Announcements/Reminders

- Class news group: rcfnews.cs.umass.edu::cmpsci.edlab.cs377

Last Class: Processes

- A process is the unit of execution.
- Processes are represented as Process Control Blocks in the OS
  - PCBs contain process state, scheduling and memory management information, etc
- A process is either New, Ready, Waiting, Running, or Terminated.
- On a uniprocessor, there is at most one running process at a time.
- The program currently executing on the CPU is changed by performing a context switch
- Processes communicate either with message passing or shared memory

Today: Threads

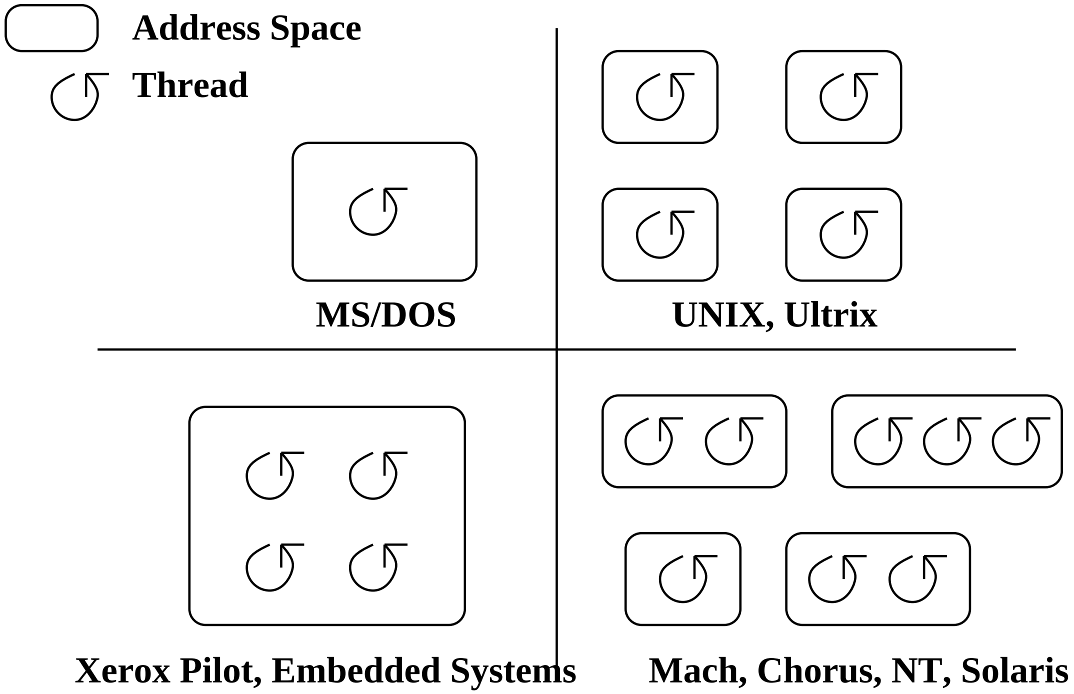
- What are threads?
- Where should we implement threads? In the kernel? In a user-level threads package?
- How should we schedule threads (or processes) onto the CPU?

Processes versus Threads

- **A process** defines the address space, text, resources, etc.,
- **A thread** defines a single sequential execution stream within a process (PC, stack, registers).
- Threads extract the *thread of control* information from the process
- Threads are bound to a single process.
- Each process may have multiple threads of control within it.
  - The address space of a process is shared among all its threads
  - No system calls are required to cooperate among threads
  - Interprocess communication is simpler than message passing and (process-level) shared-memory

Classifying Threaded Systems

Operating Systems can support one or many address spaces, and one or many threads per address space.



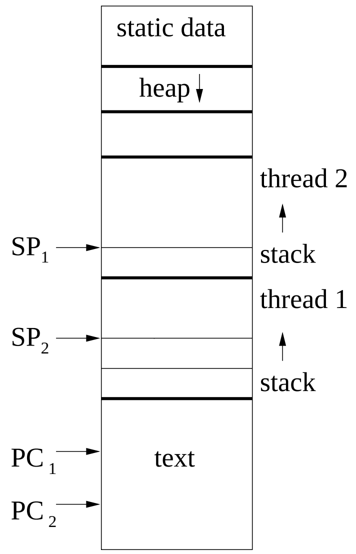
Example Threaded Program

```
main()
    global in, out, n, buffer[n];
    in = 0; out = 0;
    fork_thread (producer());
    fork_thread (consumer());
end

producer
    repeat
        nextp = produced item
        while in+1 mod n = out do no-op
        buffer[in] = nextp; in = (in+1) mod n

consumer
    repeat
        while in = out do no-op
        nextc = buffer[out]; out = (out+1) mod n
        consume item nextc
```

One possible memory layout:



Memory

- Forking a thread can be a system call to the kernel, or a procedure call to a thread library (user code).

Kernel Threads

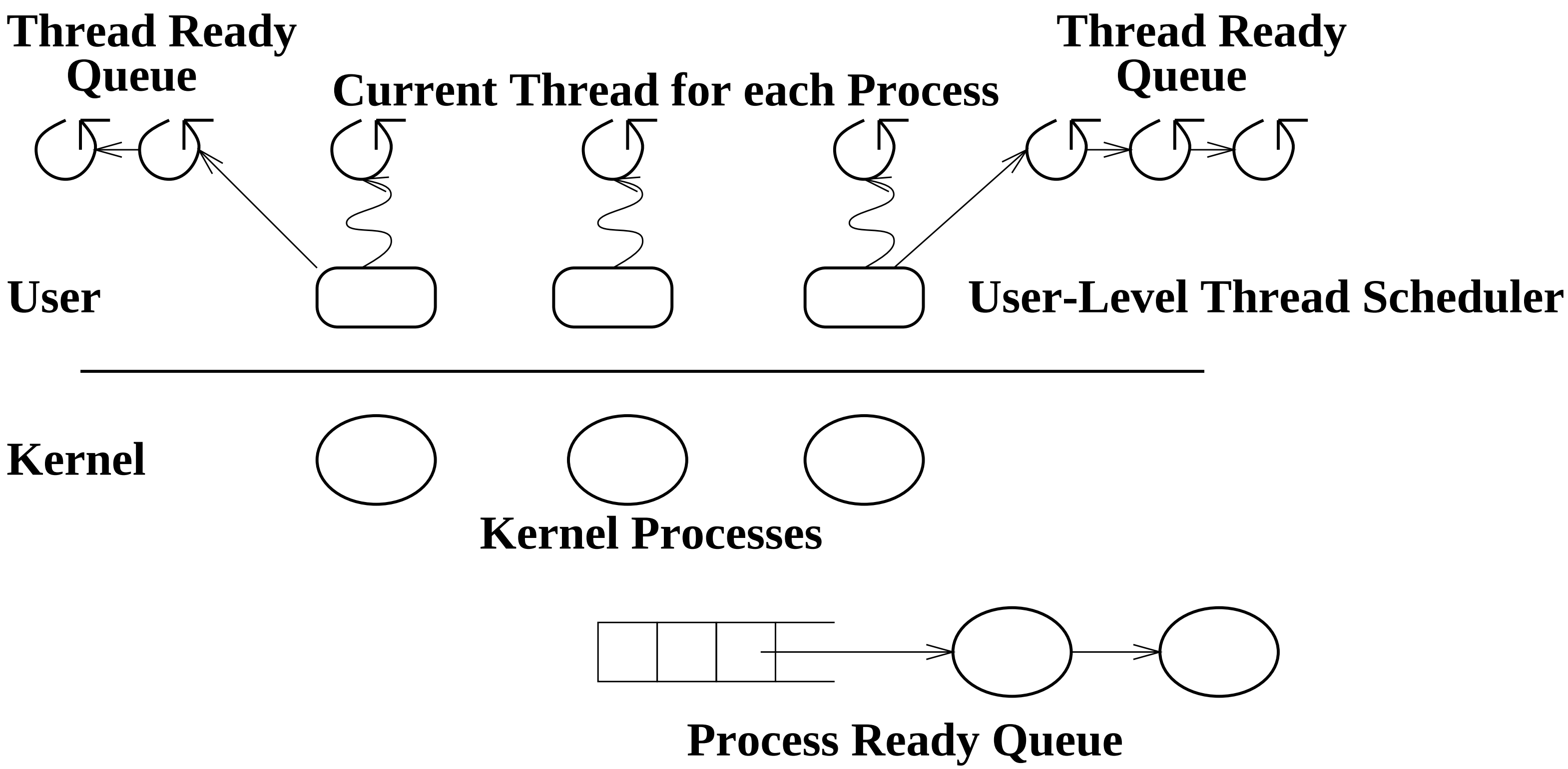
- A **kernel thread**, also known as a **lightweight process**, is a thread that the operating system knows about.
- Switching between kernel threads of the same process requires a small context switch.
  - The values of registers, program counter, and stack pointer must be changed.
  - Memory management information does not need to be changed since the threads share an address space.
- The kernel must manage and schedule threads (as well as processes), but it can use the same process scheduling algorithms.

⇒ Switching between kernel threads is slightly faster than switching between processes.

User-Level Threads

- A **user-level thread** is a thread that the OS does *not* know about.
- The OS only knows about the process containing the threads.
- The OS only schedules the process, not the threads within the process.
- The programmer uses a *thread library* to manage threads (create and delete them, synchronize them, and schedule them).
  - This is the case for Java.

User-Level Threads



User-Level Threads: Advantages

- There is no context switch involved when switching threads.
- User-level thread scheduling is more flexible:
  - A user-level code can define a problem dependent thread scheduling policy.
  - Each process might use a different scheduling algorithm for its own threads.
  - A thread can voluntarily give up the processor by telling the scheduler it will *yield* to other threads.
- User-level threads do not require system calls to create them or context switches to move between them

⇒ User-level threads are typically much faster than kernel threads

User-Level Threads: Disadvantages

- Since the OS does not know about the existence of the user-level threads, it may make poor scheduling decisions:
  -
  -

  ⇒ Solving these problems requires communication between the kernel and the user-level thread manager.

- Since the OS just knows about the process, it schedules the process the same way as other processes, regardless of the number of user threads. (For kernel threads, the more threads a process creates, the more time slices the OS will dedicate to it.)

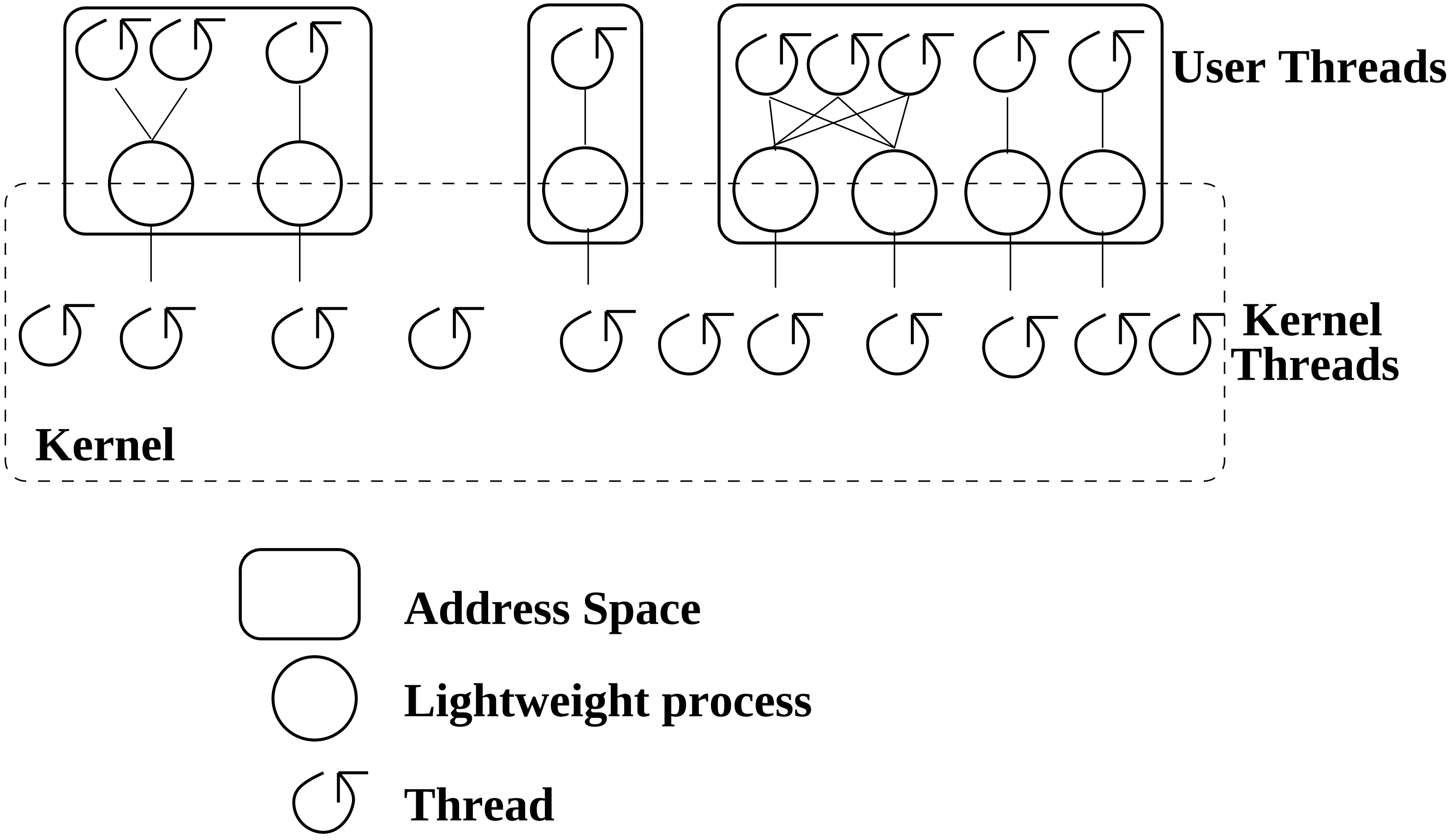
User-Level Threads: Disadvantages

- Since the OS does not know about the existence of the user-level threads, it may make poor scheduling decisions:
  - It might run a process that only has idle threads.
  - If a user-level thread is waiting for I/O, the entire process will wait.

  $\Rightarrow$ Solving these problems requires communication between the kernel and the user-level thread manager.

- Since the OS just knows about the process, it schedules the process the same way as other processes, regardless of the number of user threads. (For kernel threads, the more threads a process creates, the more time slices the OS will dedicate to it.)

Example: Kernel and User-Level Threads in Solaris



More Examples of Kernel and User-Level Threads

Ultrix: 1 thread per address space

Topaz: multiple kernel threads per address space

FastThreads: multiple user threads per address space

	Ultrix	Topaz	FastThreads
Fork	11,320	1208	39
Signal/Wait	1846	229	52

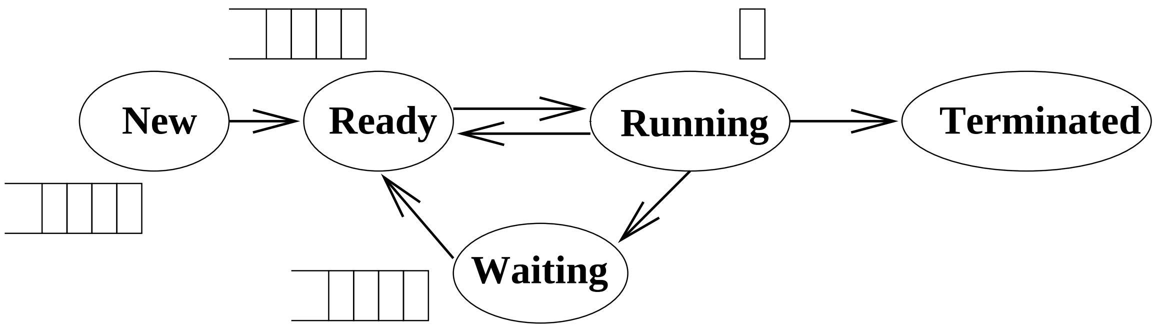
Operation times in Microseconds on a MIPS 3000

⇒ User-level thread operations are orders of magnitude faster than similar kernel thread operations

Scheduling Processes

- **Multiprogramming:** having more than one process at a time that is ready to run enables the OS to increase system utilization and throughput by overlapping I/O and CPU activities.

- Process Execution State



- All of the processes that the OS is currently managing reside in one and only one of these state queues.

Scheduling Processes

- **Long Term Scheduling::** How does the OS determine the degree of multiprogramming, i.e., the number of jobs executing at once in the primary memory? How does the OS determine which processor on which to place a job?

- **Short Term Scheduling:** How does (or should) the OS select a process from the ready queue to execute?
  - Policy Goals
  - Policy Options
  - Implementation considerations

Short Term Scheduling

- The kernel runs the scheduler at least when:
  1. A process switches from running to waiting,
  2. A process switches from waiting to ready,
  3. An interrupt occurs, or
  4. A process is created or terminated.
- **Non-preemptive system:** the scheduler must wait for one of these events
- **Preemptive system:** the scheduler can interrupt a running process

Criteria for Comparing Scheduling Algorithms:

CPU Utilization The percentage of time that the CPU is busy.

Throughput The number of processes completing in a unit of time.

Turnaround time The length of time it takes to run a process from initialization to termination, including all the waiting time.

Waiting time The total amount of time that a process is in the ready queue.

Response time The time between when a process is ready to run and its next I/O request.

Note: these are all quantitative metrics by which we can measure a scheduler's performance.

Scheduling Policies

Ideally, choose a CPU scheduler that optimizes all criteria simultaneously (utilization, throughput,..), but this is not generally possible

Instead, choose a scheduling algorithm based on its ability to satisfy a policy

- Minimize average response time - provide output to the user as quickly as possible and process their input as soon as it is received.
- Minimize variance of response time - in interactive systems, predictability may be more important than a low average with a high variance.

Scheduling Policies (cont)

- Maximize throughput - two components

  1. minimize overhead (OS overhead, context switching)
  2. efficient use of system resources (CPU, I/O devices)

- Minimize waiting time - give each process the same amount of time on the processor. This might actually increase average response time.

- Fairness - share the CPU in an equitable fashion.
  It turns out that fairness is a tradeoff against average response time. We can get better average response time by making the system less fair.

Measuring Scheduling Policy Performance:

Simplifying Assumptions

- One process per user
- One thread per process
- Processes are independent

  Researchers developed these algorithms in the 70's when these assumptions were more realistic; it is still an open problem as to how to relax these assumptions.

Scheduling Algorithms: A Snapshot

FCFS: First Come, First Served

Round Robin: Use a time slice and preemption to alternate jobs.

SJF: Shortest Job First

Multilevel Feedback Queues: Round robin on each priority queue.

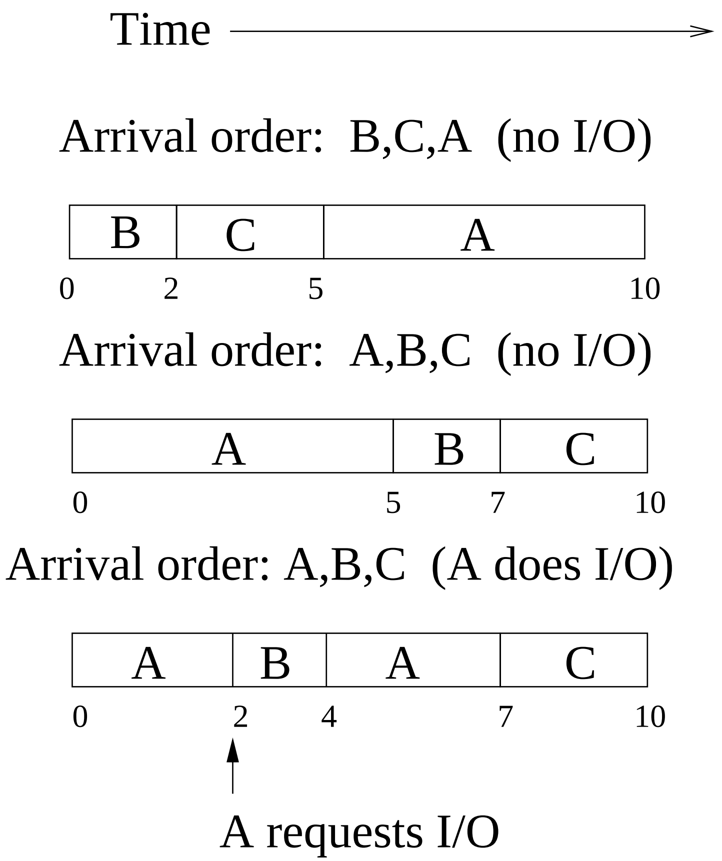
Lottery Scheduling: Jobs get tickets and scheduler randomly picks winning ticket.

Scheduling Policies

FCFS: First-Come-First-Served (or FIFO: First-In-First-Out)

- The scheduler executes jobs to completion in arrival order.
- In early FCFS schedulers, the job did not relinquish the CPU even when it was doing I/O.
- We will assume a FCFS scheduler that runs when processes are blocked on I/O, but that is non-preemptive, i.e., the job keeps the CPU until it blocks (say on an I/O device).

FCFS Scheduling Policy: Example



- If processes arrive 1 time unit apart, what is the average wait time in these three cases? How about turnaround time?

FCFS Scheduling Policy: Example

Case 1

- Turnaround time: $(2 + 4 + 8)/3 = 4.67$
- Wait time: $(0 + 1 + 3)/3 = 1.33$

Case 2

- Turnaround time: $(5 + 6 + 8)/3 = 6.33$
- Wait time: $(0 + 4 + 5)/3 = 3$

Case 3

- Turnaround time: $(7 + 3 + 8)/3 = 6$
- Wait time: (assume A does 2 units of I/O) $(0 + 1 + 5)/ = 2$

FCFS: Advantages and Disadvantages

Advantage: simple

Disadvantages:

- Average wait time is highly variable as short jobs may wait behind long jobs.
- May lead to poor overlap of I/O and CPU since CPU-bound processes will force I/O bound processes to wait for the CPU, leaving the I/O devices idle.

Summary

- Thread: a single execution stream within a process
- Switching between user-level threads is faster than between kernel threads since a context switch is not required.
- User-level threads may result in the kernel making poor scheduling decisions, resulting in slower process execution than if kernel threads were used.
- Many scheduling algorithms exist. Selecting an algorithm is a policy decision and should be based on characteristics of processes being run and goals of operating system (minimize response time, maximize throughput, ...).

Next Time

- Friday: Threads in Java (so finish reading Chapter 5)
- Monday: More scheduling (Chapter 6)