default actions for processes, *some* signals can be overridden
- A process can define what happens when, for example, it's interrupted
- First, you need a function that will get called when a particular signal happens
 - It must take an **int** (the signal) and return **void**
- Example that prints "I don't want to die!" and then exits

```
static void
handler(int signal)
{
    write(STDOUT_FILENO, "I don't want to die!\n", 21);
    exit(0);
}
```

Overriding the signal handler

- Once you've written the custom signal handler, you have to override it with the **sigaction()** function:

```
int sigaction(int signal, const struct sigaction *action,  
struct sigaction *old);
```

- The **action** parameter is a **struct sigaction** with a function pointer to the new handler
- The old parameter is **NULL** unless you want to find out what the old signal handler was

Overriding example

- The following code overrides the **SIGINT** signal with the handler from a couple of slides back
- Then it goes into an infinite loop until someone interrupts it (like with **Ctrl-C**)

```
int
main (int argc, char *argv[])
{
    struct sigaction sa; // Struct we'll add the handler to
    memset(&sa, 0, sizeof(sa)); // Zero out the contents first
    sa.sa_handler = handler;

    // Override SIGINT handler
    if (sigaction (SIGINT, &sa, NULL) == -1)
        printf ("Failed to overwrite SIGINT.\n");

    printf ("Entering loop\n");
    while (1); // Loop until signal
    return 0;
}
```

Reborn like a phoenix

- It's sort of cool that we can make a handler print something special before crashing the program
- But we can also do some code to handle the signal and then jump back to a safe location
 - Away from blocked I/O or an infinite loop
 - Somewhere that's been marked and is still on the stack
- To do that, we need two functions

```
// Set jump location
int sigsetjmp(sigjmp_buf context, int mask);

// Jump to location
int siglongjmp(sigjmp_buf context, int value);
```

Full example

```
sigjmp_buf context;


static void handler(int signal)
{
    write(STDOUT_FILENO, "I don't want to die!\n", 21);
    siglongjmp (context, 1); // Jumps to marked location with value 1 (insane!)
}

int main (int argc, char *argv[])
{
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa)); sa.sa_handler = handler;

    if (sigaction (SIGINT, &sa, NULL) == -1)
        printf ("Failed to overwrite SIGINT.\n");

    if (sigsetjmp (context, 0)) // Marks location and returns 0 the first time
        printf ("Resuming execution\n");

    printf ("Entering loop\n");
    while (1); // Loop until signal
    return 0;
}
```



Pointers

Pointers

- A **pointer** is a variable that holds an address
- Often this address is to another variable
- Sometimes it's to a piece of memory that is mapped to file I/O or something else
- Important operations:
 - Reference (&) gets the address of something
 - Dereference (*) gets the contents of a pointer

Declaration of a pointer

- We typically want a pointer that points to a certain kind of thing
- To declare a pointer to a particular type

```
type * name;
```

- Example of a pointer with type `int`:

```
int * pointer;
```


Reference operator

- A fundamental operation is to find the address of a variable
- This is done with the reference operator (&)

```
int value = 5;  
int *pointer;  
pointer = &value; // pointer has value's address
```

- We usually can't predict what the address of something will be

Dereference operator

- The reference operator doesn't let you do much
- You can get an address, but so what?
- Using the dereference operator, you can read and write the contents of the address

```
int value = 5;
int* pointer;
pointer = &value;
printf("%d", *pointer); // prints 5
*pointer = 900; // value just changed!
```

Aliasing

- Java doesn't have pointers
 - But it does have references
 - Which are basically pointers that you can't do arithmetic on
- Like Java, pointers allow us to do aliasing
 - Multiple names for the same thing

```
int wombat = 10;
int* pointer1;
int* pointer2;
pointer1 = &wombat;
pointer2 = pointer1;
*pointer1 = 7;
printf("%d %d %d", wombat, *pointer1, *pointer2);
```

Pointer arithmetic

- One of the most powerful (and most dangerous) qualities of pointers in C is that you can take arbitrary offsets in memory
- When you add to (or subtract from) a pointer, it jumps the number of bytes in memory of the size of the type it points to

```
int a = 10;  
int b = 20;  
int c = 30;  
int* value = &b;  
value++;  
printf("%d", *value); // What does it print?
```

Arrays are pointers too

- An array **is** a pointer
 - It is pre-allocated a fixed amount of memory to point to
 - You can't make it point at something else
- For this reason, you can assign an array directly to a pointer

```
int numbers[] = {3, 5, 7, 11, 13};  
int* value;  
  
value = numbers;  
value = &numbers[0]; // Exactly equivalent  
  
value = &numbers; // What about this?
```

Surprisingly, pointers are arrays too

- Well, no, they aren't
- But you can still use array subscript notation ([]) to read and write the contents of offsets from an initial pointer

```
int numbers[] = {3, 5, 7, 11, 13};  
int* value = numbers;  
  
printf("%d", value[3] ); // prints 11  
printf("%d", *(value + 3) ); // prints 11  
value[4] = 19; // changes 13 to 19
```

void pointers

- What if you don't know what you're going to point at?
- You can use a **void***, which is an address to...something!
- You have to cast it to another kind of pointer to use it
- You can't do pointer arithmetic on it
- It's not useful very often

```
char s[] = "Hello World!";  
void* address = s;  
int* thingy = (int*)address; // Uh-oh  
printf("%d\n", *thingy);
```

Functions that can change arguments

- In general, data is passed **by value**
- This means that a variable cannot be changed for the function that calls it
- Usually, that's good, since we don't have to worry about functions screwing up our data
- It's annoying if we need a function to return more than one thing, though
- Passing a pointer is equivalent to passing the original data **by reference**

Example

- Let's imagine a function that can change the values of its arguments

```
void swapIfOutOfOrder (int *a, int *b)
{
    if (*a > *b)
    {
        int temp = *a;
        *a = *b;
        *b = temp;
    }
}
```

How do you call such a function?

- You have to pass the addresses (pointers) of the variables directly

```
int x = 5;  
int y = 3;  
swapIfOutOfOrder (&x, &y); // Will swap x and y
```

- With normal parameters, you can pass a variable or a literal
- However, you **cannot** pass a reference to a literal

```
swapIfOutOfOrder (&5, &3); // Impossible
```

malloc()

- Memory can be allocated dynamically using a function called **malloc()**
 - Similar to using **new** in Java or C++
 - **#include <stdlib.h>** to use **malloc()**
- Dynamically allocated memory is on the heap
 - It doesn't disappear when a function returns
- To allocate memory, call **malloc()** with the number of bytes you want
- It returns a pointer to that memory, which you cast to the appropriate type

```
int* data = (int*)malloc(sizeof(int));
```

Allocating arrays

- It's common to allocate an array of values dynamically
- The syntax is exactly the same, but you multiply the size of the type by the number of elements you want

```
int i = 0;
int *array = (int*)malloc (sizeof(int)*100);
for (i = 0; i < 100; ++i) // Initialize for fun
    array[i] = i + 1;
```

Pointers to structs

- We can define a pointer to a struct variable
 - We can point it at an existing struct
 - We can dynamically allocate a struct to point it at

```
struct student bob;  
struct student *studentPointer;  
strcpy(bob.name, "Bob Blobberwob");  
bob.GPA = 3.7;  
bob.ID = 100008;  
studentPointer = &bob;  
(*studentPointer).GPA = 2.8;  
studentPointer = (struct student*)malloc(sizeof(struct  
student));
```

Arrow notation

- As we saw on the previous slide, we have to dereference a struct pointer and then use the dot to access a member

```
struct student* studentPointer = (struct student*)  
    malloc(sizeof(struct student));  
(*studentPointer).ID = 3030;
```

- This is cumbersome and requires parentheses
- Because this is a frequent operation, dereference + dot can be written as an arrow (->)

```
studentPointer->ID = 3030;
```

Passing structs to functions

- If you pass a struct directly to a function, you are passing it by value
 - A copy of its contents is made
- It is common to pass a struct by pointer to avoid copying and so that its members can be changed

```
void flip (struct point *value)
{
    double temp = value->x;
    value->x = value->y;
    value->y = temp;
}
```

calloc()

- One problem with `malloc()` is that the memory it allocates is filled with garbage
- Like `malloc()`, `calloc()` allocates memory, but it also zeroes all of it out
- Many programmers think it's safer to use `calloc()` in *all* situations where you would use `malloc()`
- There's a slight syntax difference:
 - `calloc()` takes two arguments: number of elements and size of each one

```
// malloc() version
int *array1 = (int*)malloc (sizeof(int)*100);
// equivalent calloc() version
int *array2 = (int*)calloc (100, sizeof(int));
```


realloc()

- For a dynamic array, it can be useful to grow an existing chunk of memory if it's too small
- You could allocate an entirely new, bigger chunk of memory, copy everything from the old memory over, and then free the old memory
 - This is what you *have* to do in Java
- C provides a slick function, **realloc()**, that does all of that for you
 - Arguments: memory to resize, new size
 - Return value: resized memory

```
if(size == capacity)
{
    capacity *= 2;
    array = realloc(array, capacity*sizeof(int));
}
array[size] = element;
++size;
```

free ()

- C isn't garbage collected like Java
- If you allocate something on the stack, it disappears when the function returns
- If you allocate something on the heap, you have to deallocate it with **free ()**
- **free ()** does *not* set the pointer to be **NULL**
 - But you can (and should) afterwards

```
char *things = (char*)malloc (100*sizeof(char));  
// Do stuff with things  
free(things);  
things = NULL;
```

Pointer practice

- Given that `i` has type `int` and `p` and `q` have type `int*`, which of the following will cause a compiler error?

a) `p = &i;`

b) `p = *&i;`

c) `p = &*i;`

d) `i = *&*p;`

e) `i = *&p;`

f) `i = &*p;`

g) `p = &*&i;`

h) `q = *&*p;`

i) `i = **&p;`

j) `q = *&p;`

k) `q = &*p;`

Interprocess Communication

Message passing

- There are many IPC approaches, but they can all be categorized as either **message passing** or **shared memory**
- Message passing:
 - Sender prepares a message
 - Sender makes a system call to request a data transfer
 - Kernel copies the message into a buffer
 - Receiver makes a system call to retrieve the data
 - Receiver copies the message into its own memory

Shared memory

- Shared memory IPC is completely different
- The processes decide on a chunk of virtual memory that will be used for IPC
- The processes make system calls to request that this memory is shared
- Once it's shared, processes can read and write from shared memory just like any other data in the program
- Mediation through the kernel isn't needed after the memory is shared

Pros and cons of message passing

- Message passing requires:
 - A system call to read
 - A system call to write
 - Copying the message into kernel memory
 - Copying the message into receiver memory
- Thus, sending lots of messages can cause a lot of overhead
- However, sending a small number of messages can be less expensive than setting up shared memory
- Message passing naturally handles the problem of synchronization
 - Making sure that timing doesn't corrupt memory

Pros and cons of shared memory

- It's computationally expensive to set up the shared memory
- But that's a one-time cost
- If two processes are sharing lots of messages, it can be more efficient to use a shared memory system
- Perhaps the more significant problem with shared memory is synchronization
 - Processes reading and writing the same memory can leave the memory in an inconsistent state
 - If one process executes $x += 100$ while another executes $x -= 100$, the result could be the correct x or the incorrect $x + 100$ or $x - 100$
- Tools must be used to guarantee synchronization

IPC taxonomy

- Using the categories from the previous slide, we can list all of the IPC techniques that will be covered in this class

| Technique | Model | Purpose | Granularity | Network |
|--------------------|-----------------|-----------------|-------------|---------|
| Pipe/FIFO | Message passing | Data exchange | Byte stream | Local |
| Socket | Message passing | Data exchange | Either | Either |
| Message queue | Message passing | Data exchange | Structured | Local |
| shm() | Shared memory | Data exchange | None | Local |
| Memory-mapped file | Shared memory | Data exchange | None | Local |
| Signal | Message passing | Synchronization | None | Local |
| Semaphore | Message passing | Synchronization | None | Local |

- Signals are a very limited form of IPC

Pipes

Pipes

- Pipes are a way to do message passing between two processes
 - The bytes flow in one direction
 - There's a different file descriptor for each end
 - Think of it like a pipe where water is poured into one end and comes out the other
- Internally, the shell uses pipes to communicate between two programs when you use the | operator on the command line

```
sort foo.txt | grep -i error | head -n 10
```

Pipe details

- Pipes only go in one direction
 - One end is the reading end, and the other is the writing end
- Pipes preserve order
 - The bytes read come out in the same order they were written
- Pipes have limited capacity
 - If a pipe is full, trying to write to the pipe will block until more is read
- Pipes are unstructured
 - It's all just bytes, so the processes have to know what kind of data to expect
- Messages smaller than **PIPE_BUF** are sent atomically
 - Two processes writing messages to a pipe will not get their messages garbled

Pipe mechanics

- The `pipe()` function takes an `int` array of length 2 to hold file descriptors corresponding to the ends of the pipe

```
int pipe (int pipefd[2]);
```

- It's convention to use element 0 for reading and element 1 for writing
- For piping between parent and child, the call to `pipe()` happens before the `fork()`, so that both have clones of the same file descriptors
- One process reads from the pipe and the other writes
- Each process closes the end that they're not using

Pipe example

```
int pipefd[2];
char buffer[10];
memset (buffer, 0, sizeof (buffer));
int result = pipe (pipefd); // Open the pipe
assert (result >= 0);

pid_t child_pid = fork (); // Create child process
assert (child_pid >= 0);
if (child_pid == 0)
{
    close (pipefd[1]); // Child closes writing end
    ssize_t bytes_read = read (pipefd[0], buffer, 10); // Read from pipe
    if (bytes_read <= 0)
        exit (1);

    printf ("Child received: '%s'\n", buffer);
    exit (0);
}

close (pipefd[0]); // Parent closes the reading end
strncpy (buffer, "hello", sizeof (buffer));
printf ("Parent is sending '%s'\n", buffer);
write (pipefd[1], buffer, sizeof (buffer)); // Parent sends "hello"
wait (NULL); // Wait for child to terminate
```

Pipes and shell commands

- Let's go back to our command-line example:

```
sort foo.txt | grep -i error | head -n 10
```

- What's happening behind the scenes?
- The shell is calling **fork ()** and **exec ()** to run each of those processes
- Then, each process is linked to the next one with a pipe
- But how do those arbitrary processes know to read from or write to a pipe?
- They don't**, so the shell magically changes **stdout** or **stdin** to pipe file descriptors



dup2 ()

- The **dup2 ()** function closes a new file descriptor and replaces it with an old file descriptor

```
int dup2 (int oldfd, int newfd) ;
```

- This function is used by the shell to close their **stdin** or **stdout** and replace it with an end of a pipe
- The syntax is confusing:
 - The first file descriptor continues to function
 - All uses of the second one are performed with the first one

FIFOs

FIFOs

- Pipes are great for parent and child processes
 - Create the pipes in the parent, use them in the children
- But what if two unrelated processes want to share a pipe?
- **FIFOs** or **named pipes** are pipes associated with a file name
- These files can be seen in the file system, but they're special files intended only for use as pipes
- Naming:
 - In Linux, it's common to put these files in the `/tmp/` directory
 - It's important to pick a file name that's unlikely to collide with other FIFOs

The `mkfifo()` function

- The `mkfifo()` function is used to create a FIFO

```
int mkfifo (const char *path, mode_t mode);
```

- The `mode` is a bitwise OR of the permissions you want the FIFO to have (who can read and write)
- Using it creates the FIFO (which looks like a file), but programs still have to open it to use it and close it when done
- After the FIFO is done being used, the `unlink()` function removes the path from the file system

```
int unlink (const char *path);
```

FIFO example reader

- The following code creates a FIFO and reads `int` values until it gets a 0

```
const char *FIFO = "/tmp/MY_FIFO";
assert (mkfifo (FIFO, S_IRUSR | S_IWUSR) == 0);
int fifo = open (FIFO, O_RDONLY); // Open FIFO, delete if fails
if (fifo == -1)
{
    fprintf (stderr, "Failed to open FIFO\n");
    unlink (FIFO);
    return 1;
}

bool done = false;
while (!done)
{
    int value = 0;
    if (read (fifo, &value, sizeof (int)) == sizeof (int)) {
        if (value == 0)
            done = true;
        else
            printf ("%d\n", value);
    }
}
close (fifo);
unlink (FIFO);
```

FIFO example writer

- The following code opens the FIFO and writes 6 `int` values to it

```
const char *FIFO = "/tmp/MY_FIFO";

int fifo = open (FIFO, O_WRONLY);
assert (fifo != -1);

for (int index = 5; index >= 0; index--)
{
    write (fifo, &index, sizeof (int));
    sleep (1); // Sleep for a second before writing more
}

close (fifo);
```

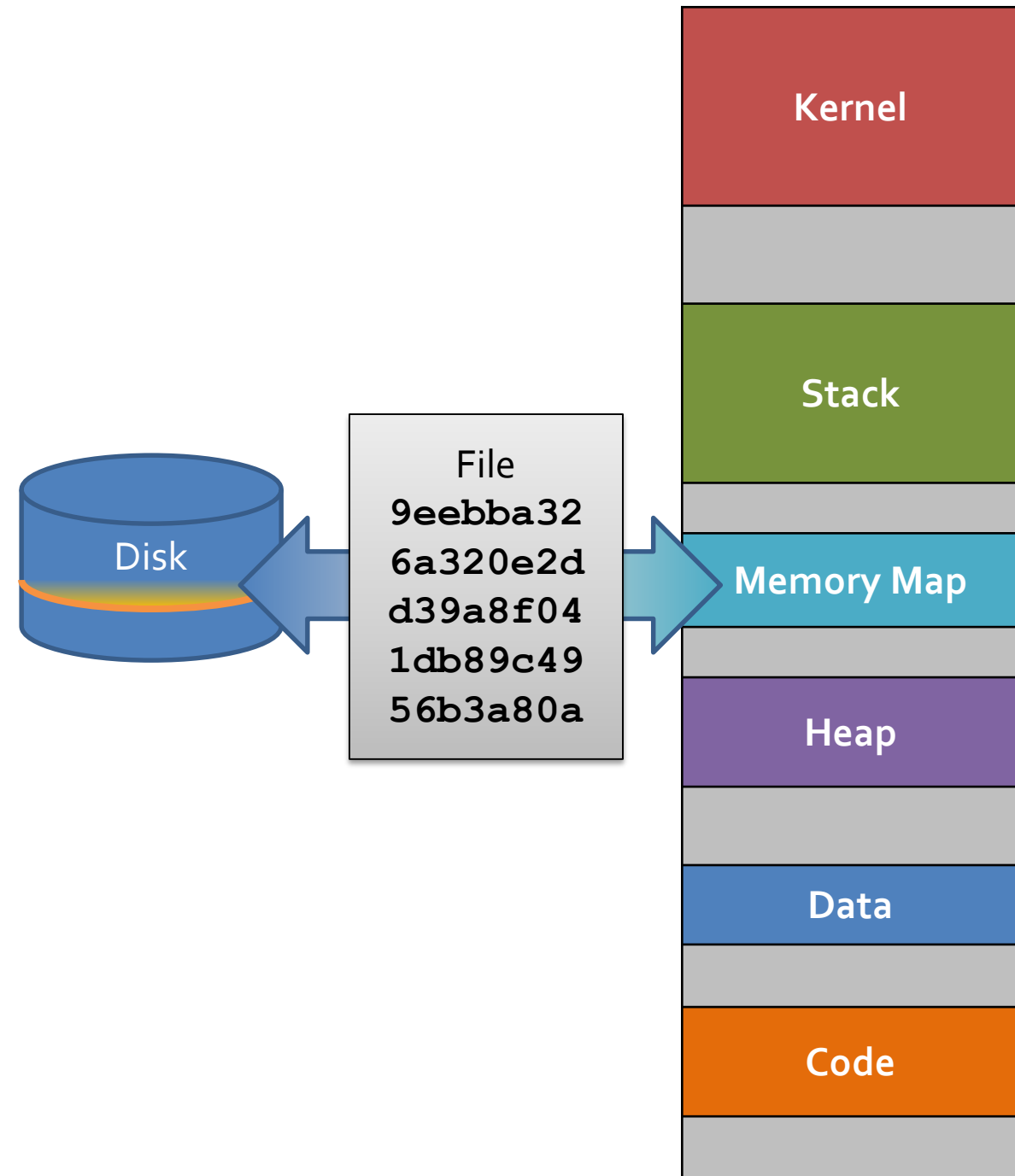
Memory-Mapped Files

Memory-mapped files

- Having covered pipes and FIFOs, we'll jump to the other side of the fence and talk about shared memory
- One shared memory technique are **memory-mapped files**
- A normal file is *mapped* into the virtual memory of a process
- Data can be read and written into that memory using normal pointer operations
 - And the data will magically get read and written to the file!
- One process can use memory-mapped files to interact with a file without using **read()** or **write()** calls
- But two or more processes can use memory-mapped files to exchange data directly

Visualization

- There's actually a special segment we haven't talked about in virtual memory before used just for memory mapping
 - Between the heap and the stack
- The virtual memory system is able to read only needed parts of the file into memory (often a page at a time)
- Storing data into this memory is eventually written back to the file



Advantages

- Over regular file access
 - Multiple processes can have read-only access to a common file
 - Often done with shared libraries, so that many different processes are able to access, for example, the same code for `printf()`
 - Programs can sometimes be simpler because there's no need to use `fseek()` to jump around a file
 - Reading files can be more efficient because the file contents don't have to be copied into the kernel's buffer cache
- Compared to other kinds of IPC
 - Writable memory-mapped files are fast for IPC
 - Unlike message passing, data continues to exist and can be read repeatedly

Mechanics

- The `mmap()` function returns memory mapped to a particular file descriptor

```
void *mmap (void *addr, size_t length, int prot, int flags,  
int fd, off_t offset);
```

- `addr` is a suggestion for where the memory goes but should usually be **NULL**
- `length` is how many bytes to map
- `prot` are flags shown on the right that can be combined
- `flags` are **MAP_SHARED** or **MAP_PRIVATE** (and others), depending on whether the area is shared
- `fd` is an open file descriptor for a file
- `offset` is the starting point inside the file

| Protection | Actions permitted |
|-------------------------|------------------------|
| <code>PROT_NONE</code> | May not be accessed |
| <code>PROT_READ</code> | Region can be read |
| <code>PROT_WRITE</code> | Region can be modified |
| <code>PROT_EXEC</code> | Region can be executed |

Other useful functions

- The `munmap ()` function unmaps an existing map

```
void munmap (void *addr, size_t length);
```

- `addr` is the start of the mapped address
- `length` is how much to unmap
- The `msync ()` function synchronizes the file with the mapped memory

```
void msync (void *addr, size_t length, int flags);
```

- `MS_ASYNC` flag returns immediately and `MS_SYNC` waits for the sync to complete

Example

- The following example checks to make sure that the 2nd, 3rd, and 4th bytes of an executable are "ELF", a marker of the executable and linking format used by Linux

```
int fd = open ("/bin/bash", O_RDONLY);
assert (fd != -1);

struct stat file_info;
assert (fstat (fd, &file_info) != -1);

// Map whole file for reading, unshared
char *mapping = mmap (NULL, file_info.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
assert (mapping != MAP_FAILED);

// Bytes 1 - 3 of the file must be 'E', 'L', 'F'
if (mapping[1] == 'E' && mapping[2] == 'L' && mapping[3] == 'F')
    printf("Valid executable!\n");
else
    printf("Invalid executable!\n");

munmap (mapping, file_info.st_size); // Unmap file and close it
close (fd);
```

POSIX IPC

POSIX IPC

- POSIX IPC function refer to IPC object named with a string that follows a particular format:
 - It must start with a slash
 - It must have one or more non-slash characters
 - Example: **/comp3400_mqueue**
- Object names must be unique
- These objects often appear as files in the file system, but you shouldn't interact with them using normal file commands
- POSIX IPC connections also have two other (familiar) values:
 - **oflag**: Access needed, a bitwise OR of flags like **O_RDONLY**, **O_WRONLY**, **O_RDWR**, **O_CREAT**, and **O_EXCL**
 - **mode**: Permissions, a bitwise OR of flags like **S_IWUSR** and **S_IRGRP**

Message queues

- Message queues are a form of message-passing IPC
- But don't we already have pipes and FIFOs?
- Differences from pipes:
 - Messages are sent as units: one whole message is retrieved at a time
 - Message queues use identifiers, not file descriptors, requiring special functions instead of **read()** and **write()**
 - Messages have priorities, not just first-in-first-out
 - Messages exist in the kernel, so killing off the sending process won't destroy them
- The big difference is structure:
 - Pipes and FIFOs send bytes, and the reader can read any number of available bytes at a time
 - Message queues send messages as units

POSIX message queue functions

- `mqd_t mq_open (const char *name, int oflag, ...
/* mode_t mode, struct mq_attr *attr */);`
 - Open (and possibly create) a POSIX message queue.
- `int mq_getattr(mqd_t mqdes, struct mq_attr *attr);`
 - Get the attributes associated with a given message queue
- `int mq_close (mqd_t mqdes);`
 - Close a message queue
- `int mq_unlink (const char *name);`
 - Remove a message queue's name (and the message queue itself, when all processes close it)
- `int mq_send (mqd_t mqdes, const char *msg_ptr,
size_t msg_len, unsigned int msg_prio);`
 - Send a message with a given length and priority
- `ssize_t mq_receive (mqd_t mqdes, char *msg_ptr,
size_t msg_len, unsigned int *msg_prio);`
 - Receive a message into a buffer and get its priority

Message queue sending example

- The following code creates a message queue and sends "WOMBAT"

```
mqd_t mqd = mq_open ("/comp3400_mq", O_CREAT | O_EXCL | O_WRONLY, 0600,  
    NULL); // mq_open() requires four arguments when creating  
  
if (mqd == -1) // Check for error  
{  
    perror ("mq_open failed");  
    exit (1);  
}  
  
mq_send (mqd, "WOMBAT", 7, 10); // Send WOMBAT (7 chars) with priority 10  
mq_close (mqd);
```

- Priority increases as the number increases
- Priorities start at 0 and go up to at least 31, but some systems go as high as 32768
- Read documentation to find out how many priority levels there are

Warning!

- With pipes and FIFOs, it's common to create a fixed-size buffer and then read into it, usually only filling part of it
- With message queues, you have to read *exactly* the size of a message that's waiting for you
 - If not, the read will fail
- Two strategies:
 - Use a system where the sizes of messages are always the same
 - Use the `mq_getattr()` function to get the attributes of a message waiting in the message queue and create a buffer exactly the right size to read it

Message queue receiving example

- The following code reads the "WOMBAT" message sent by the other code
- It uses `mq_getattr()` to find out how big of a buffer it needs

```
mqd_t mqd = mq_open ("/comp3400_mq", O_RDONLY); // Only two arguments to open
assert (mqd != -1);

struct mq_attr attr;
assert (mq_getattr (mqd, &attr) != -1); // Get attributes

char *buffer = calloc (attr.mq_msgsize, 1); // Allocate buffer with size
assert (buffer != NULL);

unsigned int priority = 0;
if ((mq_receive (mqd, buffer, attr.mq_msgsize, &priority)) == -1) // Get message
    printf ("Failed to receive message\n");
else
    printf ("Received [priority %u]: '%s'\n", priority, buffer);

free (buffer);
buffer = NULL;
mq_close (mqd);
```

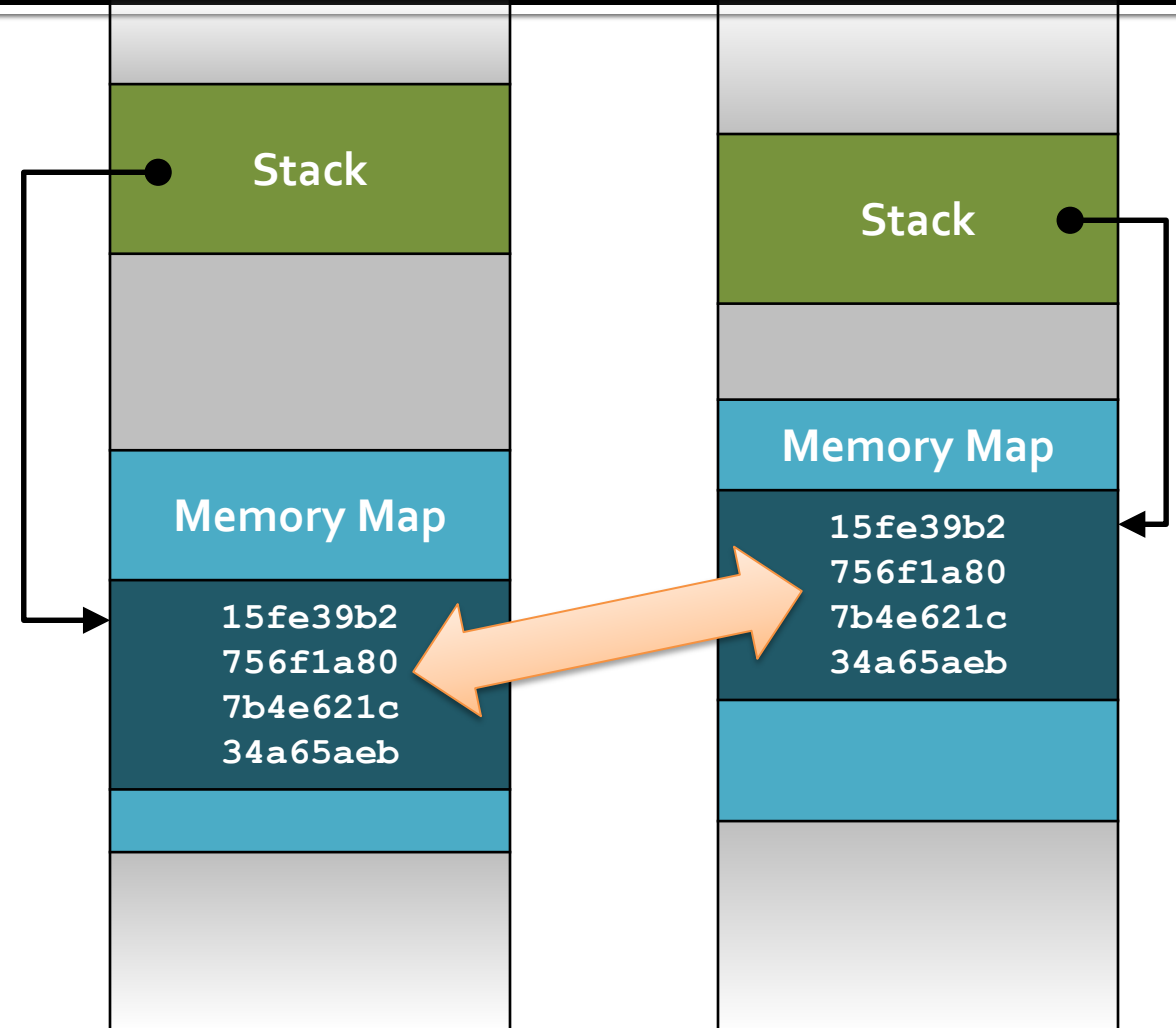
Shared Memory

Shared memory

- Shared memory is pretty much the same as using memory-mapped files
 - Except that there's no file associated with the share
 - So there's no persistent record of the memory
- To share memory, create a shared memory object (like a file, but isn't) with **shm_open()**
- The size of this object is often resized with **ftruncate()**
- Then, this shared memory object is mapped with **mmap()**, as was done with memory mapped files
- To delete the shared memory object, use **shm_unlink()**

Visualization

- The shared memory mapping means that a region of memory in one process exactly corresponds to memory in another region of memory in another process
- It's unlikely that the mapped memory will be in the same location in virtual memory for the two processes



Functions

```
int shm_open (const char *name, int oflag, mode_t mode);
```

- **name** gives the name of the object
- **oflag**: Access needed, a bitwise OR of flags like `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, and `O_EXCL`
- **mode**: Permissions, a bitwise OR of flags like `S_IWUSR` and `S_IRGRP`

```
int shm_unlink (const char *name);
```

- **name** is the object to delete

```
int ftruncate (int fd, off_t length);
```

- **fd** is a descriptor for the object or file to resize
- **length** is the new size

Example of memory mapping

- First, let's imagine a struct declaration for structs that contain permission information

```
struct permission
{
    int user;
    int group;
    int other;
};
```


Example of memory mapping continued

- A parent process:
 - Creates a memory-mapped object
 - Stretches it to be exactly the right size
 - Maps some memory to this object

```
int shmfd = shm_open ("/comp3400_shm", O_CREAT | O_EXCL | O_RDWR,  
    S_IRUSR | S_IWUSR);  
assert (shmfd != -1);  
  
// Resize to hold one struct  
assert (ftruncate (shmfd, sizeof (struct permission)) != -1);  
  
// Map the object into memory  
struct permission *perm = mmap (NULL, sizeof (struct permission),  
    PROT_READ | PROT_WRITE, MAP_SHARED, shmfd, 0);  
assert (perm != MAP_FAILED);
```

Example of memory mapping continued

- Fork the process
- Then, the child process:
 - Sets values in the struct
 - Unmaps the memory
 - Closes the object

```
pid_t child_pid = fork();
if (child_pid == 0)
{
    perm->user = 6;
    perm->group = 4;
    perm->other = 0;

    // Unmap and close the child's shared memory
    munmap (perm, sizeof (struct permission));
    close (shmfd);
    exit(0);
}
```

Example of memory mapping finished

- Finally, the parent process:
 - Waits for the child to finish
 - Outputs the data stored by the child
 - Unmaps the memory and closes the object
 - Deletes the object

```
wait (NULL); // Wait for the child to finish

// Read from mapped memory
printf ("Permission bit-mask: 0%d%d%d\n",
        perm->user, perm->group, perm->other);

munmap (perm, sizeof (struct permission)); // Unmap
close (shmfd); // Close object
shm_unlink ("/comp3400_shm"); // Delete object
```

Semaphores

Synchronization

- Both of the kinds of shared-memory IPC we've talked about often need synchronization
- **Synchronization** means controlling when reads and writes happen to avoid getting meaningless results
- In the previous example, a parent process waited for the child process to finish writing (and die) before reading
- In general, doing so is undesirable:
 - Many communicating processes do not have a parent/child relationship
 - Waiting for a process to die means that there can't be back-and-forth communication

Semaphores

- **Semaphores** are a simple kind of synchronization
- Internally, they have a counter
- If a process calls wait on a semaphore and the semaphore's value is 0 or lower, the process will get blocked
- When another process calls post and the counter goes up, a blocked process will resume (decrementing the counter back to 0 first)
- Many processes can be waiting on a single semaphore, but only one will resume per call to post
- Waiting on a semaphore is also called decrementing, downing, or P
- Posting on a semaphore is also called incrementing, upping, or V

Example

- Processes A and B have access to shared memory
- A is writing data, and B wants to read after the writing is done
- A and B also have access to a semaphore initialized to 0
- A increments the semaphore after it finishes writing
- B decrements the semaphore before reading
- Everything works out:
 - If B decrements the semaphore before A increments, B will block until A is done
 - If A increments the semaphore before B tries to decrement it, the semaphore will already be 1, so B will decrement it but not block

Semaphore functions

```
sem_t *sem_open (const char *name, int oflag,  
/* mode_t mode, unsigned int value */ );
```

- Return (and possibly create) a named semaphore, using the usual **oflag** and **mode** flags
- **value** determines the initial value of the semaphore (often 0)

```
int sem_wait (sem_t *sem);
```

- Block if the semaphore's value is 0, decrement after continuing

```
int sem_post (sem_t *sem);
```

- Increment the semaphore's value, unblocking a process if the value is 0

```
int sem_close (sem_t *sem);
```

- Close a semaphore

```
int sem_unlink (const char *name);
```

- Delete a semaphore

Trying or waiting

- Using a semaphore can be frustrating if you wanted to do other stuff and get blocked
- Instead of calling `sem_wait()`, there are two alternatives:

```
int sem_trywait (sem_t *sem);
```

- Tries to decrement the semaphore but gives an error code if it would block

```
int sem_timedwait (sem_t *sem, struct timespec *time);
```

- Waits on the semaphore but waits only for the amount of time specified in the `struct timespec`

Unnamed semaphores

- In order to avoid worrying about names, it's also possible to create unnamed semaphores, using the following functions:

```
int sem_init (sem_t *sem, int pshared, unsigned int value);
```

- Create an unnamed semaphore
- **pshared** is 0 if used only by threads of the same process and non-zero is shared by different processes

```
int sem_destroy (sem_t *sem);
```

- Delete an unnamed semaphore

Upcoming

Next time...

- Review up to Exam 2

Reminders

- **Department celebration today!**
 - Point patio
 - 11:30 a.m. – 1:30 p.m.
- CS Club study session for all finals today!
 - Point 113
 - 4:15 – 5:15 p.m.
- Work on Assignment 8
 - **Due Friday before midnight!**