COS 318: Operating Systems

Implementing Threads

Jaswinder Pal Singh and a Fabulous Course Staff
Computer Science Department
Princeton University

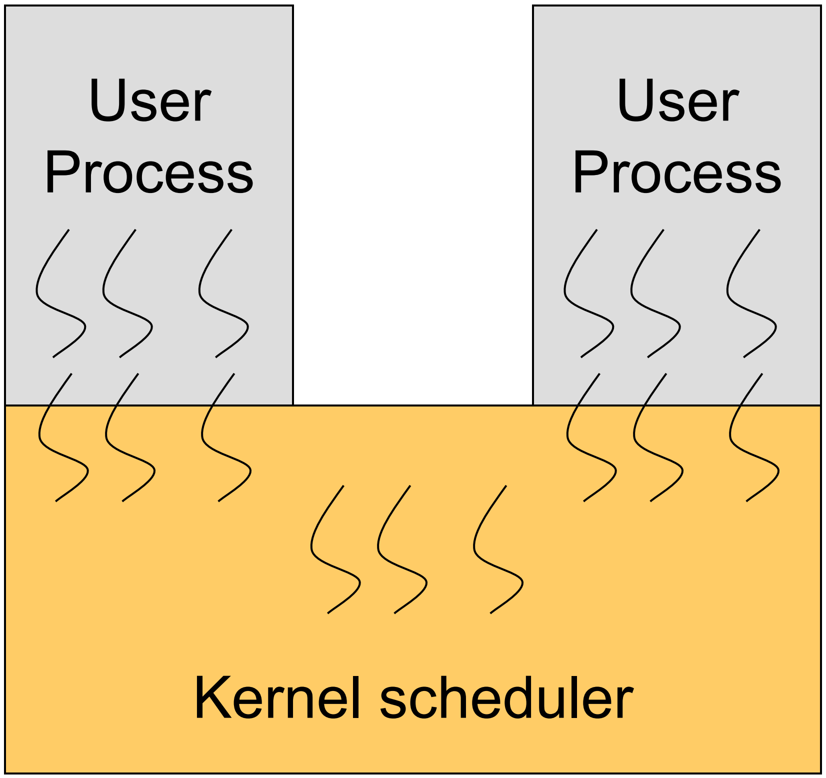
(http://www.cs.princeton.edu/courses/cos318/)

Today's Topics

- Thread implementation
  - Non-preemptive versus preemptive threads
  - Kernel vs. user threads

OS Scheduler

- Kernel consists of
  - Boot loader
  - BIOS
  - Key drivers
  - Threads
  - Scheduler
  - …
- Scheduler
  - Scheduler schedules threads on context switch
  - (Amounts to scheduling processes, when scheduler sees only one thread per process)
  - Uses a ready queue, to hold all ready threads



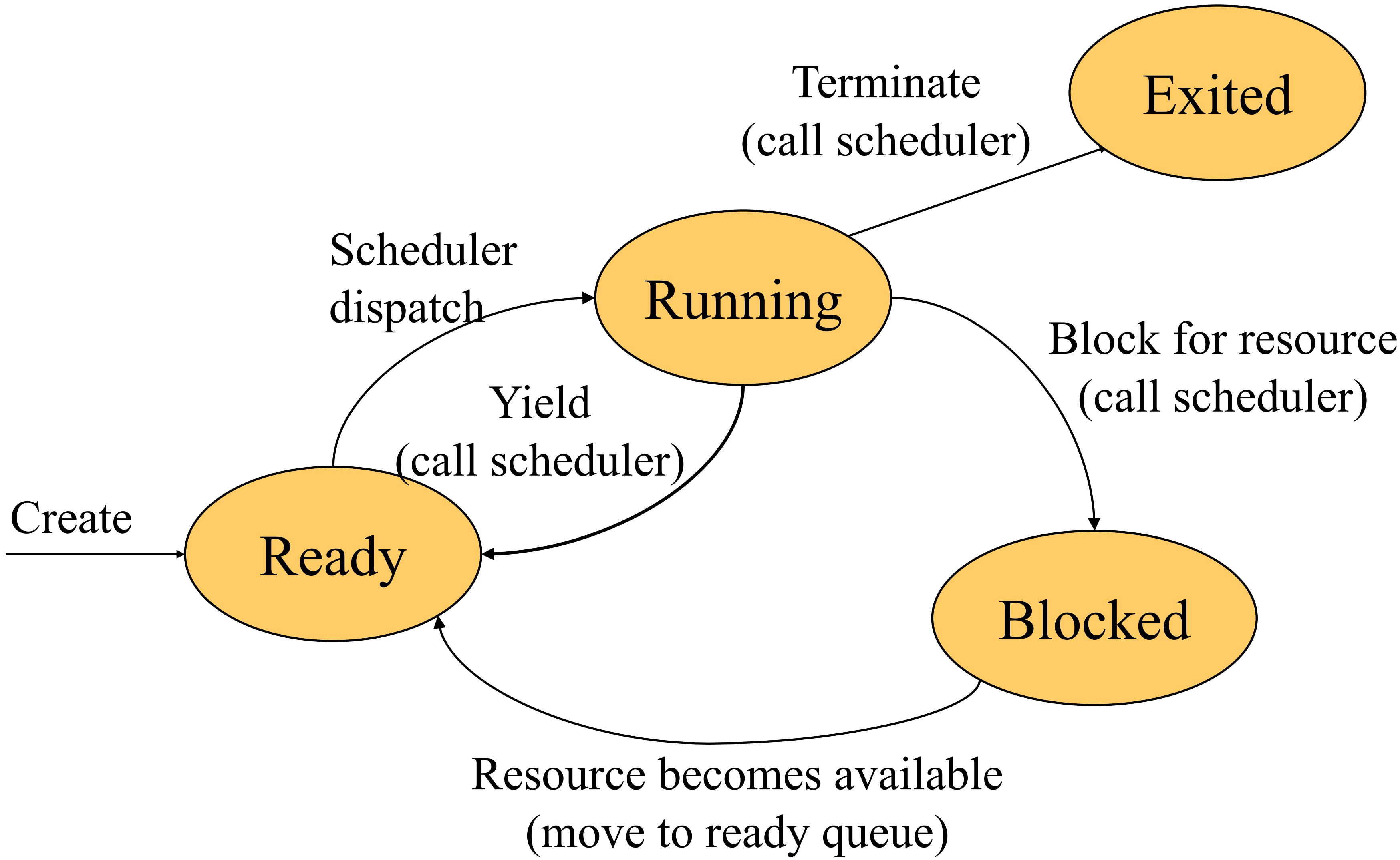
Thread Context Switching Decisions

- What to switch to?
  - Scheduling algorithm
- What to save and restore?
  - Schedule in a thread in the same address space (thread context switch)
  - Schedule in a thread in a different address space (process context switch)
- When to switch?
  - Voluntary
    - Q: Write two examples of times when a thread might voluntarily switch out
  - Involuntary
    - Q: Write two examples of times when a thread might be involuntarily switched out

Thread Context Switching Decisions

- What to switch to?
  - Scheduling algorithm
- What to save and restore?
  - Schedule in a thread in the same address space (thread context switch)
  - Schedule in a thread in a different address space (process context switch)
- When to switch?
  - Voluntary
    - Thread yields or blocks, e.g. for a resource like disk, a synchronization variable etc
    - Thread_join (wait for a target process, e.g. child, to terminate)
  - Involuntary
    - Interrupt or exception
    - Some other thread of higher priority needs to run

Non-Preemptive Scheduling



Non-Preemptive Scheduling (contd.)

- A non-preemptive scheduler is invoked by a thread calling a yield, block, join or similar

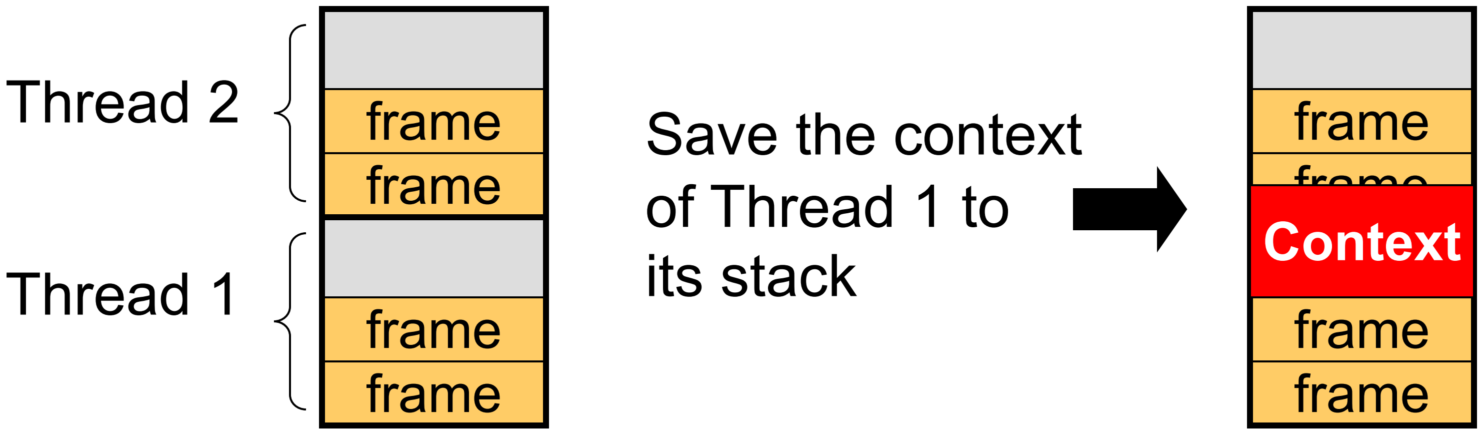
- Simplest form of scheduler: When invoked:

  save current process/thread state
  choose next process/thread to run
  dispatch (load PCB/TCB and jump to it)

- Scheduler can be viewed as just another kernel thread

Where and How to Save Thread Context?

- Save the context on the thread's stack
  - Many processors have a special instruction to do it efficiently
  - But, need to deal with the overflow problem



- Check before saving
  - Make sure that the stack has no overflow problem
  - Copy it to the TCB residing in the kernel heap
  - Not so efficient, but no overflow problems

Thread Control Block (TCB)

- Current state
  - Ready: ready to run
  - Running: currently running
  - Blocked: waiting for resources
- Registers
- Status (EFLAGS)
- Program counter (EIP)
- Stack

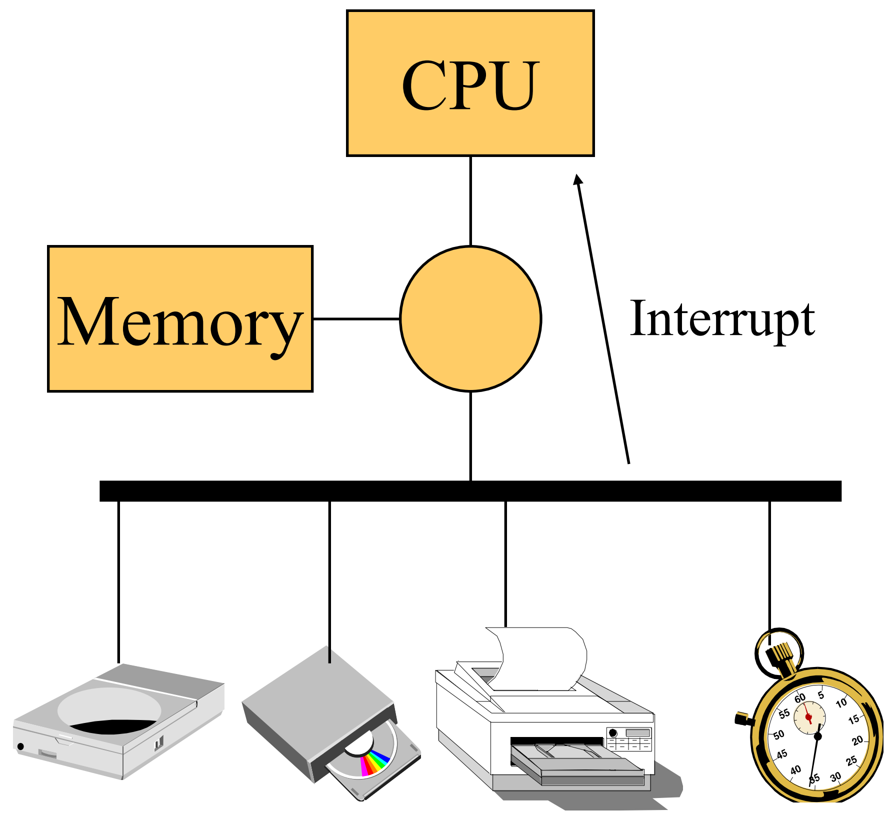
Voluntary thread context switch

- Save registers on old stack
- Switch to new stack, new thread
- Restore registers from new stack
- Return
- Exactly the same with kernel threads or user threads

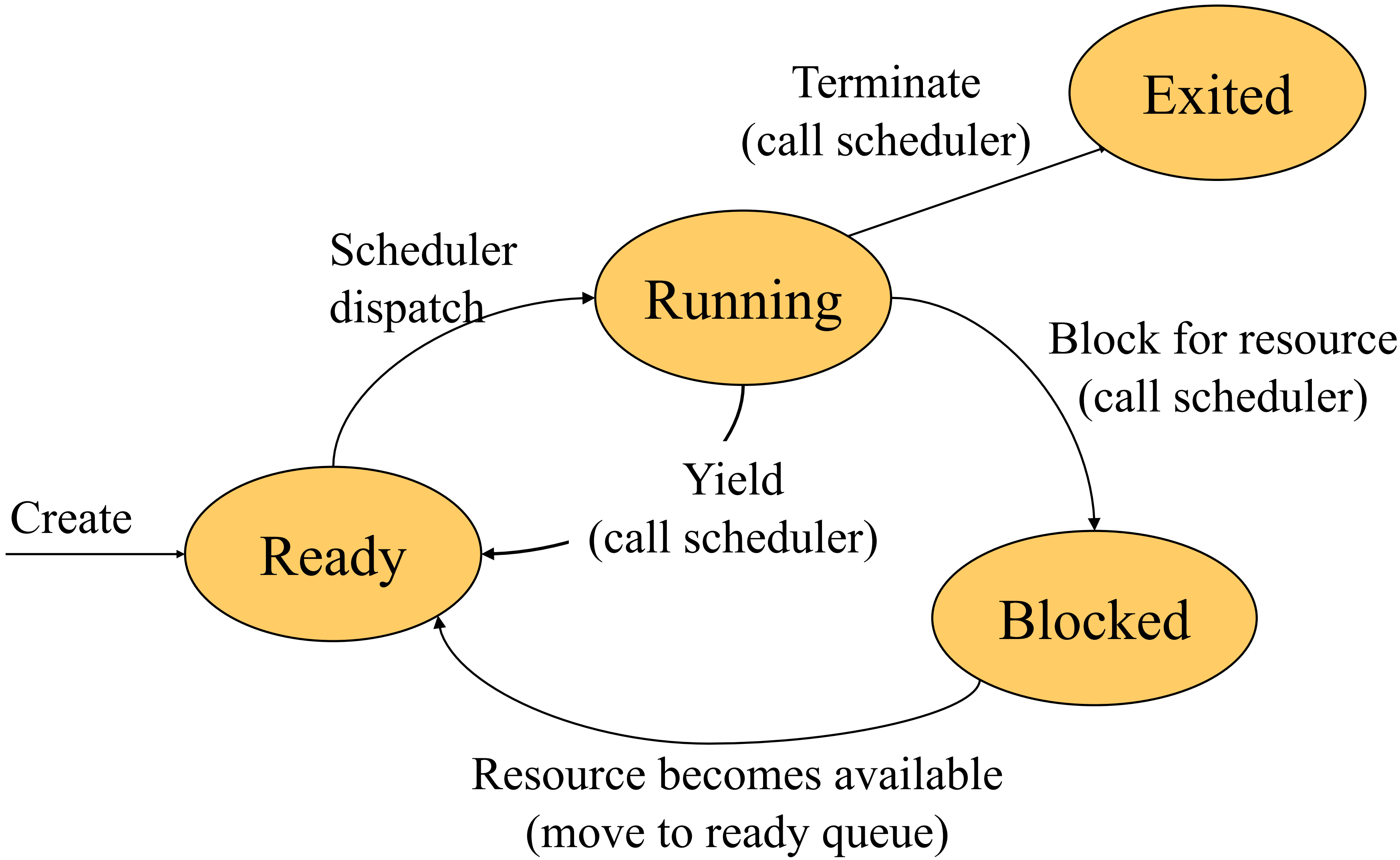
```
// We enter as oldThread, but we return as newThread.
// Returns with newThread's registers and stack.

void thread_switch(oldThreadTCB, newThreadTCB) {
    pushad;                  // Push general register values onto the old stack.
    oldThreadTCB->sp = %esp; // Save the old thread's stack pointer.
    %esp = newThreadTCB->sp; // Switch to the new stack.
    popad;                   // Pop register values from the new stack.
    return;
}
```

Preemption

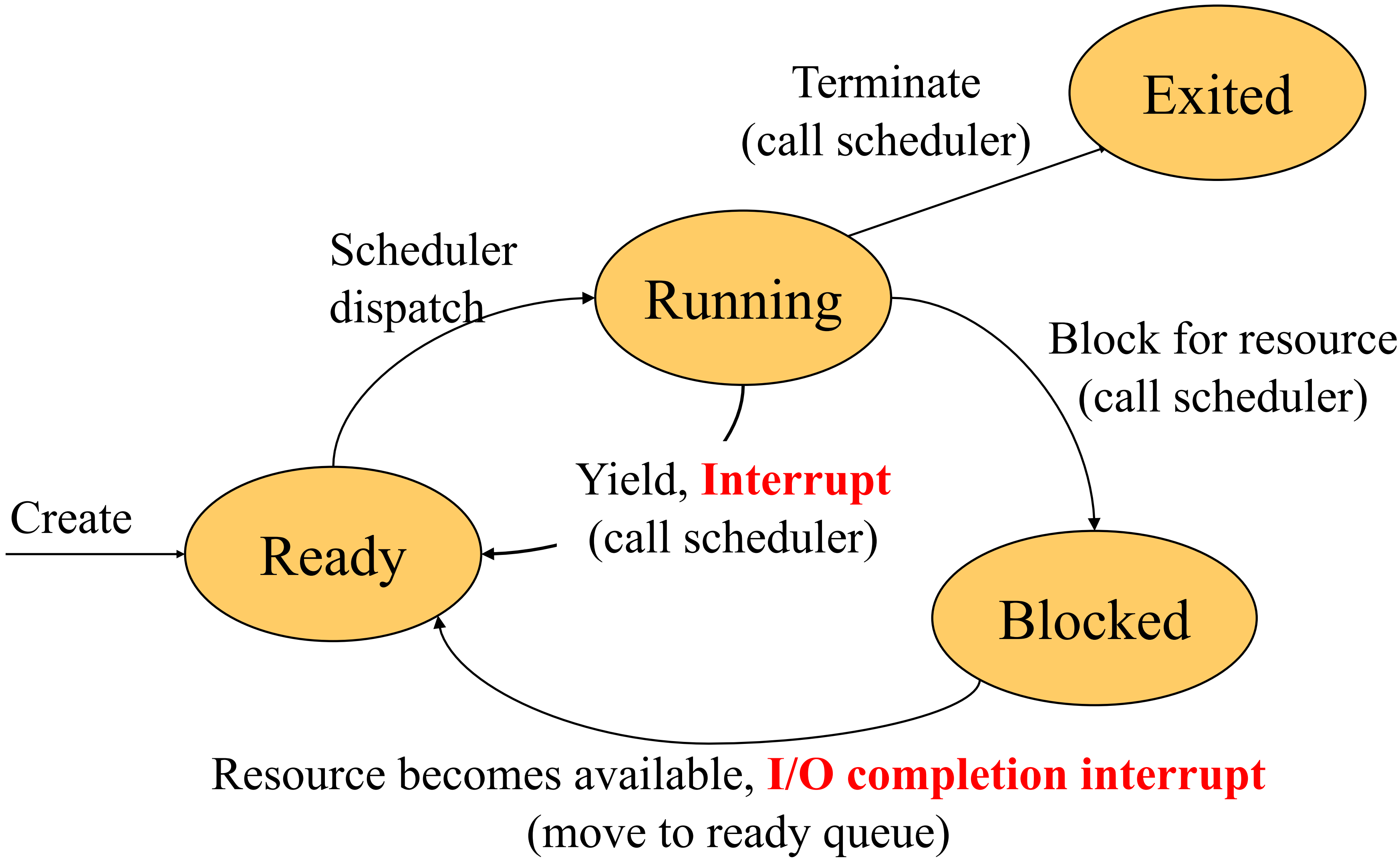
- Why?
  - Timer interrupt for CPU management
  - Asynchronous I/O completion
- When is CPU interrupted?
  - Between instructions
  - Within an instruction, except atomic ones
- Manipulate interrupts
  - Disable (mask) interrupts
  - Enable interrupts
  - Non-Maskable Interrupts



Recall: Non-Preemptive Scheduling



State Transitions for Preemptive Scheduling

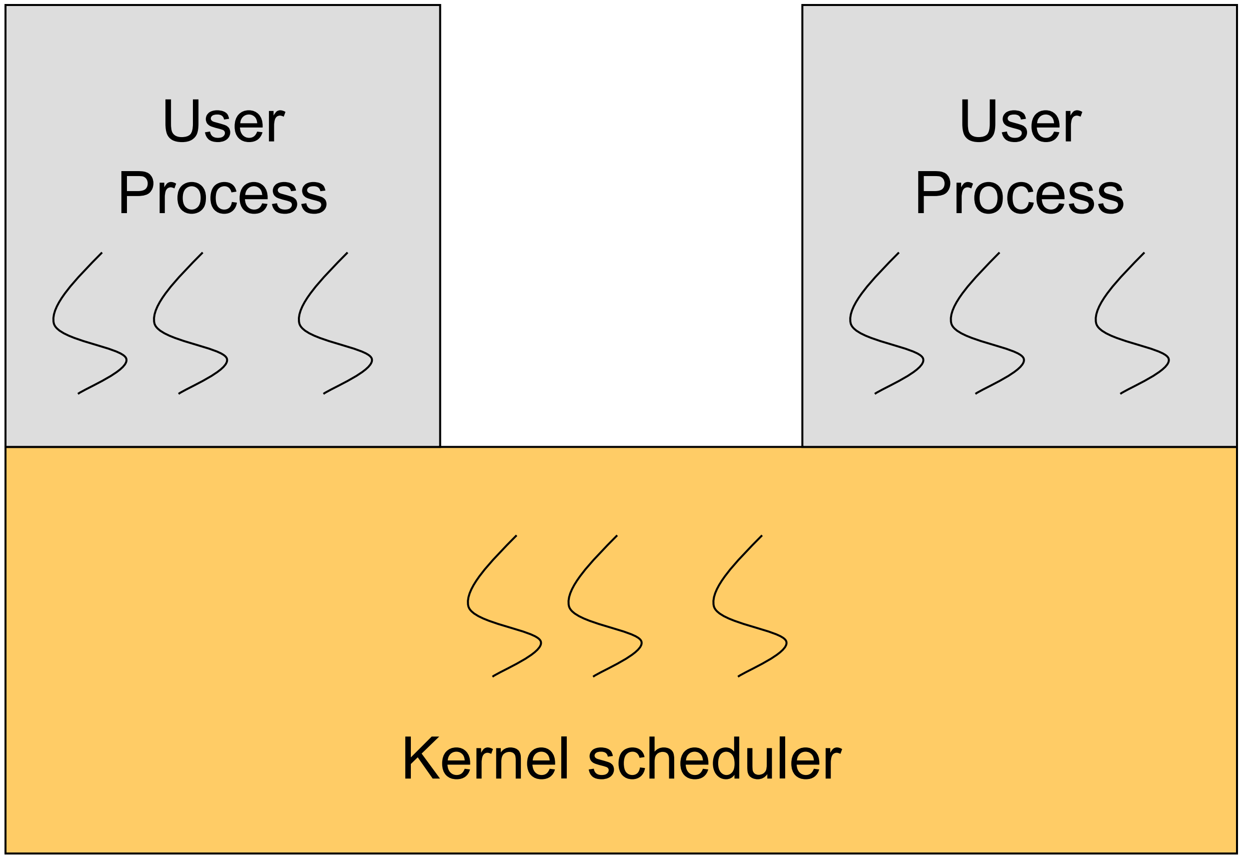


Interrupt Handling for Preemptive Scheduling

- Timer interrupt handler:
  - Save the current process / thread to its PCB / TCB
  - Call scheduler
- I/O interrupt handler:
  - Save the current process / thread to its PCB / TCB
  - Do the I/O job
  - Call scheduler

- Issues
  - Disable/enable interrupts
  - Make sure that it works on multiprocessors

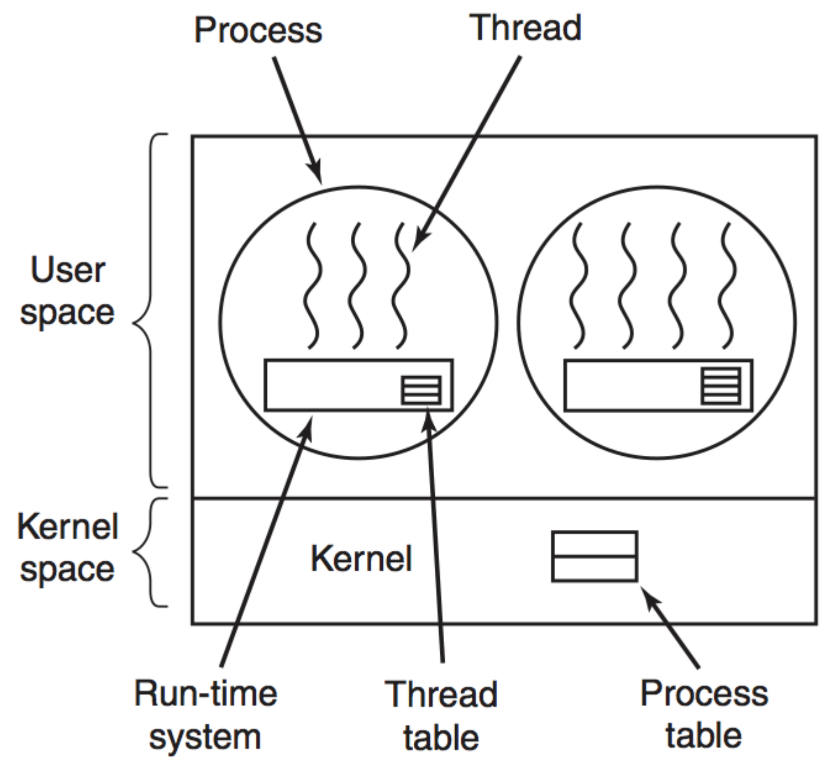
User- and Kernel-level Threads



- Threads at user level (in user space, user mode) and at kernel level
- User level threads map to kernel level threads, which are all the operating system really knows about

User-level Threads

- Managed by user-level runtime software, run in user mode
- Kernel knows only about user processes, not user threads, i.e. assumes one thread per processs
- Thread calls are user-level
- Context switch at user-level

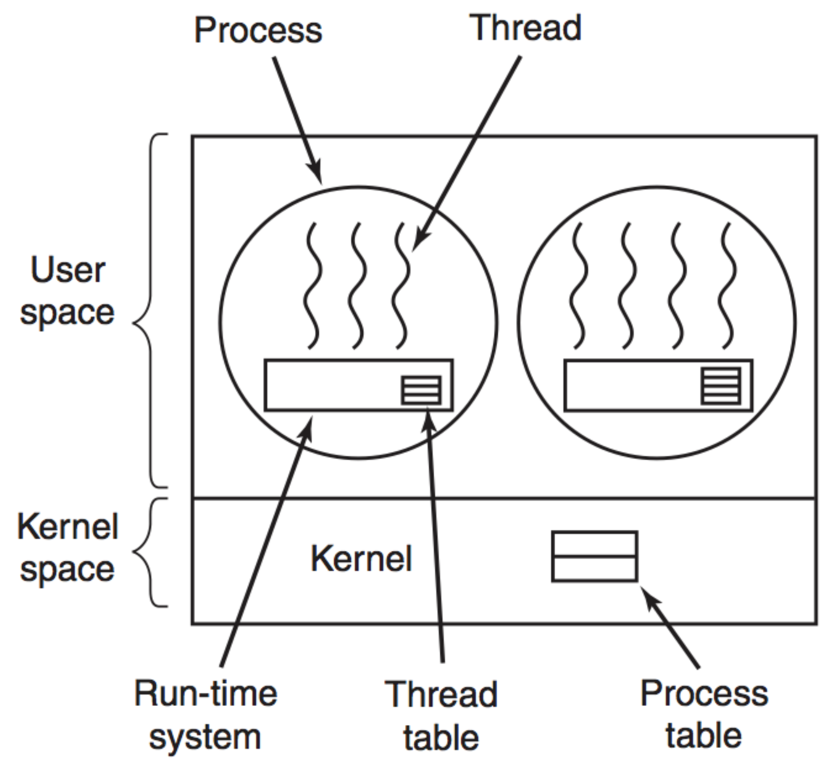


+ Fast (could be as fast as function call)
+ Can have custom user-level schedulers
+ Lower kernel complexity
+ Can implement on kernels that are single-threaded

− Entire process blocks when one thread blocks
− Kernel makes suboptimal decisions about scheduling
  − OS modifications to overcome this
− Hard to do pure on mPs

User-level Threads

- Managed by user-level runtime software, run in user mode
- Kernel knows only about user processes, not user threads, i.e. assumes one thread per processs
- Thread calls are user-level
- Context switch at user-level

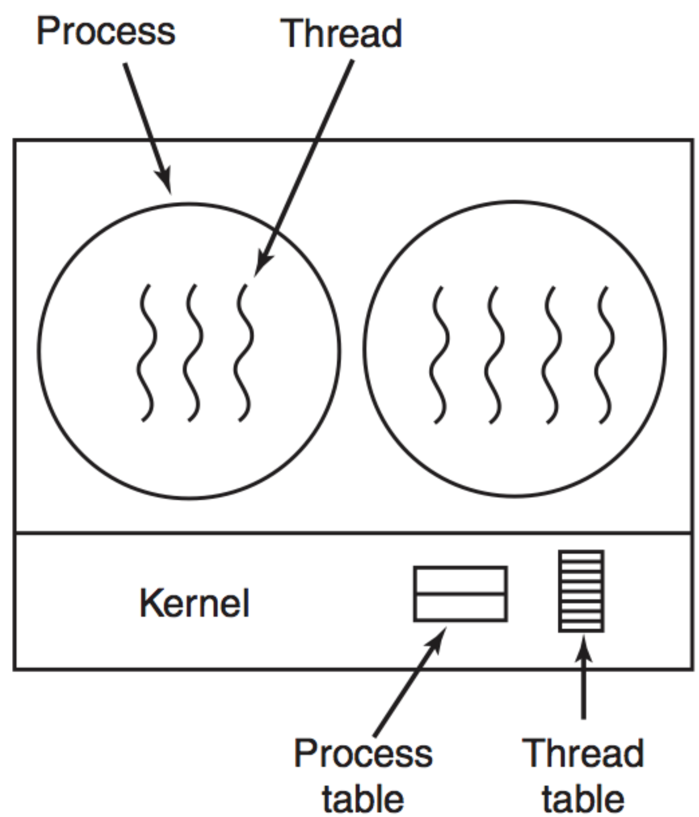


+ Fast (could be as fast as function call)
+ Can have custom user-level schedulers
+ Lower kernel complexity
+ Can implement on kernels that are single-threaded

- − Entire process blocks when one thread blocks
- − Kernel makes suboptimal decisions about scheduling
  - OS modifications to overcome this
- − Hard to do pure on mPs

Kernel Threads

- Managed by OS, run in kernel mode
- Invoking thread API causes system call
- Context switch invokes OS
- PCB per process and TCB per thread in kernel



+ Kernel has knowledge of threads so can optimize better
  - E.g. give more CPU time to processes with more threads, or threads that are not idle
+ When one thread in a process blocks, others can still run
  - Important when threads block frequently

– High overhead
– More complex OS

Implementation Models for User-level Threads

- User threads are mapped to kernel threads
  - Can think of it as a kernel thread per “virtual processor”
  - (need at least one kernel-level thread per core)
- Simpler typical cases are 1:1 and many to one
- In general, m user threads mapped to n kernel threads
  - Certain user level threads bound to a subset of kernel threads
  - Dynamically change-able no. of kernel threads for user process (but needs more communication mechanisms up/down), etc.

Summary

- Non-preemptive threads issues
  - Scheduler
  - Where to save contexts
- Preemptive threads
  - Interrupts can happen any where!
- Kernel vs. user threads