csci3411: Operating Systems

Lecture 3: **System structure and Processes**

Gabriel Parmer

Some slide material from Silberschatz and West

System Structure

- *System Structure* – How different parts of software

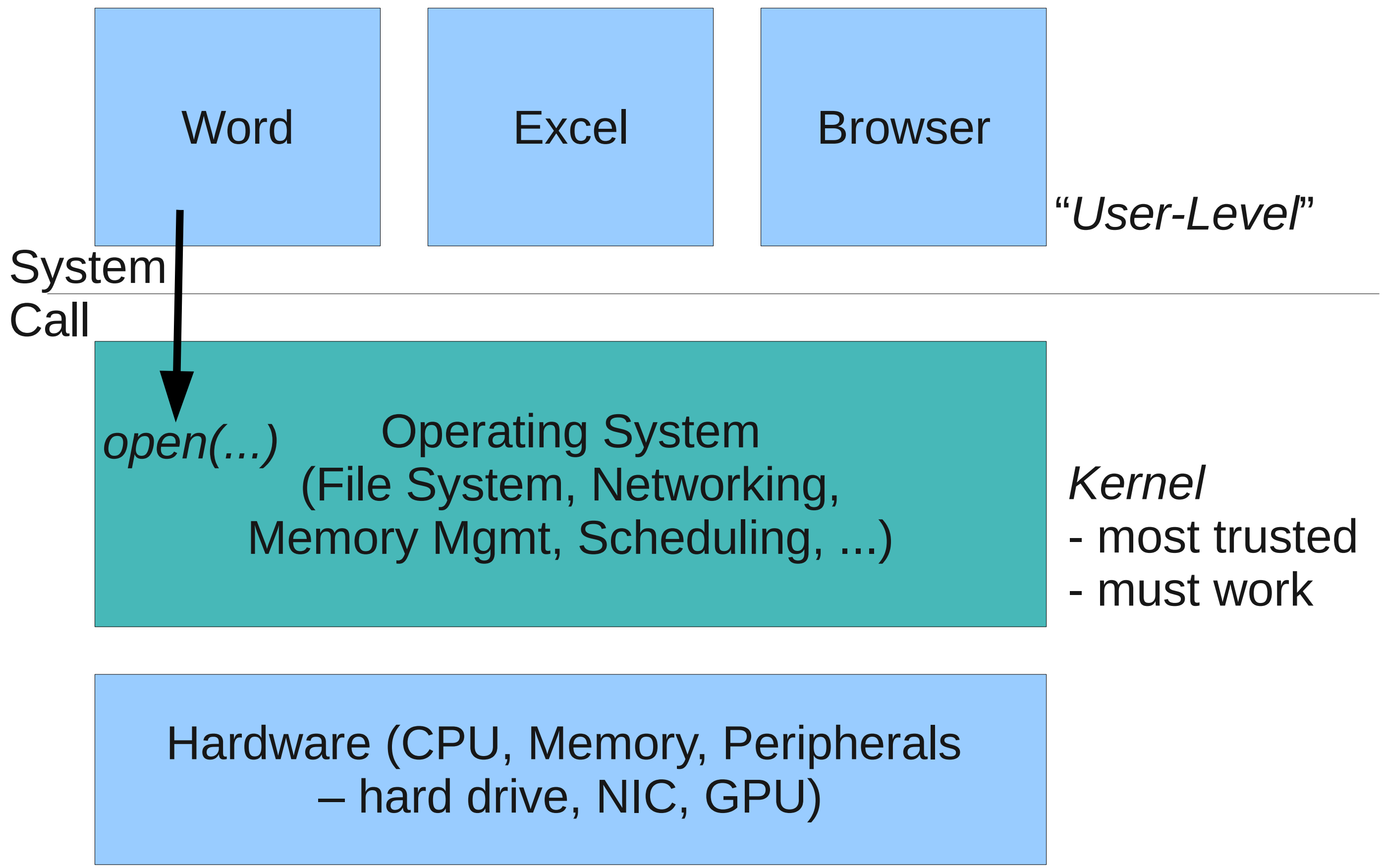
    1) Are separated from each other (*Why?)*

    2) Communicate

- How does a system use
    - dual mode
    - v*irtual address spaces*
- Implications on
    - Security/Reliability
    - Programming style/Maintainability

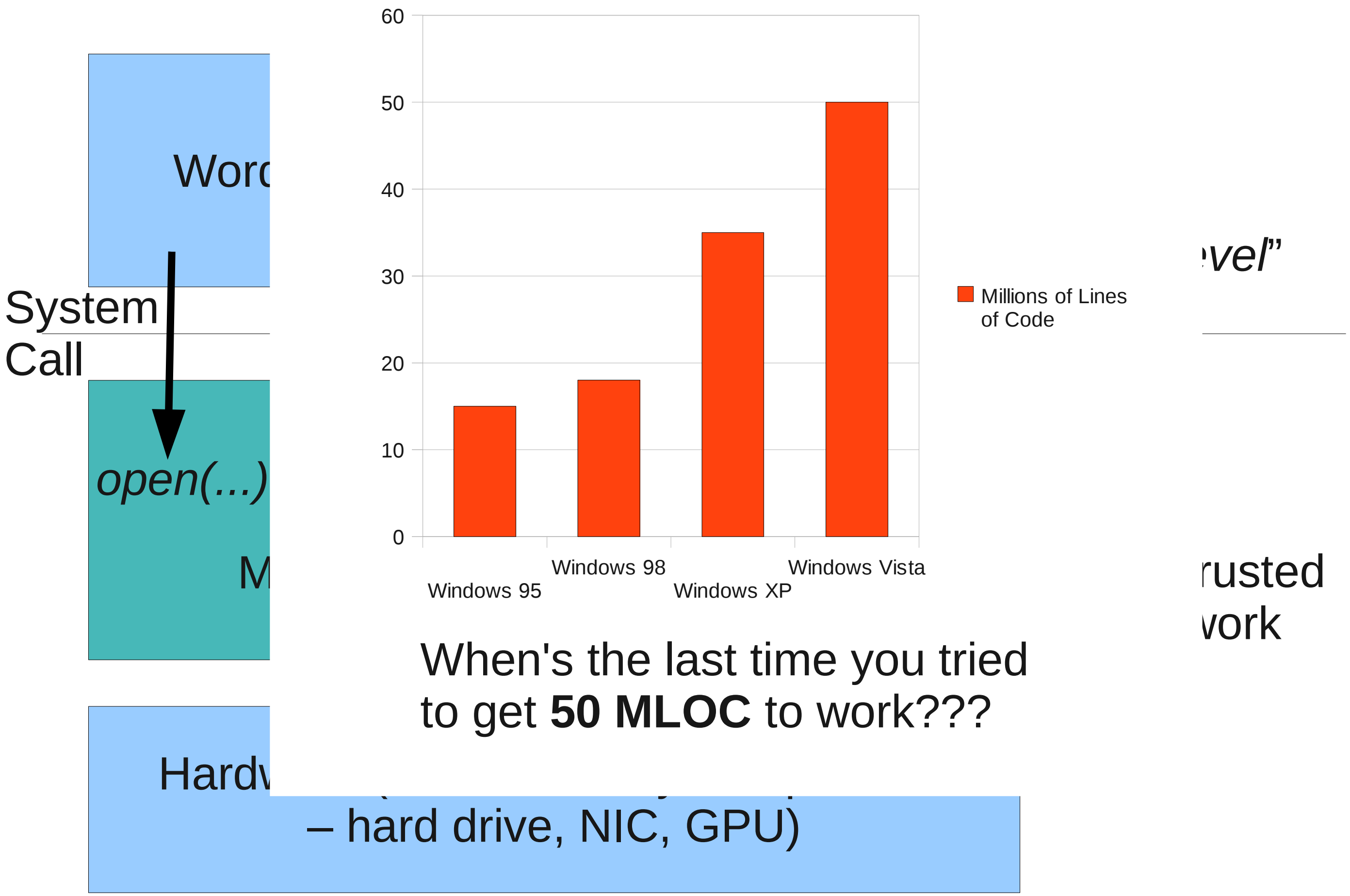
Monolithic System Structure

- Includes Unix/Windows/OSX



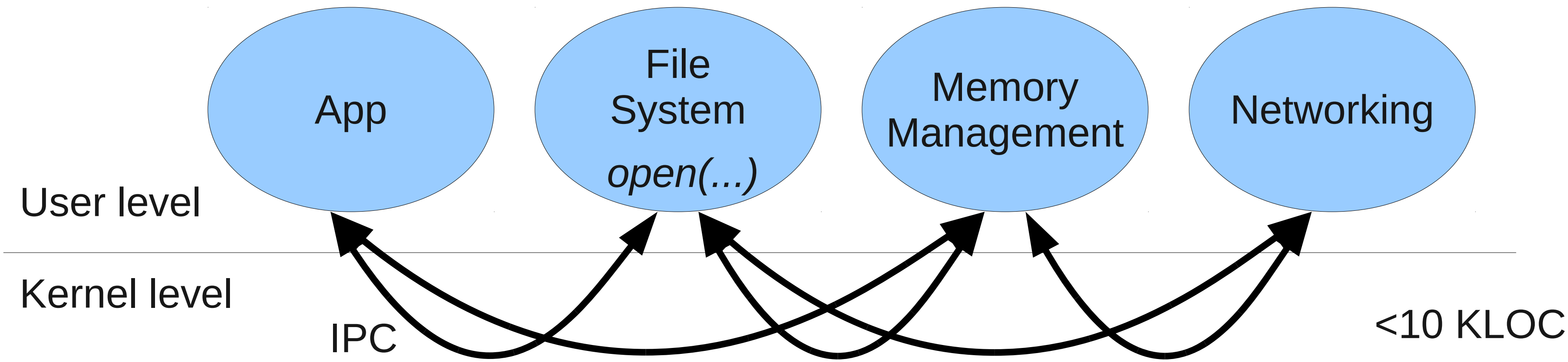
Monolithic System Structure

- Includes Unix/Windows/OSX



When's the last time you tried to get **50 MLOC** to work???

Microkernel System Structure



- Moves functionality from the kernel to "*user*" space
- Communication takes place between user *servers* using inter-process communication (IPC)
- Benefits:
  - Easier to add functionality
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments: performance! (why?)

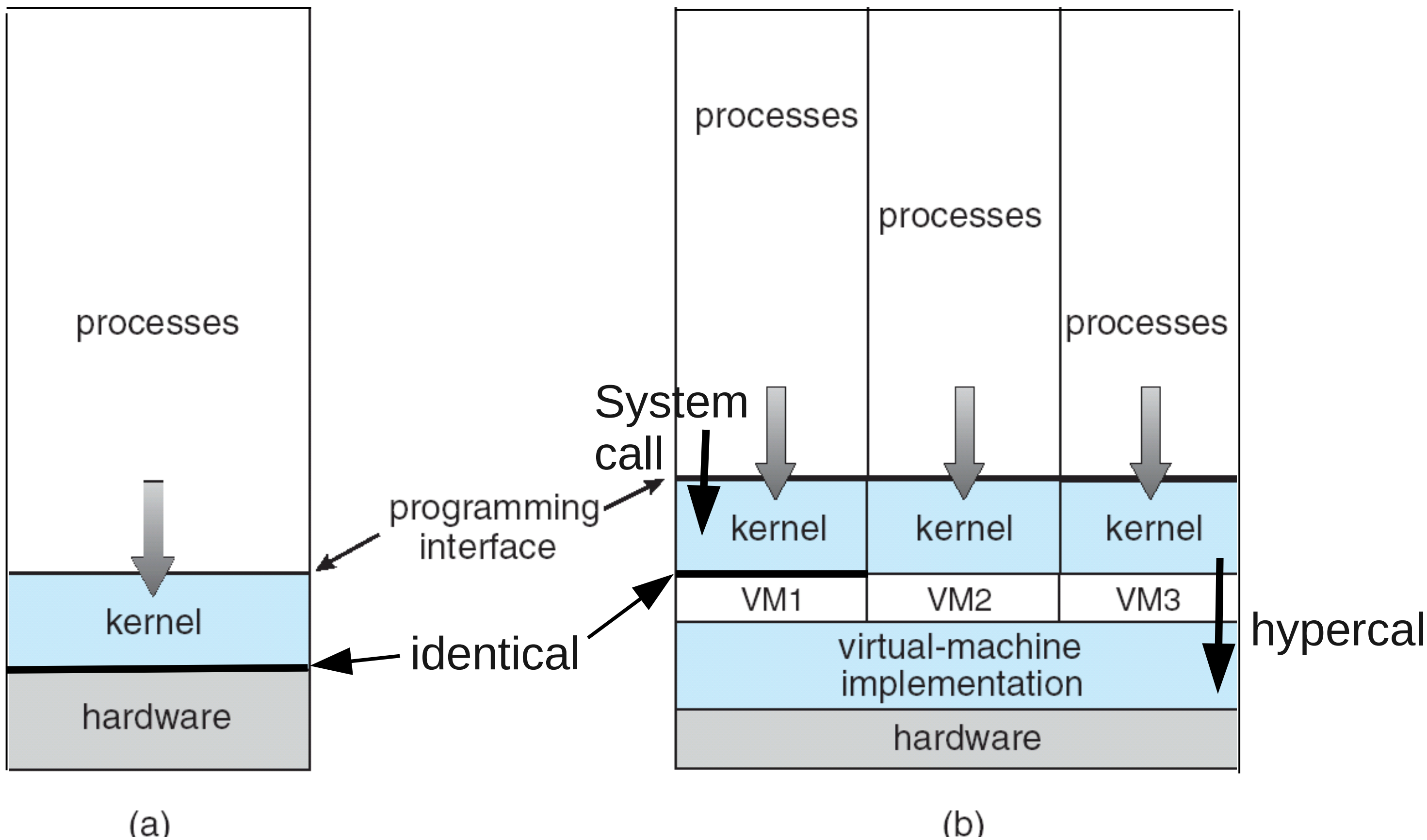
Virtual Machines I

- Do you know what these are?
- What is the structure of VMs?

Virtual Machines II

- A virtual machine *host* (the kernel) provides an interface *identical* to the underlying bare hardware
  - Other *guest* kernels execute in user-mode
  - The API for virtual machines is a copy of the machine!

Virtual Machines III



(a) non-virtual machine (b) virtual machine

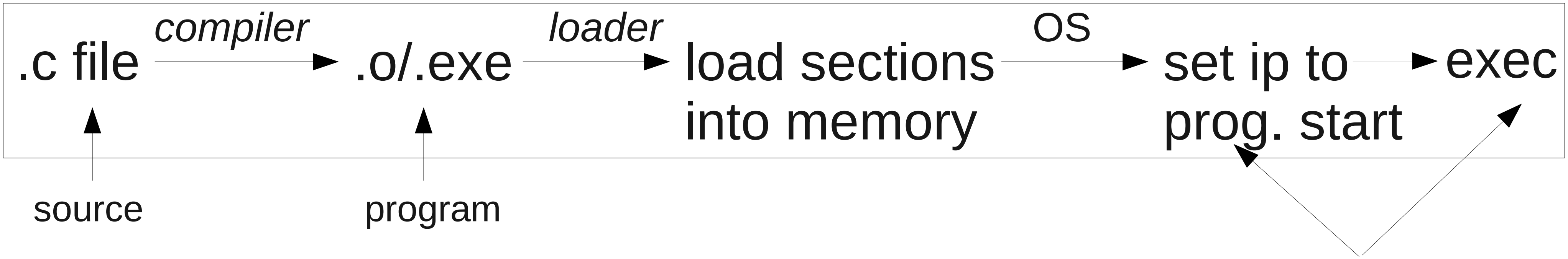
Virtual Machine: Benefits

- Fundamentally, multiple operating systems share the same hardware
- Protected from each other
- Some sharing of files
- Communicate with each other via networking
- Useful for development, testing
- *Consolidation* of many low-resource use systems onto fewer busier systems

CPU/Memory Abstraction

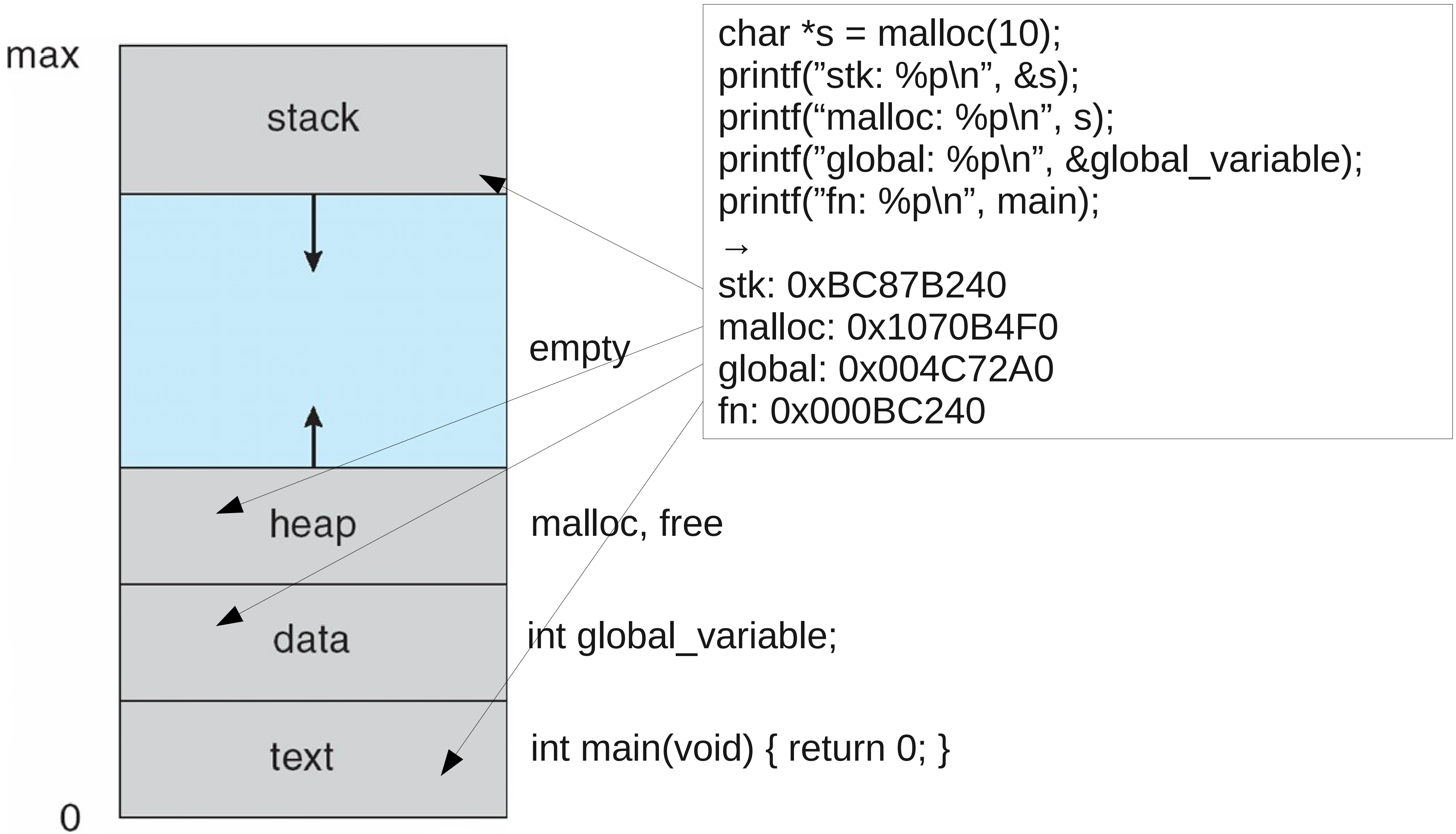
- Hardware provides
  - Sequential execution
  - Interrupts
- OS should provide
  - Multiple flows of sequential execution (diff apps)
  - Each app should have its own memory “space”
  - Protection between these applications
    - Security
    - Fault isolation

Processes

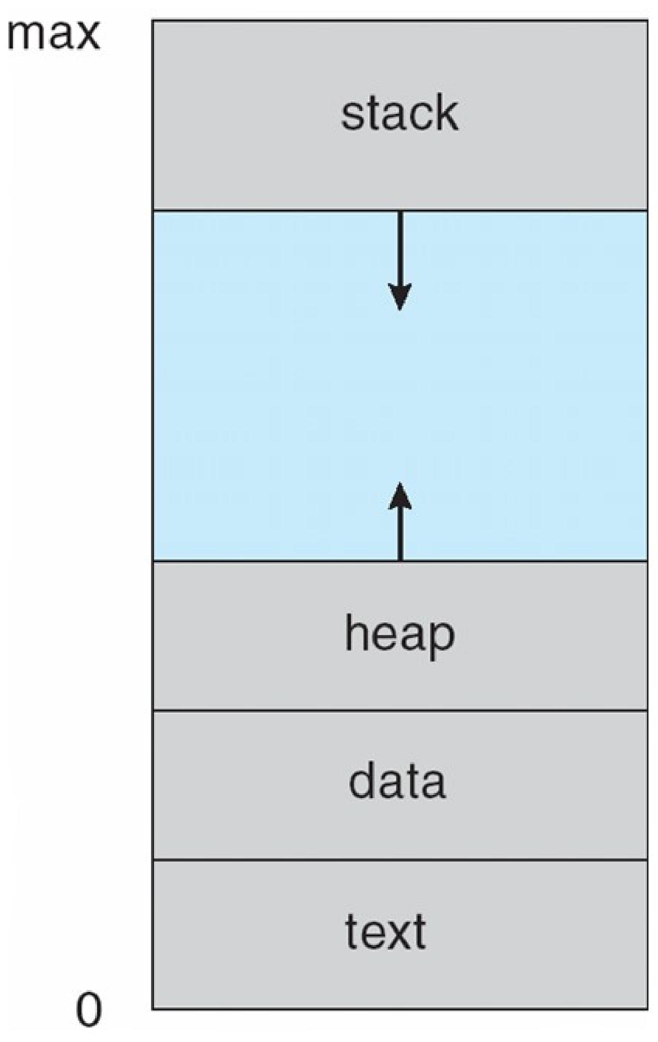


- An executable program (seen in *ls*)
  - passive collection of code and data; kept in file
- UNIX Process: active entity that includes (seen in *ps*)
  - Registers (instruction counter, stack pointer, etc..)
  - Execution stack
  - Heap
  - Data and text (code) segments

Process in Memory

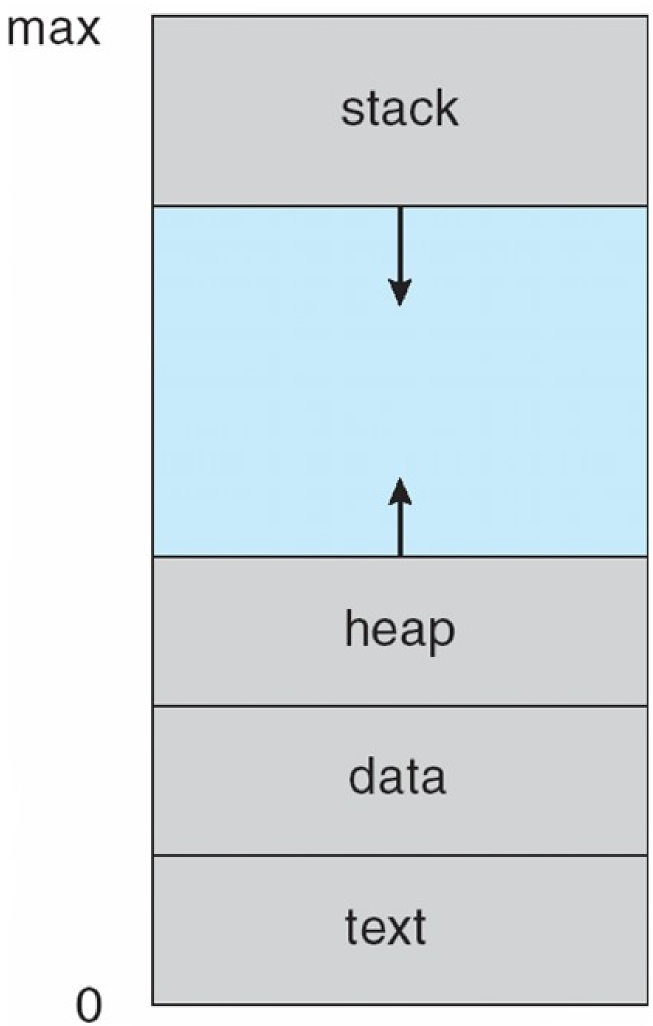


OS Support for Process Memory



- OS uses HW to provide virtual address space (VAS)
  - Each process thinks it has all memory
    - OS abstraction!!!
  - Provides protection between processes
  - Only subset of that address space is populated by actual memory

OS Support for Process Memory II



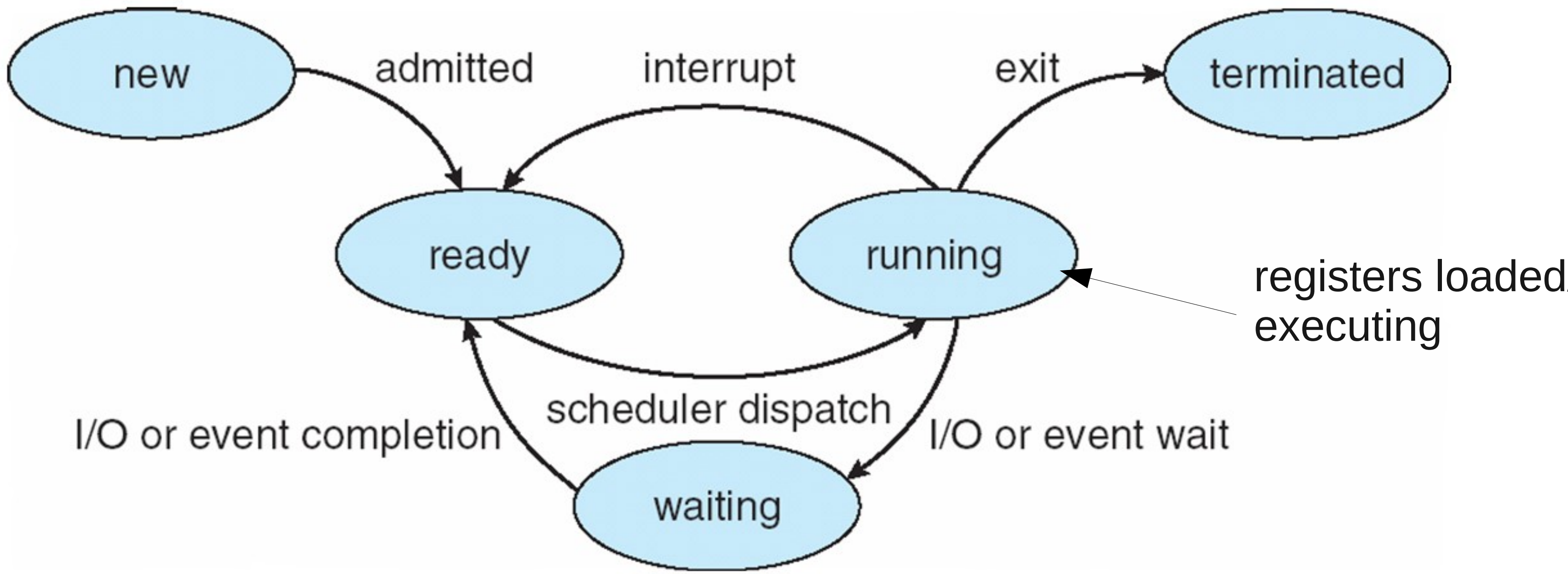
- Kernel must manage virtual address spaces
  - Create mapping between virtual and actual memory
  - Switch between apps == switch between VAS
    - Only mode 0 can switch VAS!

Process Control Block (PCB)

- Kernel, per-process, data-structure includes:
  - CPU registers (including instruction counter)
  - Scheduling state (priority)
  - Memory management information (amount of memory allocated, virtual address space mapping, stack location)
  - CPU accounting info (exec time at user/kernel level)
  - File info (open files)
  - Process state

Process States

- As process executes, the kernel changes its state

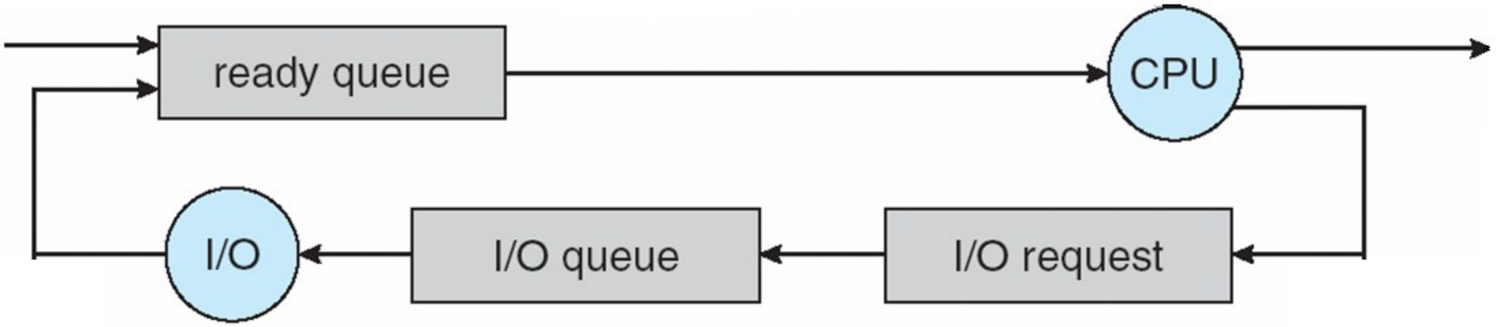


- Many processes in system
    - If one is in *running*, what states are the others in?
    - Give an example of why a process would go from *running* → *waiting*
    - Why would *running* + interrupt → waiting

Process Queues

- Process/Job queue – all processes in system
- Scheduling runqueue – procs in *ready* state
  - Waiting to execute
  - Scheduler chooses next process to run
- Device queues – processes waiting for I/O completion (interrupts)
  - Typically one queue per device
- Processes migrate between queues

Process Migration between Queues



Process Scheduling

- Choose which process to *dispatch* next given
  - Process priority (compared to other ready/runnable processes)
  - Remaining process timeslice (CPU allocation)
- Two general types of processes
  1) CPU bound: most time on CPU, not waiting for I/O
  2) I/O bound: short bursts of CPU usage, most time spent waiting on I/O
- What keeps a single CPU-bound process from monopolizing the CPU?

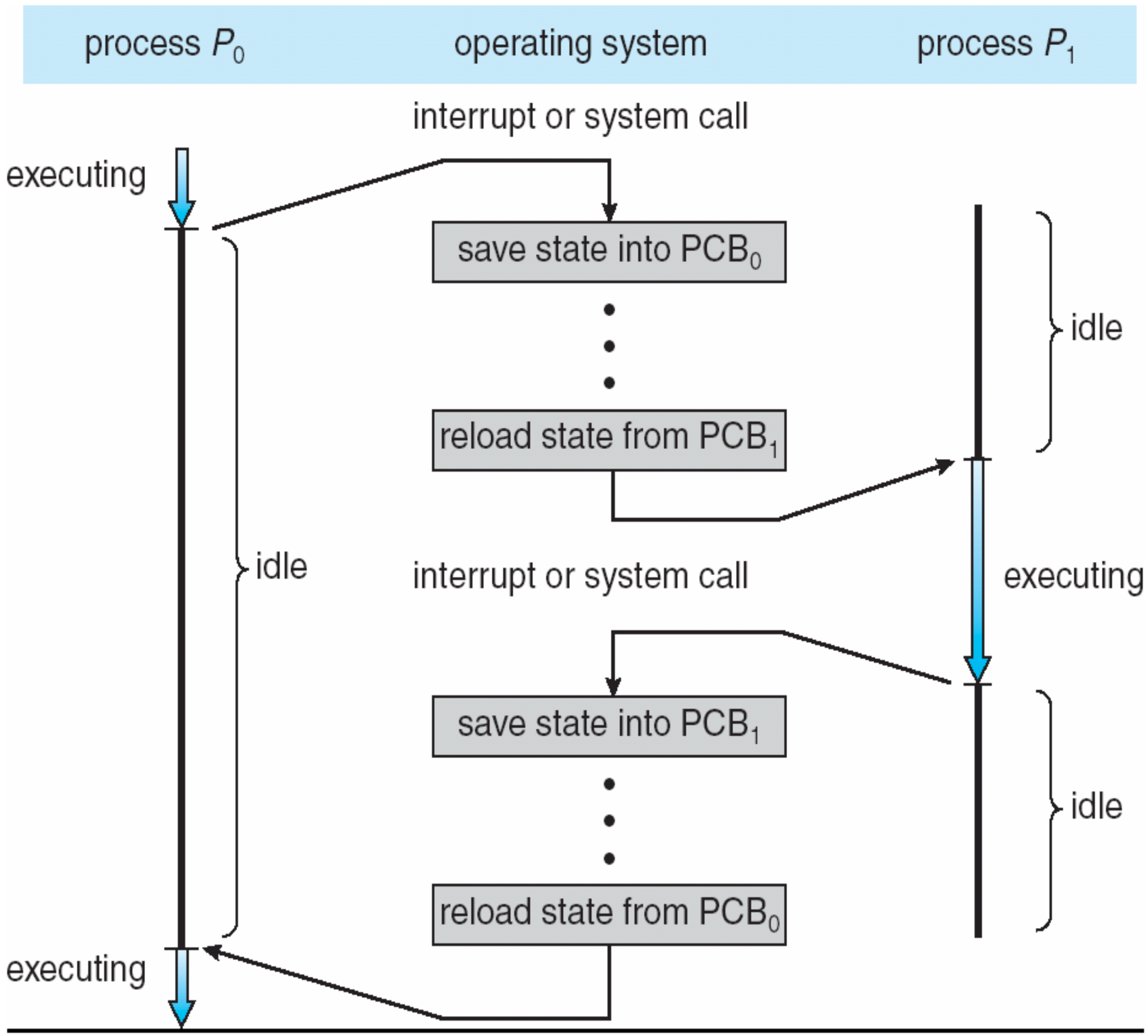
Timer Interrupts

- Interrupt from on-processor time keeping device
  - e.g. 100 times a second in Linux, every 10 milliseconds
- Allows kernel to “keep time”
  - Track amount of execution of different processes
  - Schedule accordingly
- Process' *timeslice* typically a multiple of a timer interrupt's inter-arrival time

Single CPU → Many Processes

- Scheduler decides which process to run next
- *Dispatcher* actually switches from the current process, to the next (chosen by the scheduler)
  - Ready state → running state
- *Context switch* time is overhead; should be minimal
- What is involved in a context switch? What needs to be saved and restored?

Single CPU → Many Processes II



Context Switch Implementation

```
struct thread *current, *next;
switch_regs(current, next)
```

```
switch_regs:
	/* save first thread's registers */
	mov %a, current->regs.a
	…
	mov %sp, current->regs.sp
	mov post_switch, current->regs.ip

	/* load next thread's registers! */
	mov next->regs.a, %a
	…
	mov next->regs.sp, %sp
	jmp next->regs.ip
post_switch:
	ret
```

%a is the first register
%sp is the stack pointer

Process Operations

- Creation (fork)
- Termination (exit)
- Coordination (wait)

Process Creation: fork()

- *Parent* process may fork() a *child* process
- Parent may share system resources with child
  - Open files
- Parent and child execute concurrently
- Parent can wait() for children to finish execution
- Parent can kill() its children

- Process hierarchy
  - Which is the first process? Where does a “shell” fit in?

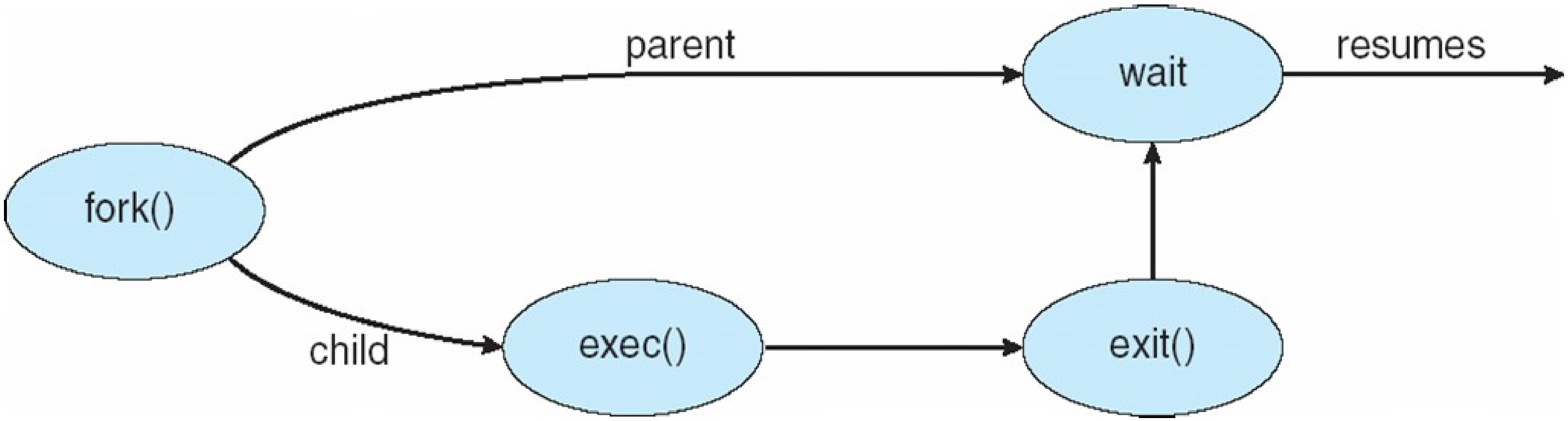
Process Creation: fork() II

- fork() creates a copy of the parent's address space for the child
  - Copying all memory can be expensive!
- Often intention is to *exec*ute new program
  - exec() or execve() system calls load program from disk into current process
- So why copy all memory?
  - COW – copy on write memory sharing
  - vfork() – stop parent's execution till we exec()

Process Termination: exit()

- Release current process' resources back to the system, discontinue execution
- Takes argument: status/return value
    - Same as returning integer from main function
- Process might stick around with status/return value until parent wait()'s
    - wait() returns the status of the child process
    - “zombie” process – new process state

fork/join style (or fork/wait)



C Example of Fork Usage

```
int main()
{
     pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process: execute “ls” */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
            int status;
            /* parent will wait for the child to complete */
            wait(&status); /* or wait_pid(pid, &status, 0) */
            printf ("Child Complete");
            exit(0);
    }
    return 0;
}
```