UNIVERSITY OF CALIFORNIA UCRIVERSIDE

Monolithic Kernels, Sleeping Beauties, and Processes

CS202, Advanced Operating Systems

(Some slides from Hung-Wei Tseng, and Heng Yin)

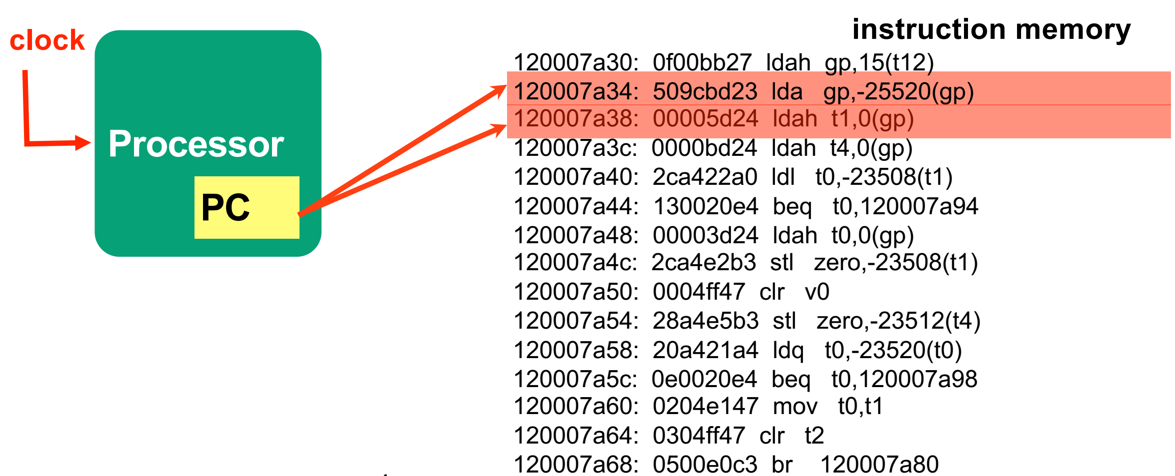
The goal of an OS

Recap: What modern operating systems support?

- **Virtualize** hardware/architectural resources
  - Easy for programs to interact with hardware resources
  - Share hardware resource among programs
  - Protect programs from each other (security)
- Execute multithreaded programs **concurrently**
  - Support multithreaded programming model
  - Execute multithreaded programs efficiently
- Store data **persistently**
  - Store data safely
  - Secure

Recap: How processor executes a program

- The program counter (PC) tells where the upcoming instruction is in the memory
- Processor fetches the instruction, decode the instruction, execute the instruction, present the instruction results according to clock signals
- The processor fetches the next instruction whenever it’s safe to do so



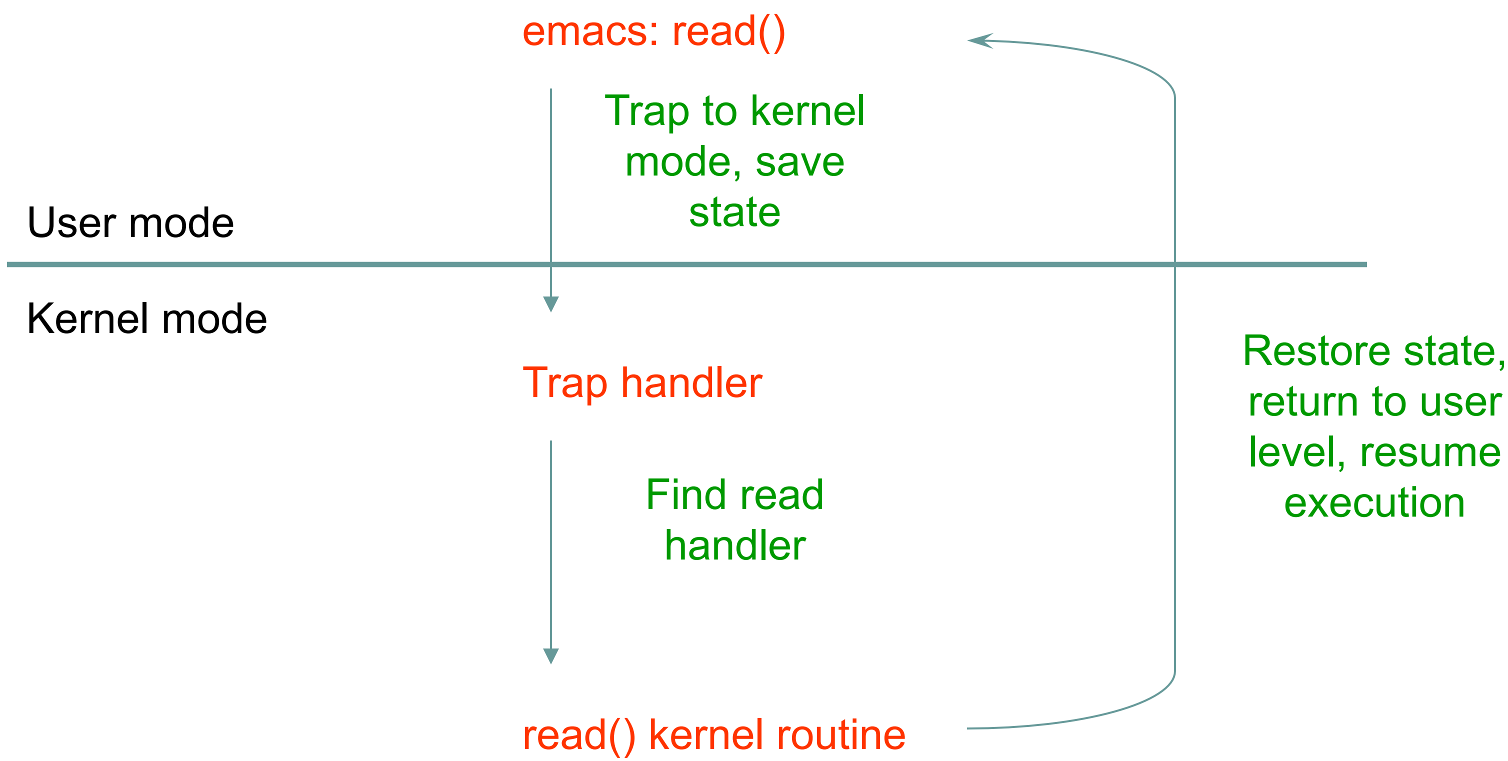
Monolithic Kernel/Sleeping Beauty

- Sleeping beauty: Direct Controlled Execution
  - Program running directly on hardware
- But I thought that is insecure?
  - Yes! We hide anything dangerous in the OS
  - Program has to ask for permission
    - System calls
- OS is an event handler
  - Any event occurs, hardware securely traps to OS
  - OS figures out who woke it up and handles the situation

System Calls

- For a user program to do something "privileged" (e.g., I/O) it must call an OS procedure
  - Known as crossing the protection boundary, or a protected procedure call
- Hardware provides a system call instruction that:
  - Causes an exception, which invokes a kernel handler
    - Passes a parameter determining the system routine to call
  - Saves caller state (PC, regs, mode) so it can be restored
    - Why save mode?
  - Returning from system call restores this state

System Call


emacs: read()
Trap to kernel mode, save state
User mode
Kernel mode
Trap handler
Find read handler
read() kernel routine
Restore state, return to user level, resume execution

Categorizing Events

	Unexpected	Deliberate
Synchronous	fault	syscall trap
Asynchronous	interrupt	software interrupt

- Interrupts signal asynchronous events
  - I/O hardware interrupts
  - Software and hardware timers

Timer

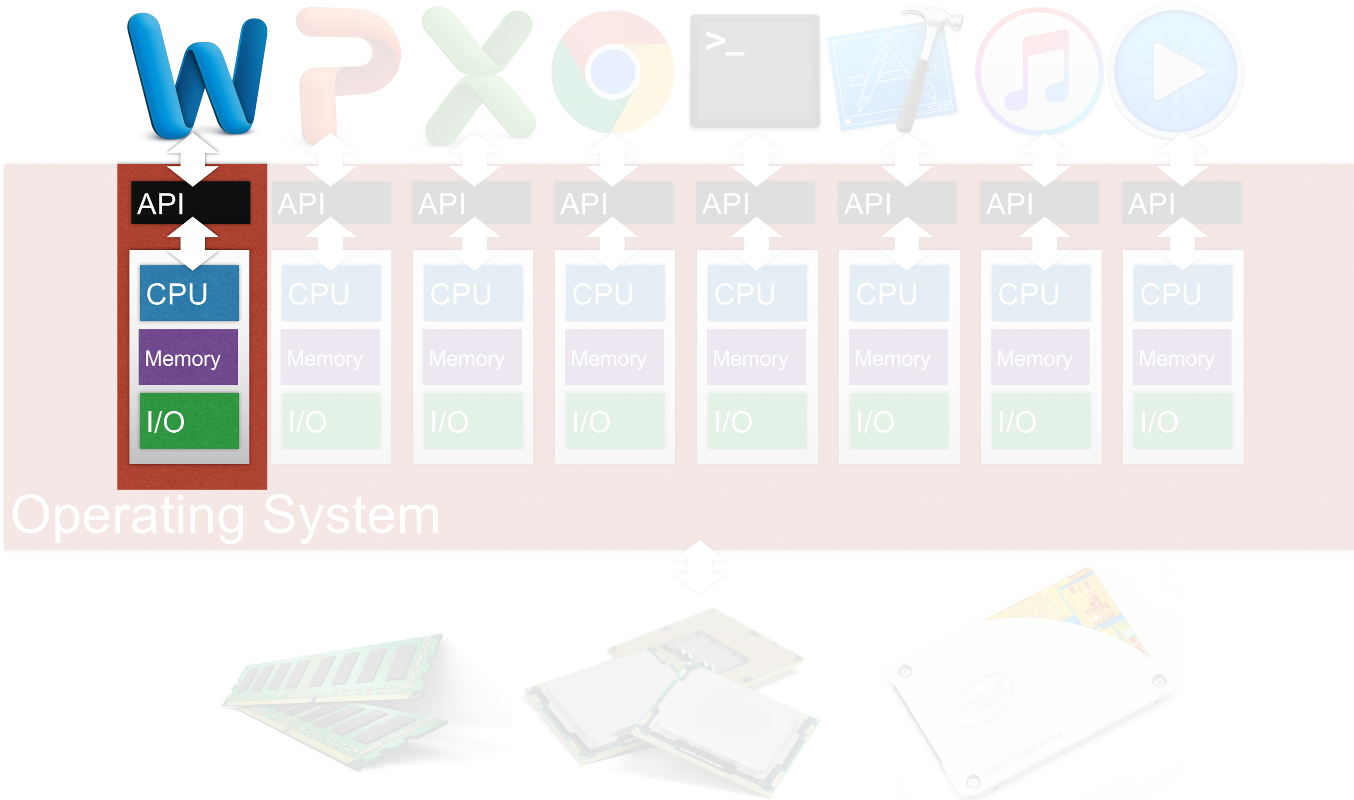
- The key to a timesharing OS

- The fallback mechanism by which the OS reclaims control
  - Timer is set to generate an interrupt after a period of time
    - Setting timer is a privileged instruction
    - When timer expires, generates an interrupt
      - Handled by the OS, forcing a switch from the user program
    - Basis for OS scheduler *(more later…)*

- Also used for time-based functions (e.g., *sleep()*)

The goal of an OS

The idea of an OS: virtualization



The idea: virtualization

- The operating system presents an illusion of a virtual machine to each running program and maintains architectural states of a von Neumann machine
  - Processor
  - Memory
  - I/O
- Each virtualized environment accesses architectural facilities through some sort of application programming interface (API)
- Dynamically map those virtualized resources into physical resources

Demo, Virtualization

```
double a;

int main(int argc, char *argv[])
{
    int cpu, status, i;
    int *address_from_malloc;
    cpu_set_t my_set;        // Define your cpu_set bit mask.
    CPU_ZERO(&my_set);       // Initialize it all to 0, i.e. no CPUs selected.
    CPU_SET(4, &my_set);     // set the bit that represents core 7.
    sched_setaffinity(0, sizeof(cpu_set_t), &my_set); // Set affinity of this process to the defined mask, i.e. only 7.
    status = syscall(SYS_getcpu, &cpu, NULL, NULL);
                                                  getcpu system call to retrieve the executing CPU ID
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s process_nickname\n",argv[0]);
        exit(1);
    }

    srand((int)time(NULL)+(int)getpid());
    a = rand();                          create a random number

    fprintf(stderr, "\nProcess %s is using CPU: %d. Value of a is %lf and address of a is %p\n",argv[1], cpu, a, &a);
    sleep(1);
                                                  print the value of a and address of a

    fprintf(stderr, "\nProcess %s is using CPU: %d. Value of a is %lf and address of a is %p\n",argv[1], cpu, a, &a);
    sleep(3);

                                                  print the value of a and address of a again after sleep
    return 0;
}
```

Virtualization Demo

```
Process C is using CPU: 4. Value of a is 685161796.000000 and address of a is 0x6010b0
Process A is using CPU: 4. Value of a is 217757257.000000 and address of a is 0x6010b0
Process B is using CPU: 4. Value of a is 2057721479.000000 and address of a is 0x6010b0
Process D is using CPU: 4. Value of a is 1457934803.000000 and address of a is 0x6010b0
Process C is using CPU: 4. Value of a is 685161796.000000 and address of a is 0x6010b0
Process A is using CPU: 4. Value of a is 217757257.000000 and address of a is 0x6010b0
Process B is using CPU: 4. Value of a is 2057721479.000000 and address of a is 0x6010b0
Process D is using CPU: 4. Value of a is 1457934803.000000 and address of a is 0x6010b0
```

Different values

The same processor!

Different values are preserved

The same memory address!

Why virtualization

- How many of the following statement is true about why operating systems virtualize running programs?
  ① Virtualization can help improve the utilization and the throughput of the underlying hardware
  ② Virtualization may allow the system to execute more programs than the number of physical processors installed in the machine
  ③ Virtualization may allow a running program or running programs to use more than install physical memory
  ④ Virtualization can improve the latency of executing each program
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

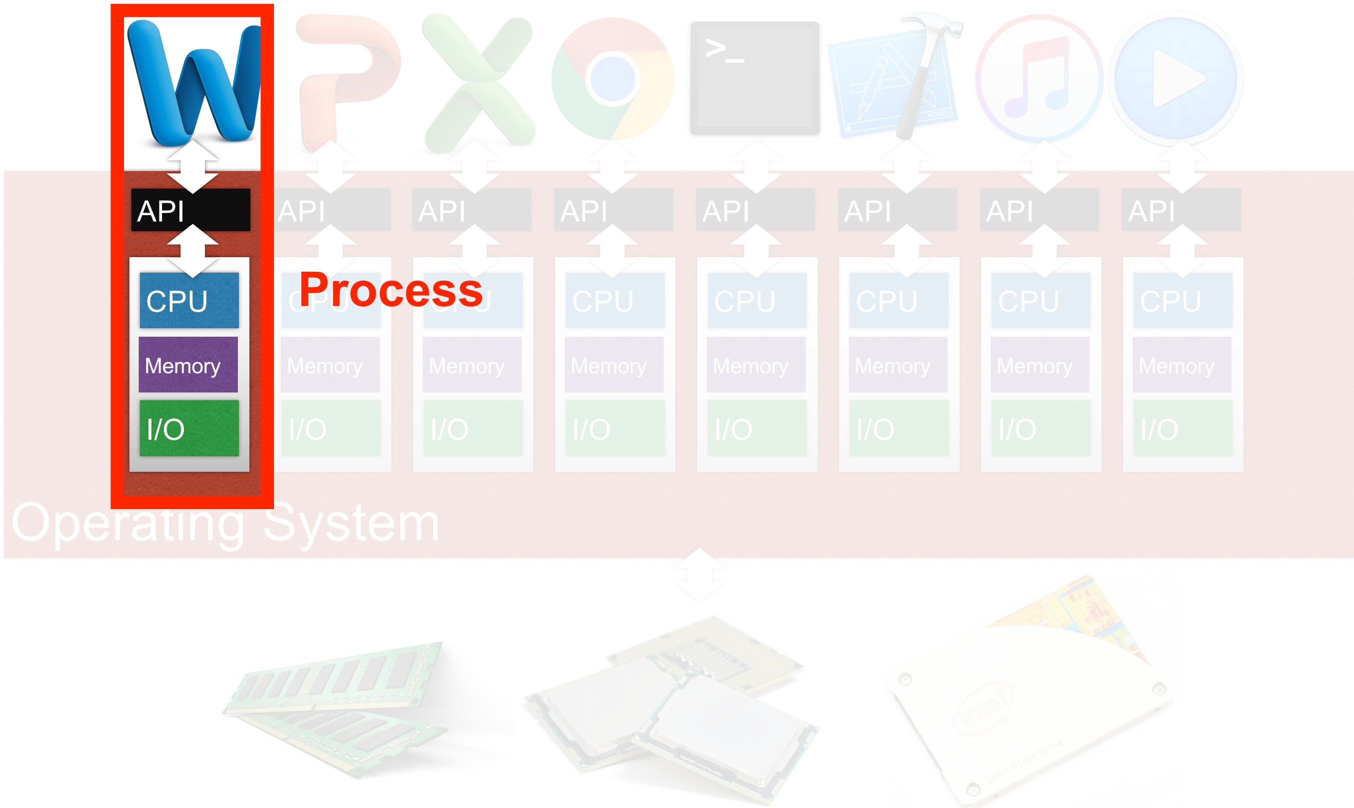
Why virtualization

- How many of the following statement is true about why operating systems virtualize running programs?
  1. Virtualization can help improve the utilization and the throughput of the underlying hardware
  2. Virtualization may allow the system to execute more programs than the number of physical processors installed in the machine
  3. Virtualization may allow a running program or running programs to use more than install physical memory
  4. Virtualization can improve the latency of executing each program

  Make programs less machine-dependent

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Virtualizing the CPU: Processes



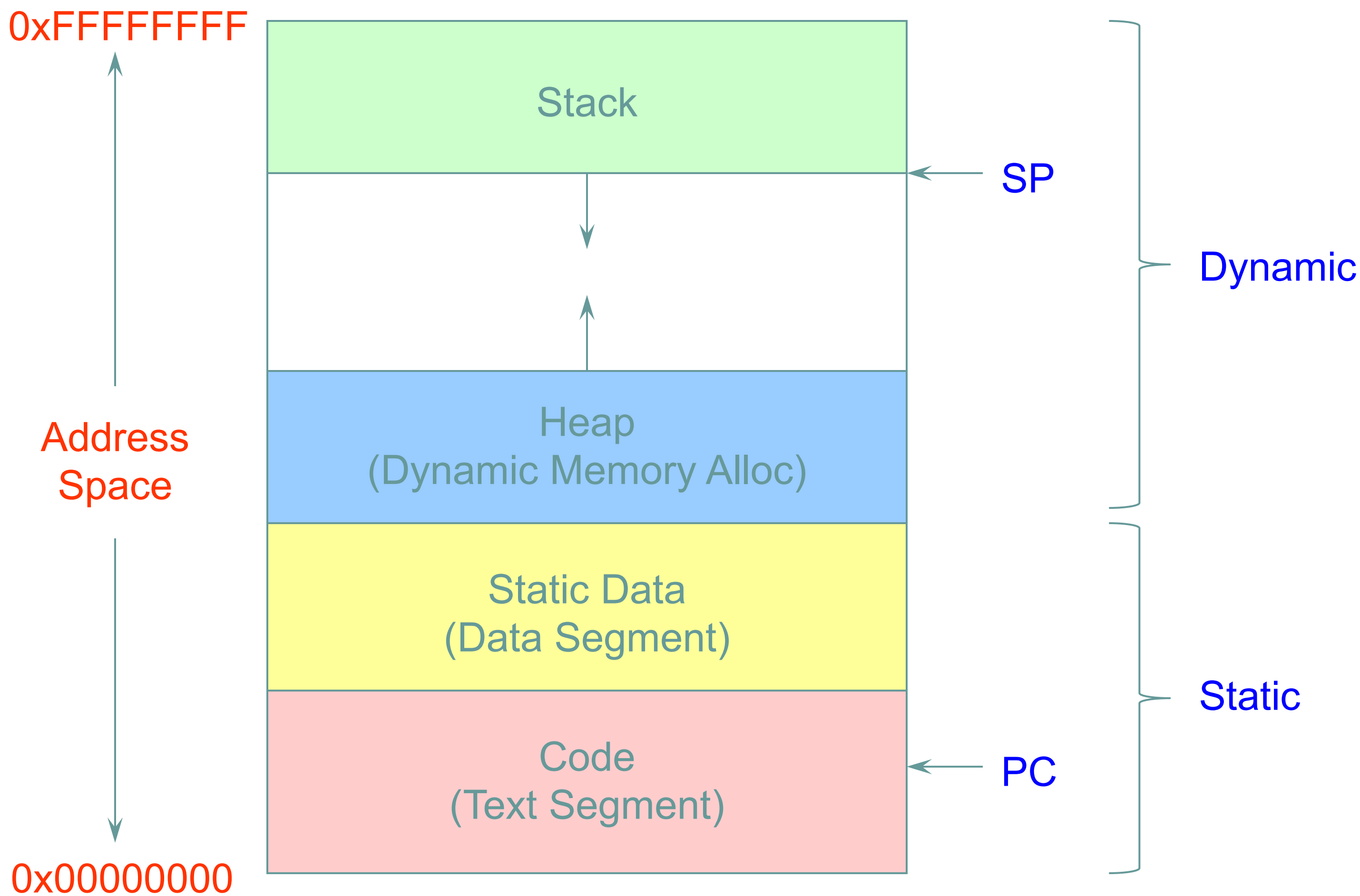
The Process

- The process is the OS abstraction for execution
  - It is the unit of execution
  - It is the unit of scheduling
- A process is a program in execution
  - Programs are static entities with the potential for execution
  - Process is the animated/active program
    - Starts from the program, but also includes dynamic state
    - As the representative of the program, it is the "owner" of other resources (memory, files, sockets, …)
- How does the OS implement this abstraction?
  - How does it share the CPU?

Process Components

- A process contains all the state for a program in execution
  - An address space containing
    - Static memory:
      - The code and input data for the executing program
    - Dynamic memory:
      - The memory allocated by the executing program
      - An execution stack encapsulating the state of procedure calls
  - Control registers such as the program counter (PC)
  - A set of general-purpose registers with current values
  - A set of operating system resources
    - Open files, network connections, etc.
- A process is named using its process ID (PID)

Address Space (memory abstraction)



What the OS must track for a process?

- Which of the following information does the OS need to track for each process?
  A. Stack pointer
  B. Program counter
  C. Process state
  D. Registers
  E. All of the above

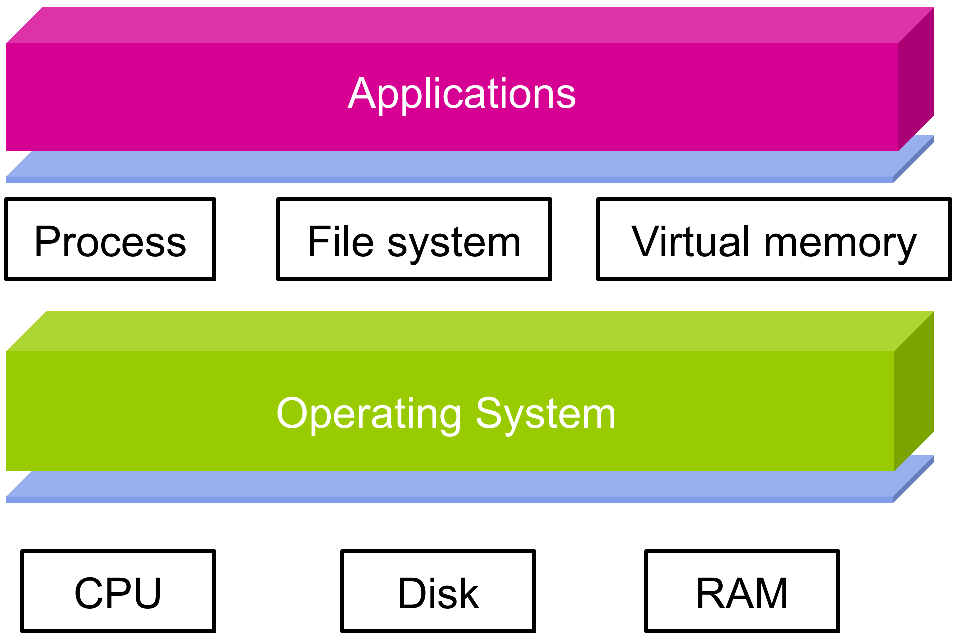
What the OS must track for a process?

- Which of the following information does the OS need to track for each process?
  A. Stack pointer
  B. Program counter
  C. Process state
  D. Registers
  E. All of the above

  - You also need to keep other process information like an unique process id, process states, I/O status, and etc…

Processes

OS Abstractions



Today, we start discussing the first abstraction that enables us to virtualize (i.e., share) the CPU – processes!

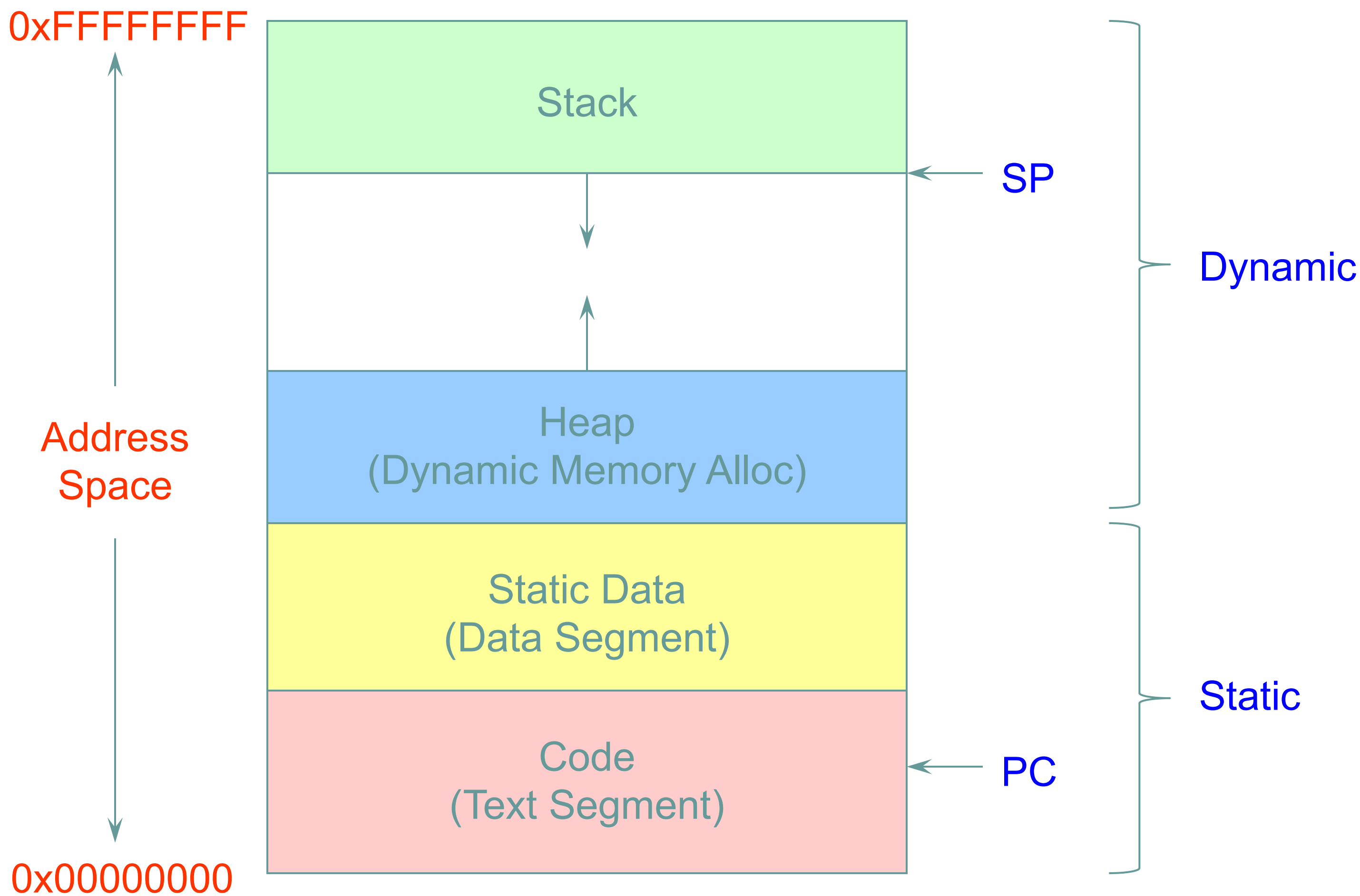
The Process

- The process is the OS abstraction for execution
  - It is the unit of execution
  - It is the unit of scheduling
- A process is a program in execution
  - Programs are static entities with the potential for execution
  - Process is the animated/active program
    - Starts from the program, but also includes dynamic state
    - As the representative of the program, it is the “owner” of other resources (memory, files, sockets, …)
- How does the OS implement this abstraction?
  - How does it share the CPU?

Process Components

- A process contains all the state for a program in execution
  - An address space containing
    - Static memory:
      - The code and input data for the executing program
    - Dynamic memory:
      - The memory allocated by the executing program
      - An execution stack encapsulating the state of procedure calls
  - Control registers such as the program counter (PC)
  - A set of general-purpose registers with current values
  - A set of operating system resources
    - Open files, network connections, etc.
- A process is named using its process ID (PID)

Address Space (memory abstraction)



Process Execution State

- A process is born, executes for a while, and then dies
- The process execution state that indicates what it is currently doing
  - Running: Executing instructions on the CPU
    - It is the process that has control of the CPU
    - How many processes can be in the running state simultaneously?
  - Ready: Waiting to be assigned to the CPU
    - Ready to execute, but another process is executing on the CPU
  - Waiting: Waiting for an event, e.g., I/O completion
    - It cannot make progress until event is signaled (disk completes)

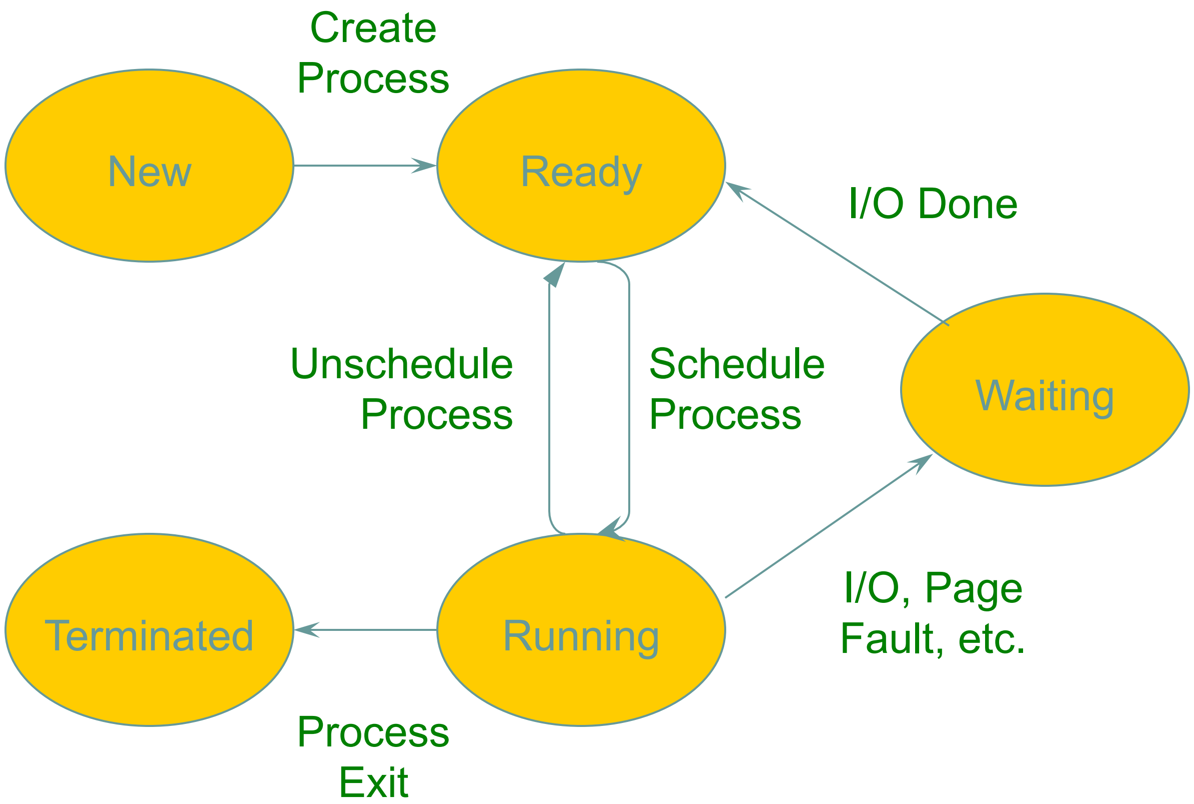
Execution state (cont'd)

- As a process executes, it moves from state to state
  - Unix "ps -x": STAT column indicates execution state

```
PROCESS STATE CODES
       Here are the different values that the s, stat and state output specifiers (header "S
       D    uninterruptible sleep (usually IO)
       R    running or runnable (on run queue)
       S    interruptible sleep (waiting for an event to complete)
       T    stopped, either by a job control signal or because it is being traced.
       W    paging (not valid since the 2.6.xx kernel)
       X    dead (should never be seen)
       Z    defunct ("zombie") process, terminated but not reaped by its parent.

       For BSD formats and when the stat keyword is used, additional characters may be displ
       <    high-priority (not nice to other users)
       N    low-priority (nice to other users)
       L    has pages locked into memory (for real-time and custom IO)
       s    is a session leader
       l    is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
       +    is in the foreground process group.
```

Execution State Graph



How does the OS support this model?

Three issues:

1. How does the OS represent a process in the kernel?
   - The OS data structure representing each process is called the Process Control Block (PCB)

2. How do we pause and restart processes?
   - We must be able to save and restore the full machine state

3. How do we keep track of all the processes in the system?
   - A lot of queues!

PCB Data Structure

- PCB also is where OS keeps all of a process' hardware execution state when the process is not running
  - Process ID (PID)
  - Execution state
  - Hardware state: PC, SP, regs
  - Memory management
  - Scheduling
  - Accounting
  - Pointers for state queues
  - Etc.
- This state is everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware

Xv6 struct proc

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Linear address of proc's pgdir
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // Switch here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

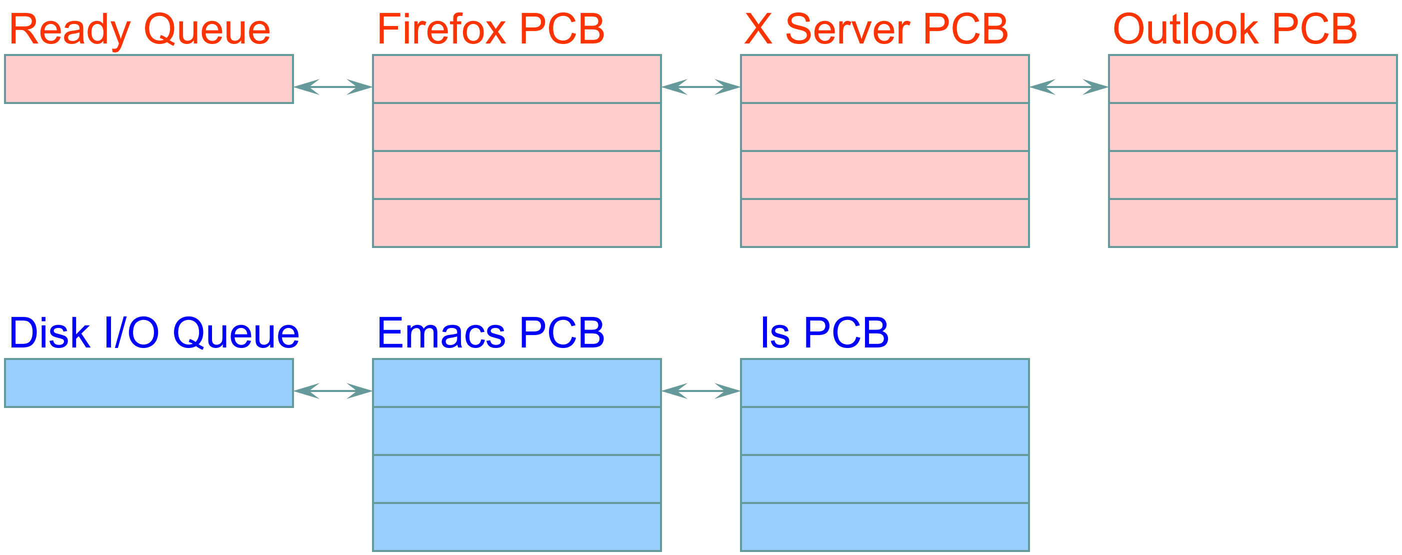
How to pause/restart processes?

- When a process is running, its dynamic state is in memory and some hardware registers
  - Hardware registers include Program counter, stack pointer, control registers, data registers, …
  - To be able to stop and restart a process, we need to completely restore this state
- When the OS stops running a process, it saves the current values of the registers (usually in PCB)
- When the OS restarts executing a process, it loads the hardware registers from the stored values in PCB
- Changing CPU hardware state from one process to another is called a context switch
  - This can happen 100s or 1000s of times a second!

How does the OS track processes?

- The OS maintains a collection of queues that represent the state of all processes in the system
- Typically, the OS at least one queue for each state
  - Ready, waiting, etc.
- Each PCB is queued on a state queue according to its current state
- As a process changes state, its PCB is unlinked from one queue and linked into another

State Queues



Console Queue

Sleep Queue

.

.

.

There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)