Operating Systems

Lecture 7-8
Processes/Thread Management

Department of Computer Science & Technology
Tsinghua University

Outline

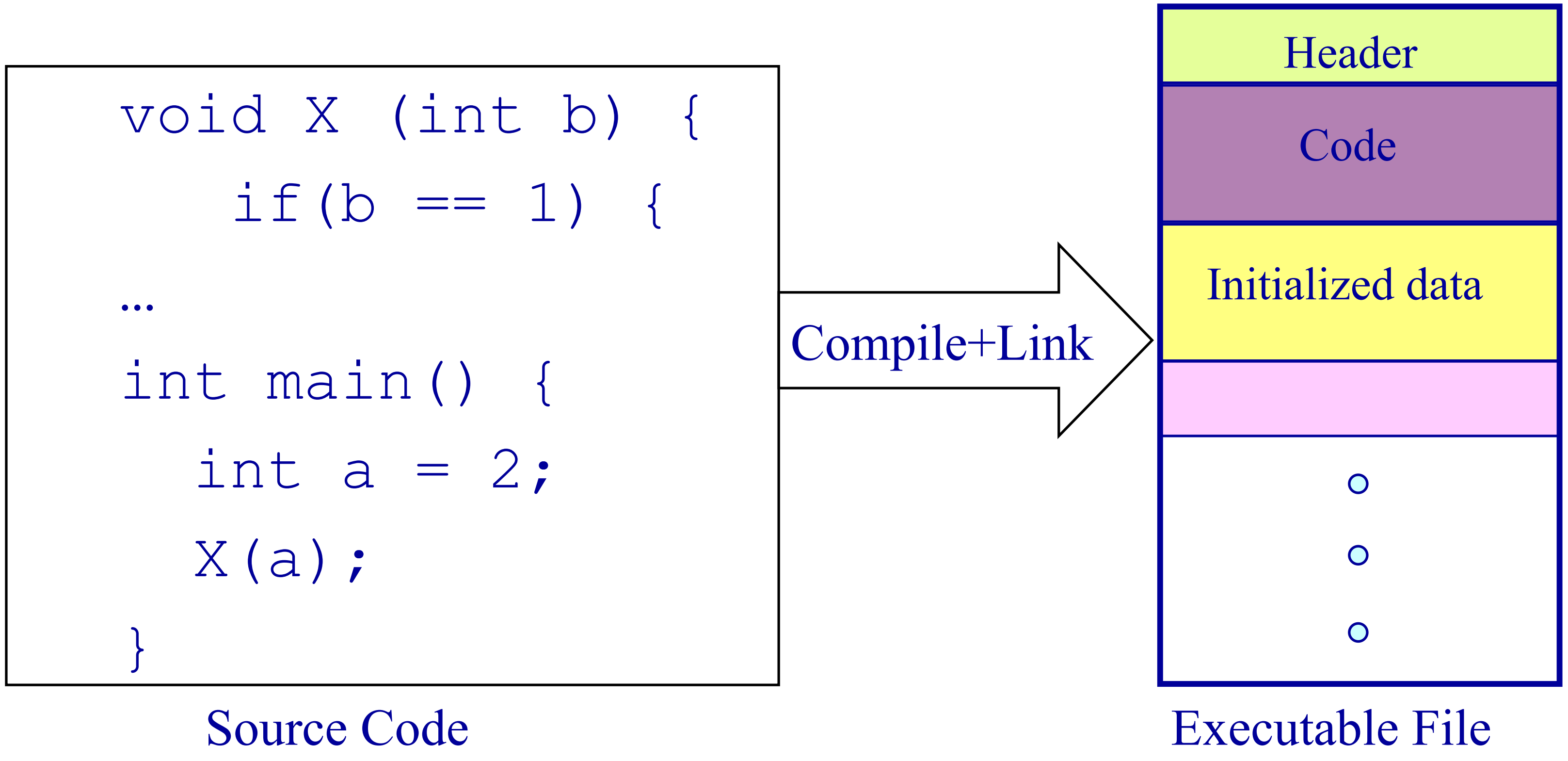
- What is a Process?
- Process Control Block
- Process Life Cycle
- The Concept of Thread
- Example Multithreaded Programs
- Thread Implementations
- Context Switch
- Process Control

What is a Process?

- An OS abstraction that supports running programs
  - Basic unit of execution in an operating system
- A process is a program during execution.
  - Program = static file (image)
  - Process = executing program = program + execution state.
- Different processes may run several instances of the same program
  - I run ls, you run ls – same program, different processes
- At a minimum, process execution requires following resources:
  - Memory to contain the program code and data
  - A set of CPU registers to support execution

From Program to Process

- We write a program in e.g., C.
- A compiler turns that program into an instruction list.
- A linker builds an executable file (code + data)
- A loader loads the executable file into memory (make ready to run)



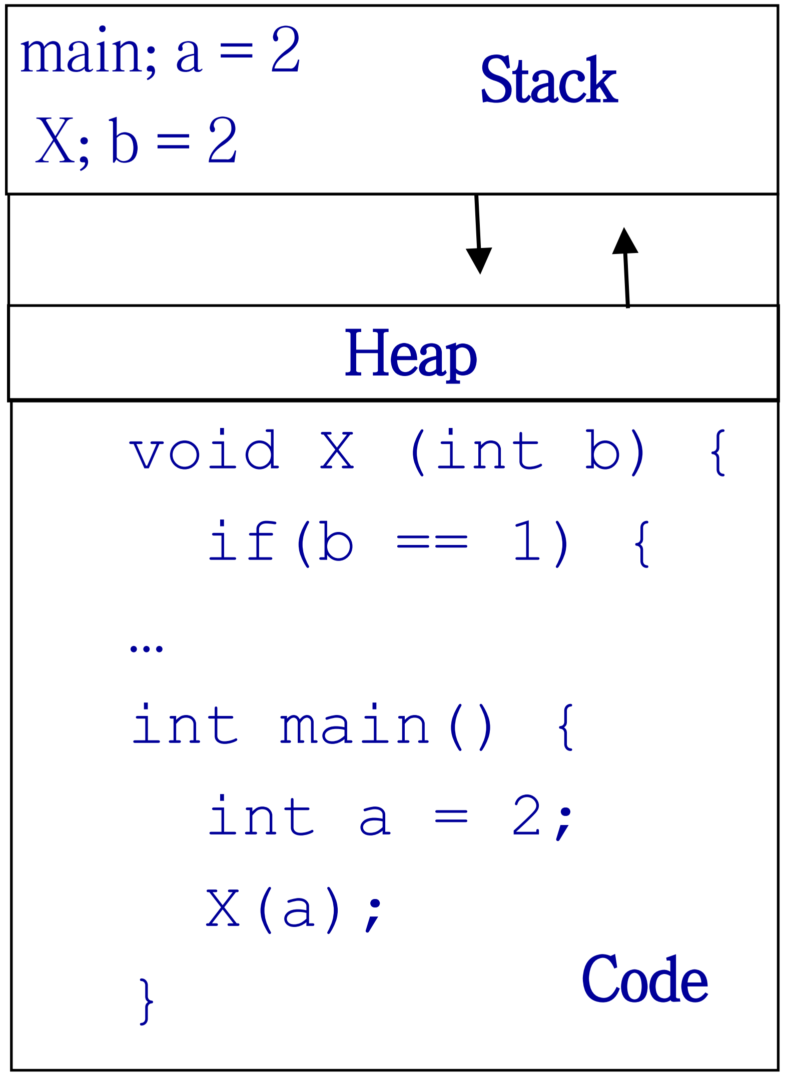
```
void X (int b) {
    if(b == 1) {
...
int main() {
   int a = 2;
   X(a);
}
```

Process in Memory

- What you wrote

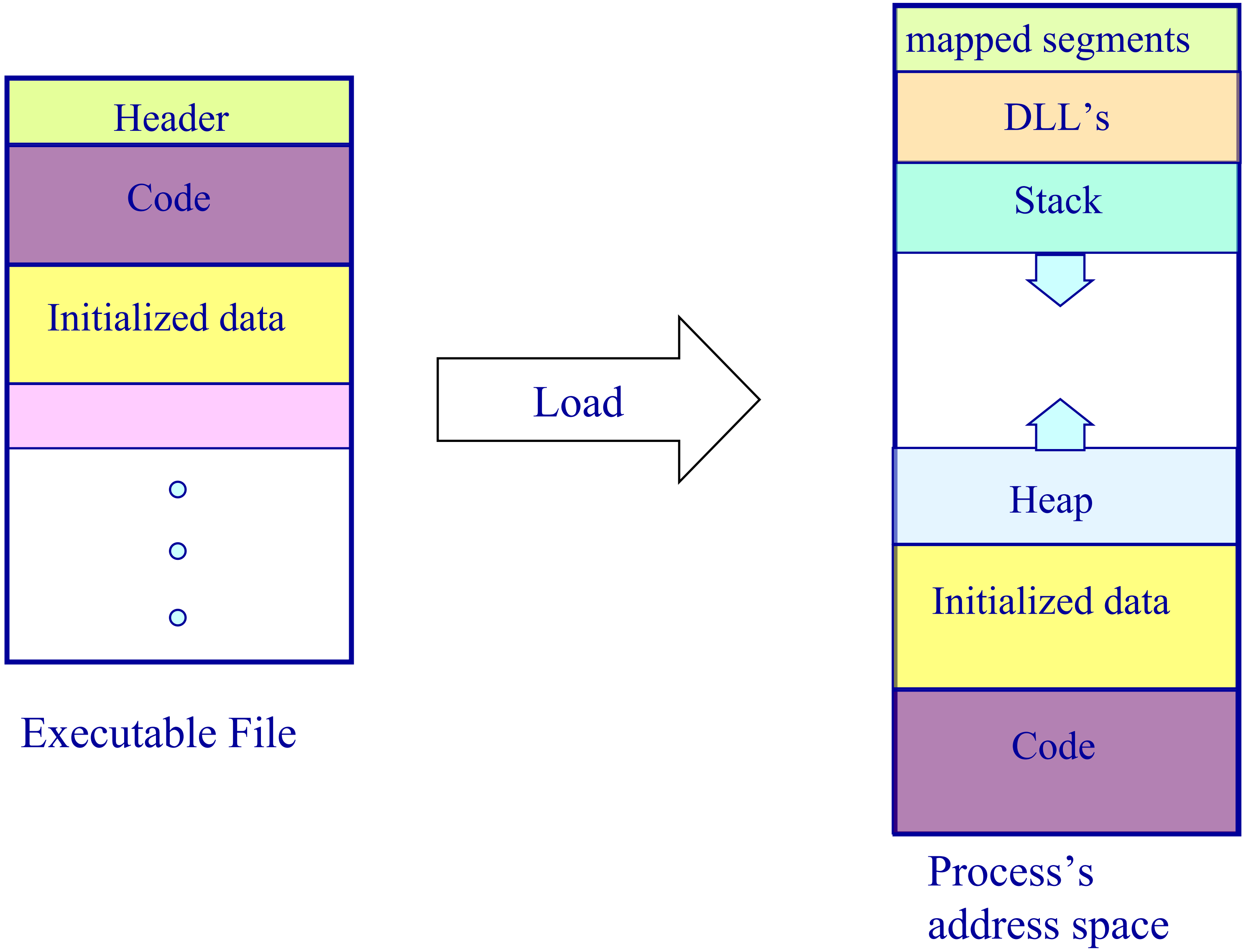
```
void X (int b) {
  if(b == 1) {
…
int main() {
  int a = 2;
  X(a);
}
```

- What is in memory.



```
void X (int b) {
  if(b == 1) {
…
int main() {
  int a = 2;
  X(a);
}
```

Anatomy of a Process

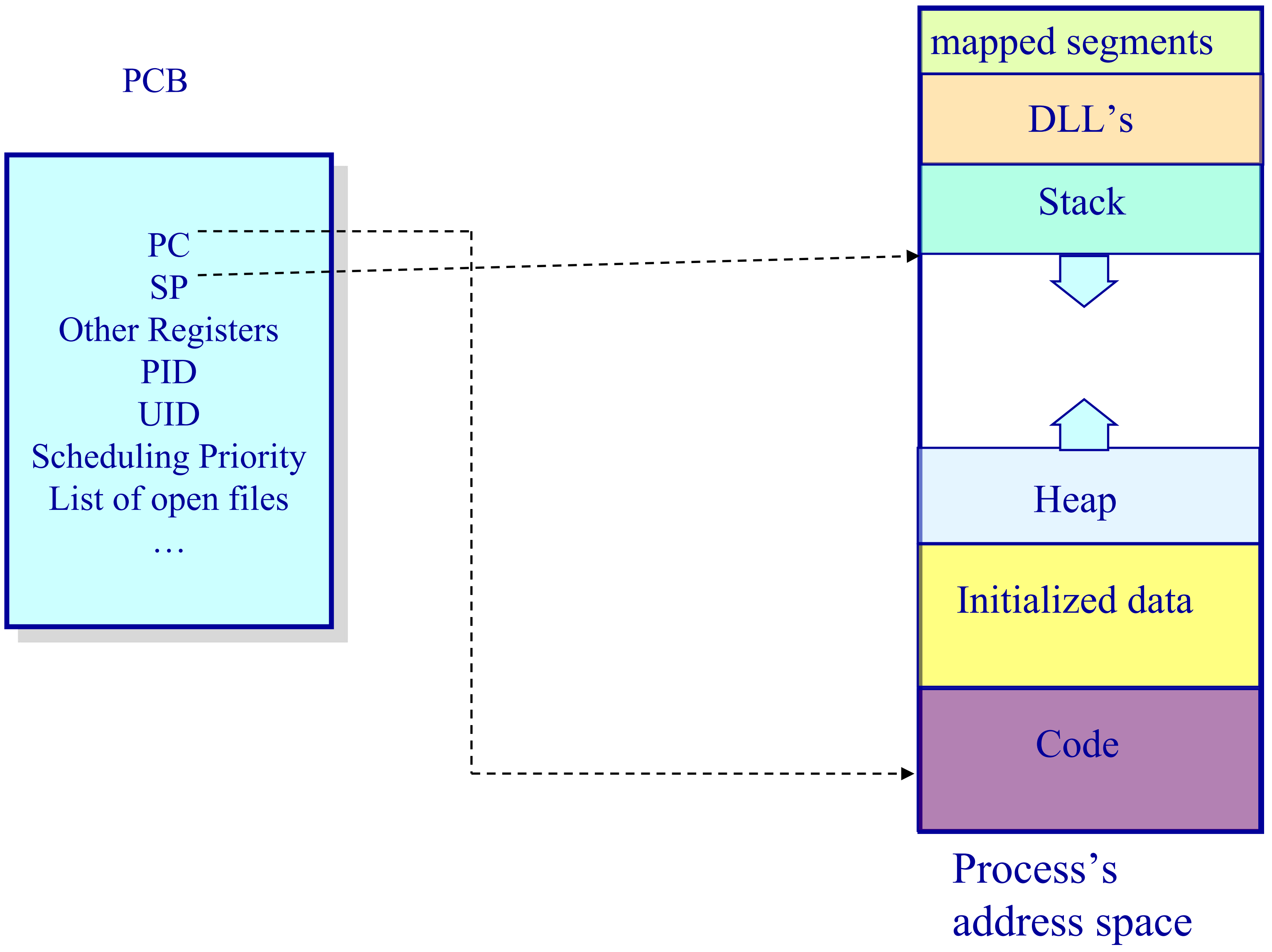


Outline

Keeping track of a process in OS

- A process has code.
  - OS must track program counter (code location).
- A process has a stack.
  - OS must track stack pointer.
- OS stores state of processes’ computation in a process control block (PCB).
  - E.g., each process has an identifier (process identifier, or PID)
- Data (program instructions, stack & heap) resides in memory, metadata is in PCB.

Process Control Block



Processes and Process Management
Details for running a program

- A program consists of code and data
- On running a program, the loader:
  - reads and interprets the executable file
  - sets up the process's memory to contain the code & data from executable
  - pushes "argc", "argv" on the stack
  - sets the CPU registers properly & calls "_start()"
- Program starts running at _start()

```
_start(args) {
    ret = main(args);
    exit(ret)
}
```

  we say "process" is now running, and no longer think of "program"

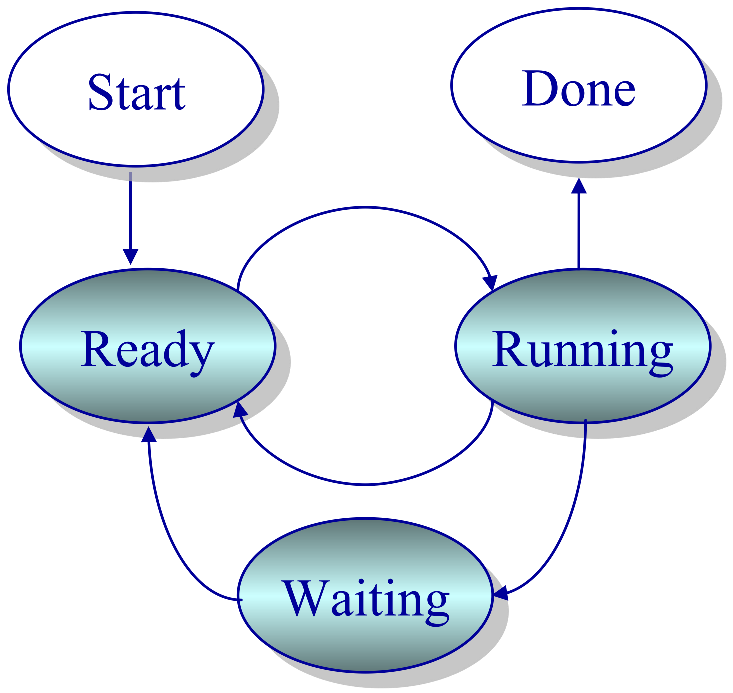
- When main() returns, OS calls "exit()" which destroys the process and returns all resources

Outline

- What is a Process?
- Process Control Block
- Process Life Cycle
- The Concept of Thread
- Example Multithreaded Programs
- Thread Implementations
- Context Switch
- Process Control

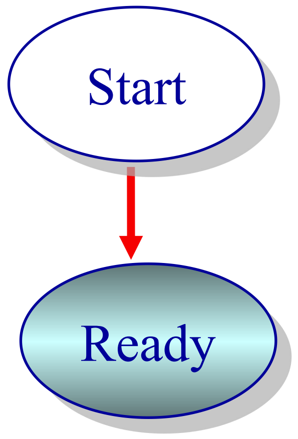
Process Life Cycle

- Processes are always either Running, Ready (to execute) or Waiting (for an event to occur)



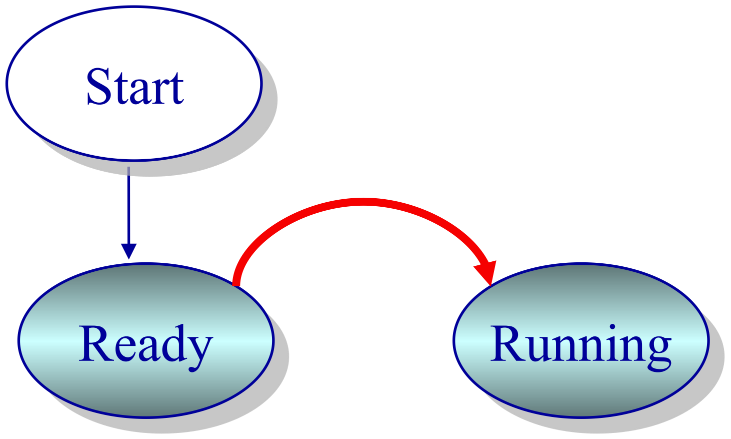
Process Life Cycle

- Process is created at Start and transitions to Ready when it becomes runnable



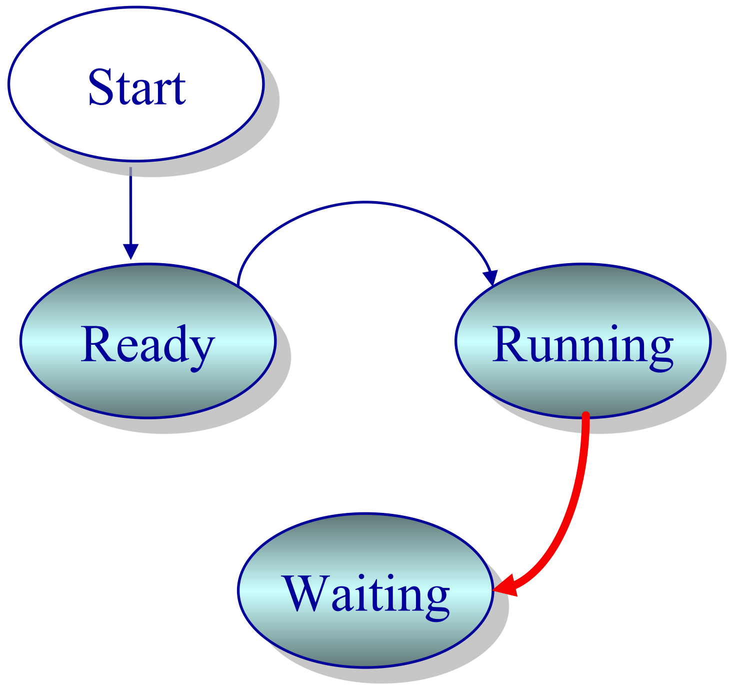
Process Life Cycle

- Process transitions from Ready to Running when kernel schedules it



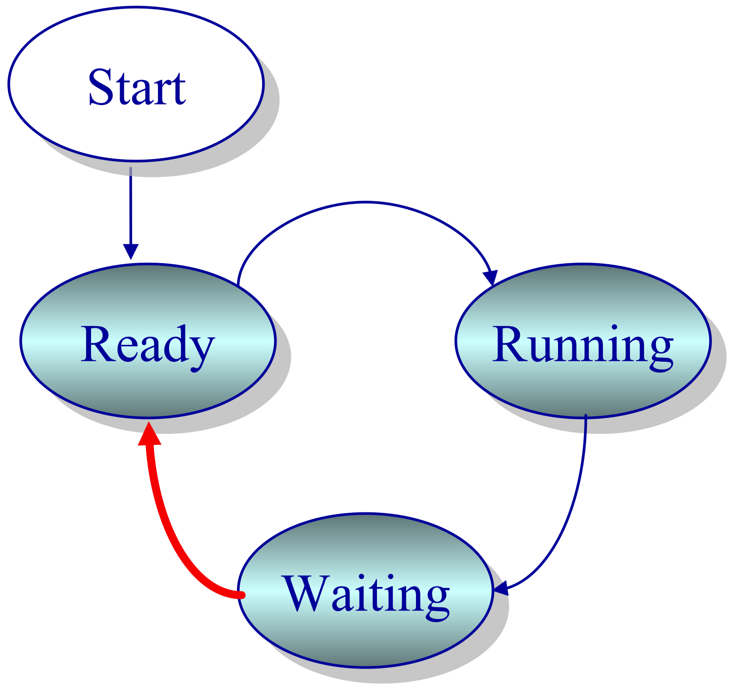
Process Life Cycle

- Process transitions from Running to Waiting when it is blocked, waiting for an event to occur (e.g., waiting for an I/O to finish)



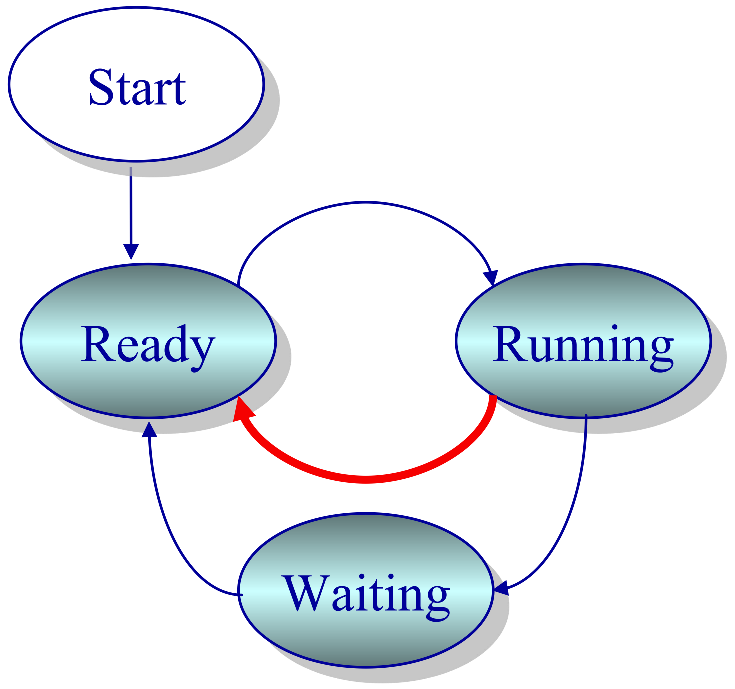
Process Life Cycle

- Process transitions from Waiting to Ready when the event occurs (e.g., I/O completion)



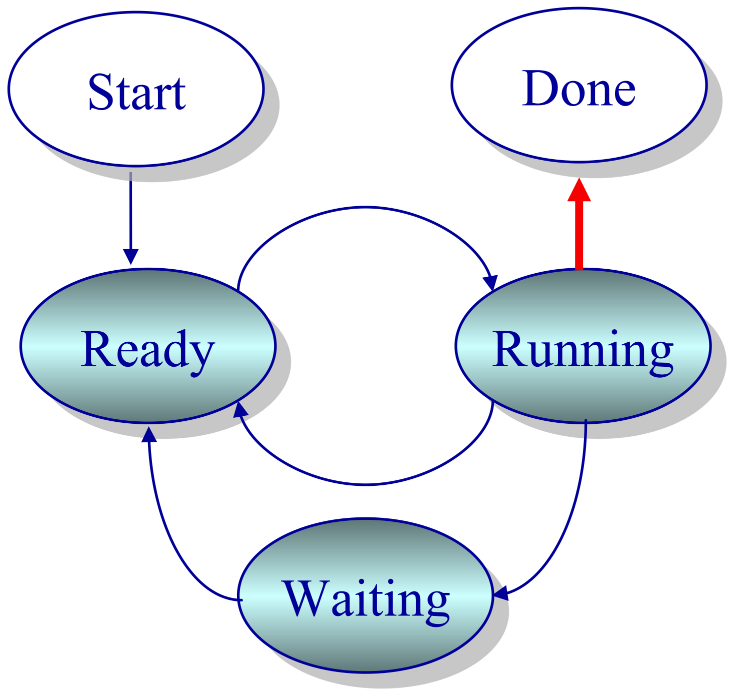
Process Life Cycle

- Process transitions from Running to Ready on an interrupt and pre-emptive scheduling



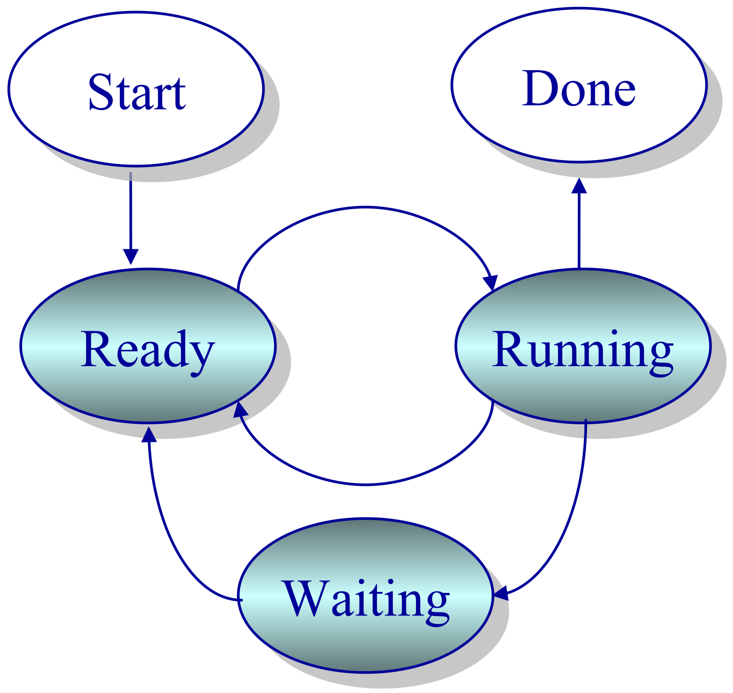
Process Life Cycle

- Process transitions from Running to Done on exit()

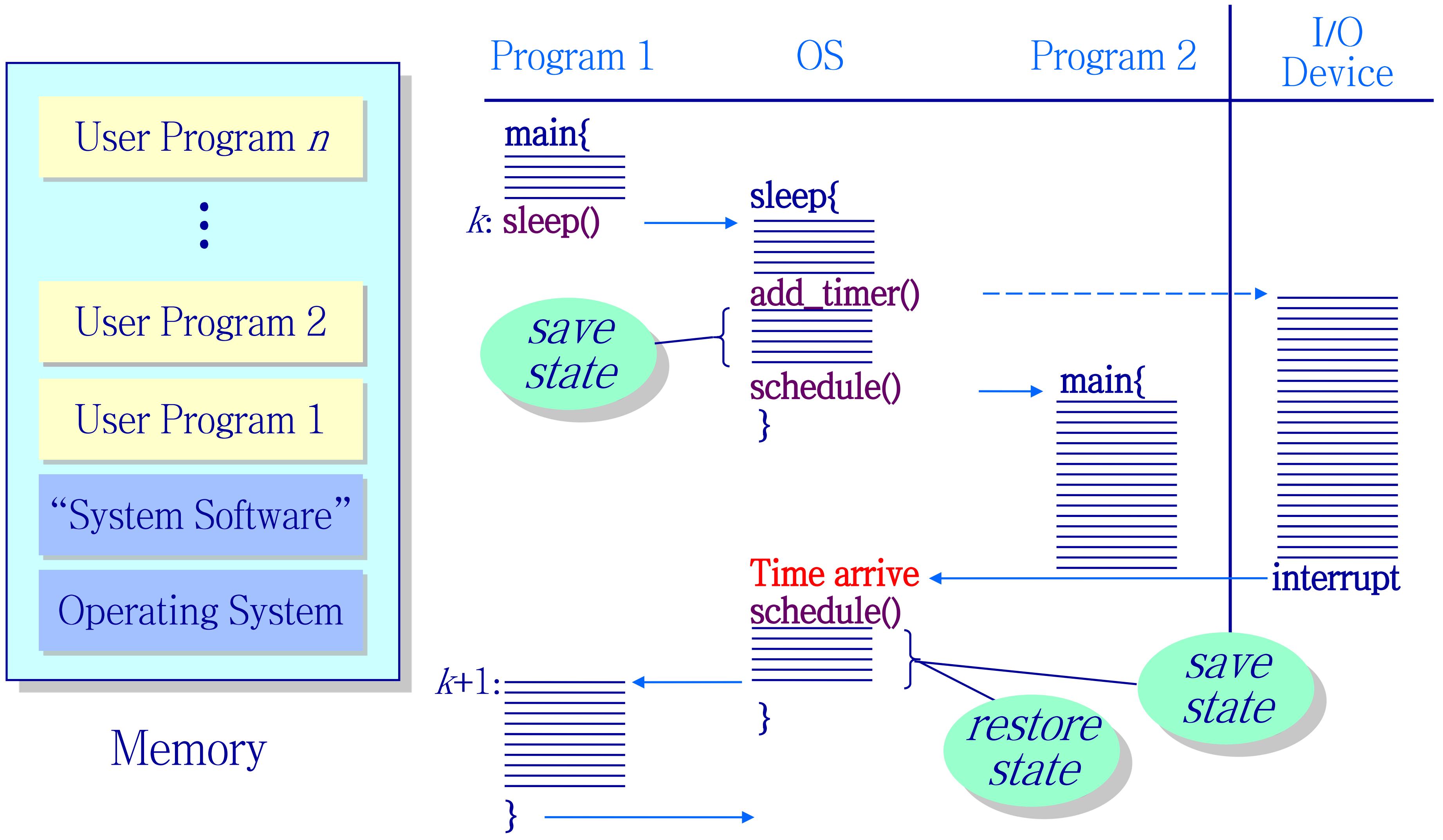


Example Process State Transition

- What happens on a sleep() system call?



Process Contexts (process sleep)



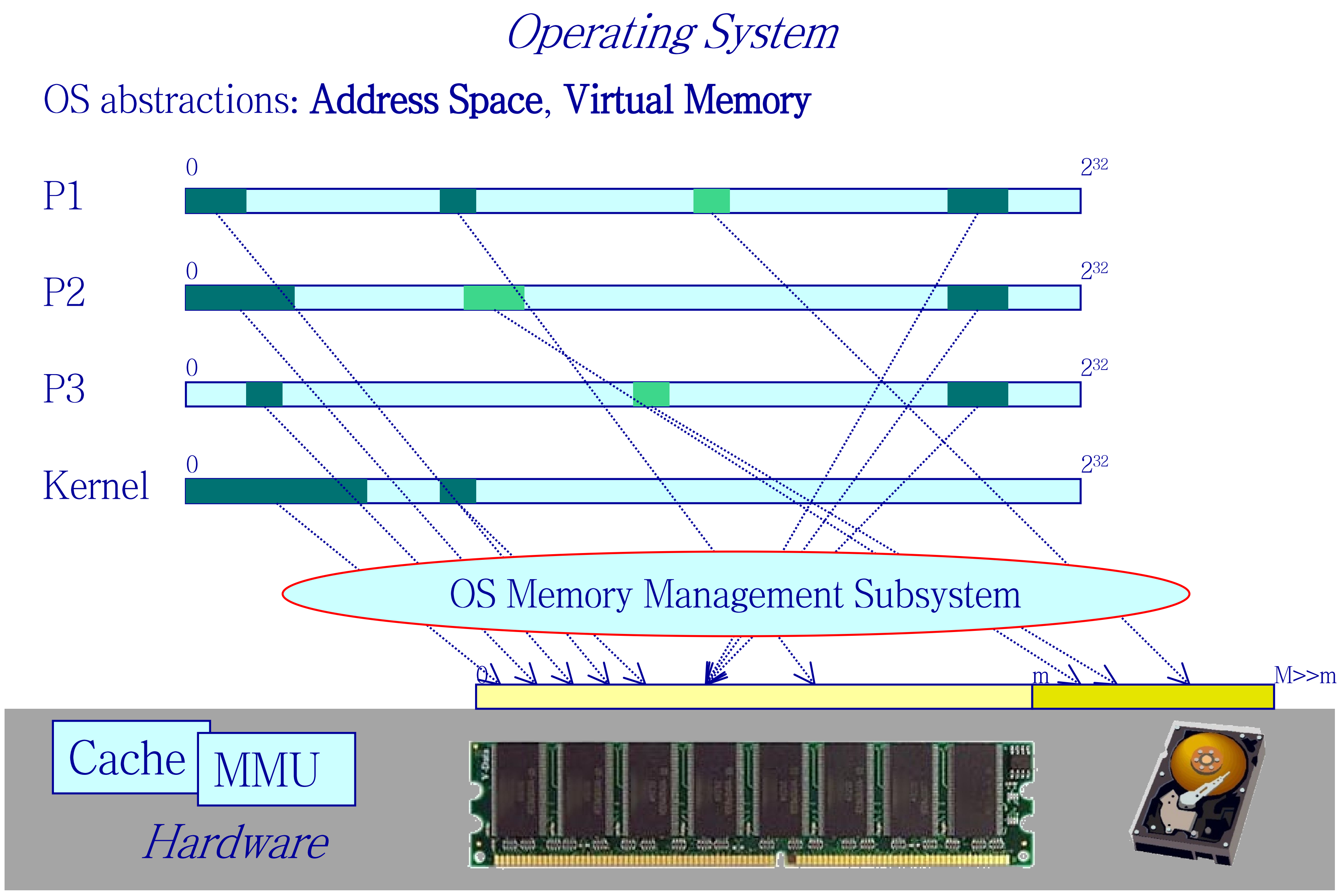
Outline

- Background
- The Concept of Thread
- Example Multithreaded Programs
- Thread Implementations
  - Kernel Threading
  - User-level Threading
- Multithreading in Real Life
  - Windows Thread
  - Posix Thread
- Multiple thread and CPU Architecture
  - Instruction-Level Parallelism
  - Data-Level Parallelism
  - Thread-Level Parallelism

The Notion of Concurrency

- "Thread" of execution
  - Sequential execution of a stream of instructions at a CPU
- Uniprogramming: one thread at a time
  - Early OS (MSDOS, etc.)
- Multiprogramming: multiple threads at a time
  - Modern OS
  - Sometimes called "multitasking"
- The basic problem of concurrency: multiplexing
  - Hardware: limited set of resources (CPU, memory, I/O)
  - Multiprogramming: each thread thinks it owns the whole thing
  - OS has to manage concurrency

Managing Space

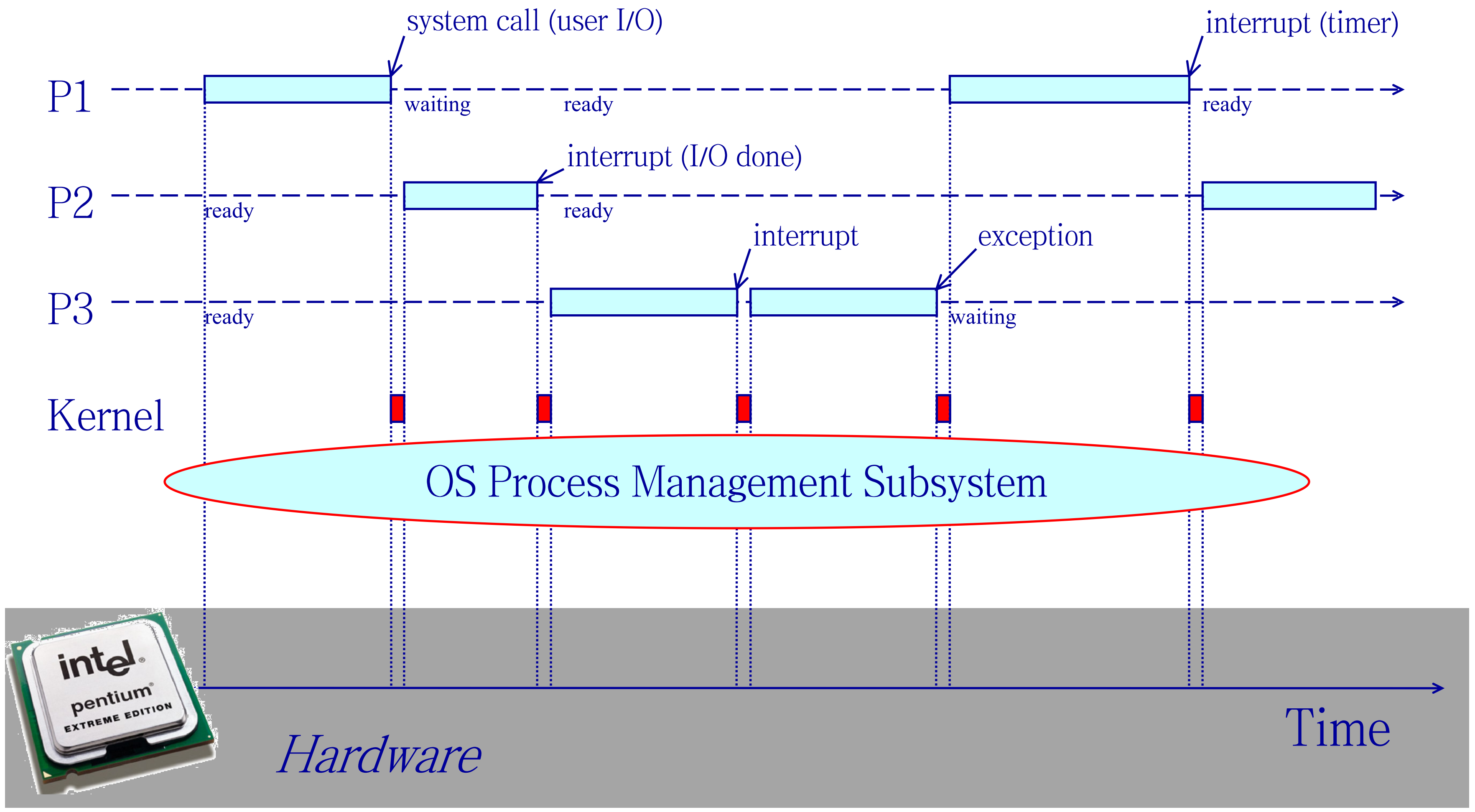

Operating System
OS abstractions: Address Space, Virtual Memory
P1
P2
P3
Kernel
0
2^{32}
OS Memory Management Subsystem
0
m
M>>m
Cache
MMU
Hardware

OS

Managing Time

Operating System

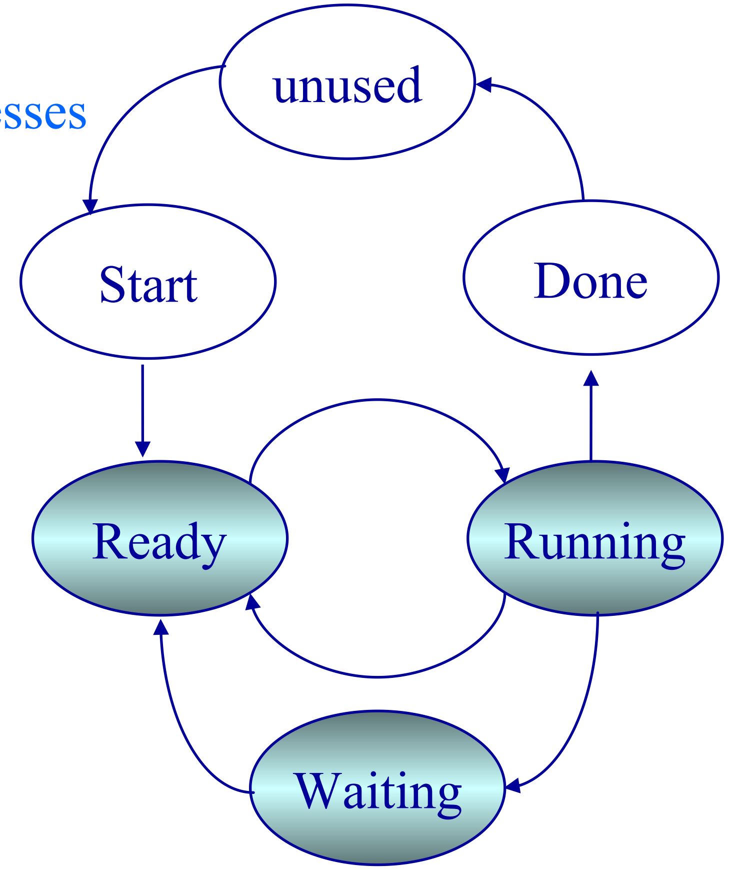
OS abstraction: **Process**, **Thread**



OS

Before …

- Process = Program + Execution State
  - Process is a sequential execution in its own address space
- PCB (Process Control Block)
  - Kernel data structure to manage processes
- Process life cycle
  - Ready, Running, Waiting
- Context and context switch
  - Save the execution state
- API
  - fork() and exec()



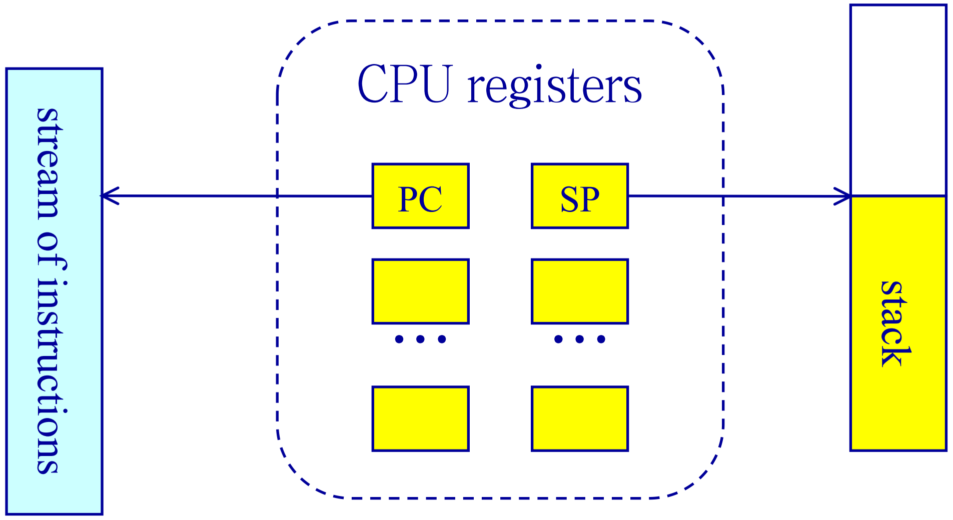
Outline

Two Concepts in a "Process"

- The "Process" abstraction
  - Process is a sequential execution in its own address space
  - It combines two concepts: concurrency and protection
- Concurrency
  - A "thread" of execution independent of other processes
- Protection
  - Each process defines an address space, which identifies all addresses that can be touched by the process
- From Process to Thread
  - Thread: a sequential execution of a program (or a stream of instructions), in *some* address space
  - Separate the concepts of concurrency from protection

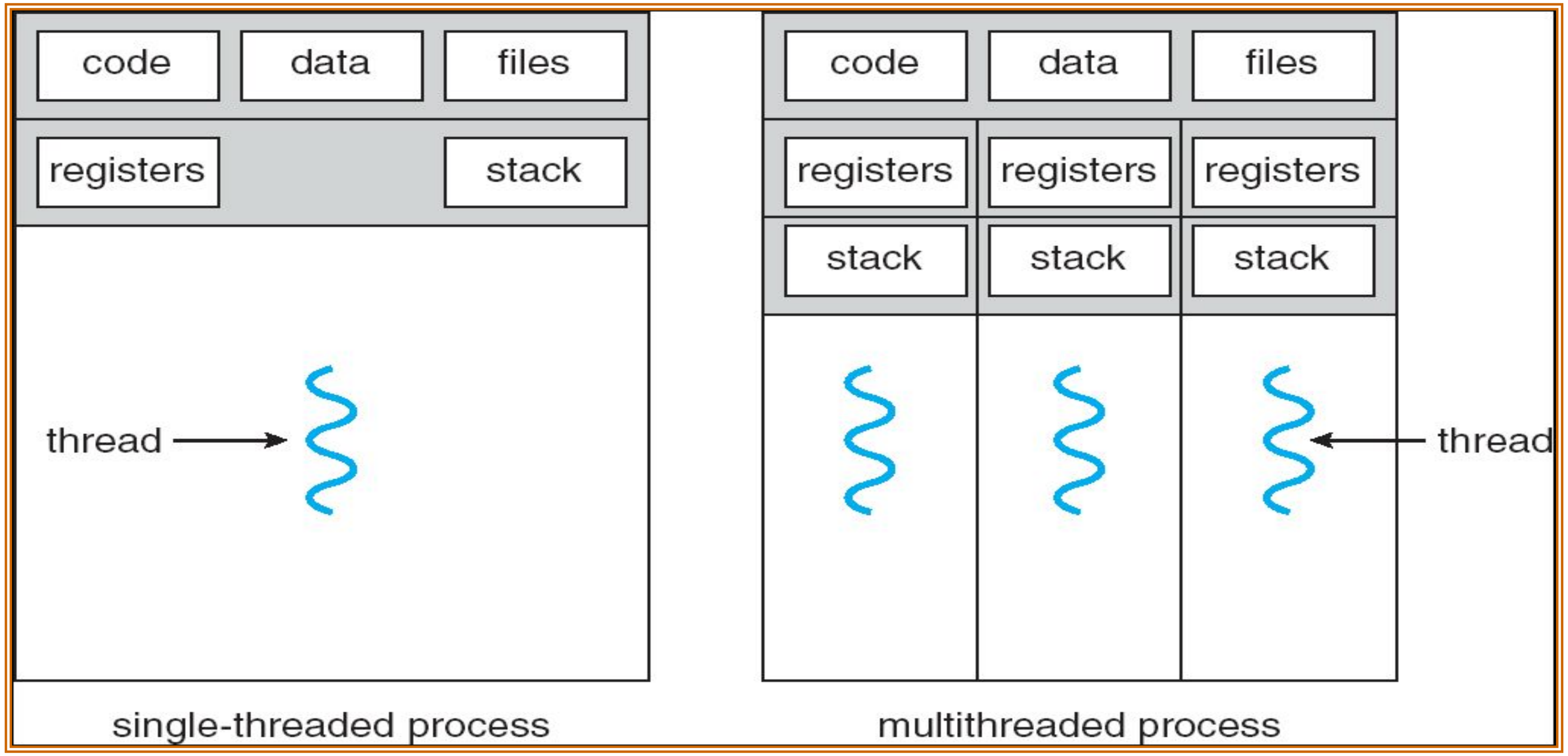
The Concept of Thread

- An OS abstraction
  - A sequential execution of a stream of instructions
- Resources associated with thread
  - Program Counter (PC), Stack Pointer (SP), plus a set of other CPU registers & flags
  - Each thread must have its own stack



OS

Single and Multithreaded Processes



- Maximum one thread per process (address space)
- Example: traditional Unix (no concept of thread)
- But doesn’t prevent user to add own thread support in user program (user-level threading)

- Support more than one threads per process
- A single program made up of a number of different concurrent activities

From Process to Thread

- Roughly, Process = Thread(s) + Address Space
  - One or more threads in a single address space
  - Thread: encapsulate concurrency
  - Address space: encapsulate protection
- Usually need OS support for threads
  - Managing threads
  - Scheduling/switching among threads
- Example systems that support threads:
  - OS-supported: Sun’s LWP, POSIX’s threads
  - Language-supported: Modula-3, Java, ErLang

Thread States

- Individual state for each thread
  - CPU registers (must save/restore during context switch)
  - Stack (how do we save/restore this?)
- Shared by all threads in a process
  - Contents of memory (MMU translation states)
  - I/O states
  - Other OS book keeping data (open files, network connections, etc)
- Threads are lightweight (c.f. process)
  - No thread-specific heap or data segment (unlike process)
  - Therefore, context switching between threads is much cheaper than for a process

Outline

- What is a Process?
- Process Control Block
- Process Life Cycle
- The Concept of Thread
- Example Multithreaded Programs
- Thread Implementations
- Context Switch
- Process Control

OS

Example Multithreaded Programs

- Server programs
  - Web servers, file servers, network servers, database servers, application servers, etc.
  - Why multithreading? concurrent requests from network, from concurrent users, etc.
- Embedded systems
  - Elevators, machines, etc.
  - Single program, multiple concurrent operations
- Operating system kernel?
  - Yes for most modern OS
  - Have to deal with concurrent requests

Multithreading

- Why multithreaded programs?
  - Single program, multiple concurrent operations
  - Have to serve multiple requests, multiple users
  - Take advantage of algorithmic parallelism
- Technology trend: concurrent programming
  - The world is going multi-core
  - Parallel programming: split program into multiple threads for performance gain
- Multiple threads or multiple processes?
  - Depends.

OS

Web Server Example

- Non-threaded version

```
Loop {
    block for new connection;
    ForkNewProcess(WebServer, new_connection);
}
```

- Threaded version

```
Loop {
    block for new connection;
    ForkNewThread(new_connection);
}
```

- Advantages
  - Share file caches kept in memory, results of CGI scripts, etc.
  - Low per-request overhead (threads are much cheaper to create than process)

Threads vs. Processes

Threads

- No data segment or heap
- Multiple can coexist in a process
- Share code, data, heap and I/0
- Have own stack and registers, but no isolation from other threads in the same process
- Inexpensive to create
- Inexpensive context switching

Processes

- Have data/code/heap and other segments
- Include at least one thread
- Have own address space, isolated from other processes'
- Expensive to create
- Expensive context switching

Outline

- What is a Process?
- Process Control Block
- Process Life Cycle
- The Concept of Thread
- Example Multithreaded Programs
- Thread Implementations
- Context Switch
- Process Control

OS

Thread Implementations

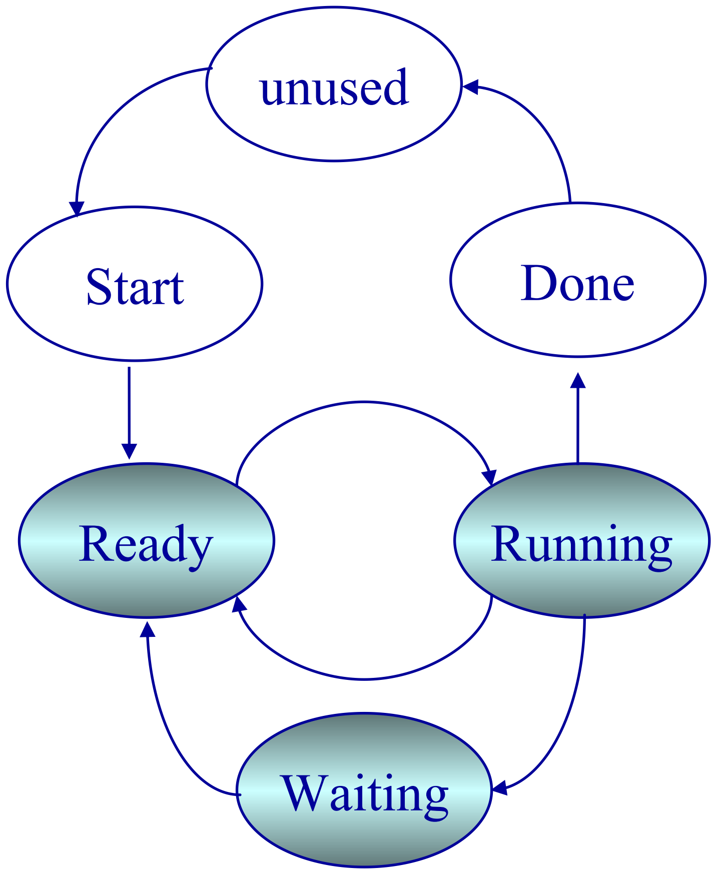
- Kernel multithreading
  - Operating system supports multiple threads per process
  - OS kernel manage and schedule the threads
- User-level multithreading
  - User program implements its own threading with some user-space threading library
  - System may or may not have kernel threading, but kernel does not know about the user-level threads
- Chip-level multithreading
  - Architecture (Hardware) support for multithreading

Kernel Threading

- New kernel data structure: TCB (Thread Control Block)
  - Execution state: PC, SP, CPU registers
  - Scheduling info: lifecycle, priority, etc.
  - Pointer to enclosing process (PCB)
  - Plus others
- Like process, thread has state (in lifecycle) and will be scheduled by CPU scheduler

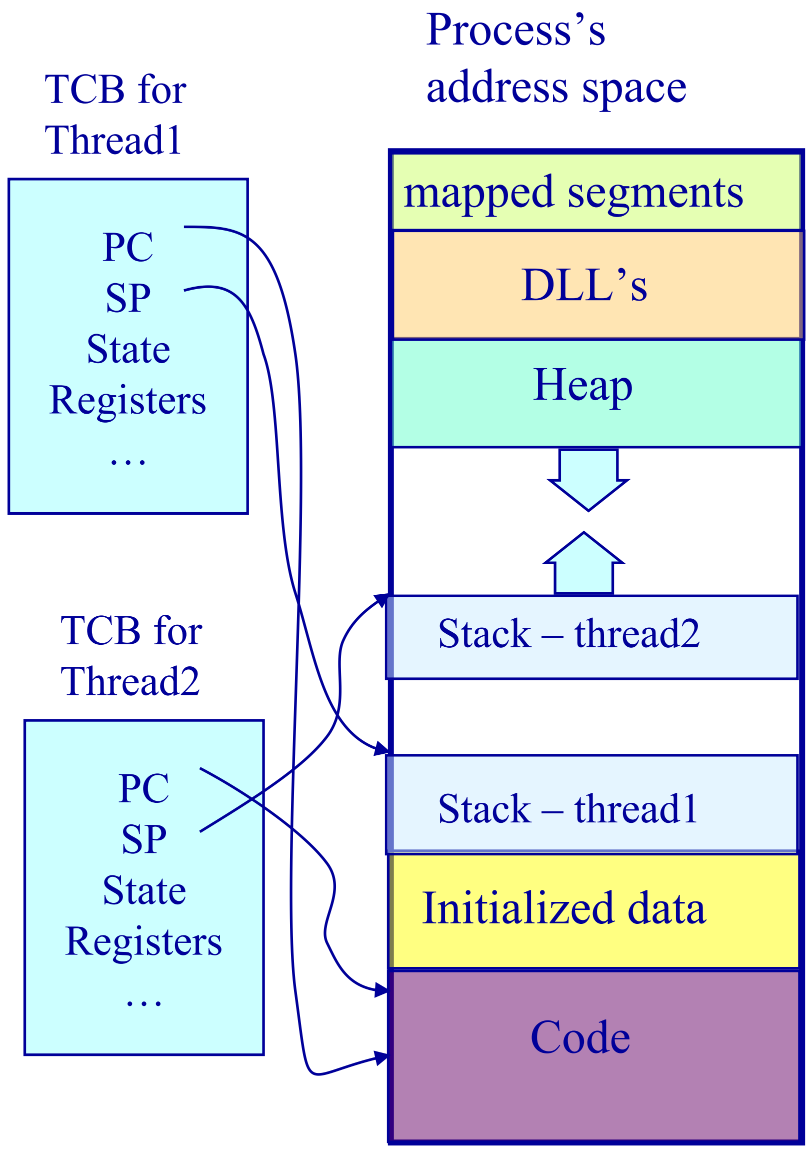
Threads' Life Cycle

- Threads (just like processes) go through a sequence of *start*, *ready*, *running*, *waiting*, and *done* states



Implementing Thread Support in OS Kernel

- PCB contains process-specific information
  - Owner, PID, heap pointer, priority, active thread, and pointers to thread information
- TCB contains thread-specific information
  - SP, PC, CPU registers thread state, pointer to PCB, ···



OS

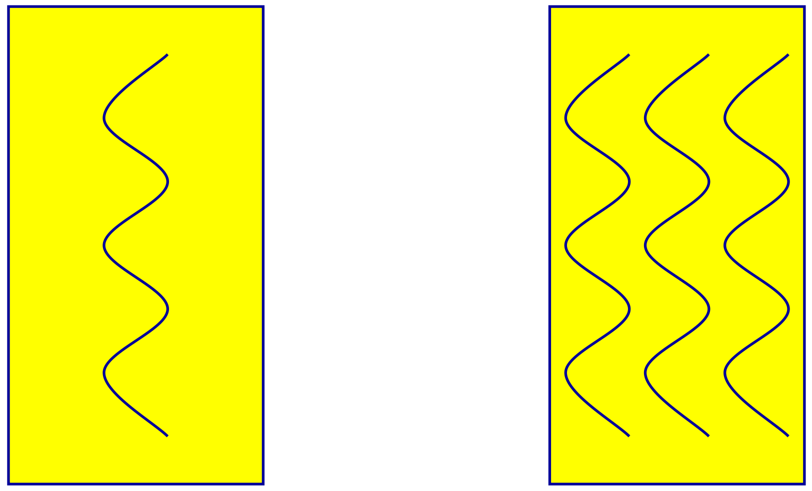
Implementing Threads

```
CreateThread(pointer_to_procedure, arg0, …) {
// allocate a new TCB and stack
      TCB tcb = new TCB();
     Stack stack = new Stack();
// initialize TCB and stack with initial register values and address of first
      instruction
      tcb.pc = Stub;
      tcb.stack = stack;
      tcb.arg0reg = pointer_to_procedure;
      tcb.arg1reg = arg0;
      …
// Tell the dispatcher about the newly created thread
      ReadyQ.add(tcb);
}

Stub(proc, arg0, arg1, …) {
                (*proc)(arg0, arg1, …);
                DeleteCurrentThread();
}
```

Summary of Threads

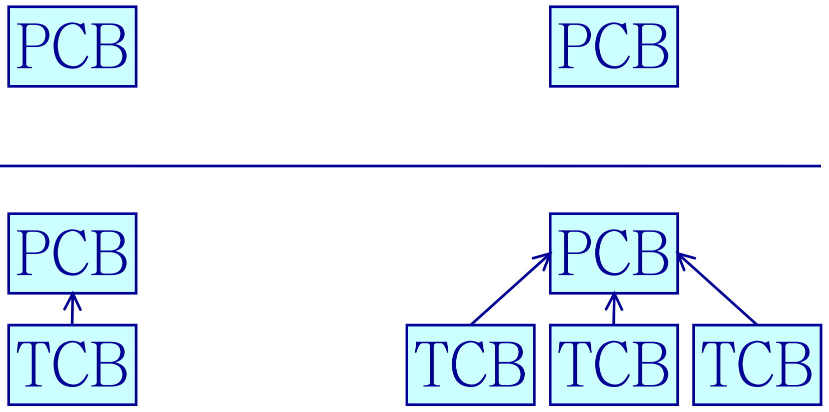
Process



Kernel data structure

Single-threading OS kernel

Multi-threading OS kernel
(virtually all modern OS)



OS

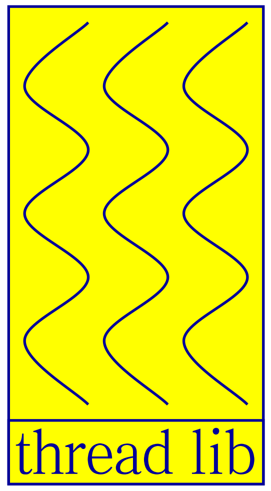
User-level Threading

- Motivation
  - Threads are a useful programming abstraction
  - Implement thread creation/scheduling using procedure calls to a user-level library rather than system calls
- User-level threading
  - User-level library implementations for
    - 尌 CreateThread(), DestroyThread(), Yield(), ···
  - User-level library performs the same set of actions of corresponding system calls
  - Main difference: thread management is under the control of user-level library
- What happens if a user-level thread makes a system call?

OS

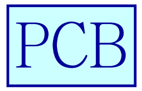
User-level Threading

Process

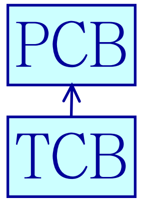


Kernel data structure

Single–threading OS kernel



Multi–threading OS kernel



OS

User-level Threading

- Benefits:
  - Faster context switch (no need to cross into kernel)
  - Thread scheduling is more flexible
    - 封 Can use application-specific scheduling policy
    - 封 Each process can use a different scheduling algorithm
    - 封 Threads voluntarily give up CPU
- Drawbacks:
  - OS is unaware of the existence of user-level threads
    - 封 Poor scheduling decisions
    - 封 If a user-level thread waits for I/O – entire process waits
  - OS schedules processes independent of number of threads within a process

User-level Threading vs Kernel Threading

- User-level threading
  - OS does not know about user-level threads
  - OS is only aware of the process that contains threads
  - OS schedules processes, not threads
  - Programmer uses a threads library to manage threads (create, delete, synchronize and schedule)
- Kernel threading
  - OS knows and tracks kernel threads
  - Switching threads within same process is inexpensive
  - Kernel uses process scheduling algorithms to manage threads

Scheduler Activations (best of both worlds)

- Why not a user level thread scheduler that spawns a kernel thread for blocking operations?
  - But how do we know if an operation will block?
  - read() might block, or data might be in page cache.
  - Any memory reference might cause a page fault to disk.
- Solution : Scheduler Activations
  - Kernel tells user when a thread is going to block, via an upcall.
  - Kernel can provide a kernel thread to run the user-level upcall handler (or preempt user thread).
  - User-level scheduler suspends blocking thread and can give back kernel thread it was running on.

Thread Pools

- Control multiprogramming level
  - Maintain a bounded "pool" of worker threads (controlling the maximum number of threads)
- Web server example

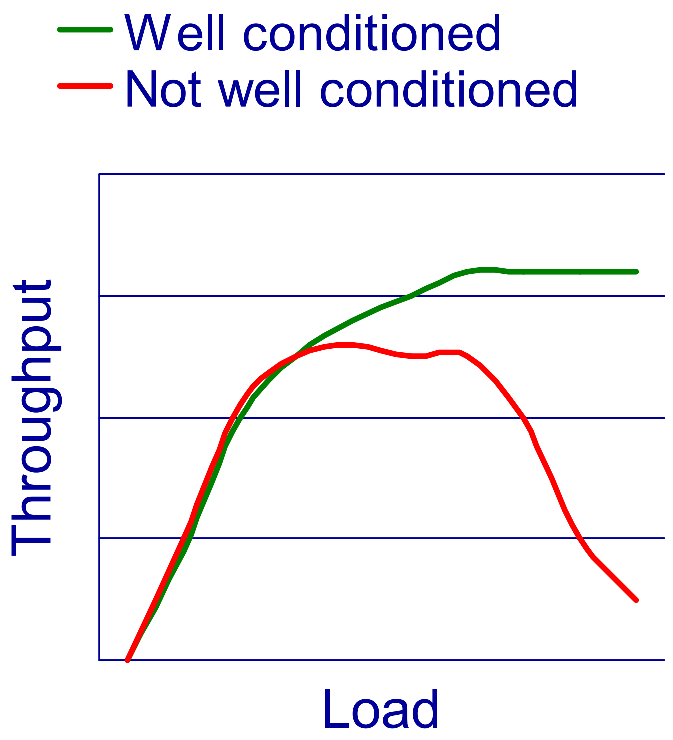
```
Master:
    loop {
        wait until an incoming connection
        equeue(q, new_connection);
        wakeup(q);
    }

Worker:
    loop {
        waiton(q);
        new_connection = dequeue(q);
        service  new_connection;
    }
```

OS

Thread or Process Pool

- Creating a thread or process for each unit of work (e.g., user request) is dangerous
  - High overhead to create & delete thread/process
  - Can exhaust CPU & memory resource
- Thread/process pool controls resource use
  - Allows service to be well conditioned.



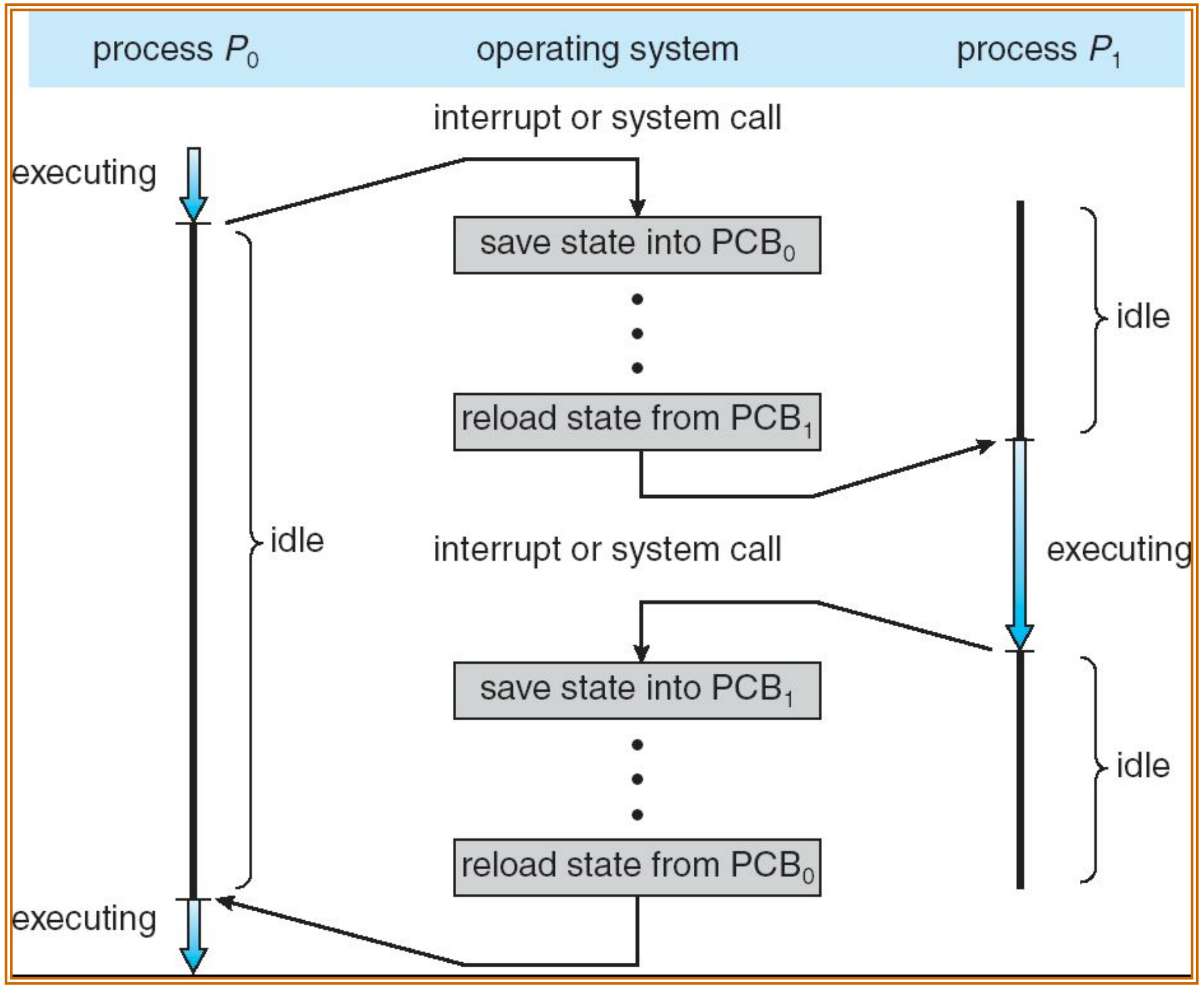
Outline

- What is a Process?
- Process Control Block
- Process Life Cycle
- The Concept of Thread
- Example Multithreaded Programs
- Thread Implementations
- Context Switch
- Process Control

Context Switch

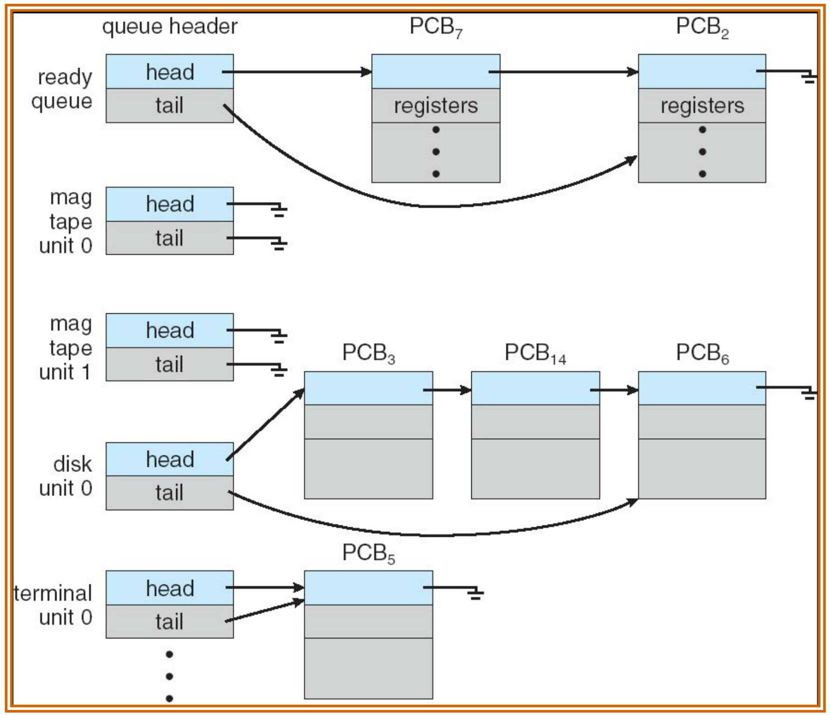
- Stop current running process (move from Running to another state) and schedule another process (put to Running state)
  - Must save various portions of the process context before switching.
  - Must be able to restore them later so that the process cannot tell that it was ever suspended.
  - Must be fast (context switches are very frequent)
- What context needs to be saved?
  - Registers (PC, SP, …), CPU states, …
  - Sometimes can be time-consuming and we should avoid if possible

Context Switch Illustration



Keeping Track of Processes

- OS has PCBs for active processes.
- OS puts PCB on an appropriate queue.
  - Ready to run queue.
  - Waiting for I/O queue (Queue per device).
  - Zombie queue.



Outline

- What is a Process?
- Process Control Block
- Process Life Cycle
- The Concept of Thread
- Example Multithreaded Programs
- Thread Implementations
- Context Switch
- Process Control
  - fork()
  - exec()
  - wait()
  - exit()

How Do Programmers Use Processes?

- How to build a fast, multi-process web server
  - Main process waits for a network connection
  - Main process accepts connection. OS represents open connection with a FILE DESCRIPTOR
  - Main process starts a new process for this connection
  - Main process must pass new process the file descriptor for the open connection
- Simple CreateProcess system call is insufficient
  - Process is program + process state
  - Process state can be as little as initial stack contents, or anything in the PCB (open files, network connections, security credentials)

The Genius of Separating Fork/Exec

- Life with `CreateProcess(filename);`
  - But I want to close all file descriptors in the child.
    `CreateProcess(filename, CLOSE_FD);`
  - And I want to change the child's environment.
    `CreateProcess(filename, CLOSE_FD, new_envp);`
  - Etc.
- **`fork()`** = split this process into 2 (new PID)
- **`exec()`** = overlay this process with new program
  (PID does not change)

The Genius of Separating Fork/Exec

- Decoupling fork and exec lets you do anything to the child's process environment without adding it to the CreateProcess API.

```
int pid = fork();                    // create a child
if(pid == 0) {                       // child continues here
    // Do anything (unmap memory, close net connections…)
    exec("program", argc, argv0, argv1, …);
}
```

- **`fork()`** creates a child process that inherits:
  - identical copy of all parent's variables & memory
  - identical copy of all parent's CPU registers (except one)
- Parent and child execute at the same point after **`fork()`** returns:
  - for the child, fork() returns 0
  - for the parent, fork() returns the process identifier of the child
  - fork() return code a convenience, could always use getpid()

Unix fork() example

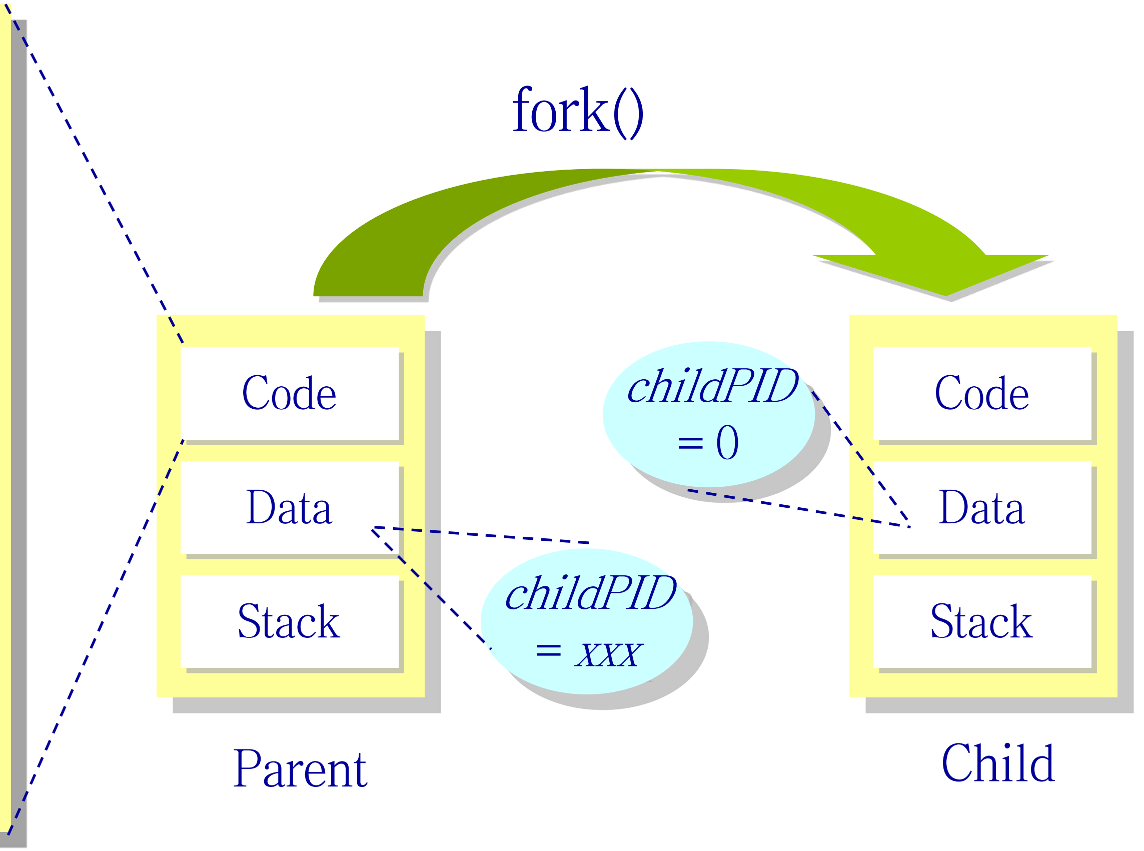
- The execution context for the child process is a *copy* of the parent's context at the time of the call
  - fork() returns child PID in parent, and 0 in child

```
main {
  int childPID;
  S1;

  childPID = fork();

  if(childPID == 0)
   <code for child process>
  else {
    <code for parent process>
    wait();
  }

  S2;
}
```



General Purpose Process Creation

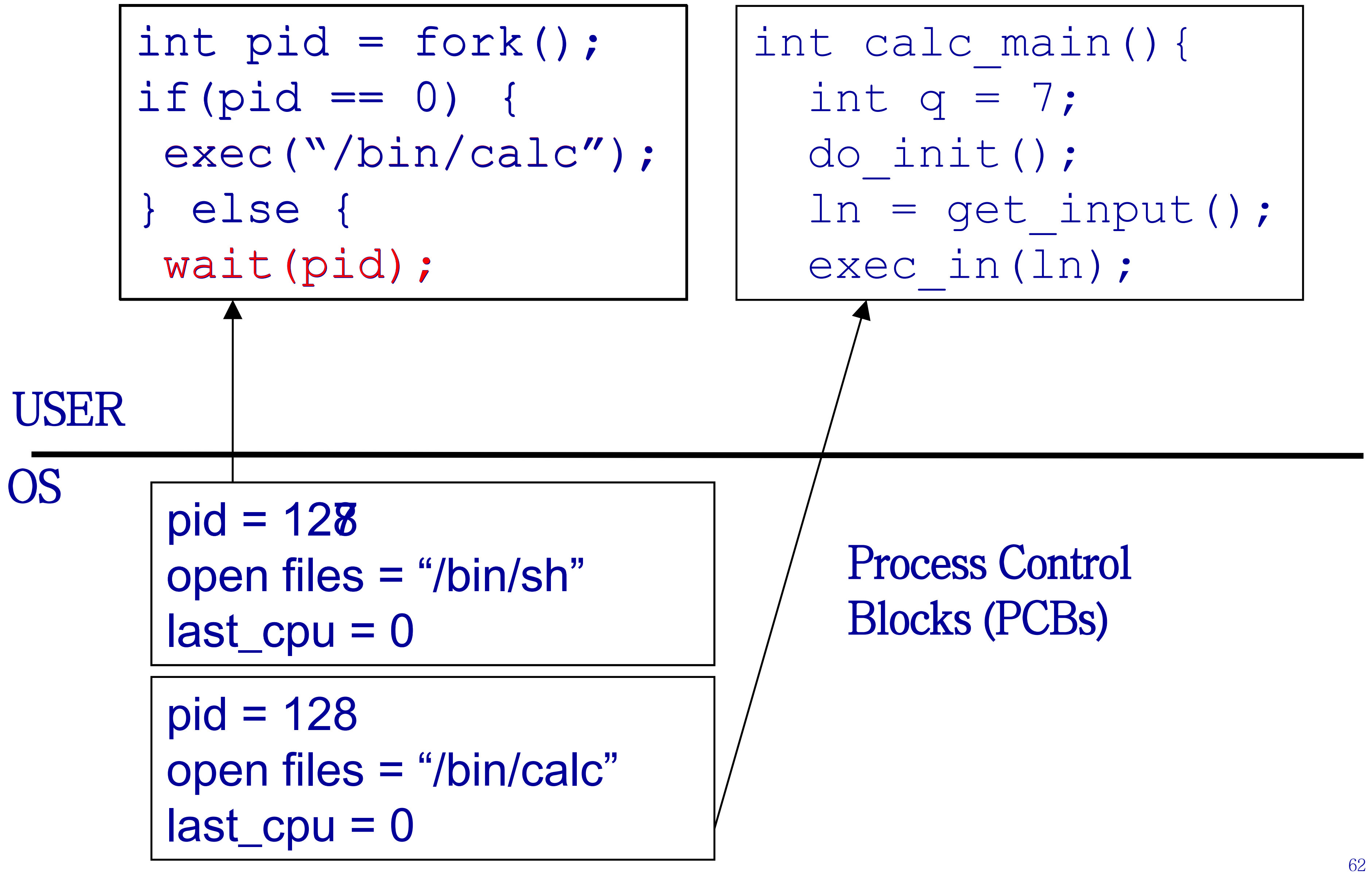
In the parent process:

```
main()
…
int pid = fork();                          // create a child
if(pid == 0) {                             // child continues here
      exec_status = exec("calc", argc, argv0, argv1, …);
     printf("Why would I execute?");
}
else {                                     // parent continues here
    printf("Whose your daddy?");
    …
   child_status = wait(pid);
}
```

C Program Forking Separate Process

```
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 */
                execlp("/bin/ls", "ls", NULL);
      }
      else { /* parent process */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
      }
}
```

A shell forks and then execs a calculator



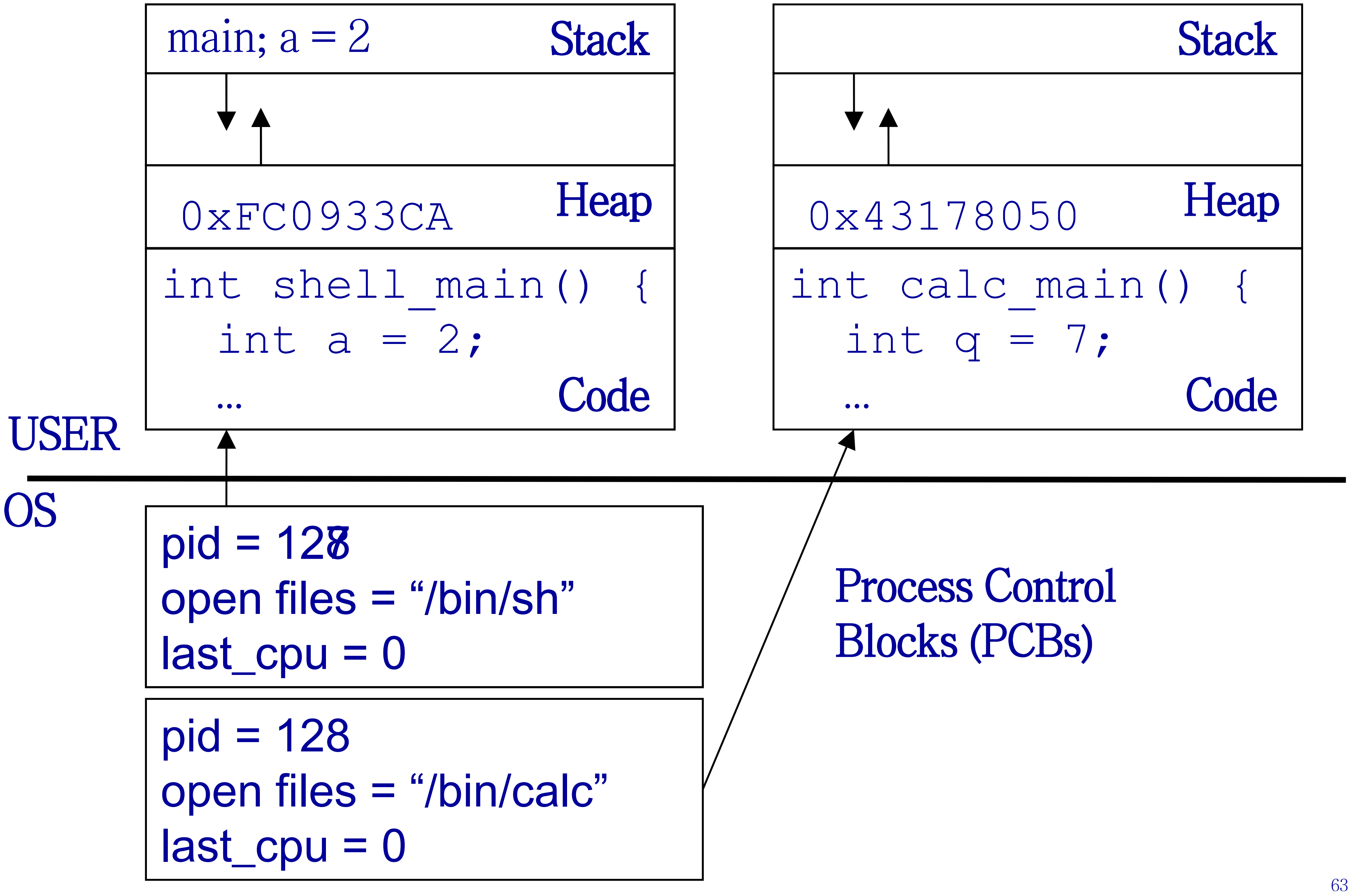
```
int pid = fork();
if(pid == 0) {
 exec("/bin/calc");
} else {
 wait(pid);
```

```
int calc_main(){
  int q = 7;
  do_init();
  ln = get_input();
  exec_in(ln);
```

pid = 127
open files = "/bin/sh"
last_cpu = 0

pid = 128
open files = "/bin/calc"
last_cpu = 0

A shell forks and then execs a calculator



Program Loading: exec()

- The exec() call allows a process to “load” a different program and start execution at main (actually _start).
- It allows a process to specify the number of arguments (argc) and the string argument array (argv).
- If the call is successful
  - it is the same process …
  - but it runs a different program !!
- Code, stack & heap is overwritten
  - Sometimes memory mapped files are preserved.

At what cost, fork()?

- Simple implementation of fork():
  - allocate memory for the child process
  - copy parent's memory and CPU registers to child's
  - *Expensive* !!
- In 99% of the time, we call exec() after calling fork()
  - the memory copying during fork() operation is useless
  - the child process will likely close the open files & connections
  - overhead is therefore high
  - Why not combine them in one call (OS/2, Windows)?
- vfork()
  - a system call that creates a process "without" creating an identical memory image
  - sometimes called lightweight fork()
  - child process should call exec() almost immediately
  - No use now if we use Copy on Write (COW) technology

The wait() System Call

- A child program returns a value to the parent, so the parent must arrange to receive that value

- The wait() system call serves this purpose
  - it puts the parent to sleep waiting for a child's result
  - when a child calls exit(), the OS unblocks the parent and returns the value passed by exit() as a result of the wait call (along with the pid of the child)
  - if there are no children alive, wait() returns immediately
  - also, if there are zombies waiting for their parents, wait() returns one of the values immediately (and deallocates the zombie)

Orderly Termination: exit()

- After the program finishes execution, it calls *exit*()
- This system call:
  - takes the "result" of the program as an argument
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - checks if parent is alive:
    - If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the zombie/defunct state
    - If not, it deallocates all data structures, the process is dead
  - cleans up all waiting zombies
- Process termination is the ultimate garbage collection (resource reclamation).

Process Control

OS must include calls to enable special control of a process:

- Priority manipulation:
  - nice(), which specifies base process priority (initial priority)
  - In UNIX, process priority decays as the process consumes CPU
- Debugging support:
  - ptrace(), allows a process to be put under control of another process
  - The other process can set breakpoints, examine registers, etc.
- Alarms and time:
  - Sleep puts a process on a timer queue waiting for some number of seconds, supporting an alarm functionality

Tying it All Together: The Unix Shell

```
while(! EOF) {
read input
handle regular expressions
int pid = fork();                           // create a child
if(pid == 0) {                              // child continues here
    exec("program", argc, argv0, argv1, …);
}
else {                                      // parent continues here
…
}
```

- Translates <CTRL-C> to the kill() system call with SIGKILL
- Translates <CTRL-Z> to the kill() system call with SIGSTOP
- Allows input-output redirections, pipes, and a lot of other stuff that we will see later