CSI 4500 Operating Systems

Lecture 2

Processes and Threads

Today's Goals

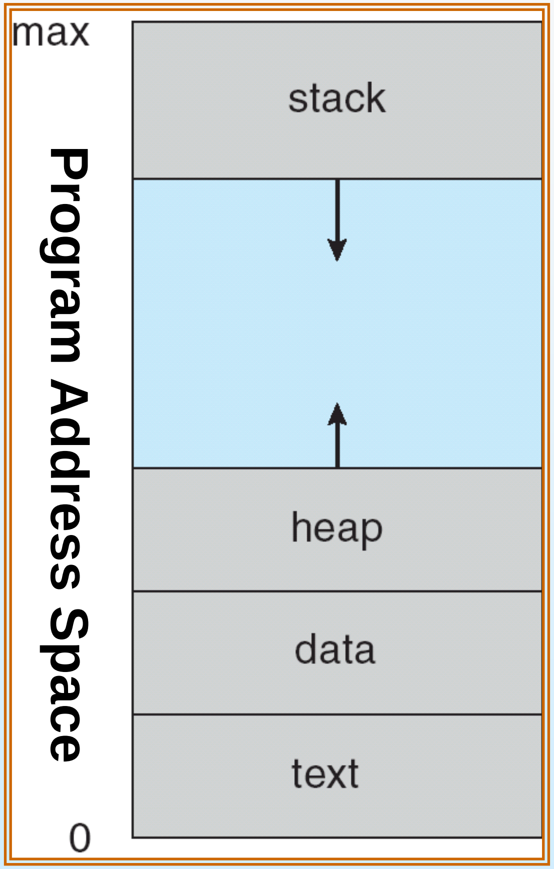
- What are Processes?
  - Process Concept
  - Process Scheduling
  - Operations on Processes
- Understanding Threads
  - Thread Dispatching
  - Beginnings of Thread Scheduling

Process Concept

- Process – a program in execution
  - Operating system abstraction to represent what is need to run a program
- A process includes:
  - Sequential Program Execution Instruction Stream
    - Code executed as a *single, sequential* stream of execution
    - Includes State of CPU registers
  - Protected Resources:
    - Main Memory State (contents of Address Space)
    - I/O state (i.e. file descriptors)
- Processes can be described as either:
  - **I/O-bound process**
    - spends more time doing I/O than computations
    - many short CPU bursts
  - **CPU-bound process**
    - spends more time doing computations
    - few very long CPU bursts

Process in Memory

- Address space ⇒ the set of accessible addresses + state associated with them:
  - For a 32-bit processor there are 2^{32} bits addresses
- Read or write operation to an address?
  - Regular memory access
  - Exception
  - I/O operation



Memory Layout of a Process

- **Text segment** contains the machine-language instructions of the program run by a process.
- **Initialized data segment** contains global and static variables that explicitly initialized.
- **Uninitialized data segment** contains global and static variables that not explicitly initialized
- **Stack** contains stack frames. One stack frame is allocated for each currently called function. A frame stores the function's local variables (automatic variables), arguments, and return value.
- **Heap** is an area of memory can be dynamically allocated at run time.

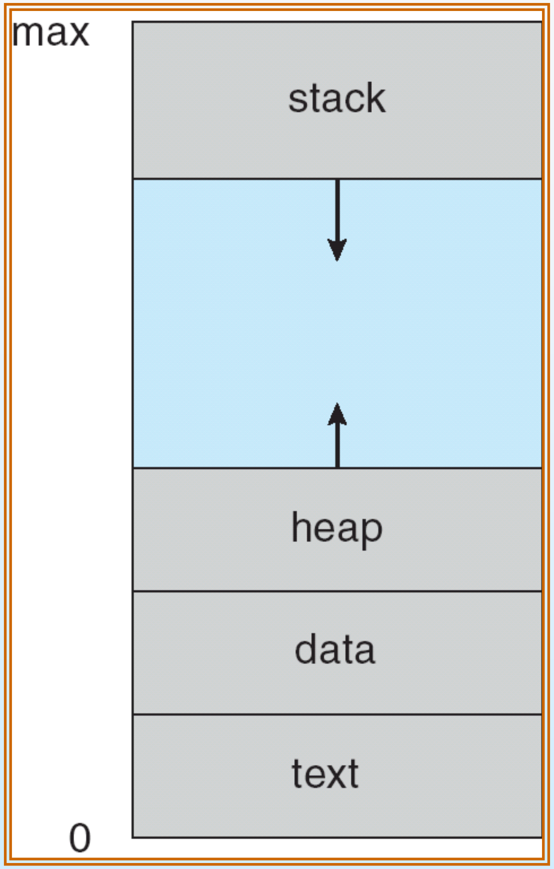
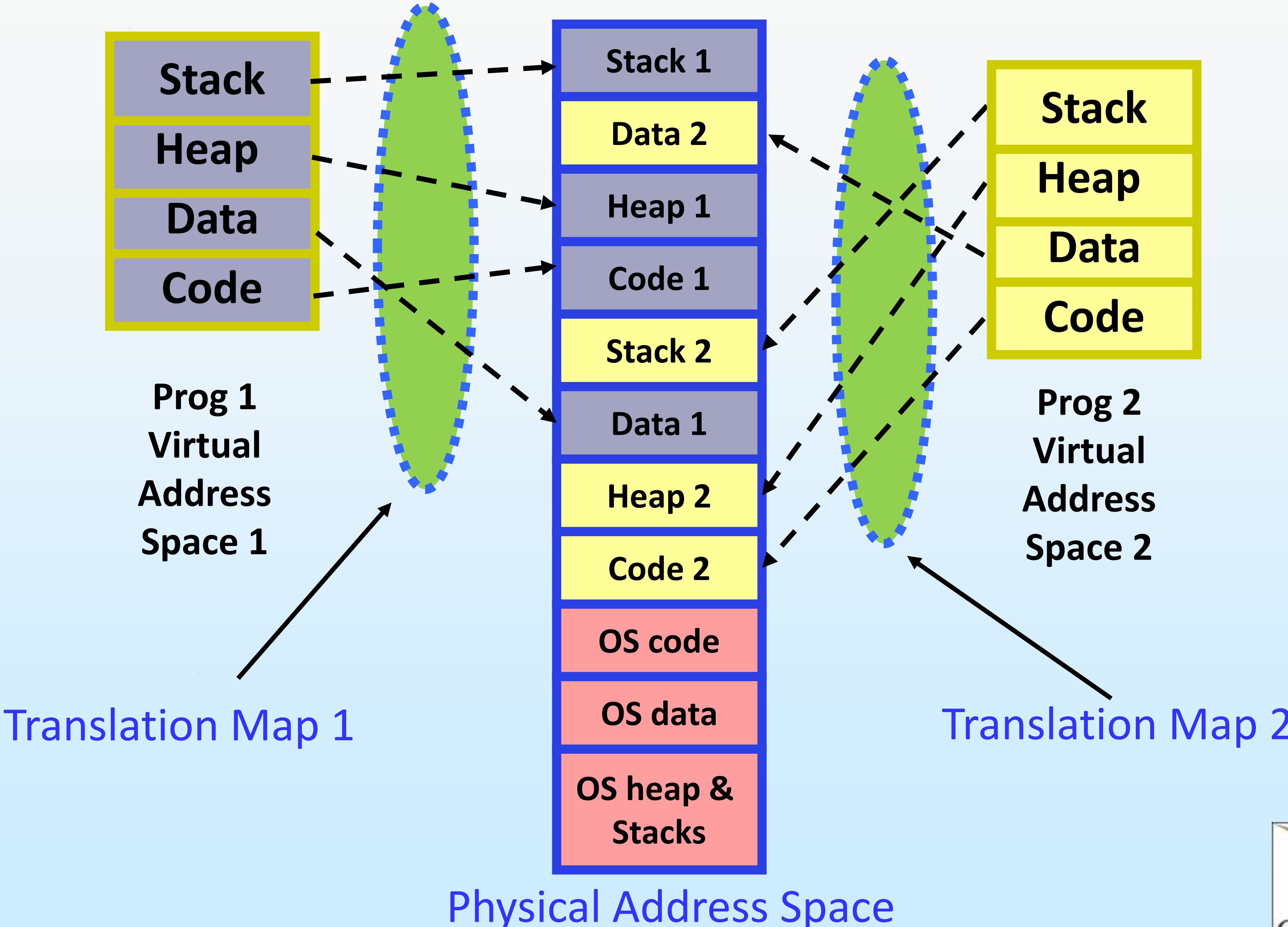
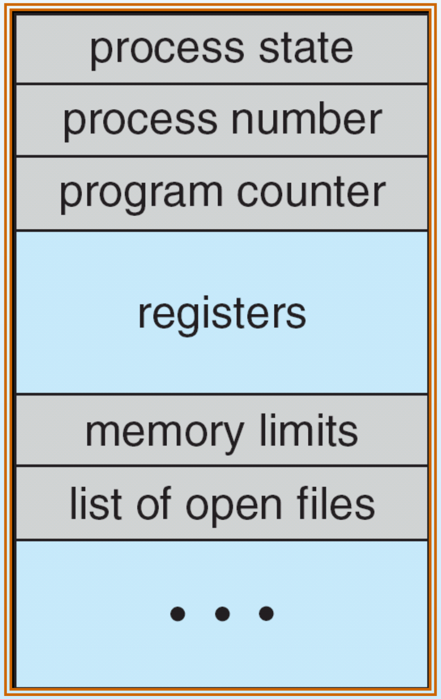


Illustration of Separate Address Space



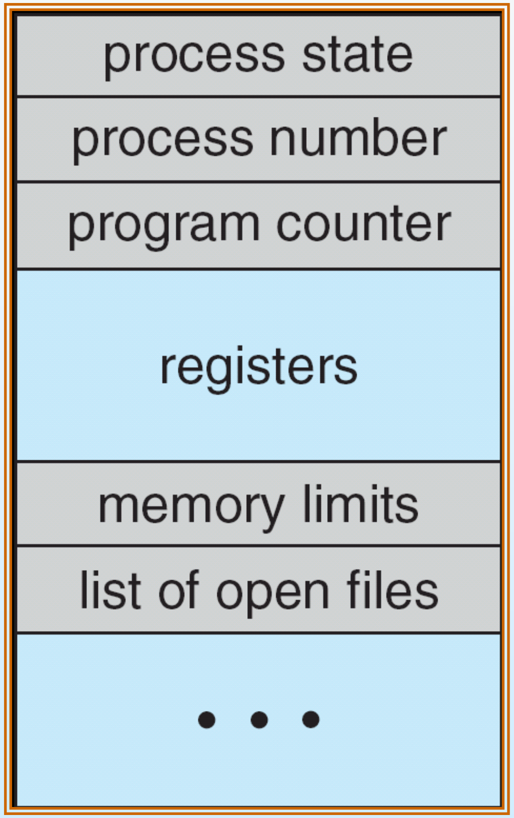
Process Control Block (PCB)

- The current state of process held in a process control block (PCB):
  - This is a "snapshot" of the execution and protection environment
  - Information associated with each process
    - Process state
    - Program counter
    - CPU registers
    - CPU scheduling information
    - Memory-management information
    - Accounting information
    - I/O status information

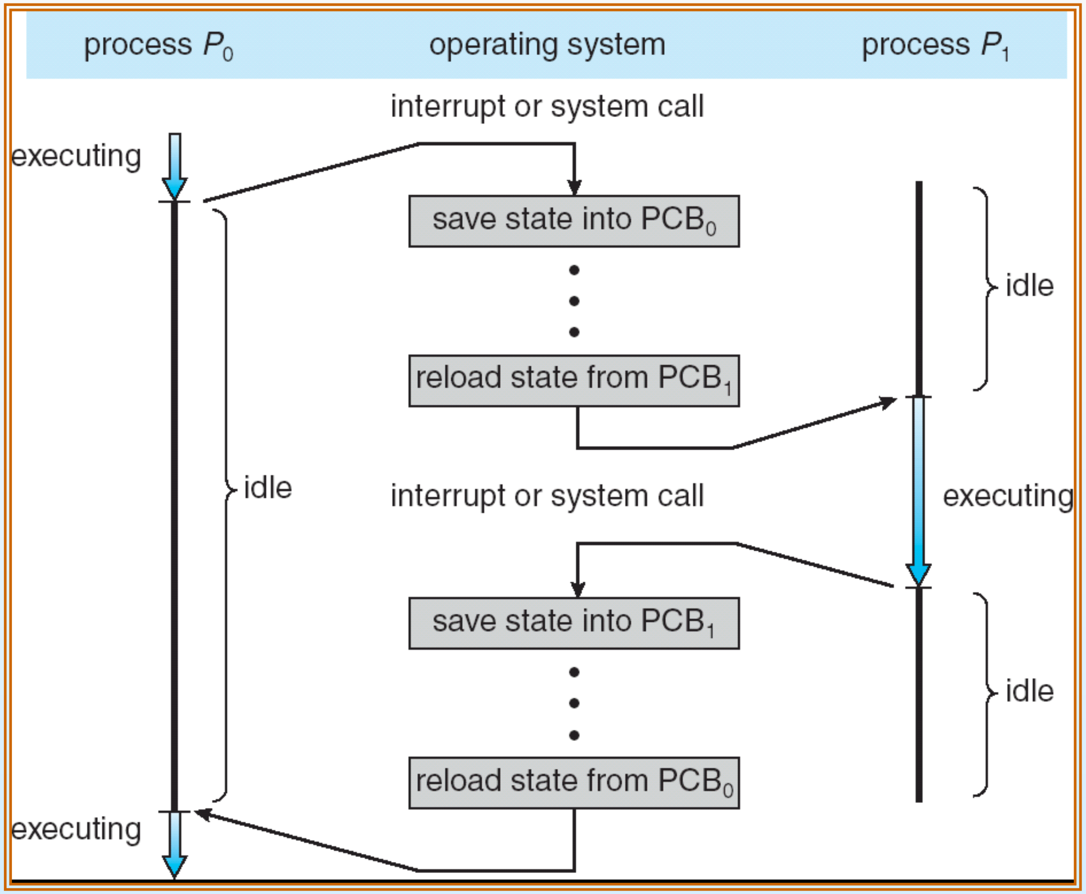


How do we multiplex processes?

- Process control block (PCB):
  - Only one PCB active at a time
- Assign CPU time to different processes (Scheduling):
  - Only one process "running" at a time
  - Give more time to important processes (Priority)
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
  - Sample mechanisms:
    - Memory Mapping: Give each process their own address space
    - Kernel/User duality: Arbitrary multiplexing of I/O through system calls



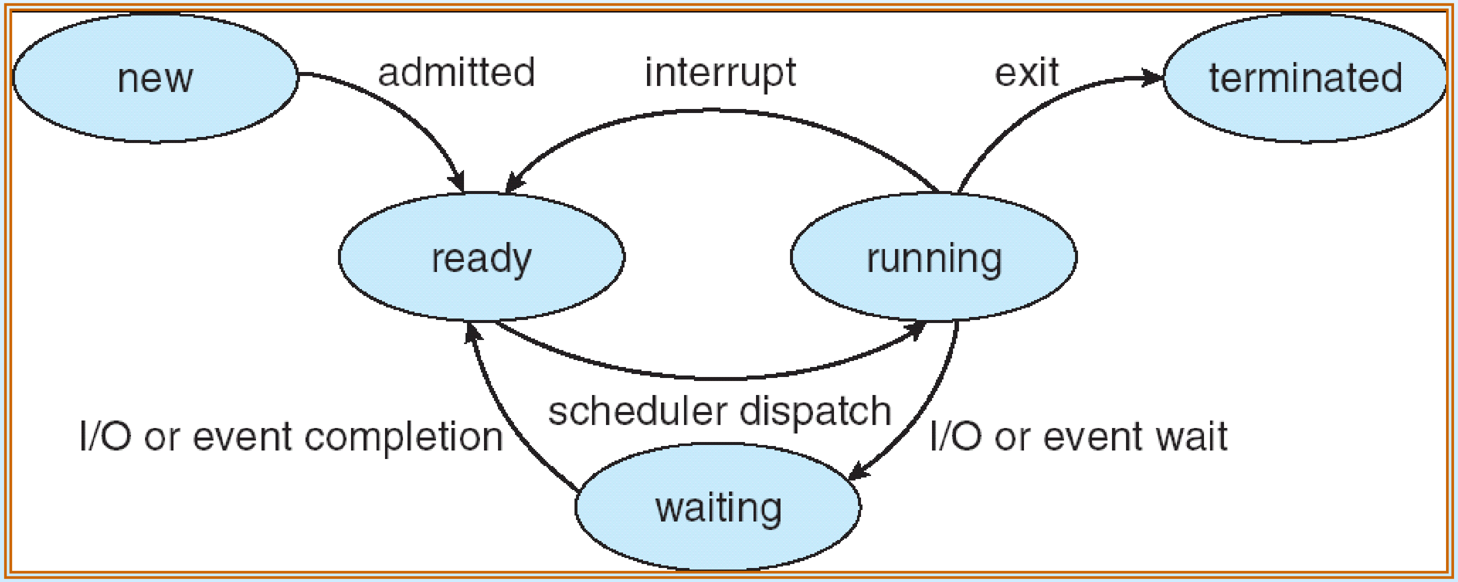
CPU Switch From Process to Process



- This is also called a "context switch"
- During context switch system does no useful work
  - Overhead dependent on hardware support
  - What are they?

Process State

- As a process executes, it changes ***state***
  - **new**: The process is being created
  - **ready**: The process is waiting to run
  - **running**: Instructions are being executed
  - **waiting**: Process waiting for some event to occur
  - **terminated**: The process has finished execution



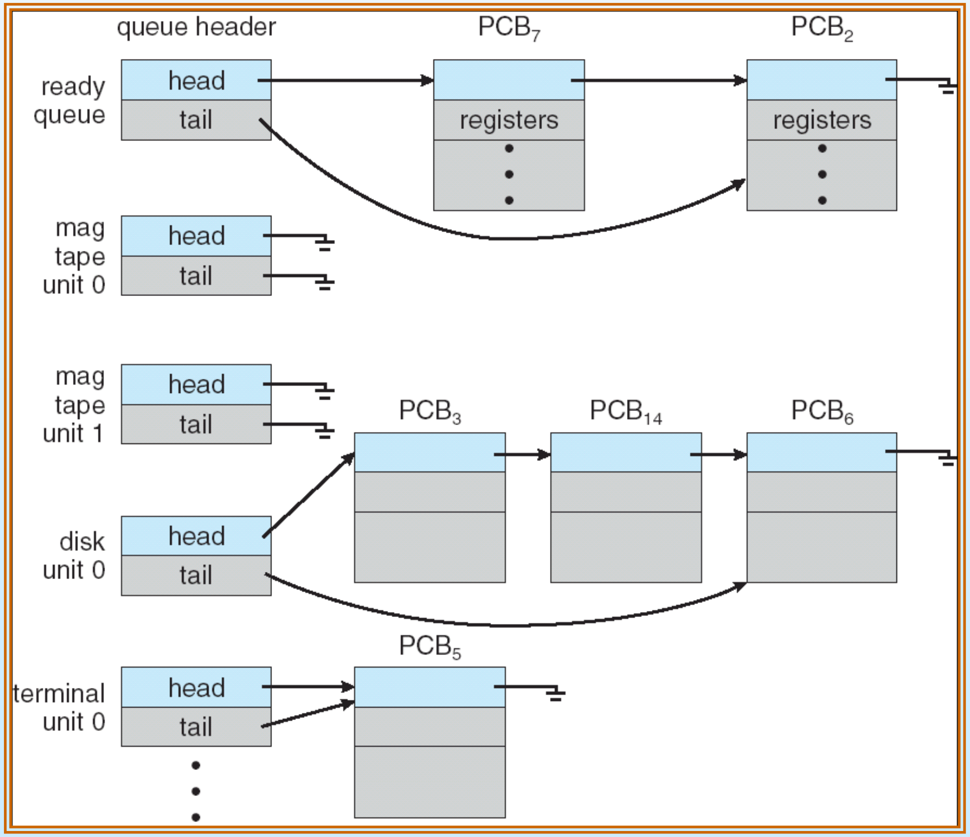
Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- **Scheduling**

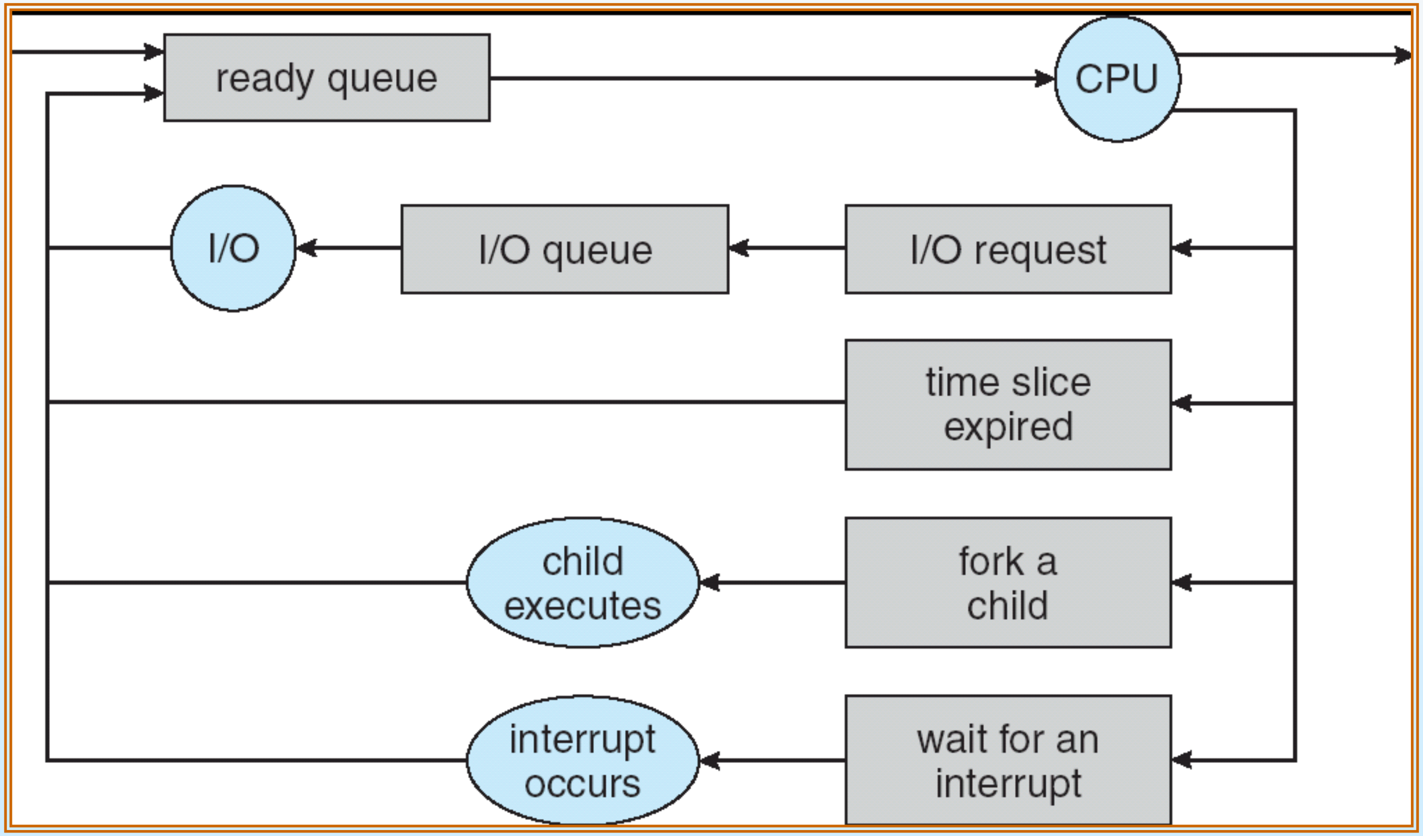
  Processes (PCBs)

  migrate among the

  various queues

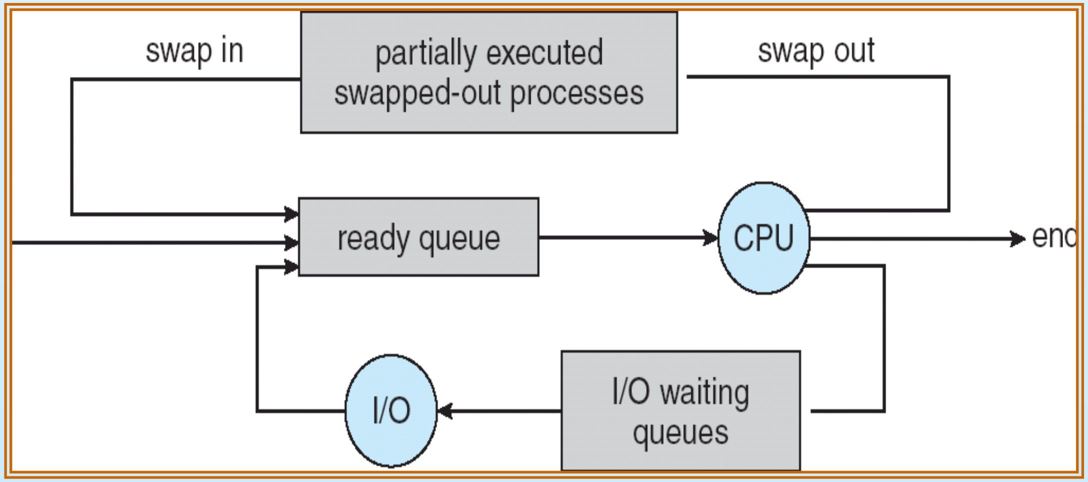


Representation of Process Scheduling



Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked very infrequently
  - The long-term scheduler controls the *degree of multiprogramming*
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)
- **Medium-term scheduler**

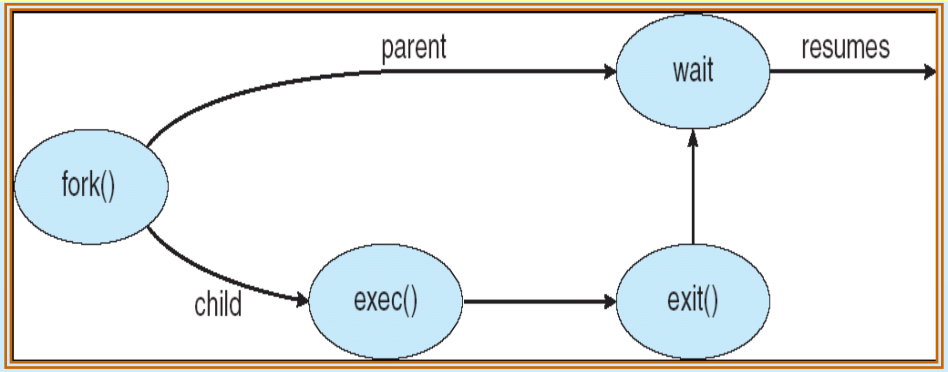


How to Create a Process?

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Must construct new PCB
- Resource sharing strategies
  - Parent and children share all resources (I/O states, address space information)
  - Children share subset of parent's resources
  - Parent and child share no resources (Unix *exec*())
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation – C example

```
int main() {
pid_t  pid;
    pid = fork(); /* fork another process */
    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);
    }
}
```



Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
      - All children terminated - *cascading termination*

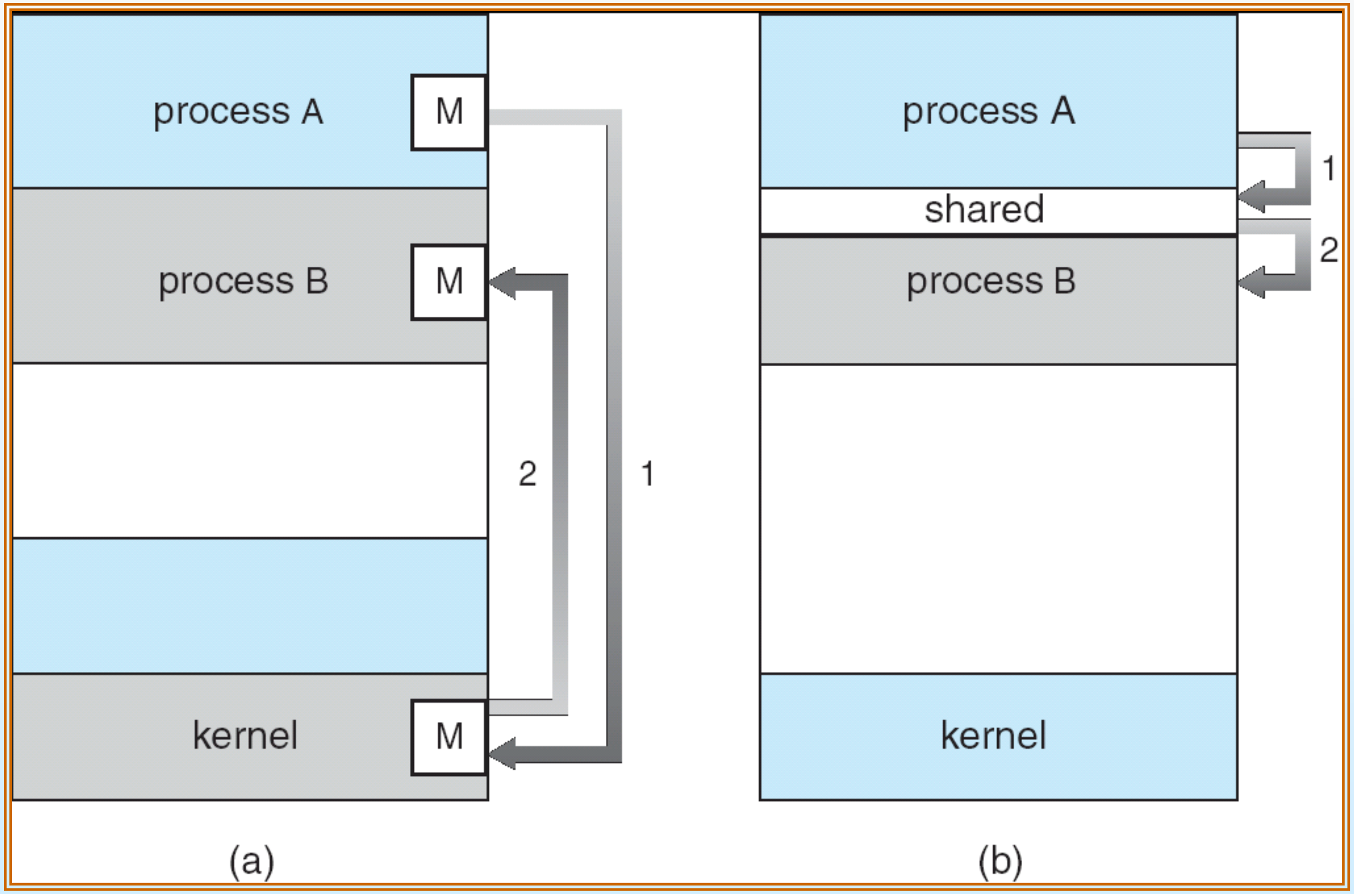
Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- Disadvantages?
  - High Creation/memory Overhead
  - (Relatively) High Context-Switch Overhead

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – **processes communicate with each other without resorting to shared variables**
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Communications Models



(a) Message Passing (b) Shared-Memory Mapping

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q

- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - **send**(*A, message*) – send a message to mailbox A
  - **receive**(*A, message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver.
    - Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking

- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available

- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** has the sender send the message and continue
  - **Non-blocking receive** has the receiver receive a valid message or null

Buffering

- Queue of messages attached to the link
  - Zero capacity – 0 messages
    - *Sender must wait for receiver (rendezvous)*
  - Bounded capacity – finite length of *n* messages
    - *Sender must wait if link full*
  - Unbounded capacity – infinite length
    - *Sender never waits*

Bounded-Buffer – Shared Memory Solution

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Share Data

```
while (true) {
    /* Produce an item */
    while (((in = (in + 1) % BUFFER SIZE count)   == out)
        ;   /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```

Insert()

```
while (true) {
    while (in == out)
        ; // do nothing -- nothing to consume
    item = buffer[out]; // remove an item from the buffer
    out = (out + 1) % BUFFER SIZE;
    return item;
}
```

Remove()

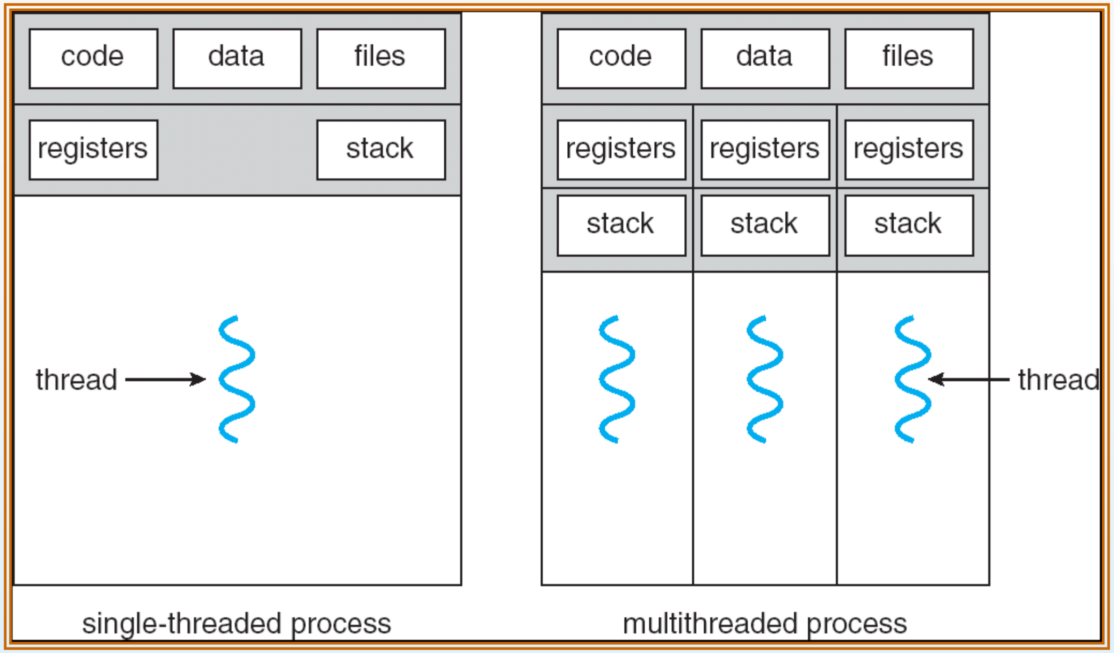
Process Summary

- Processes have two parts
  - Sequence of Execution Stream
  - Resources
- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (`yield()`, I/O operations) or involuntary (timer, other interrupts)
- Protection accomplished restricting access:
  - Memory mapping isolates processes from each other
  - Dual-mode for isolating I/O, other resources
- Cooperating of Processes
  - Shared Memory Communication
  - Message Communication

Multiple Threads within a Process

- Process: Operating system abstraction to represent what is needed to run a single (**multithreaded**) program
- Two parts:
  - Multiple Threads
    - Each thread is a *single, sequential stream of execution*
  - Protected Resources:
    - Main Memory State (contents of Address Space)
    - I/O state (i.e. file descriptors)
- Why separate the concept of a thread from that of a process?
  - Heavyweight Process ≡ Process with only one thread

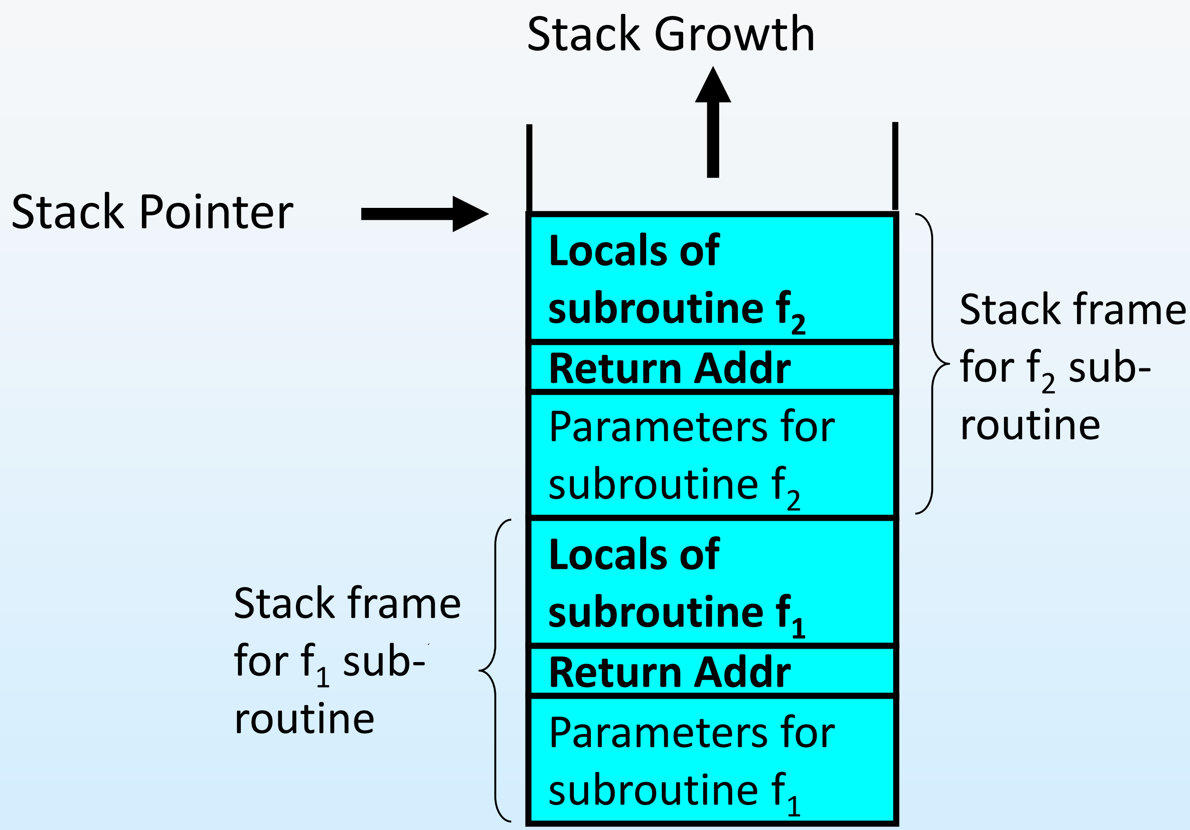
Single and Multithreaded Processes



- Threads encapsulate concurrency
  - “Control” component of a process
- Address spaces encapsulate protection
  - “Passive” component of a process

Recall: Call Stack Example

```
f1(int tmp) {
  …
  f2(a, b);
  …
}
f2(int a,
float b) {
  int i;
  …
}
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Single-Threaded Example

- Imagine the following Pseudo program:

```
main() {
        …
        PrintDigitPI("pi.txt");
        PrintDigitE("e.txt");
        …
}
```

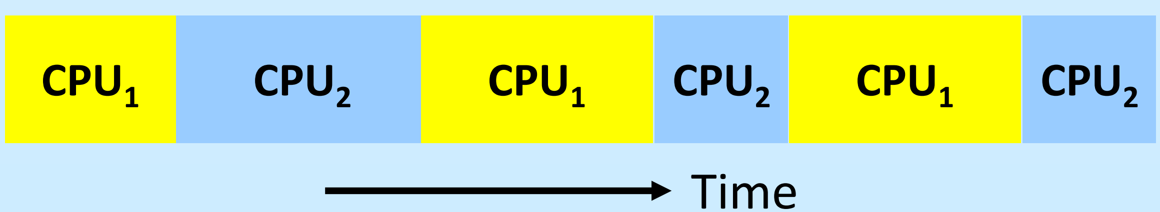
- What is the behavior here?
  - Program would never print out **E** (mathematical constant)
  - Why?
    - `PrintDigitPI()` would never finish

Use of Threads

- Version of program with Threads:

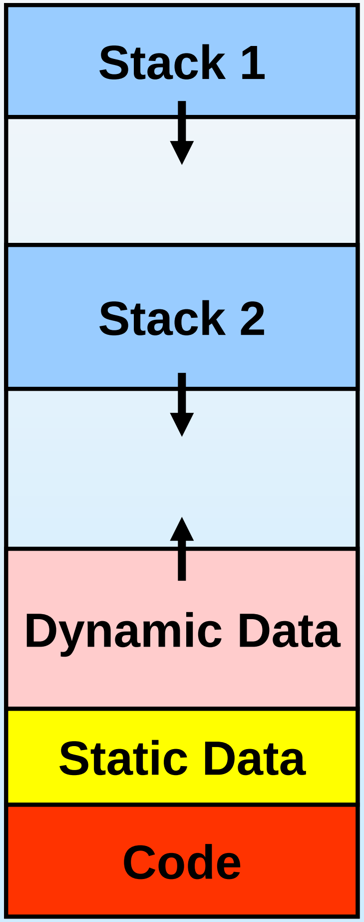
```
main() {
        …
        CreateThread(PrintDigitPI("pi.txt"));
        CreateThread(PrintDigitE("e.text"));
        …
}
```

- What does "`CreateThread()`" do?
  - Start independent thread running given procedure
- What is the program behavior now?
  - Now, you would actually see the both **PI** and **E**
  - This *should* behave as if there are two separate CPUs



Memory View of Two Threads

- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks
- Questions:
  - How do stacks locate relative to each other?
  - What maximum size for the stacks?
  - What happens if threads violate?
  - How might you catch violations?

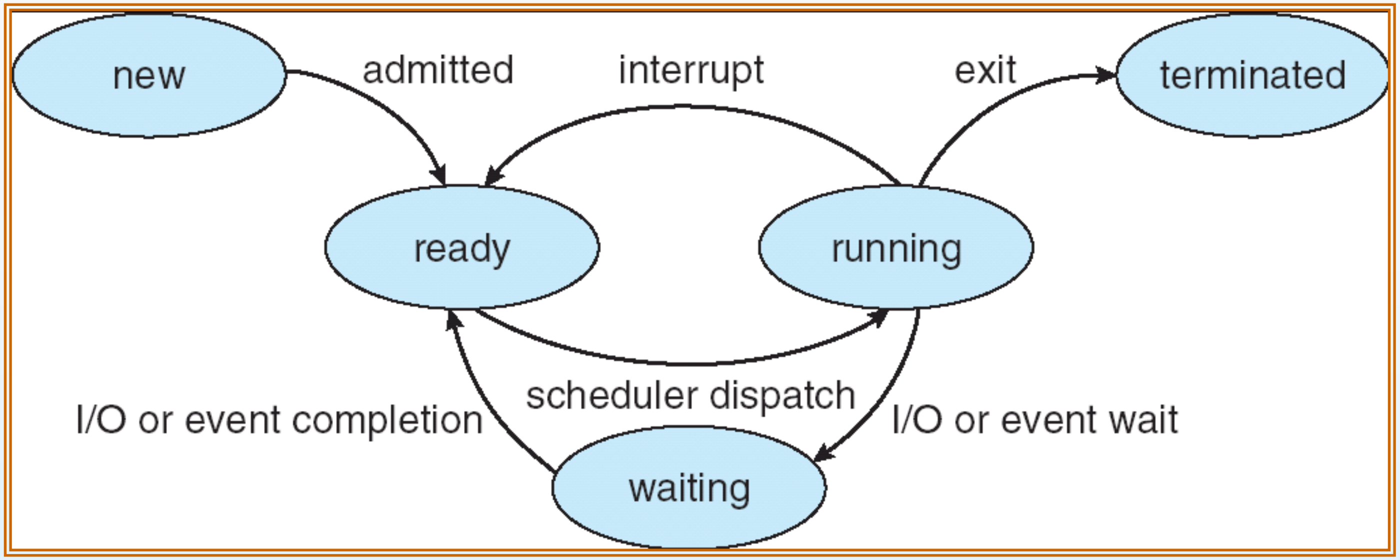


Address Space

Thread State

- Each Thread has a *Thread Control Block* (TCB)
  - **Execution State**: CPU registers, program counter, pointer to stack
  - **Scheduling info**: State, priority, CPU time
  - Various Pointers (for implementing scheduling queues)
  - Etc.
- In Java: “Thread” is a class that includes the TCB
- OS Keeps track of TCBs in protected memory
  - In Array, or Linked List, or …

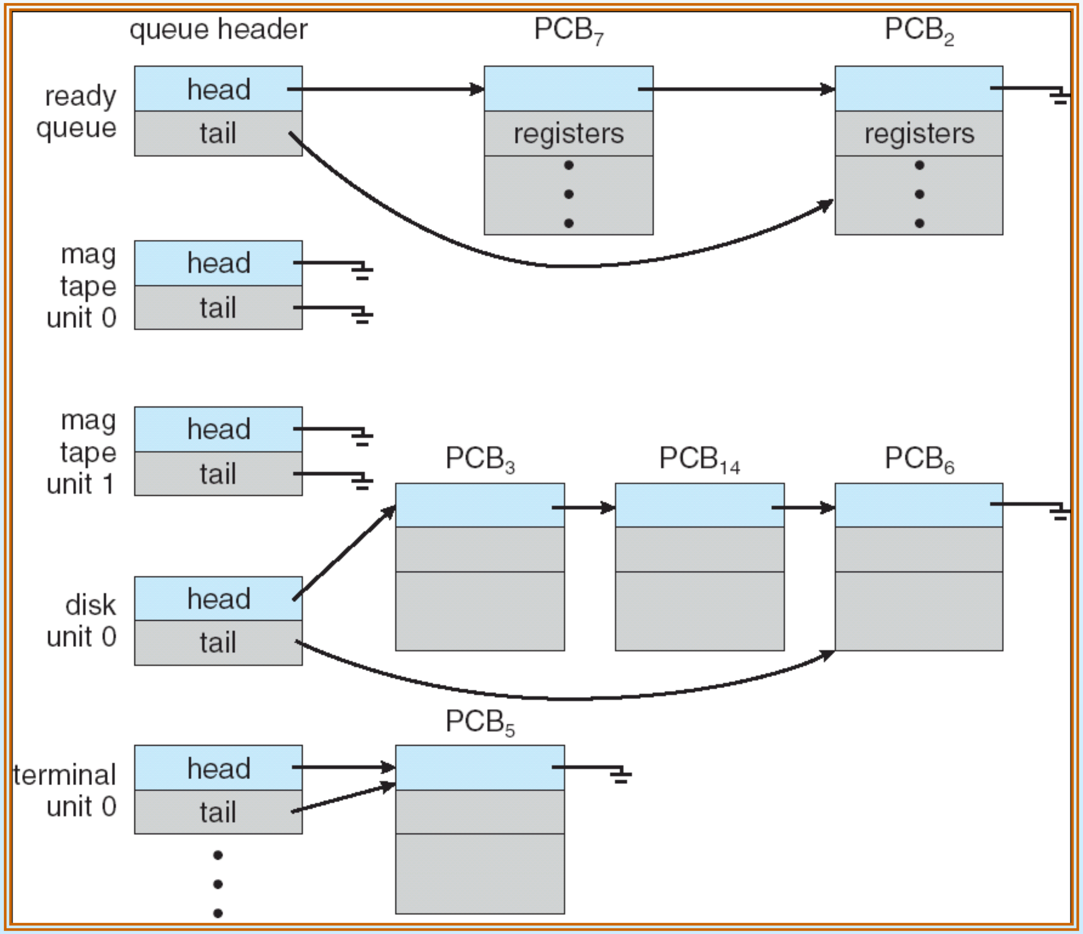
Recall: Lifecycle of a Thread (or Process)



- As a thread executes, it changes state:
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
  - terminated: The thread has finished execution
- "Active" threads are represented by their TCBs
  - TCBs organized into queues based on their state

Ready Queue And Various I/O Device Queues

- Thread not running ⇒ TCB is in some other scheduler queues
  - Queues exist based on device/signal/condition
  - Each queue may have a different scheduler policy



OS operates flow

- Conceptual view of the operating system

```
Loop {
        RunThread();
        ChooseNextThread();
        SavecurrentTCB();
        LoadStateOfCPU(newTCB);
}
```

- This is an *infinite* loop
  - One could argue that this is all that the OS does
- When we ever exit this loop???

Running a thread

- How do I run a thread?
  - Load its state (registers, PC, stack pointer) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC

- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets *preempted*

Internal Events

- Blocking on I/O
  - The action of requesting I/O implicitly yields the CPU
- Waiting on a “signal” from other thread
  - Thread asks to wait and thus yields the CPU
- Thread executes a `yield()`
  - Thread volunteers to give up CPU

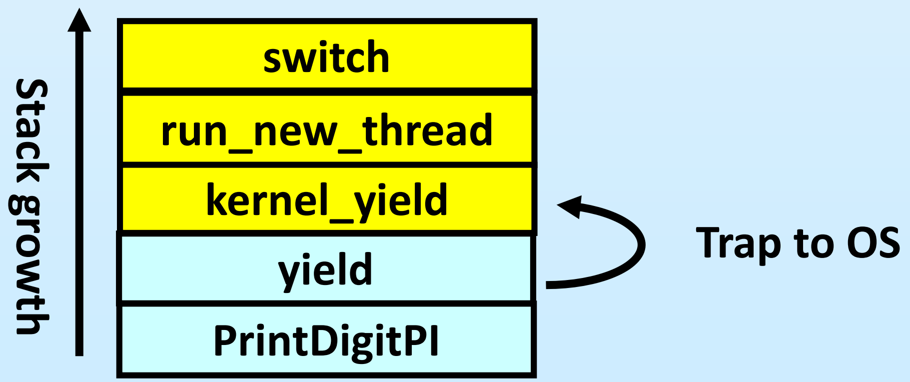
```
PrintDigitPI() {
    while(TRUE) {
        PrintNextDigit();
        yield();
    }
}
```

Stack for Yielding Thread

- How do we run a new thread?

```
run_new_thread() {
      newThread = PickNewThread();
      switch(curThread, newThread);
            …
}
```

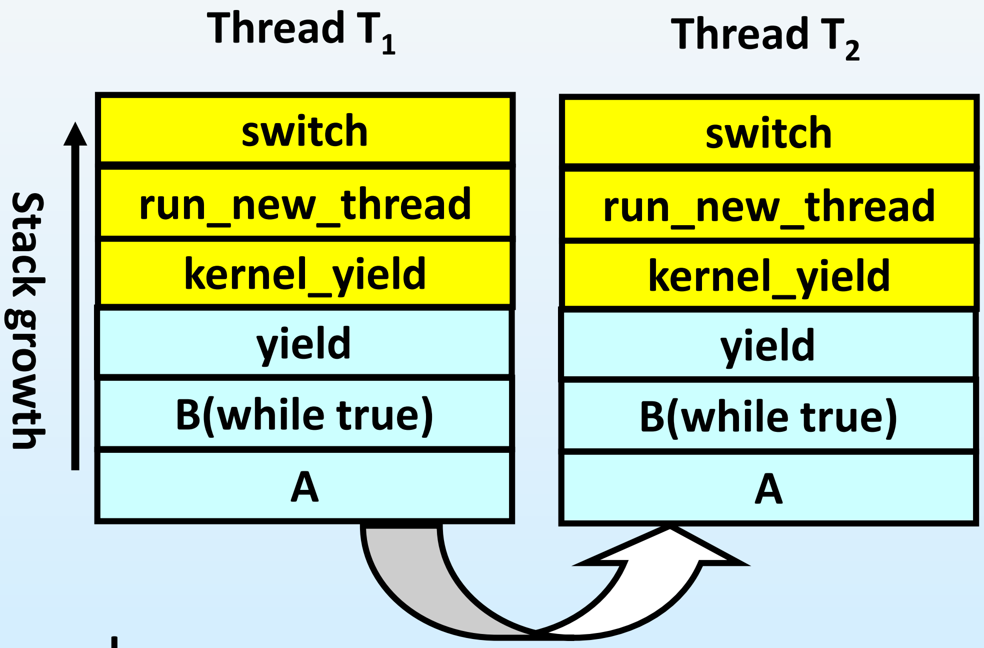
- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack
  - Maintain isolation for each thread



What do the stacks look like?

- Consider the following code blocks:

```
proc A() {
      B();
}
proc B() {
      while(TRUE){
            yield();
      }
}
```



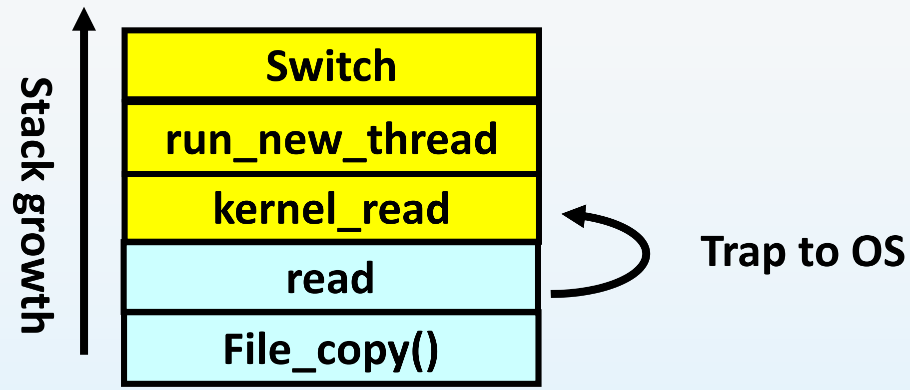
- Suppose we have two threads:
  - Threads T_1 and T_2

Saving/Restoring state

```
Switch(curThread,newThread) {
  /* Unload old thread */
  TCB[curThread].regs.r7 = CPU.r7;
                    …
  TCB[curThread].regs.r0 = CPU.r0;
  TCB[curThread].regs.sp = CPU.sp;
  TCB[curThread].regs.retpc = CPU.retpc; /*return addr*/

  /* Load and execute new thread */
  CPU.r7 = TCB[newThread].regs.r7;
                    …
  CPU.r0 = TCB[newThread].regs.r0;
  CPU.sp = TCB[newThread].regs.sp;
  CPU.retpc = TCB[newThread].regs.retpc;
  return; /* Return to CPU.retpc */
}
```

Thread blocks on I/O



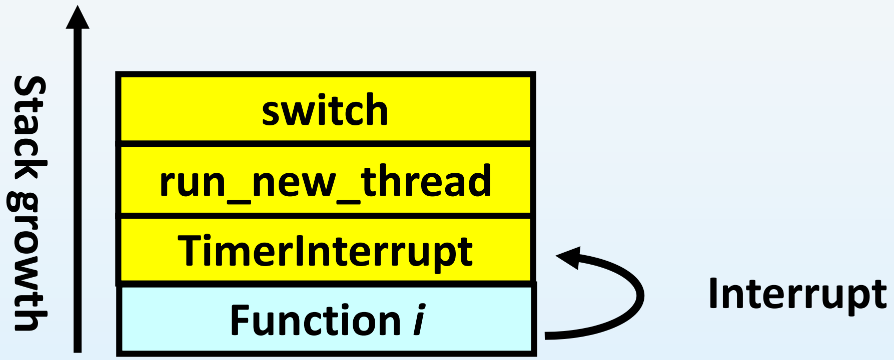
- What happens when a thread requests a block of data from the file system?
  - User code invokes a system call
  - Read operation is initiated
  - Run new thread/switch
- Thread communication similar
  - Wait for Signal/Join
  - Networking

External Events

- What happens if thread never does any I/O, never waits, and never yields control?
  - Could the `PrintDigitPI` program grab all resources and never release the processor?
    - What if it didn't print anything?
  - Must find way that dispatcher can regain control!
- Answer: Utilize External Events
  - Interrupts: signals from hardware or software that stop the running code and jump to kernel
  - Timer: like an alarm clock that goes off every some many milliseconds
- If we make sure that external events occur frequently enough, can ensure dispatcher runs

To Acquire Control via Interrupt

- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions



- Timer Interrupt routine:

```
TimerInterrupt() {
        DoPeriodicChecking();
        run_new_thread();
}
```

- I/O interrupt: same as timer interrupt except that `DoPeriodicChecking()` replaced by `ServiceIO()`.

Choosing a Thread to Run

- How does Dispatcher decide what to run next?
  - Zero ready threads – dispatcher loops
    - Alternative is to create an "idle thread"
    - Can put machine into low-power mode
  - Exactly one ready thread – easy
  - More than one ready thread: scheduling
- Possible priorities:
  - LIFO (last in, first out):
    - put ready threads on front of list, remove from front
  - FIFO (first in, first out):
    - Put ready threads on the tail of list, pick them from front
    - Fair policy
  - Priority queue:
    - keep ready list sorted by TCB priority field
  - Pick one at random

Threads Summary

- The state of a thread is contained in the TCB
  - Registers, PC, stack pointer
  - States: New, Ready, Running, Waiting, or Terminated
- Multithreading provides simple illusion of multiple CPUs
  - Switch registers and stack to dispatch new thread
  - Provide mechanism to ensure dispatcher regains control
- Switch routine
  - Can be very expensive if many registers
  - Must be very carefully constructed!
- Many scheduling options
  - Decision of which thread to run complex enough for complete lecture