# Thread Wrap-up & Scheduling

## Computer Systems Programming, Spring 2023

**Instructor:** Travis McGaha

**TAs:**

| | |
|---|---|
| Kevin Bernat | Jialin Cai |
| Mati Davis | Donglun He |
| Chandravaran Kunjeti | Heyi Liu |
| Shufan Liu | Eddy Yang |

# Poll Everywhere

**pollev.com/tqm**

- Is it possible for a single threaded program to deadlock?

**A.** Yes

**B.** No

# Upcoming Due Dates

- HW2 (Threads)
  - Released
  - Should have everything you ***need***
  - Recitation will help ☺
  - This lecture will help

# Lecture Outline

- **Condition Variables**
- Scheduling

# Aside: sleep()

- `unistd.h` defines the function:

```
unsigned int sleep(unsigned int seconds);
```

  - Makes the calling thread sleep for the specified number of seconds, resuming execution afterwards

- Useful for manipulating scheduling for testing and demonstration purposes
  - Also for asynchronous/non-blocking I/O, but not covered in this course.

- Necessary for HW2 so that auto-graders work ☹

# Thread Communication

- Sometimes threads may need to communicate with each other to know when they can perform operations

- Example: Producer and consumer threads
  - One thread creates tasks/data
  - One thread consumes the produced tasks/data to perform some operation
  - The consumer thread can only produce things once the producer has produced them

# Naïve Solution

- Consider the example where a thread must wait to be notified before it can print something out and terminate

- Possible solution: "Spinning"
  - Infinitely loop until the producer thread notifies that the consumer thread can print

- See `spinning.cc`

# Poll Everywhere

pollev.com/tqm

- Does "spinning" fix the deadlock?
  - **A.** Yes
  - **B.** No, possible deadlock
  - **C.** No, not thread safe
  - **D.** Segmentation Fault
  - **E.** We're Lost…

```
bool print_ok = false;
pthread_mutex_t lock;

void* consumer(void* arg) {

  // "spin" until print_ok
  // is true
  pthread_mutex_lock(&lock);
  while (!print_ok) {
    pthread_mutex_unlock(&lock);
    pthread_mutex_lock(&lock);
  }
  pthread_mutex_unlock(&lock);

  cout << "Ok to print :)";
  cout << endl;

  pthread_exit(nullptr);

}
```

# Condition Variables

- Variables that allow for a thread to wait until they are notified to resume

- Avoids waiting clock cycles “spinning”

- Done in the context of mutual exclusion
  - a thread must already have a lock, which it will temporarily release while waiting
  - Once notified, the thread will re-acquire a lock and resume execution

# pthreads and condition variables

- `pthread.h` defines datatype `pthread_cond_t`

- ```
  int pthread_cond_init(pthread_cond_t* cond,
                        const pthread_condattr_t* attr);
  ```
  - Initializes a condition variable with specified attributes

- ```
  int pthread_cond_destroy(pthread_cond_t* cond);
  ```
  - “Uninitializes” a condition variable – clean up when done

# pthreads and condition variables

- `pthread.h` defines datatype `pthread_cond_t`
- ```
  int pthread_cond_wait(pthread_cond_t* cond,
                        pthread_mutex_t* mutex);
  ```
  - Atomically releases the mutex and blocks on the condition variable. Once unblocked (by one of the functions below), function will return and calling thread will have the mutex locked
- ```
  int pthread_cond_signal(pthread_cond_t* cond);
  ```
  - Unblock at least one of the threads on the specified condition
- ```
  int pthread_cond_broadcast(pthread_cond_t* cond);
  ```
  - Unblock all threads blocked on the specified condition
- See `cond.cc`
  ```

# Aside: Things left out

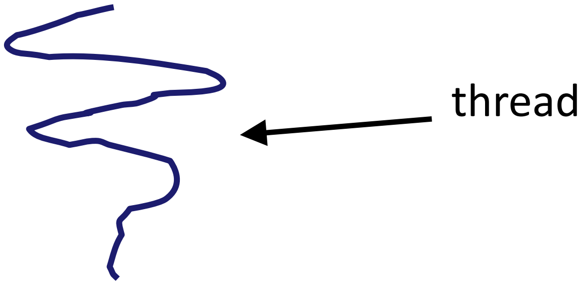
- MANY things left out of this lecture
- Synchronization methods:
  - Semaphores
  - Monitors
- Concurrency properties
  - ACID (databases)
  - CAP theorem
- A lot more concurrency stuff covered in CIS 5050 & CIS 5480 ☺

# Lecture Outline

- Condition Variables
- **Scheduling**

# Reminder: Threads

- Separate the concept of a process from the "*thread of execution*"
  - Threads are contained within a process
  - Usually called a thread, this is a sequential execution stream within a process

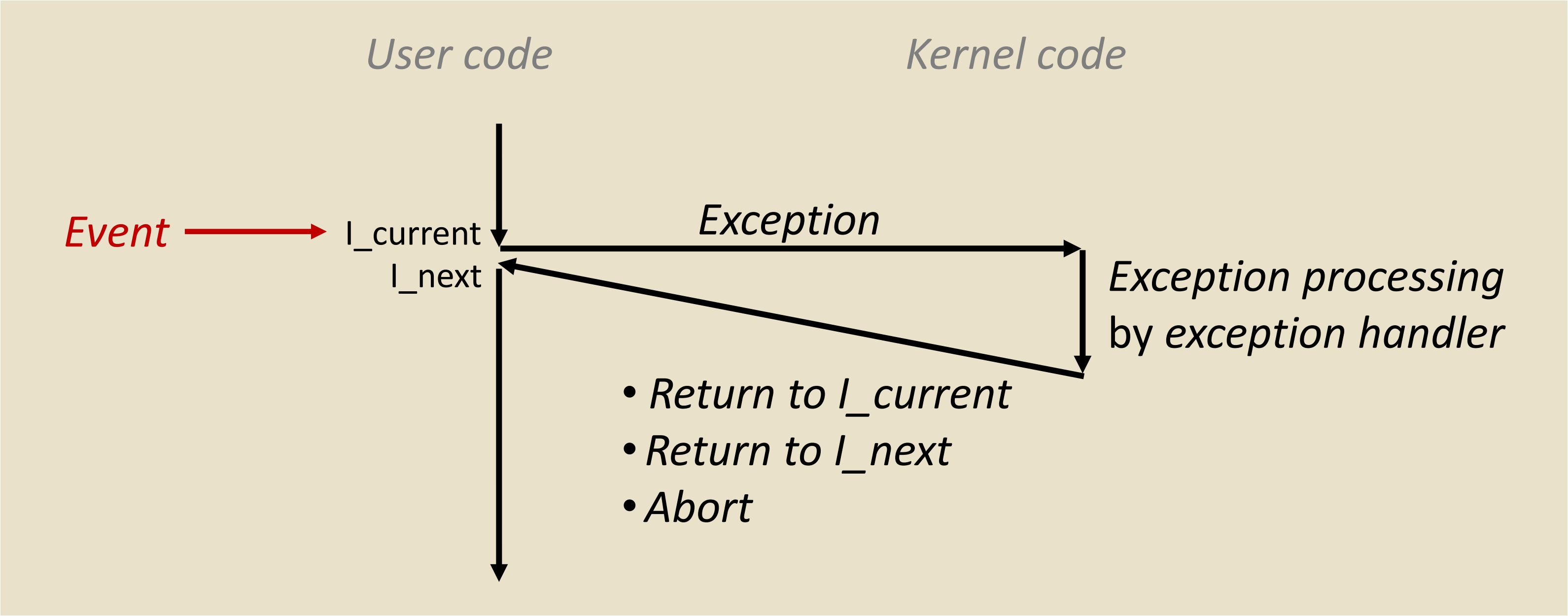



- Has its own stack, program counter & other registers

- In most modern OS's:
  - Threads are the *unit of scheduling.*

# Reminder: Exceptions

- An *exception* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



# OS as the Scheduler

- The scheduler is code that is part of the kernel (OS)

- The scheduler runs when a thread:
  - starts (“arrives to be scheduled”),
  - Finishes
  - Blocks (e.g., waiting on something, usually some form of I/O)
  - Has run for a certain amount of time

- It is responsible for scheduling other threads
  - Choosing which one to run
  - Deciding how long to run it

# Scheduler Terminology

- The scheduler has a scheduling algorithm to decide what runs next.

- Algorithms are designed to consider many factors:
  - Fairness: Every program gets to run
  - Liveness: That "something" will eventually happen
  - Throughput: amount of work completed over an interval of time
  - Wait time: Average time a "task" is "alive" but not running
  - Turnaround time: time between task being ready and completing
  - Response time: time it takes between task being ready and when it can take user input
  - Etc…

# Goals

- The scheduler will have various things to prioritize
- Some examples:
- Minimizing wait time
  - Get threads started as soon as possible
- Minimizing latency
  - Quick response times and task completions are preferred
- Maximizing throughput
  - Do as much work as possible per unit of time
- Maximizing fairness
  - Make sure every thread can execute fairly

- These goals depend on the system and can conflict

# Scheduling: Other Considerations

- It takes time to context switch between threads
  - Could get more work done if thread switching is minimized

- Scheduling takes resources
  - It takes time to decide which thread to run next
  - It takes space to hold the required data structures

- Different tasks have different priorities
  - Higher priority tasks should finish first

# Types of Scheduling Algorithms

- **Non-Preemptive:** if a thread is running, it continues to run until it completes or until it gives up the CPU
  - First come first serve (FCFS)
  - Shortest Job First (SJF)

- **Preemptive:** the thread may be interrupted after a given time and/or if another thread becomes ready
  - Round Robin
  - Priority Round Robin
  - …

# First Come First Serve (FCFS)

- Idea: Whenever a thread is ready, schedule it to run until it is finished (or blocks).
- Maintain a queue of ready threads
  - Threads go to the back of the queue when it arrives or becomes unblocked
  - The thread at the front of the queue is the next to run

# Example of FCFS

1 CPU
Job 2 arrives slightly after job 1.
Job 3 arrives slightly after job 2

- Example workload with three “jobs”:
  Job 1: 24 time units; Job 2: 3 units; Job 3: 3 units

- FCFS schedule:


- Total waiting time: 0 + 24 + 27 = 51
- Average waiting time: 51/3 = 17
- Total turnaround time: 24 + 27 + 30 = 81
- Average turnaround time: 81/3 = 27

# Poll Everywhere

**pollev.com/tqm**

- What are the advantages/disadvantages/concerns with **First Come First Serve**

# FCFS Analysis

- Advantages:
  - Simple, low overhead
  - Hard to screw up the implementation
  - Each thread will DEFINITELY get to run eventually.

- Disadvantages
  - Doesn’t work well for interactive systems
  - Throughput can be low due to long threads
  - Large fluctuations in average turn around time
  - Priority not taken into considerations

# Shortest Job First (SJF)

- Idea: variation on FCFS, but have the tasks with the smallest CPU-time requirement run first
  - Arriving jobs are instead put into the queue depending on their run time, shorter jobs being towards the front
  - Scheduler selects the shortest job ($1^{st}$ in queue) and runs till completion

# Example of SJF

1 CPU
Job 2 arrives slightly after job 1.
Job 3 arrives slightly after job 2

- Same example workload with three “jobs”:
  Job 1: 24 time units; Job 2: 3 units; Job 3: 3 units

- FCFS schedule:

```
| Job 2 | Job 3  |   Job 1          |
0       3        6                 30
```

- Total waiting time: 6 + 0 + 3 = 9
- Average waiting time: 3
- Total turnaround time: 30 + 3 + 6 = 39
- Average turnaround time: 39/3 = 13

## Poll Everywhere

pollev.com/tqm

- What are the advantages/disadvantages/concerns with **Shortest Job First**

# SJF Analysis

- Advantages:
  - Still relatively simple, low overhead
  - provably minimal average turnaround time

- Disadvantages
  - Starvation possible
    - If quick jobs keep arriving, long jobs will keep being pushed back and won't execute
  - How do you know how long it takes for something to run?
    - You CAN'T. You can use a history of past behavior to make a guess.
  - Priority not taken into considerations

# Types of Scheduling Algorithms

- **Non-Preemptive:** if a thread is running, it continues to run until it completes or until it gives up the CPU
  - First come first serve (FCFS)
  - Shortest Job First (SJF)

- **Preemptive:** the thread may be interrupted after a given time and/or if another thread becomes ready
  - Round Robin
  - Priority Round Robin
  - …

# Round Robin

- Sort of a preemptive version of FCFS
  - Whenever a thread is ready, add it to the end of the queue.
  - Run whatever job is at the front of the queue

- BUT only let it run for a fixed amount of time (quantum).
  - If it finishes before the time is up, schedule another thread to run
  - If time is up, then send the running thread back to the end of the queue.

# Example of Round Robin

- Same example workload:

  Job 1: 24 units, Job 2: 3 units, Job 3: 3 units
- RR schedule with time quantum=2:

```
|Job 1|Job 2|Job 3|Job 1|Jo2|Jo3|Job 1| …      |Job 1|
0     2     4     6     8   9   10    12,14…         30
```

- Total waiting time: (0 + 4 + 2) + (2 + 4) + (4 + 3) = 19
  - Counting time spent waiting between each "turn" a job has with the CPU
- Average waiting time: 19/3 (~6.33)
- Total turnaround time: 30 + 9 + 10 = 49
- Average turnaround time: 49/3 (~16.33)

# Poll Everywhere

pollev.com/tqm

- What are the advantages/disadvantages/concerns with **Round Robin?**

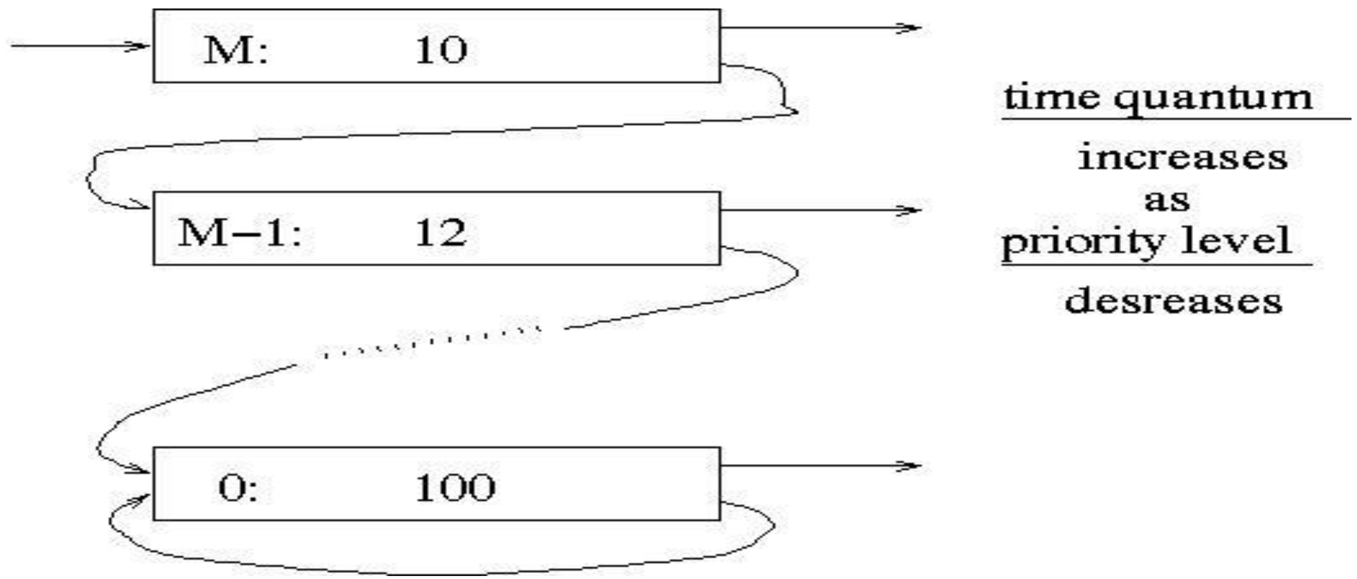
# Round Robin Analysis

- Advantages:
  - Still relatively simple
  - Can works for interactive systems
- Disadvantages
  - If quantum is too small, can spend a lot of time context switching
  - If quantum is too large, approaches FCFS
  - <u>Still assumes all processes have the same priority.</u>
- Rule of thumb:
  - Choose a unit of time so that most jobs (80-90%) finish in one usage of CPU time

# RR Variant: Priority Round Robin

- Same idea as round robin, but with multiple queues for different priority levels.
- Scheduler chooses the first item in the highest priority queue to run
- Scheduler only schedules items in lower priorities if all queues with higher priority are empty.

# RR Variant: Multi Level Feedback




- Each priority level has a ready queue, and a time quantum
- Thread enters highest priority queue initially, and lower queue with each timer interrupt
- If a thread voluntarily stops using CPU before time is up, it is moved to the end of the current queue
- Bottom queue is standard Round Robin
- Thread in a given queue not scheduled until all higher queues are empty

# Multi Level Feedback Analysis

- Threads with high I/O bursts are preferred
  - Makes higher utilization of the I/O devices
  - Good for interactive programs (keyboard, terminal, mouse is I/O)
- Threads that need the CPU a lot will sink to lower priority, giving shorter threads a chance to run
- Still have to be careful in choosing time quantum
- Also have to be careful in choosing how many layers

# Multi Level Feedback Variants: Priority

- Can assign tasks different priority levels upon initiation that decide which queue it starts in
  - E.g. OS Services should have higher priority than HelloWorld.java

- Update the priority based on recent CPU usage rather than overall cpu usage of a task
  - Makes sure that priority is consistent with recent behavior

- Many others that vary from system to system

# Why did we talk about this?

- Scheduling is fundamental to wards how computer can multi-task

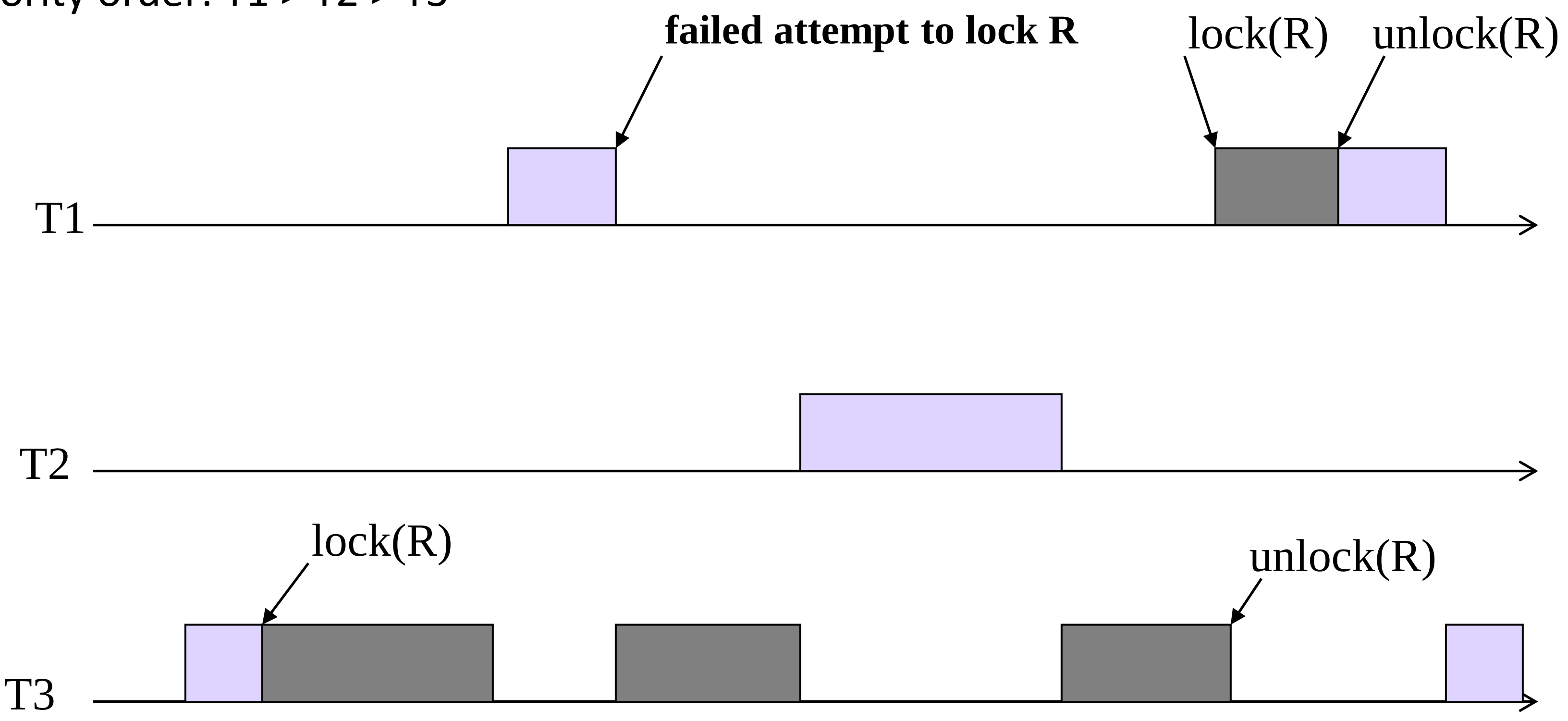
- This is a great example of how “systems” intersects with algorithms :)

- It shows up occasionally in the real world :)
  - Scheduling threads with priority with shared resources can cause a priority inversion, potentially causing serious errors.

What really happened on Mars Rover Pathfinder, Mike Jones.
http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html

# The Priority Inversion Problem

Priority order: T1 > T2 > T3




**T2 is causing a higher priority task T1 wait !**

# More

- For those curious, there was a LOT left out

- RTOS (Real Time Operating Systems)
  - For real time applications
  - CRITICAL that data and events meet defined time constraints
  - Different focus in scheduling. Throughput is de-prioritized

- Fair-share scheduling
  - Equal distribution across different users instead of by processes

- Priority Inversion