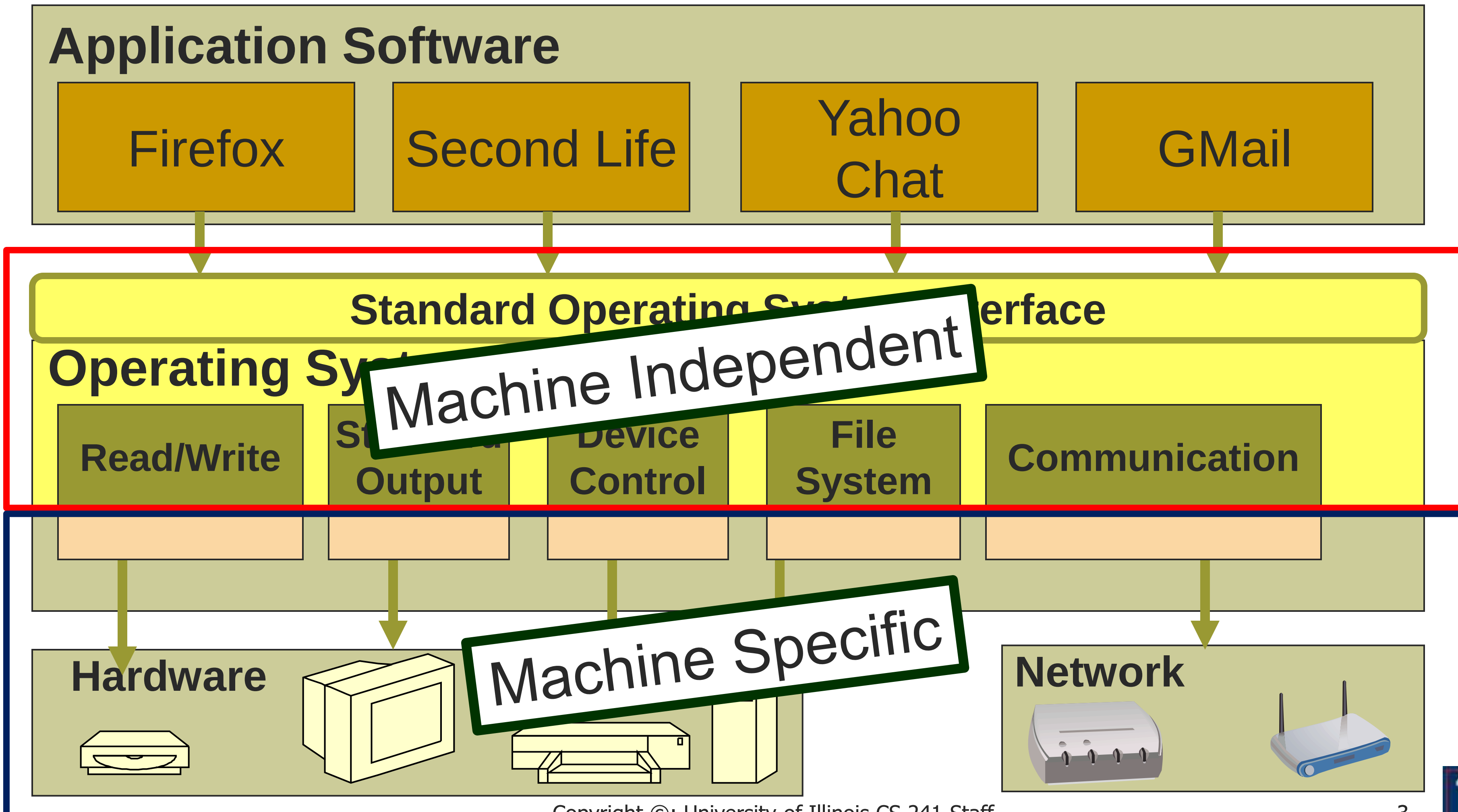
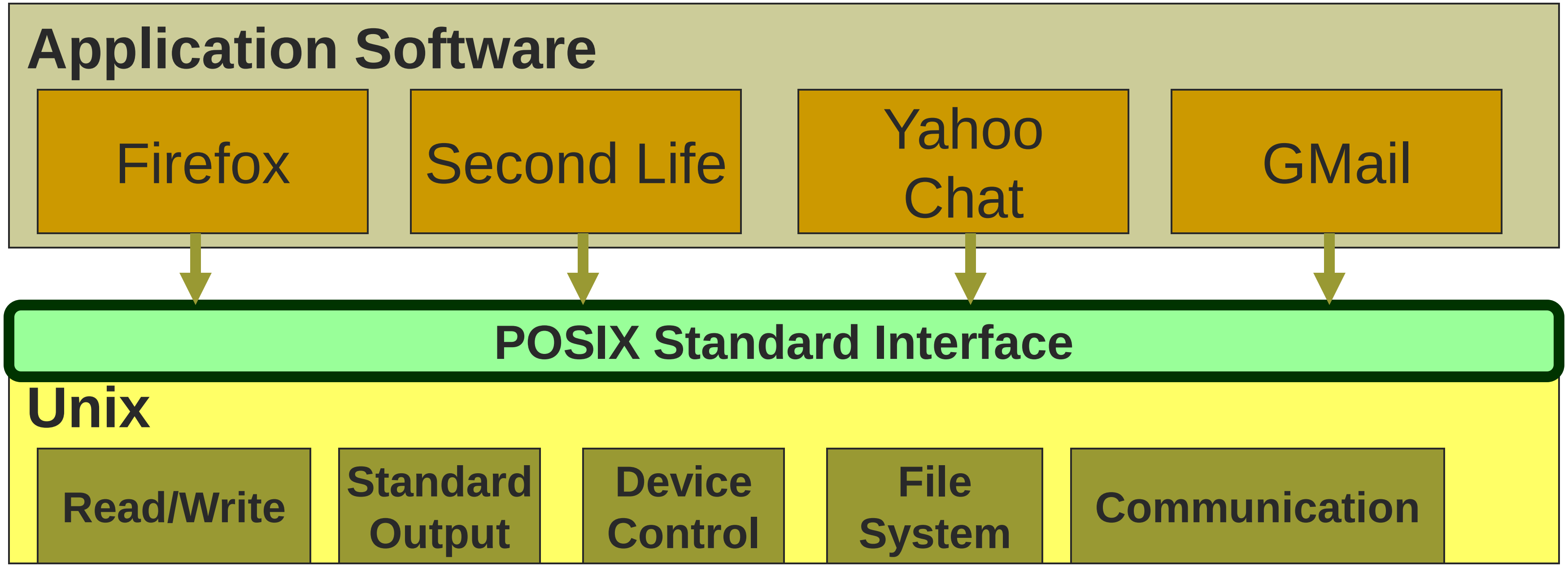
OS Structure



POSIX
The UNIX Interface Standard



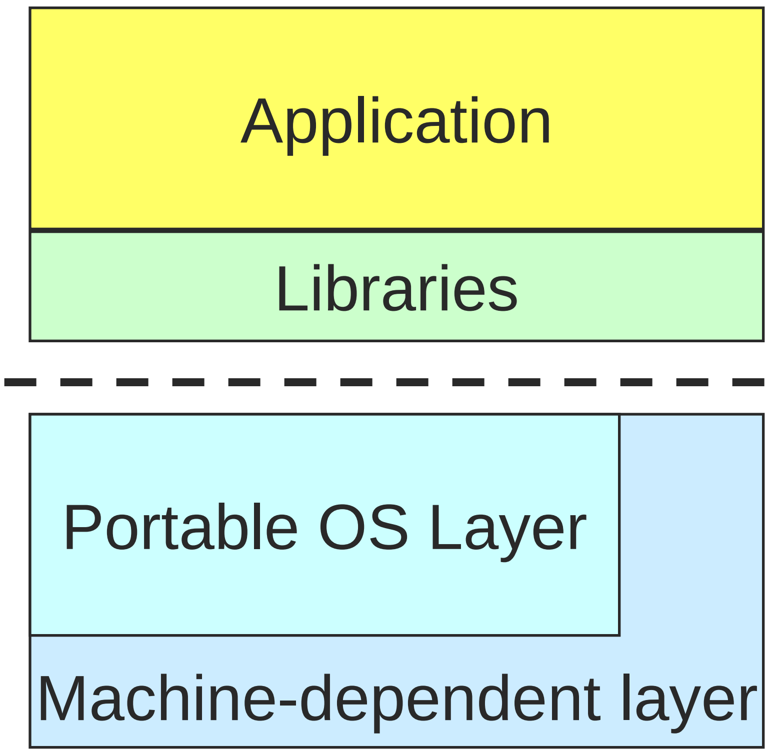
What is an Operating System?

- It is an *extended machine*
  - Hides the messy details that must be performed
  - Presents user with a virtualized and simplified abstraction of the machine, easier to use

- It is a *resource manager*
  - Each program gets time with the resource
  - Each program gets space on the resource

Copyright ©: University of Illinois CS 241 Staff

A Peek into Unix

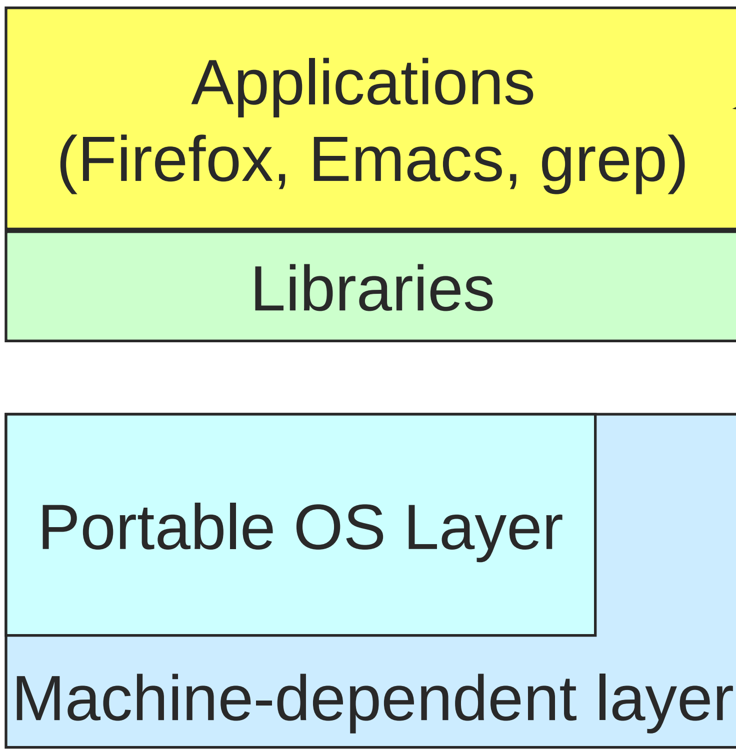


User space/level

Kernel space/level

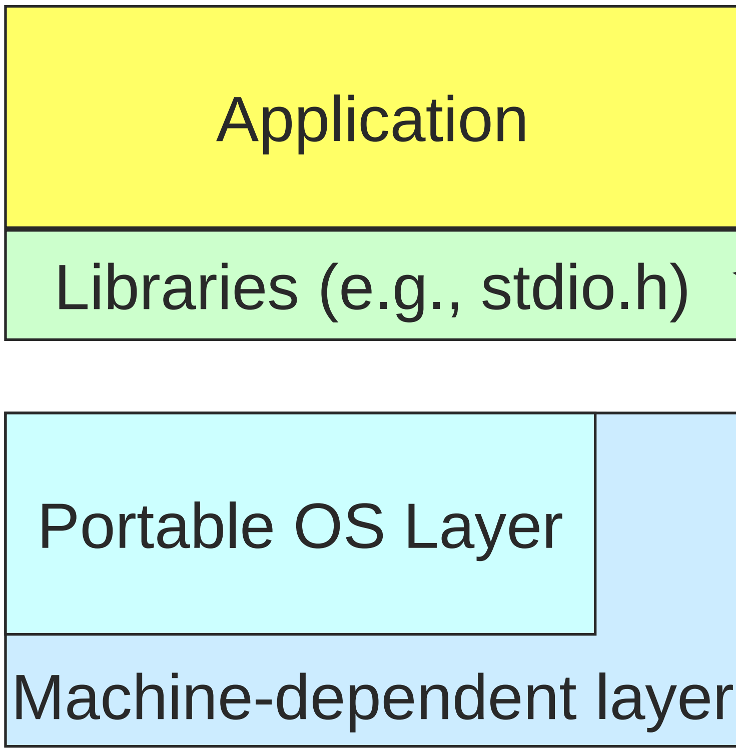
- **User/kernel modes are supported by hardware**
- **Some systems do not have clear user-kernel boundary**

Application



- Written by programmer
- Compiled by programmer
- Use function calls

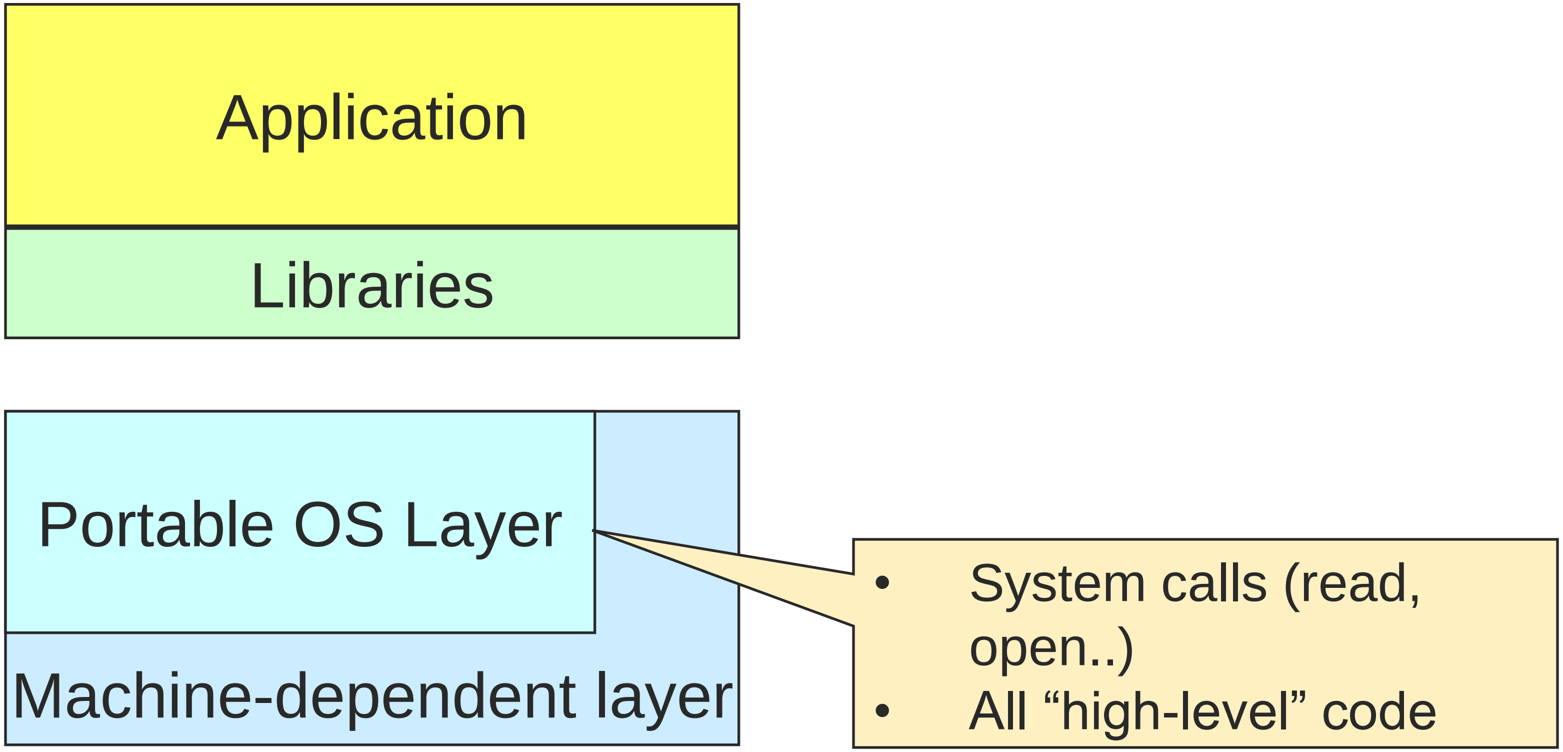
Unix: Libraries



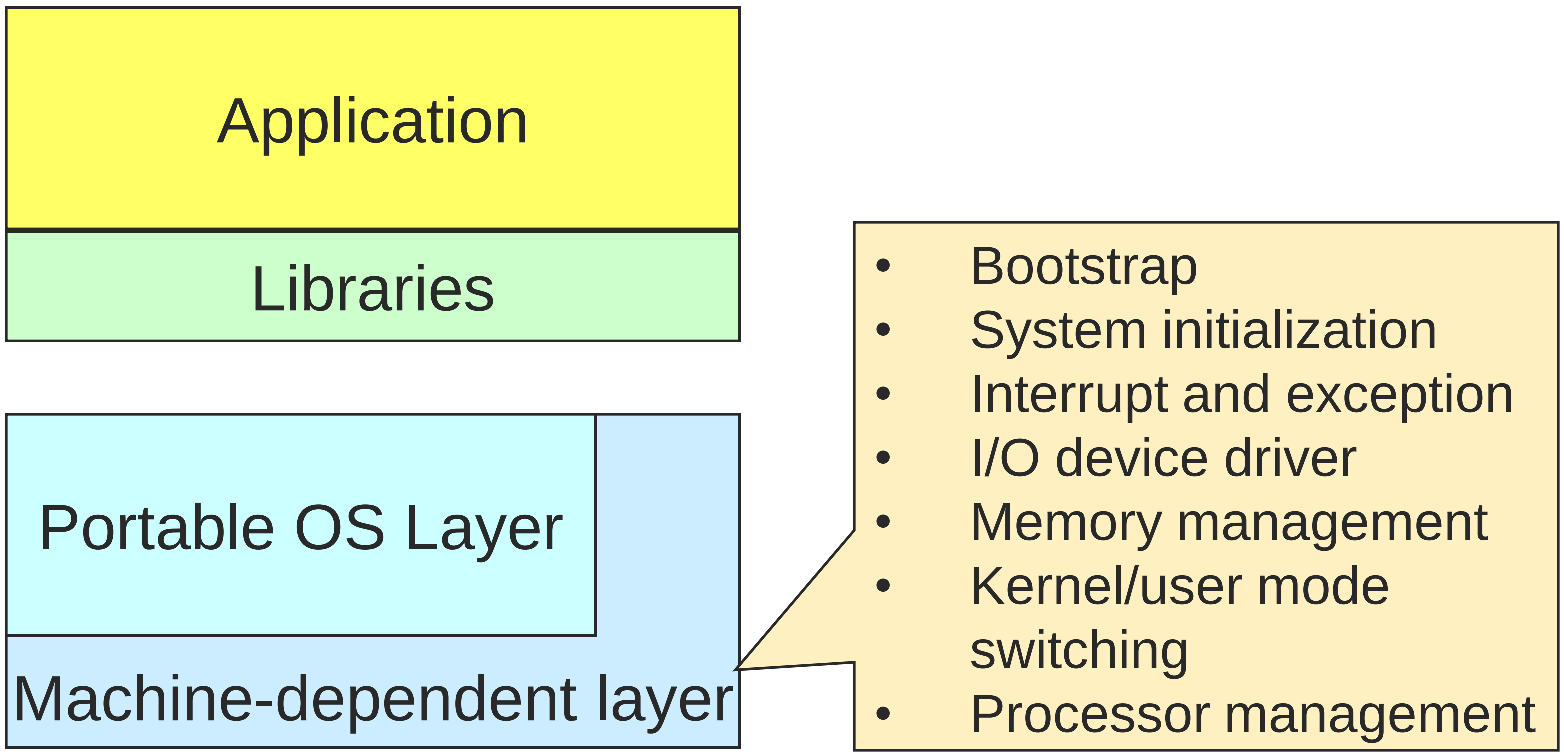
- Provided pre-compiled
- Defined in headers
- Input to linker (compiler)
- Invoked like functions
- May be “resolved” when program is loaded

Copyright ©: University of Illinois CS 241 Staff

Typical Unix OS Structure

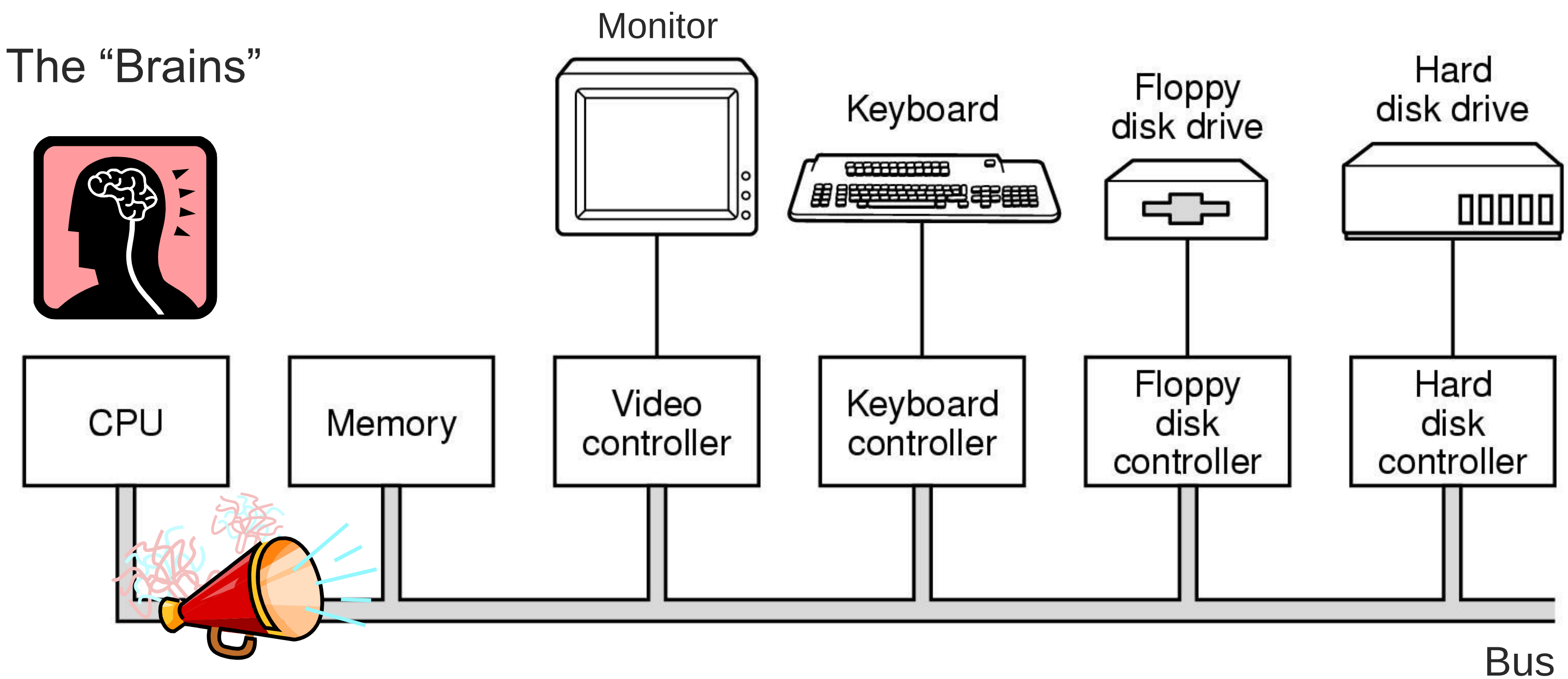


Typical Unix OS Structure



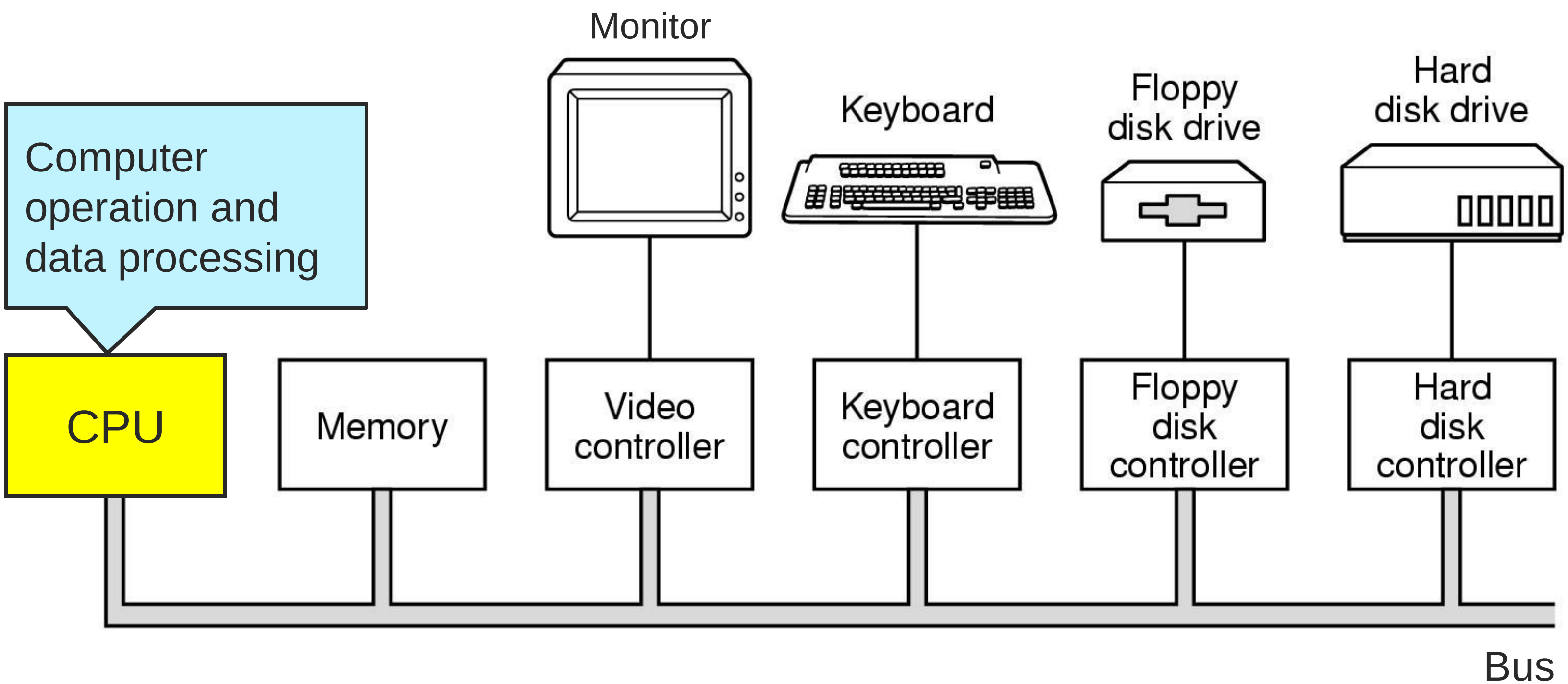
Copyright ©: University of Illinois CS 241 Staff

Computer Hardware Review



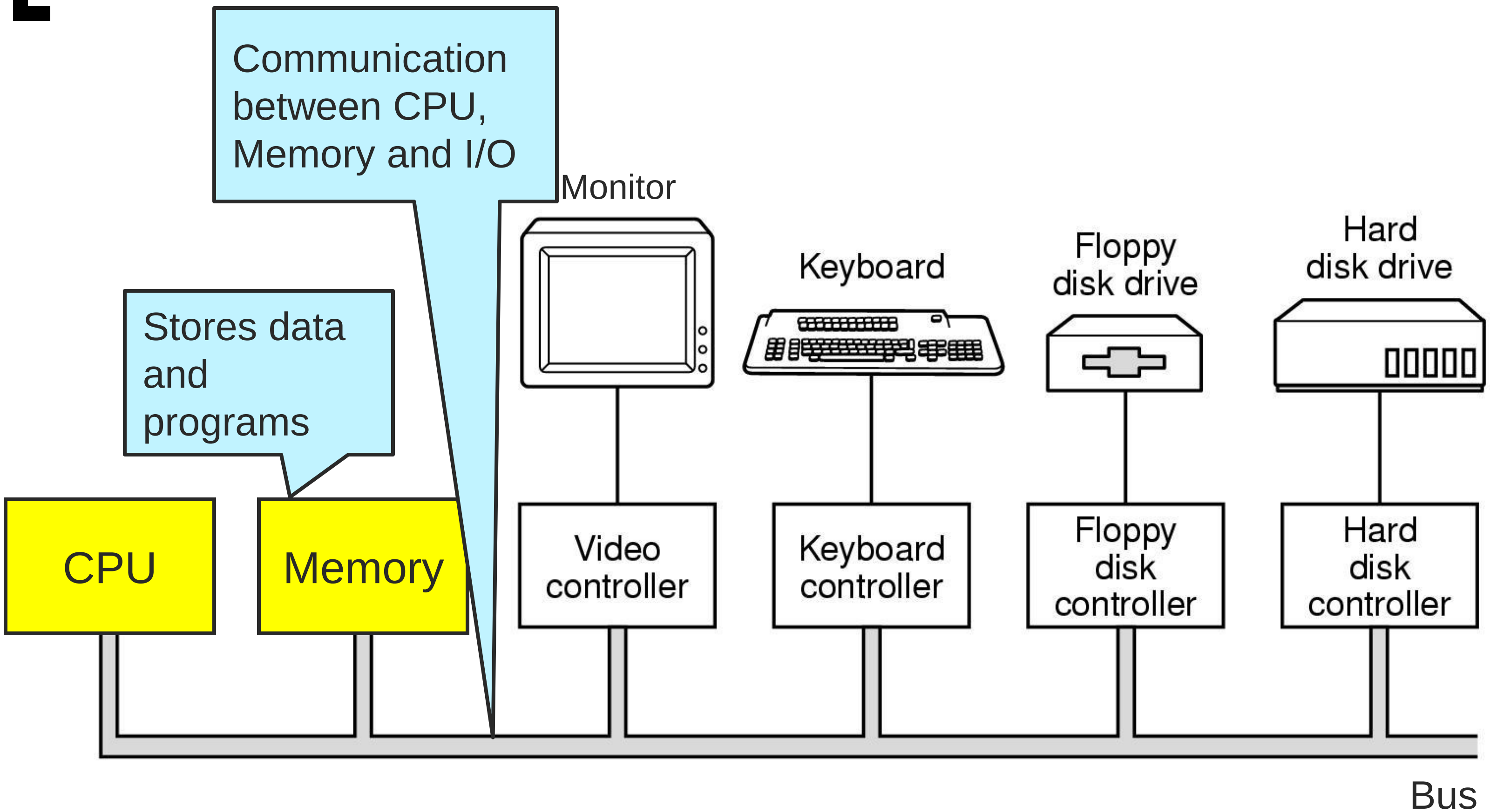
- Components of a simple personal computer

Computer Hardware Review



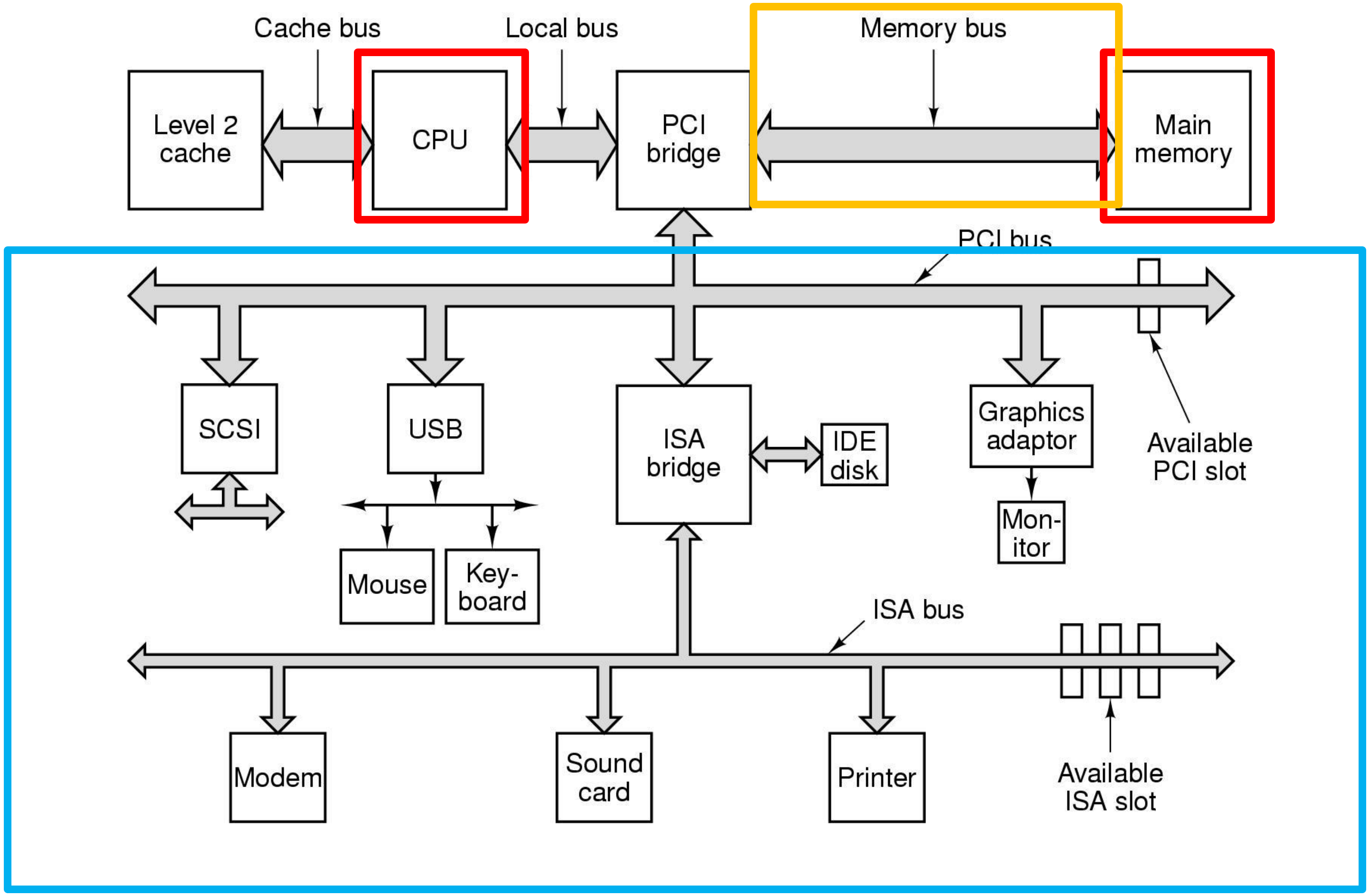
- Components of a simple personal computer

Computer Hardware Review



- Components of a simple personal computer

Early Pentium system



CPU, From CS231

- Fetch instruction from code memory
- Fetch operands from data memory
- Perform operation (and store result)
- (Check interrupt line)
- Go to next instruction

- 'Conventional CPU'
  (Ignore pipeline, optimization complexities)

CPU Registers

- Fetch instruction from code memory
- Fetch operands from data memory
- Perform operation (and store result)
- Go to next instruction

- Note: CPU must maintain certain state
  - Current instructions to fetch (program counter)
  - Location of code memory segment
  - Location of data memory segment

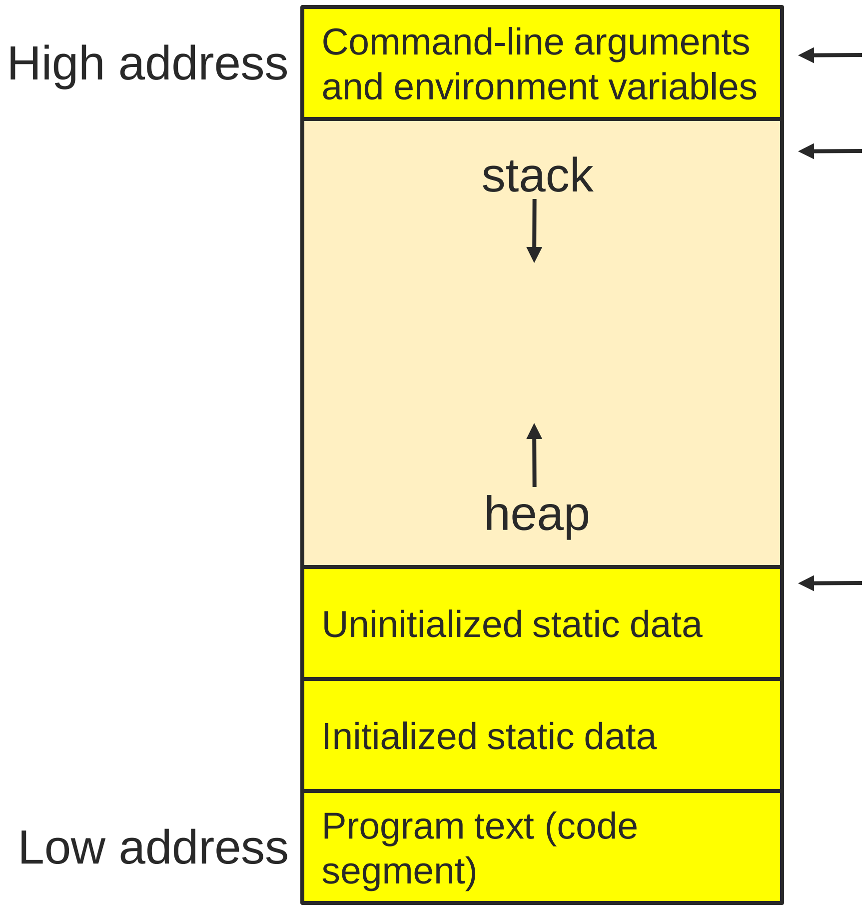
CPU Register Examples

- Hold instruction operands
- Point to start of
  - Code segment (executable instructions)
  - Data segment (static/global variables)
  - Stack segment (execution stack data)
- Point to current position of
  - Instruction pointer
  - Stack pointer

CPU Register Examples

- Hold instruction operands
- Point to start of
  - Code segment
  - Data segment
  - Stack segment
- Point to current position of
  - Instruction pointer
  - Stack pointer
    - Why stack?

Sample Layout for program image in main memory



`argc, argv`, environment

Activation record for function calls (return address, parameters, saved registers, automatic variables

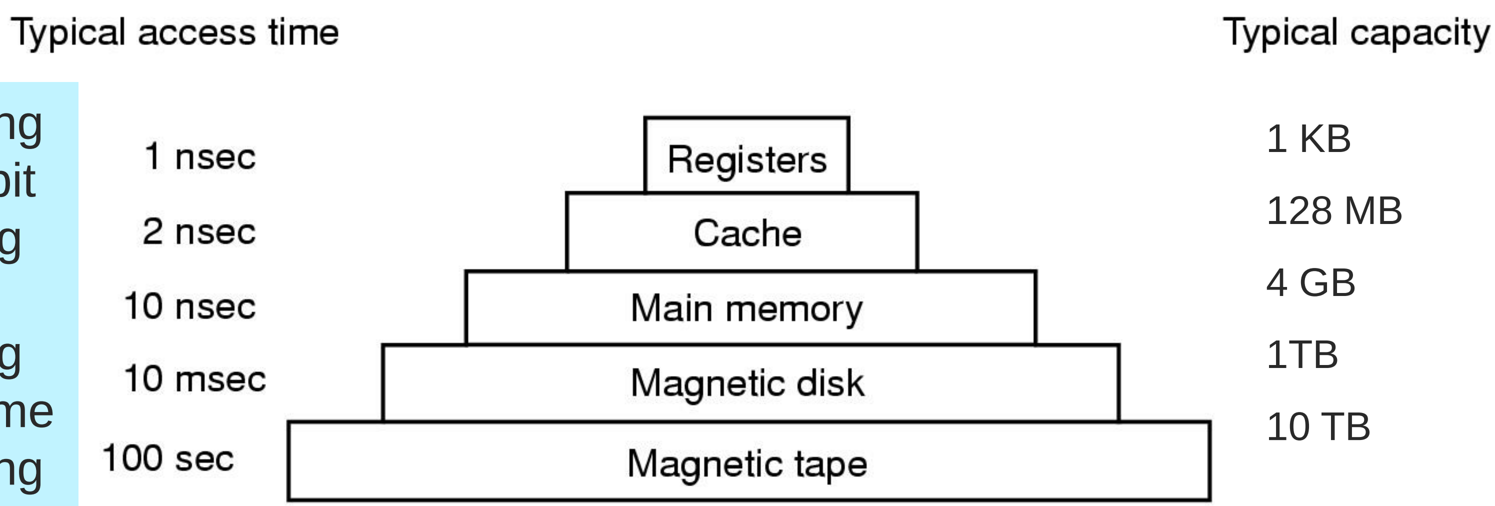
Allocations from **`malloc`** family

Processes have three segments: text, data, stack

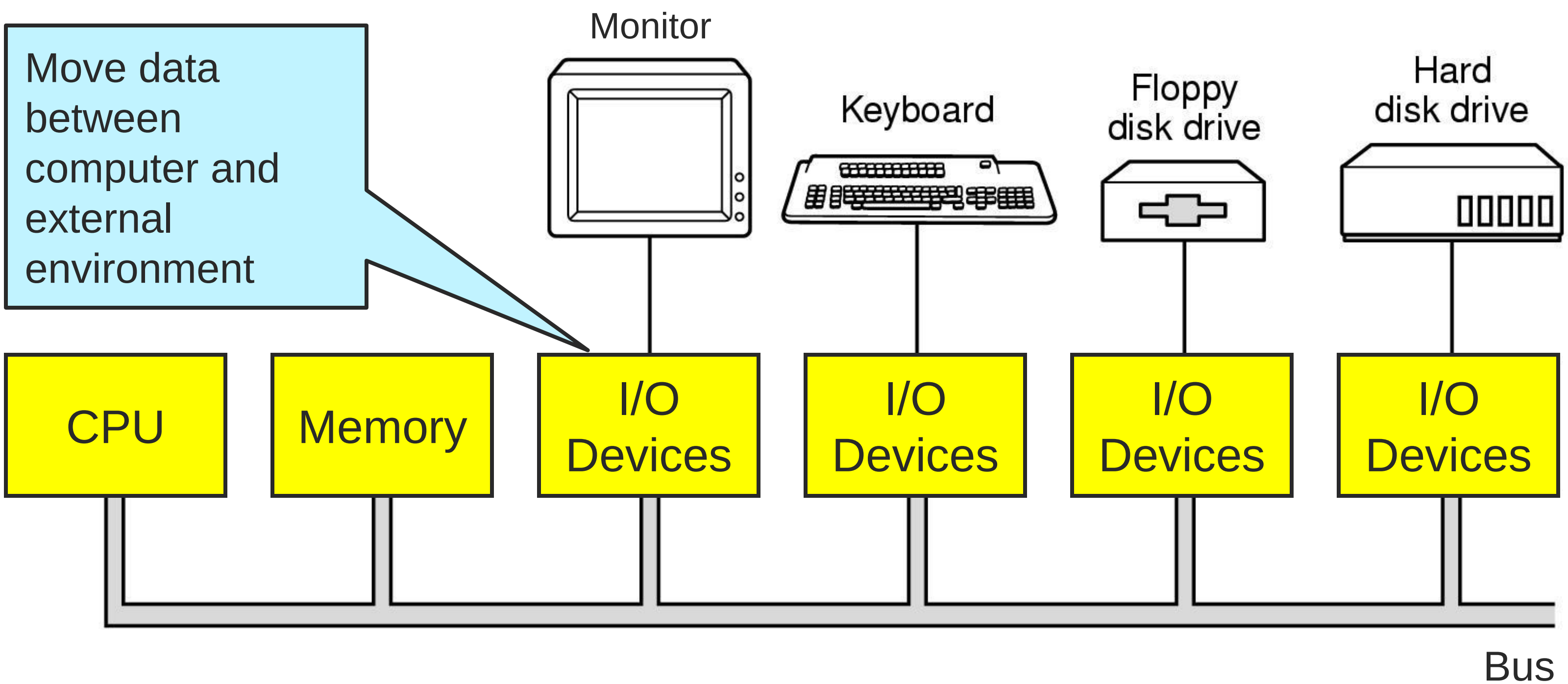
Memory Hierarchy

- Leverage locality of reference

1. Decreasing cost per bit
2. Increasing capacity
3. Increasing access time
4. Decreasing frequency of access



Computer Hardware Review



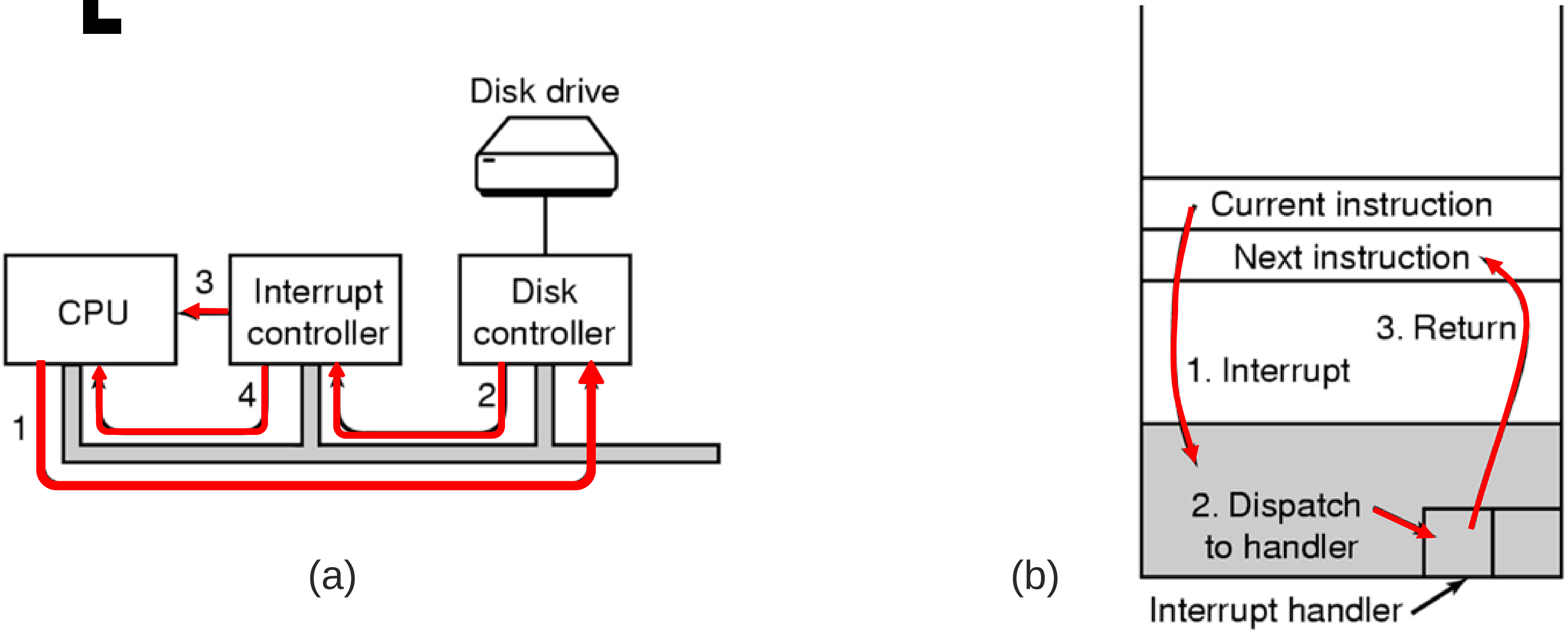
- Components of a simple personal computer

I/O Device Access

- System Calls
  - Application makes a system call
  - Kernel translates to specific driver
  - Driver starts I/O
  - Polls device for completion
- Interrupts
  - Application starts device
  - Asks for an interrupt upon completion
  - OS blocks application
  - Device controller generates interrupt

Copyright ©: University of Illinois CS 241 Staff

I/O Interrupt Mechanism



1. Application writes into device registers, Controller starts device
2. When done, device controller signals interrupt controller
3. Interrupt controller asserts pin on CPU
4. Interrupt controller puts I/O device number on bus to CPU

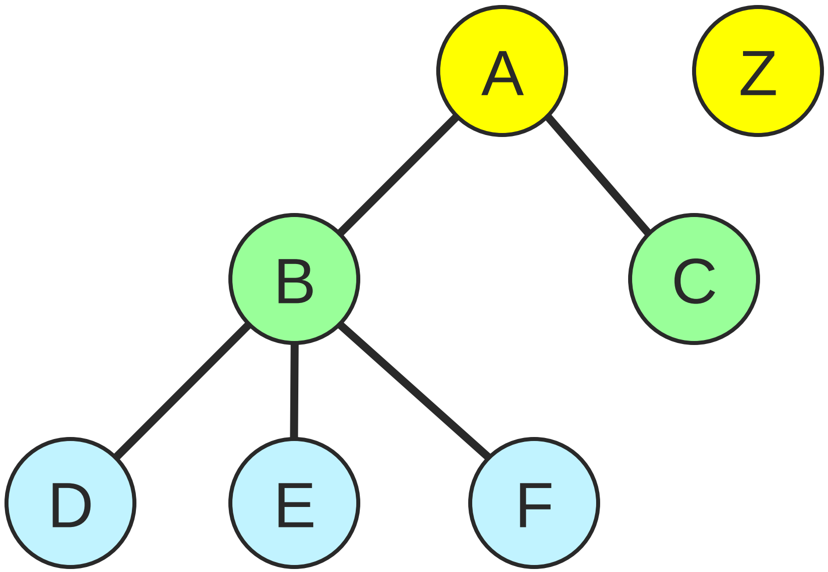
Operating System Concepts

- Shared resources
  - I have B KB of memory, but need 2B KB
  - I have N processes trying to access the disk at the same time
  - How would you control access to resources?
- Challenges
  - Who gets to use the resources?
  - How do you control fair use of the resources over time?
  - Deadlock

Copyright ©: University of Illinois CS 241 Staff

Operating System Concepts

- Process
  - An executable instance of a program
  - Only one process can use a (single-core) CPU at a time
- A process tree
  - A created two child processes, B and C
  - B created three child processes, D, E, and F



Copyright ©: University of Illinois CS 241 Staff

Operating System Concepts

- How would you switch CPU execution from one process to another?
- Solution: Context Switching
  - Store/restore state on CPU, so execution can be resumed from same point later in time
  - Triggers: multitasking, interrupt handling, user/kernel mode switching
  - Involves: Saving/loading registers and other state into a "process control block" (PCB)
  - PCBs stored in kernel memory

Copyright ©: University of Illinois CS 241 Staff

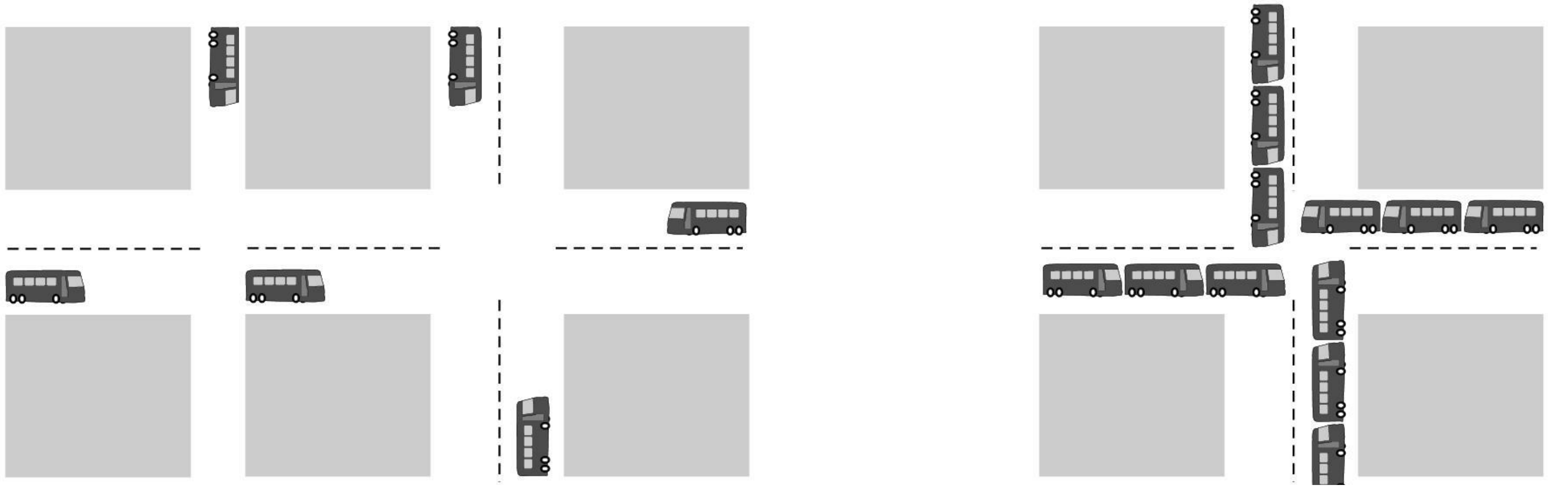
Operating System Concepts

- Context Switching
  - What are the costs involved?

Item	Time	Scaled Time in Human Terms (2 billion times slower)
Processor cycle	0.5 ns (2 GHz)	1 s
Cache access	1 ns (1 GHz)	2 s
Memory access	15 ns	30 s
Context switch	5,000 ns (5 micros)	167 m
Disk access	7,000,000 ns (7 ms)	162 days
System quanta	100,000,000 (100 ms)	6.3 years

Copyright ©: University of Illinois CS 241 Staff

Operating System Concepts

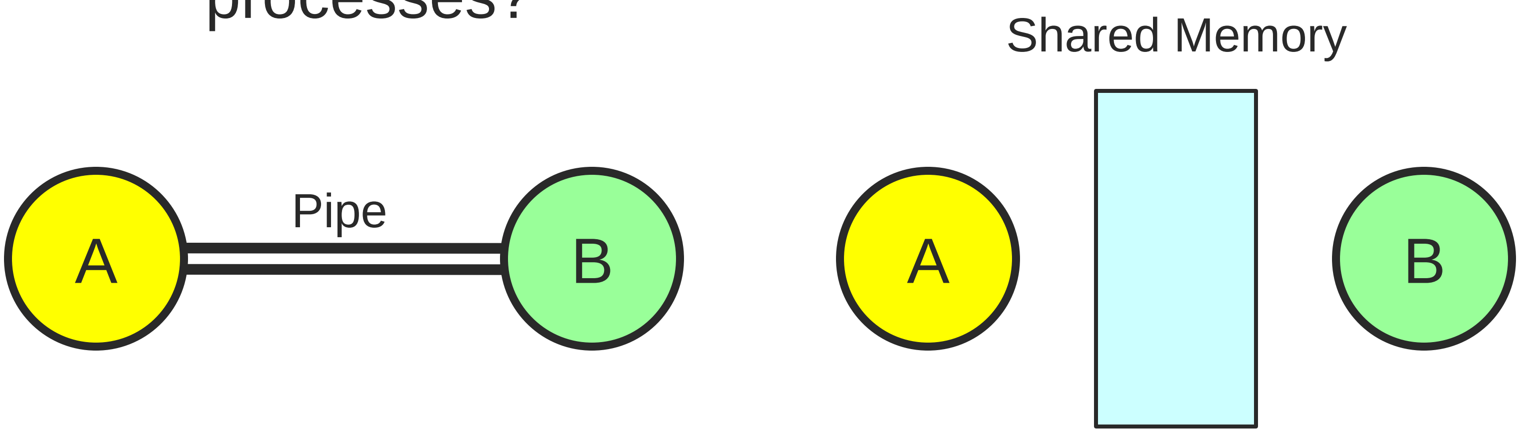


(a) A potential deadlock (b) An actual deadlock

- One challenge: Deadlock
  - Set of actions waiting for each other to finish
- Example:
  - Process A has lock on file 1, wants to acquire lock on file 2
  - Process B has a lock on file 2, wants to acquire lock on file 1

Operating System Concepts

- Inter-process Communication
  - Now process A needs to exchange information with process B
  - How would you enable communication between processes?



Copyright ©: University of Illinois CS 241 Staff

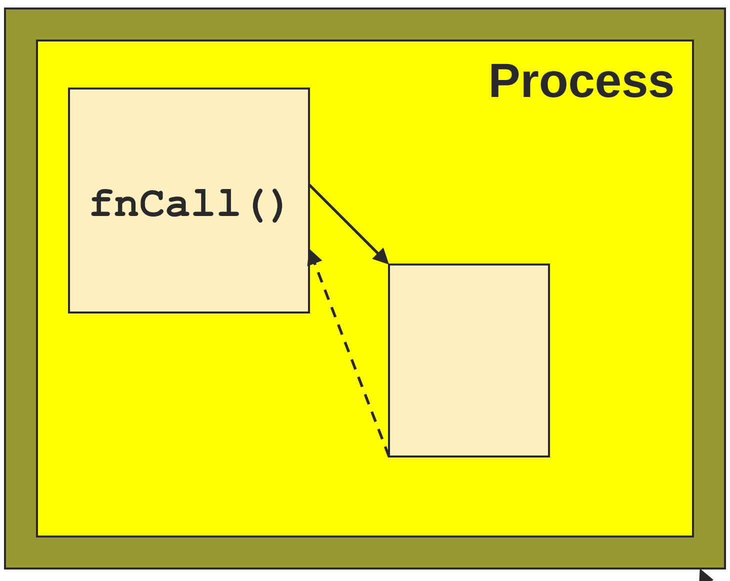
Summary

- Resource Manager
- Hardware independence
- Virtual Machine Interface
- POSIX
- Concurrency & Deadlock

Copyright ©: University of Illinois CS 241 Staff

System Calls versus Function Calls

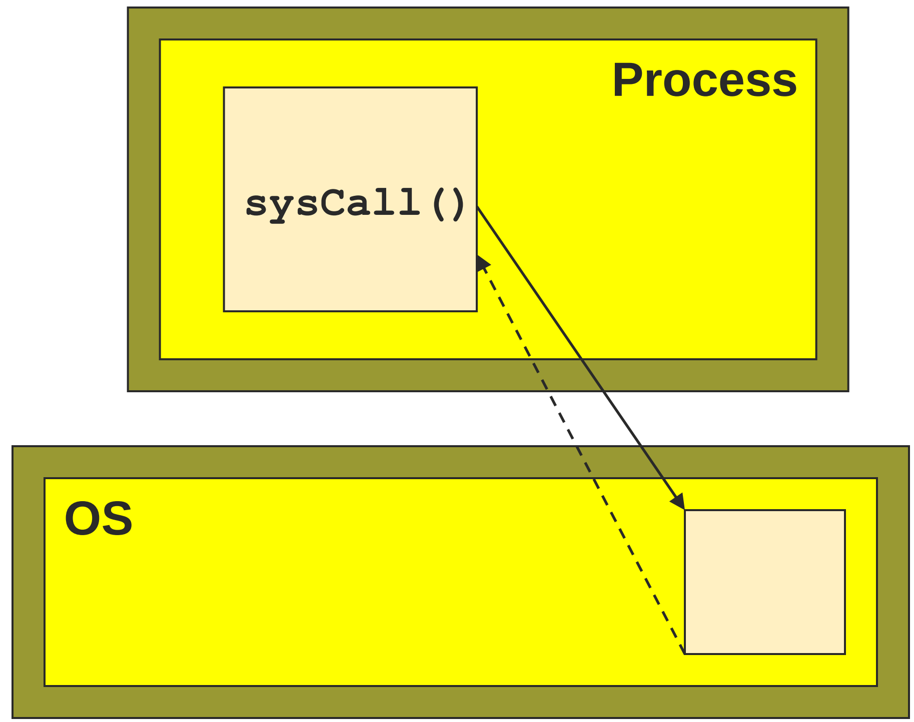
Function Call



Caller and callee are in the same Process

- Same user
- Same “domain of trust”

System Call



- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse

Copyright ©: University of Illinois CS 241 Staff

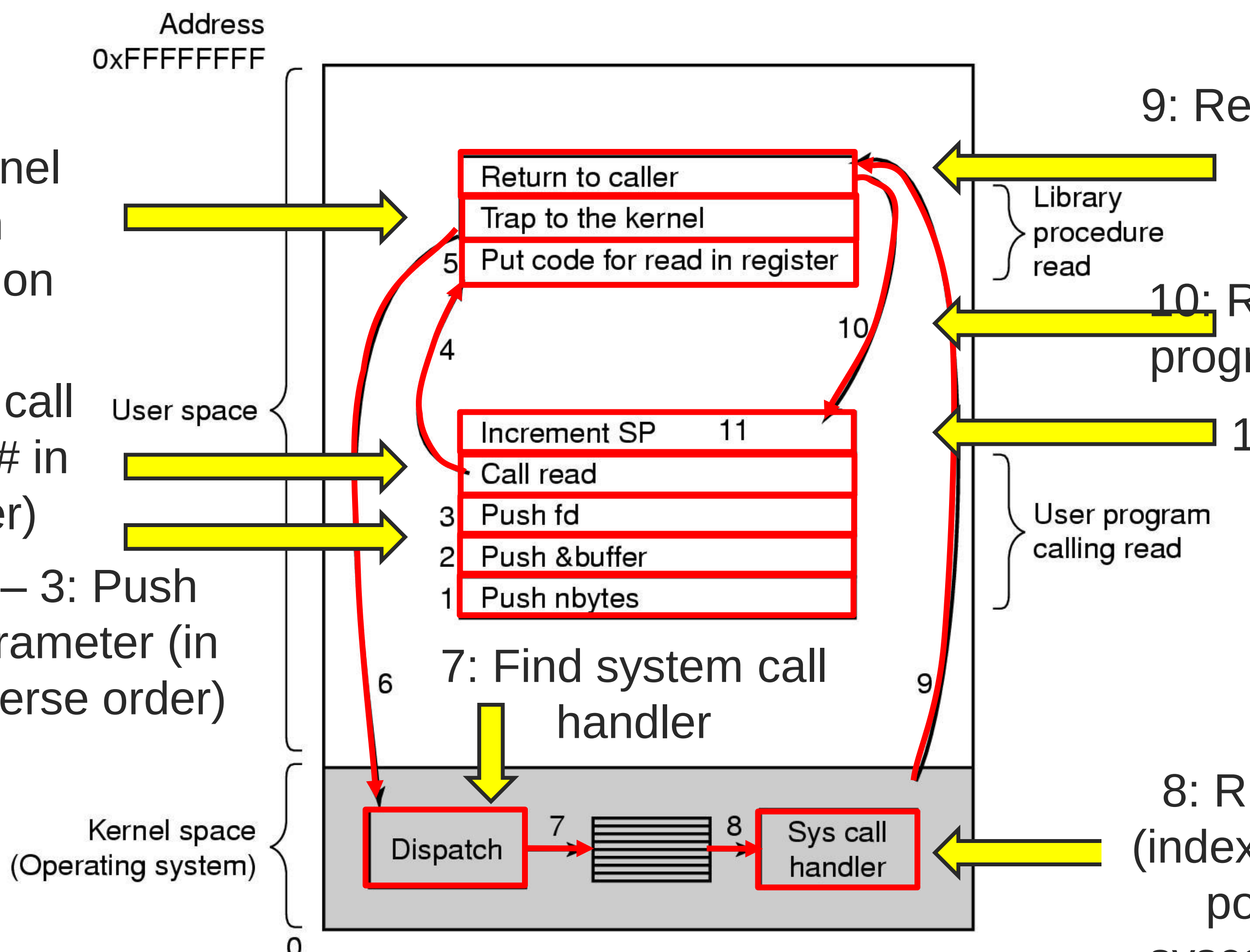
System Calls

- System Calls
  - A request to the operating system to perform some activity
- System calls are expensive
  - The system needs to perform many things before executing a system call
    - The computer (hardware) saves its state
    - The OS code takes control of the CPU, privileges are updated.
    - The OS examines the call parameters
    - The OS performs the requested function
    - The OS saves its state (and call results)
    - The OS returns control of the CPU to the caller

Copyright ©: University of Illinois CS 241 Staff

Steps for Making a System Call (Example: read call)

```
count = read(fd, buffer, nbytes);
```



6: Switch to kernel mode (return address saved on stack)

4 – 5: Library call (puts syscall # in CPU register)

1 – 3: Push parameter (in reverse order)

7: Find system call handler

9: Return to user mode

10: Return to user program (via trap)

11: Clean up

8: Run handler (index via table of pointers to syscall handles)

Examples of System Calls

- Examples
  - **`getuid()`** //get the user ID
  - **`fork()`** //create a child process
  - **`exec()`** //executing a program
- Don't mix system calls with standard library calls
  - Differences?
  - Is **`printf()`** a system call?
  - Is **`rand()`** a system call?

`man syscalls`

Major System Calls

Process Management	
`pid = fork( )`	Create a child process identical to the parent
`pid = waitpid(pid, &statloc, options)`	Wait for a child to terminate
`s = execve(name, argv, environp)`	Replace a process' core image
`exit(status)`	Terminate process execution and return status

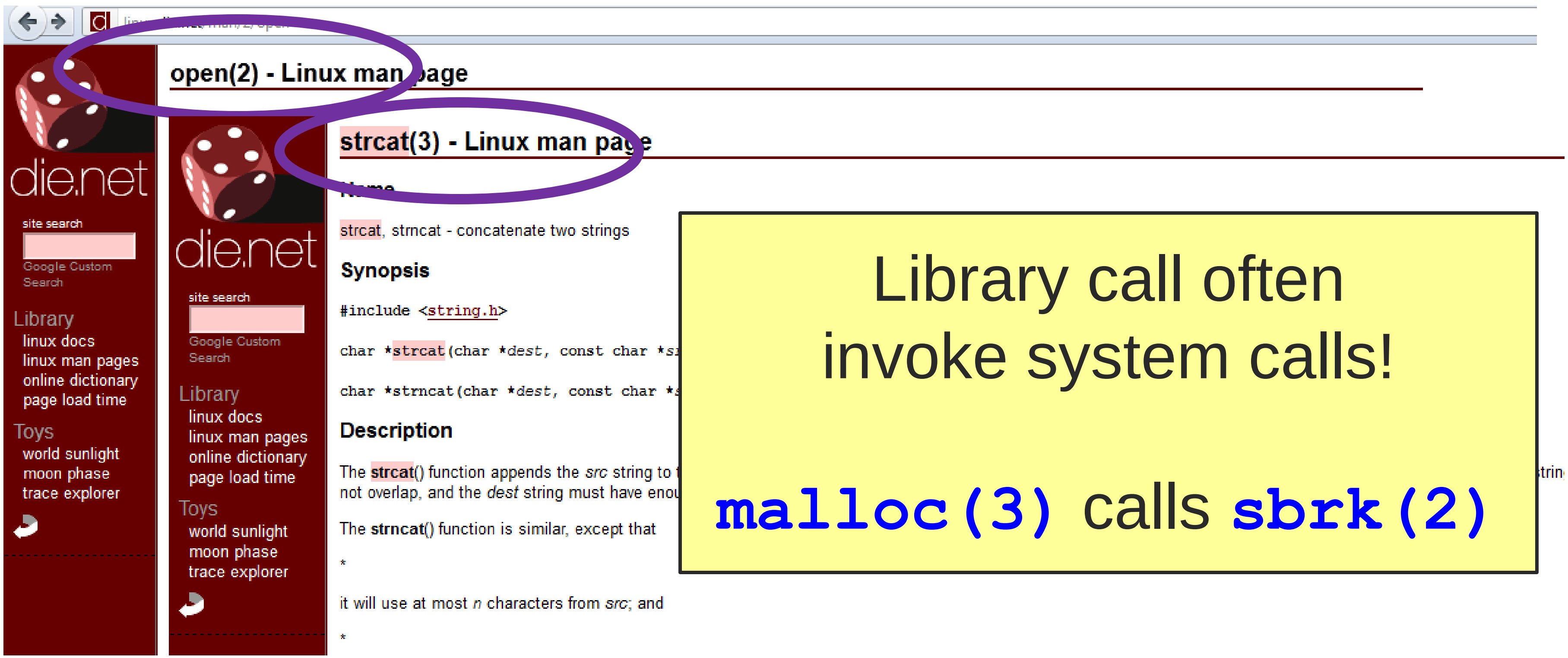
File Management	Today
`fd = open(file, how, ...)`	Open a file for reading, writing or both
`s = close(fd)`	Close an open file
`n = read(fd, buffer, nbytes)`	Read data from a file into a buffer
`n = write(fd, buffer, nbytes)`	Write data from a buffer into a file
`position = lseek(fd, offset, whence)`	Move the file pointer
`s = stat(name, &buf)`	Get a file's status information

Copyright ©: University of Illinois CS 241 Staff

Major System Calls

Directory and File System Management	
`s = mkdir(name, mode)`	Create a new directory
`s = rmdir(name)`	Remove an empty directory
`s = link(name, name)`	Create a new entry, name, pointing to name
`s = unlink(name)`	Remove a directory entry
`s = mount(special, name, flag)`	Mount a file system
`s = umount(special)`	Unmount a file system
Miscellaneous	
`s = chdir(dirname)`	Change the working directory
`s = chmod(name, mode)`	Change a file's protection bits
`s = kill(pid, signal)`	Send a signal to a process
`seconds = time(&seconds)`	Get the elapsed time since January 1, 1970

How do we know what is and what ins’t a system call?



Library call often invoke system calls!

`malloc(3)` calls **`sbrk(2)`**

- 2: System Call
- 3: Library Call

File System and I/O Related System Calls

- A file system
  - A means to organize, retrieve, and update data in persistent storage
  - A hierarchical arrangement of directories
  - Bookkeeping information (file metadata)
    - File length, # bytes, modified timestamp, etc
- Unix file system
  - Root file system starts with "/"

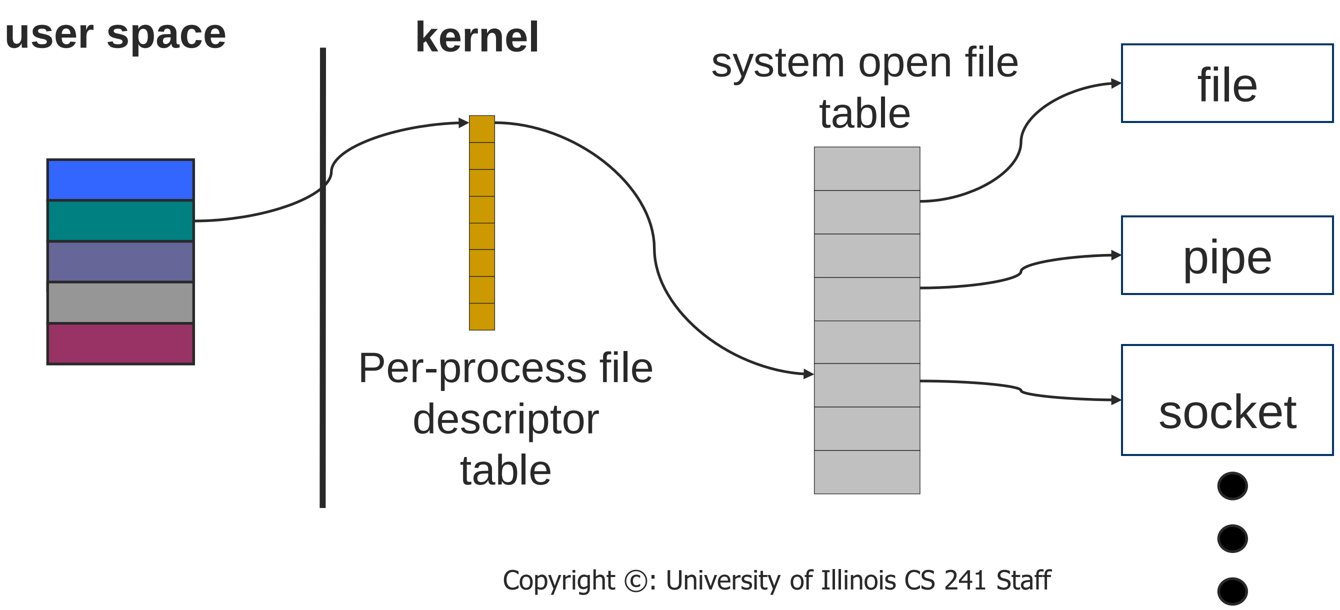
Why does the OS control I/O?

- Safety
  - The computer must try to ensure that if a program has a bug in it, then it doesn't crash or mess up
    - The system
    - Other programs that may be running at the same time or later
- Fairness
  - Make sure other programs have a fair use of device

Copyright ©: University of Illinois CS 241 Staff

Basic Unix Concepts

- Input/Output – I/O
  - Per-process table of I/O channels
  - Table entries describe files, sockets, devices, pipes, etc.
  - Table entry/index into table called “file descriptor”
  - Unifies I/O interface



Copyright ©: University of Illinois CS 241 Staff

Basic Unix Concepts

- Error Model
  - `errno` variable
    - Unix provides a globally accessible integer variable that contains an error code number
  - Return value
    - 0 on success
    - -1 on failure for functions returning integer values
    - NULL on failure for functions returning pointers
  - Examples (see `errno.h`)

```
#define EPERM     1      /* Operation not permitted */
#define ENOENT    2      /* No such file or directory */
#define ESRCH     3      /* No such process */
#define EINTR     4      /* Interrupted system call */
#define EIO       5      /* I/O error */
#define ENXIO     6      /* No such device or address */
```

System Calls for I/O

- Get information about a file

  ```
  int stat(const char* name, struct stat* buf);
  ```

- Open (and/or create) a file for reading, writing or both

  ```
  int open (const char* name, in flags);
  ```

- Read data from one buffer to file descriptor

  ```
  size_t read (int fd, void* buf, size_t cnt);
  ```

- Write data from file descriptor into buffer

  ```
  size_t write (int fd, void* buf, size_t cnt);
  ```

- Close a file

  ```
  int close(int fd);
  ```

System Calls for I/O

- They look like regular procedure calls but are different
  - A system call makes a request to the operating system by trapping into kernel mode
  - A procedure call just jumps to a procedure defined elsewhere in your program
- Some library procedure calls may themselves make a system call
  - e.g., `fopen()` calls `open()`

File: Statistics

```
#include <sys/stat.h>
int stat(const char* name, struct stat* buf);
```

- Get information about a file
- Returns:
  - 0 on success
  - -1 on error, sets `errno`
- Parameters:
  - `name`: Path to file you want to use
    - Absolute paths begin with "`/`", relative paths do not
  - `buf`: Statistics structure
    - `off_t st_size`: Size in bytes
    - `time_t st_mtime`: Date of last modification. Seconds since January 1, 1970
- Also

```
int fstat(int filedes, struct stat *buf);
```

Example - (stat())

```
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
int main(int argc, char **argv) {
    struct stat fileStat;
    if(argc != 2)
        return 1;
    if(stat(argv[1], &fileStat) < 0)
        return 1;
    printf("Information for %s\n",argv[1]);
    printf("---------------------------\n");
    printf("File Size: \t\t%d bytes\n", fileStat.st_size);
    printf("Number of Links: \t%d\n", fileStat.st_nlink);
    printf("File inode: \t\t%d\n", fileStat.st_ino);
```

Copyright ©: University of Illinois CS 241 Staff

Example - (stat())

```
    printf("File Permissions: \t");
    printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
    printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
    printf( (fileStat.st_mode & S_IRGRP) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWGRP) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXGRP) ? "x" : "-");
    printf( (fileStat.st_mode & S_IROTH) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWOTH) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n\n"); printf("The file %s a symbolic link\n",
    (S_ISLNK(fileStat.st_mode)) ? "is" : "is not");
    return 0;
}
```

Useful Macros: File types

- Is file a symbolic link
  - **`S_ISLNK`**
- Is file a regular file
  - **`S_ISREG`**
- Is file a character device
  - **`S_ISCHR`**
- Is file a block device
  - **`S_ISBLK`**
- Is file a FIFO
  - **`S_ISFIFO`**
- Is file a unix socket
  - **`S_ISSOCK`**

Copyright ©: University of Illinois CS 241 Staff

Useful Macros: File Modes

- **S_IRWXU**
  - read, write, execute/search by owner
- **S_IRUSR**
  - read permission, owner
- **S_IWUSR**
  - write permission, owner
- **S_IXUSR**
  - execute/search permission, owner
- **S_IRGRP**
  - read permission, group
- **S_IRWXO**
  - read, write, execute/search by others

Example - (`stat()`)

```
Information for testfile.sh
---------------------------
File Size: 36 bytes
Number of Links: 1
File inode: 180055
File Permissions: -rwxr-xr-x

The file is not a symbolic link
```

File: Open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char* path, int flags [, int mode ]);
```

- Open (and/or create) a file for reading, writing or both
- Returns:
  - Return value ≥ 0 : Success - New file descriptor on success
  - Return value = -1: Error, check value of `errno`
- Parameters:
  - `path`: Path to file you want to use
    - Absolute paths begin with "`/`", relative paths do not
  - `flags`: How you would like to use the file
    - `O_RDONLY`: read only, `O_WRONLY`: write only, `O_RDWR`: read and write, `O_CREAT`: create file if it doesn't exist, `O_EXCL`: prevent creation if it already exists

Example (`open()`)

```
#include <fcntl.h>
#include <errno.h>
extern int errno;

main() {
    int fd;
    fd = open("foo.txt", O RDONLY | O CREAT);
    printf("%d\n", fd);
    if (fd=-1) {
        printf ("Error Number %d\n", errno);
        perror("Program");
    }
}
```

Argument: string
Output: the string, a colon, and a description of the error condition stored in **errno**

File: Close

```
#include <fcntl.h>
int close(int fd);
```

- Close a file
  - Tells the operating system you are done with a file descriptor
- Return:
  - 0 on success
  - -1 on error, sets `errno`
- Parameters:
  - `fd`: file descriptor

Example (`close()`)

```
#include <fcntl.h>
main(){
    int fd1;

    if(( fd1 = open(“foo.txt", O_RDONLY)) < 0){
        perror("c1");
        exit(1);
    }
    if (close(fd1) < 0) {
        perror("c1");
        exit(1);
    }
    printf("closed the fd.\n");
```

Example (`close()`)

```
#include <fcntl.h>
main(){
    int fd1;

    if(( fd1 = open(“foo.txt", O_RDONLY)) < 0){
        perror("c1");
        exit(1);
    }
    if (close(fd1) < 0) {
        perror("c1");
        exit(1);
    }
    printf("closed the fd.\n");
```

After close, can you still use the file descriptor?

Why do we need to close a file?

Copyright ©: University of Illinois CS 241 Staff

File: Read

```
#include <fcntl.h>
size_t read (int fd, void* buf, size_t cnt);
```

- Read data from one buffer to file descriptor
  - Read **`size`** bytes from the file specified by **`fd`** into the memory location pointed to by **`buf`**
- Return: How many bytes were actually read
  - Number of bytes read on success
  - 0 on reaching end of file
  - -1 on error, sets **`errno`**
  - -1 on signal interrupt, sets **`errno`** to **`EINTR`**
- Parameters:
  - **`fd`**: file descriptor
  - **`buf`**: buffer to read data from
  - **`cnt`**: length of buffer

Copyright ©: University of Illinois CS 241 Staff

File: Read

`size_t read (int fd, void* buf, size_t cnt);`

- Things to be careful about
  - **`buf`** needs to point to a valid memory location with length not smaller than the specified size
    - Otherwise, what could happen?
  - **`fd`** should be a valid file descriptor returned from **`open()`** to perform read operation
    - Otherwise, what could happen?
  - **`cnt`** is the requested number of bytes read, while the return value is the actual number of bytes read
    - How could this happen?

Copyright ©: University of Illinois CS 241 Staff

Example (**read()**)

```
#include <fcntl.h>
main() {
    char *c;
    int fd, sz;

    c = (char *) malloc(100
                * sizeof(char));
    fd = open("foo.txt",
                O_RDONLY);
    if (fd < 0) {
        perror("r1");
        exit(1);
    }
    sz = read(fd, c, 10);
    printf("called
        read(%d, c, 10).
        returned that %d
        bytes  were
        read.\n", fd, sz);
    c[sz] = '\0';

    printf("Those bytes
        are as follows:
        %s\n", c);
    close(fd);
}
```

File: Write

`#include <fcntl.h>`

`size_t write (int fd, void* buf, size_t cnt);`

- Write data from file descriptor into buffer
  - Writes the bytes stored in **`buf`** to the file specified by **`fd`**
- Return: How many bytes were actually written
  - Number of bytes written on success
  - 0 on reaching end of file
  - -1 on error, sets **`errno`**
  - -1 on signal interrupt, sets **`errno`** to **`EINTR`**
- Parameters:
  - **`fd`**: file descriptor
  - **`buf`**: buffer to write data to
  - **`cnt`**: length of buffer

File: Write

`size_t write (int fd, void* buf, size_t cnt);`

- Things to be careful about
  - The file needs to be opened for write operations
  - **`buf`** needs to be at least as long as specified by **`cnt`**
    - If not, what will happen?
  - **`cnt`** is the requested number of bytes to write, while the return value is the actual number of bytes written
    - How could this happen?

Example (write())

```c
#include <fcntl.h>
main()
{
    int fd, sz;

    fd = open("out3",
        O_RDWR | O_CREAT |
        O_APPEND, 0644);
    if (fd < 0) {
        perror("r1");
        exit(1);
    }

    sz = write(fd, "cs241\n",
        strlen("cs241\n"));

    printf("called write(%d,
        \"cs360\\n\", %d).
        it returned %d\n",
        fd, strlen("cs360\n"),
        sz);

    close(fd);
}
```

File Pointers

- All open files have a "file pointer" associated with them to record the current position for the next file operation
- On open
  - File pointer points to the beginning of the file
- After reading/write m bytes
  - File pointer moves m bytes forward

Copyright ©: University of Illinois CS 241 Staff

File: Seek

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

- Explicitly set the file offset for the open file
- Return: Where the file pointer is
  - the new offset, in bytes, from the beginning of the file
  - -1 on error, sets `errno`, file pointer remains unchanged
- Parameters:
  - `fd`: file descriptor
  - `offset`: indicates relative or absolute location
  - `whence`: How you would like to use `lseek`
    - `SEEK_SET`, set file pointer to `offset` bytes from the beginning of the file
    - `SEEK_CUR`, set file pointer to `offset` bytes from current location
    - `SEEK_END`, set file pointer to `offset` bytes from the end of the file

File: Seek Examples

- Random access
  - Jump to any byte in a file
- Move to byte #16

```
newpos = lseek(fd, 16, SEEK_SET);
```

- Move forward 4 bytes

```
newpos = lseek(fd, 4, SEEK_CUR);
```

- Move to 8 bytes from the end

```
newpos = lseek(fd, -8, SEEK_END);
```

Copyright ©: University of Illinois CS 241 Staff

Example (`lseek()`)

```
c = (char *) malloc(100 *
    sizeof(char));
fd = open("foo.txt", O_RDONLY);
if (fd < 0) {
    perror("r1");
    exit(1);
}

sz = read(fd, c, 10);
printf("We have opened in1, and
    called read(%d, c, 10).\n",
    fd);
c[sz] = '\0';
printf("Those bytes are as
    follows: %s\n", c);
```

```
i = lseek(fd, 0, SEEK_CUR);
printf("lseek(%d, 0, SEEK_CUR)
    returns that the current
    offset is %d\n\n", fd, i);

printf("now, we seek to the
    beginning of the file and
    call read(%d, c, 10)\n",
    fd);
lseek(fd, 0, SEEK_SET);
sz = read(fd, c, 10);
c[sz] = '\0';
printf("The read returns the
    following bytes: %s\n", c);
…
```

Standard Input, Standard Output and Standard Error

- Every process in Unix has three predefined file descriptors
  - File descriptor 0 is standard input (**`STDIN`**)
  - File descriptor 1 is standard output (**`STDOUT`**)
  - File descriptor 2 is standard error (**`STDERR`**)
- Read from standard input,
  - **`read(0, ...);`**
- Write to standard output
  - **`write(1, ...);`**
- Two additional library functions
  - **`printf();`**
  - **`scanf();`**

Copyright ©: University of Illinois CS 241 Staff

I/O Library Calls

- Every system call has paired procedure calls from the standard I/O library:

- System Call
  - **`open`**
  - **`close`**
  - **`read/write`**
  - **`lseek`**

- Standard I/O call (**`stdio.h`**)
  - **`fopen`**
  - **`fclose`**
  - **`getchar/putchar, getc/putc, fgetc/fputc, fread/fwrite, gets/puts, fgets/fputs, scanf/printf, fscanf/fprintf`**
  - **`fseek`**