Scheduler & Threads (cont.)

Computer Operating Systems, Spring 2025

Instructors: Joel Ramirez Travis McGaha

Head TAs: Ash Fujiyama Emily Shen Maya Huizar

TAs:

Ahmed Abdellah	Bo Sun	Joy Liu	Susan Zhang	Zihao Zhou
Akash Kaukuntla	Connor Cummings	Khush Gupta	Vedansh Goenka	
Alexander Cho	Eric Zou	Kyrie Dowling	Vivi Li	
Alicia Sun	Haoyun Qin	Rafael Sakamoto	Yousef AlRabiah	
August Fu	Jonathan Hong	Sarah Zhang	Yu Cao	

Poll Everywhere

pollev.com/tqm

- How are you doing? Any questions?

Administrivia

- Penn-shell is out (this shouldn't be news)!
  - ***Full thing is due (Fri, Feb 28) (This week!)***
  - ***Done in partners***
    - Everything was covered already that you would need…
- Midterm is Thursday next week
  - Old exams and exam policies are posted on the course website
  - Review session in Recitation Thursday this week!!!!!!!!!!!! (7pm in Towne 217)
  - Some midterm review in Lecture Tuesday Next Week
  - What we get to in this lecture will be testable.
- SIGCSE TS
  - Some office hours moving around as well. Calendar updated soon.

Lecture Outline

- **Scheduler**
  - **Round robin variants**
  - Linux Scheduler
- Threads & Shared Data
  - Thread Refresher
  - Mutex
  - TSL
  - Disable Interrupts
  - Petersons

Lecture ended right before TSL

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)

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
  - Still assumes all processes have the same priority.

- Rule of thumb:
  - Choose a unit of time so that most jobs (80-90%) finish in one usage of CPU time

RR Variant: PennOS Scheduler

- In PennOS you will have to implement a priority scheduler based mostly off of round robin.

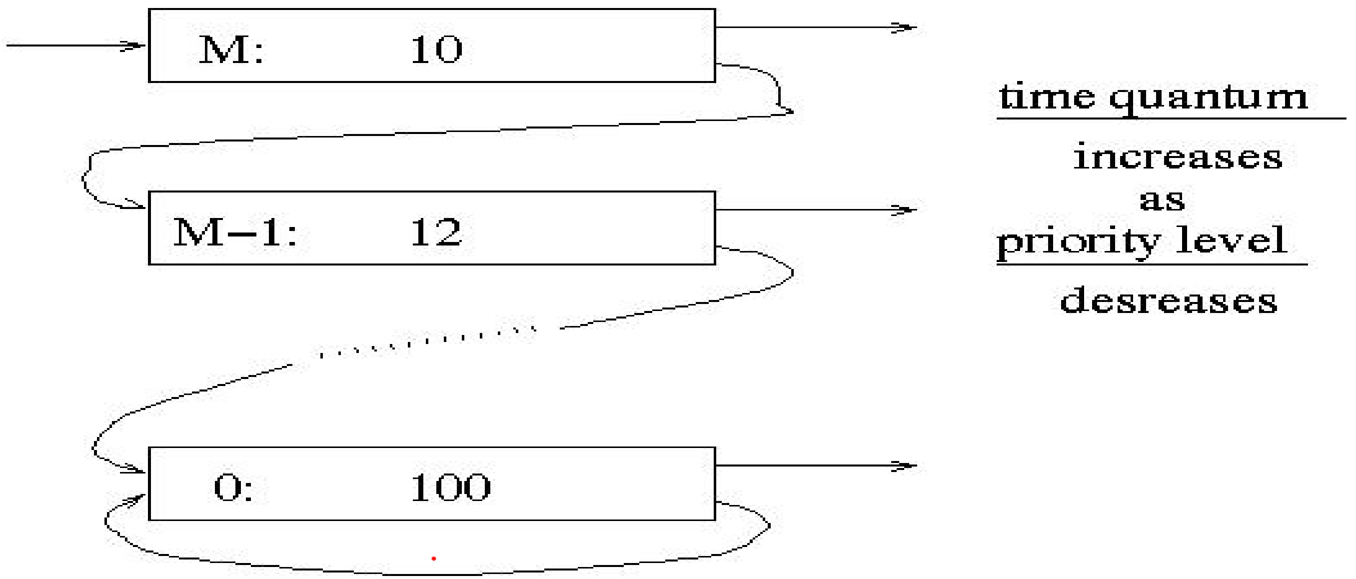
- You will have 3 queues, each with a different priority (0, 1, 2)
  - Each queue acts like normal round robin within the queue

- You spend time quantum processing each queue proportional to the priority
  - Priority 0 is scheduled 1.5 times more often than priority 1
  - Priority 1 is scheduled 1.5 times more often than priority 2

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. the scheduler 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

Lecture Outline

- **Scheduler**
  - Round robin variants
  - **Linux Scheduler**
- Threads & Shared Data
  - Thread Refresher
  - Mutex
  - TSL
  - Disable Interrupts
  - Petersons

Lecture ended right before TSL

Multiple Cores

- On a modern machine, we have multiple CPU Cores, each can run tasks
  - ***Generally*** each core has its own run-queue
  - It helps to keep threads in the same process on the same processor
    - Threads in the same process use the same memory: lower overhead
    - If we want to there are ways to make sure a thread/process is “pinned” to a CPU
      - See: Thread Affinity / Processor Affinity / CPU Pinning

- There is other stuff to balance tasks across cores, but I am leaving that out for time ☺

"Completely Fair Scheduling"

- “Fairness” – making sure that each task gets its fair share of the CPU
  - This is not always achievable
  - “Fairness, it turns out, is enough to solve many CPU-scheduling problems.”

"Completely Fair Scheduling"

- "Fairness" – making sure that each task gets its fair share of the CPU
  - This is not always achievable
  - "Fairness, it turns out, is enough to solve many CPU-scheduling problems."

THIS IS WHAT CFS IS TRYING TO REPLICATE. AS IF WE ARE ON AN "IDEAL PROCESSOR"

- Here is an example of fairness:
  - Within some "slice" of time, each task gets an equal proportion of the processor

TASK	Run Time
A	1
B	5
C	2

Task

A	1/3								
B	1/3								
C	1/3								

0 1 2 3 4 5 6 7 8

"Completely Fair Scheduling"

- "Fairness" – making sure that each task gets its fair share of the CPU
  - This is not always achievable
  - "Fairness, it turns out, is enough to solve many CPU-scheduling problems."

THIS IS WHAT CFS IS TRYING TO REPLICATE. AS IF WE ARE ON AN "IDEAL PROCESSOR"

- Here is an example of fairness:
  - Within some "slice" of time, each task gets an equal proportion of the processor

TASK	Run Time
A	1
B	5
C	2

Task	0–1	1–2	2–3	3–4	4–5	5–6	6–7	7–8	8–
A	1/3	1/3	1/3						
B	1/3	1/3	1/3						
C	1/3	1/3	1/3						

0 1 2 3 4 5 6 7 8

"Completely Fair Scheduling"

- "Fairness" – making sure that each task gets its fair share of the CPU
  - This is not always achievable
  - "Fairness, it turns out, is enough to solve many CPU-scheduling problems."

THIS IS WHAT CFS IS TRYING TO REPLICATE. AS IF WE ARE ON AN "IDEAL PROCESSOR"

- Here is an example of fairness:
  - Within some "slice" of time, each task gets an equal proportion of the processor

TASK	Run Time
A	1
B	5
C	2

Task	0–1	1–2	2–3	3–4	4–5	5–6	6–7	7–8	8–
A	1/3	1/3	1/3						
B	1/3	1/3	1/3	1/2					
C	1/3	1/3	1/3	1/2					

"Completely Fair Scheduling"

- "Fairness" – making sure that each task gets its fair share of the CPU
  - This is not always achievable
  - "Fairness, it turns out, is enough to solve many CPU-scheduling problems."

THIS IS WHAT CFS IS TRYING TO REPLICATE. AS IF WE ARE ON AN "IDEAL PROCESSOR"

- Here is an example of fairness:
  - Within some "slice" of time, each task gets an equal proportion of the processor

TASK	Run Time
A	1
B	5
C	2

Task	0–1	1–2	2–3	3–4	4–5	5–6	6–7	7–8	8–
A	1/3	1/3	1/3						
B	1/3	1/3	1/3	1/2	1/2				
C	1/3	1/3	1/3	1/2	1/2				

"Completely Fair Scheduling"

- "Fairness" – making sure that each task gets its fair share of the CPU
  - This is not always achievable
  - "Fairness, it turns out, is enough to solve many CPU-scheduling problems."

THIS IS WHAT CFS IS TRYING TO REPLICATE. AS IF WE ARE ON AN "IDEAL PROCESSOR"

- Here is an example of fairness:
  - Within some "slice" of time, each task gets an equal proportion of the processor

TASK	Run Time
A	1
B	5
C	2

Task	0–1	1–2	2–3	3–4	4–5	5–6	6–7	7–8	8–
A	1/3	1/3	1/3						
B	1/3	1/3	1/3	1/2	1/2	1	1	1	
C	1/3	1/3	1/3	1/2	1/2				

CFS – Reality

- In reality there are things that prevent us from having a “perfect multi-tasking processor”
  - Time to context switch
  - Time for the scheduler run
  - Time spent running other things in the krenel that dno’t really belong to a single task
  - Etc.

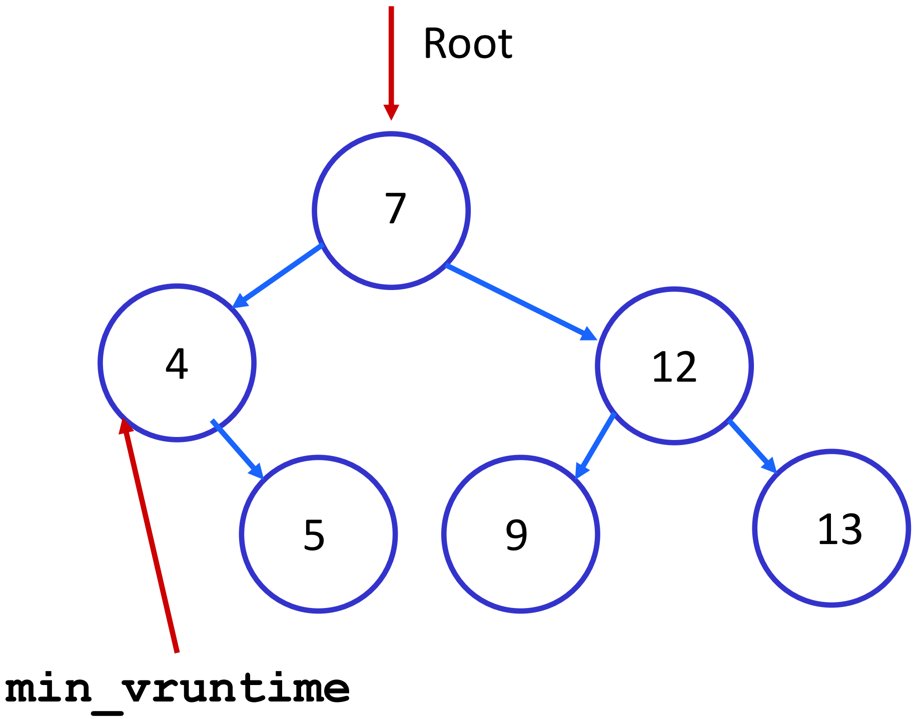
CFS – Implementation

- CFS maintains a current count for "how long has a task run" called `vruntime`.
- The runtimes of all tasks are stored by the scheduler
- Unlike round robin, a thread is not run for a fixed amount of time
  - Run a task till there is some thing with a lower `vruntime`
  - To avoid constantly switching back and forth between two tasks there is a minimum "granularity" (~2.25 miliseconds iirc)

CFS – Implementation Details

- CFS maintains a current count for "how long has a task run" called `vruntime`.

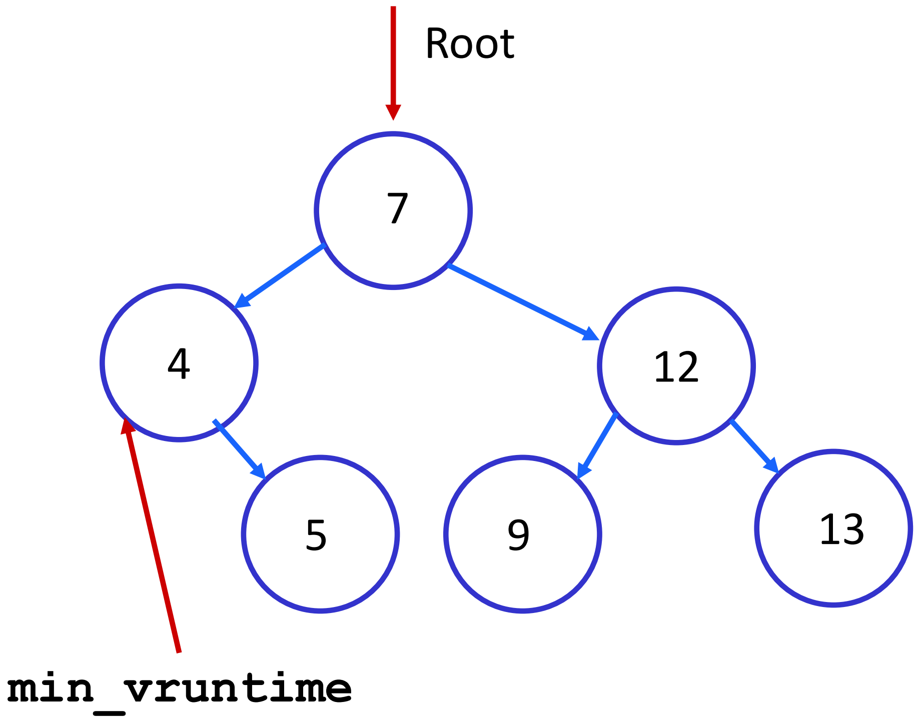
- The runtimes of all tasks are stored by the scheduler inside of a Red-Black Tree
  - Red-Black Tree is a Self balancing binary tree
  - Sorted on the vruntime for each task
  - Smallest vruntime task is the leftmost node



- Adding a node is O(log N) operation
- Pointer to leftmost node is maintained, so looking up is O(1)

CFS – Implementation Details

- CFS maintains a current count for “how long has a task run” called `vruntime`.
- On each scheduler “tick” the processor compares the current running task to the leftmost task
- If the min_vruntime is less than the current node (and granularity has passed) then start running the minimum task.



CFS – New Tasks

- New tasks haven't run on the CPU, so their **vruntime** is 0 when they are created?
  - No, instead new tasks start with their vruntime equal to the min_vruntime.
  - This way fairness is maintained between newer and older tasks.

CFS – I/O Bound Tasks

- CFS will also maintain whether a job is sleeping or blocked. Won't schedule to run those tasks and store them in a separate structure.

- CFS handles I/O bound tasks pretty well :)

- Tasks with many I/O bursts will have small usage of CPU.
  So they also have a low vruntime and have higher priority.

nice

- nice

nice

- Linux has a way to set priority with a `nice` value.
  - Each process starts with a nice value of 0
  - Nice is clamped to [-20, 19]

- The higher your nice score, the “nicer” you are
  (the task runs less often thus letting other tasks run instead of it)

- Higher nice score -> lower priority
- Lower nice score -> higher priority

CFS – Vruntime

- CFS uses vruntime as the dominant metric
  - V stands for virtual (into not real runtime)

- You may have though:
  - `curr_task->runtime += time_running`
  - This is false

- vruntime takes other things (like nice scores) into consideration
  - `curr_task->vruntime += (time_running * weight_based_on_nice)`

- CFS takes other things into consideration that make it more complex :)

Earliest Eligible Virtual Deadline First (EEVDF)

- New Linux scheduler!
  - Replaced CFS less than a year ago (April 2024)
  - Still aims for fairness, just with some different metrics

- Utilizes a new concept called “lag” (in addition to vruntime)
  - A measurement for how much time a task is “owed” if it did not get its fair share of time
  - Tasks that took more CPU time than its fair share have negative “lag”
    - Will not be considered “Eligible”. will not be run until lag >= 0
    - Sleeping / blocked tasks will not get free lag increases

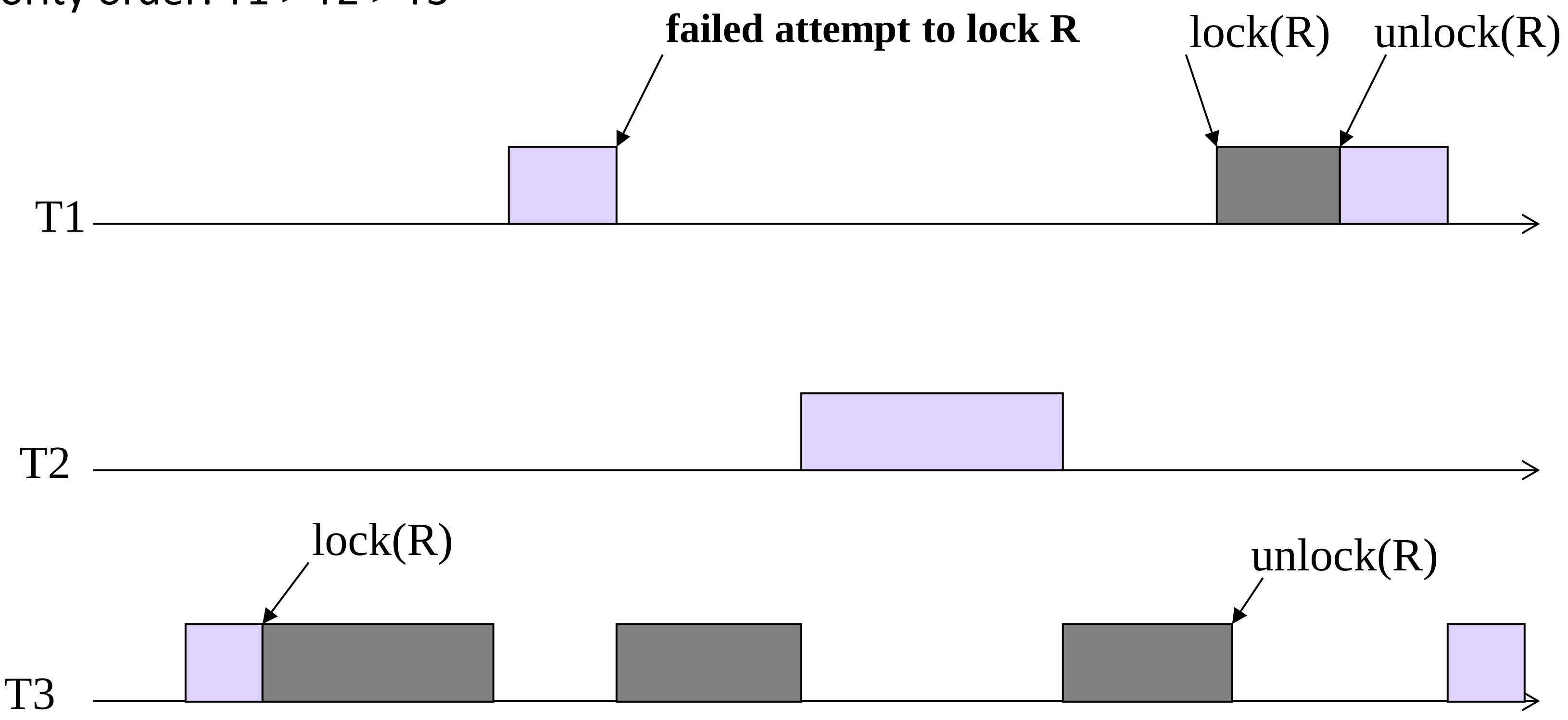
Earliest Eligible Virtual Deadline First (EEVDF)

- Not going over it due to:
  - Time in lecture, looks like it may be more complex and take longer to explain
  - It is new! Not as much information out there on it
    - I could read the Linux kernel source code, but that takes time :)))))))

- Take a look at these articles from LWN.net if you want to learn more about EEVDF
  - https://lwn.net/Articles/925371/
  - https://lwn.net/Articles/969062/

The Priority Inversion Problem

Priority order: T1 > T2 > T3



T2 is causing a higher priority task T1 wait !

Why did we talk about this?

- Scheduling is fundamental towards 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

More

- For those curious, there was a LOT left out

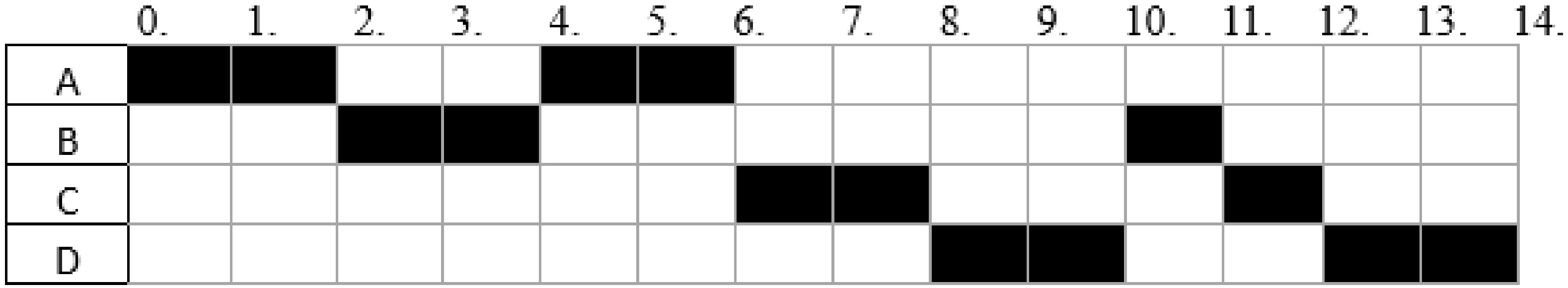
- RTOS (Real Time Operating Systems)
  - For real time applications
  - CRITICAL that data and events meet defined time constraints
  - Different focus in scheduling. Throughput is de-prioritized

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

- Etc.

More Round Robin Practice

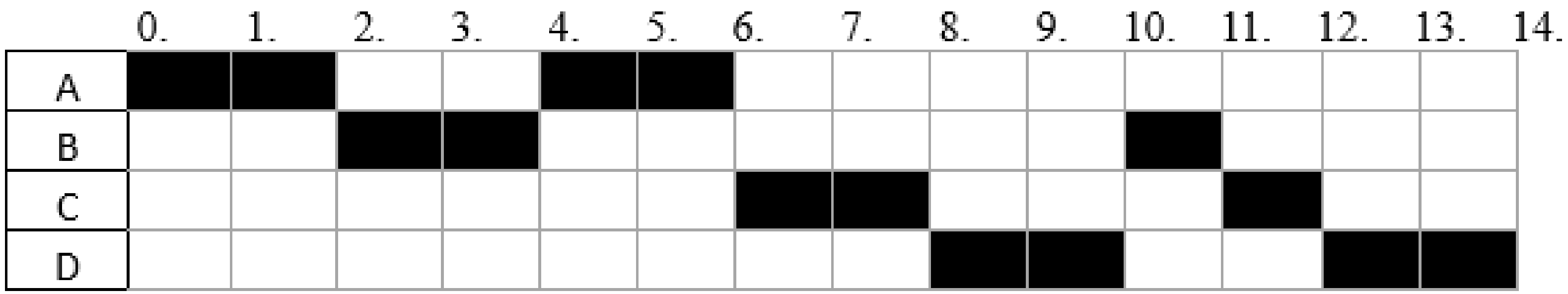
- Four processes are executing on one CPU following round robin scheduling:



- You can assume:
  - All processes do not block for I/O or any resource.
  - Context switching and running the Scheduler are instantaneous.
  - If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

Solutions at end of slide deck

More Round Robin Practice



- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

- What is the earliest time that process C could have arrived?
- Which processes are in the ready queue at time 9?
- If this algorithm used a quantum of 3 instead of 2, how many fewer context switches would there be?

Solutions at end of slide deck

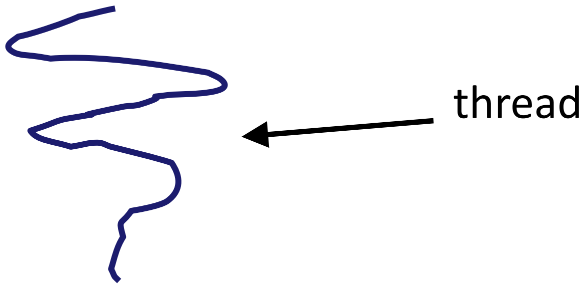
Lecture Outline

- Scheduler
  - Round robin variants
  - Linux Scheduler
- **Threads & Shared Data**
  - **Thread Refresher**
  - Mutex
  - TSL
  - Disable Interrupts
  - Petersons

Lecture ended right before TSL

Introducing 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

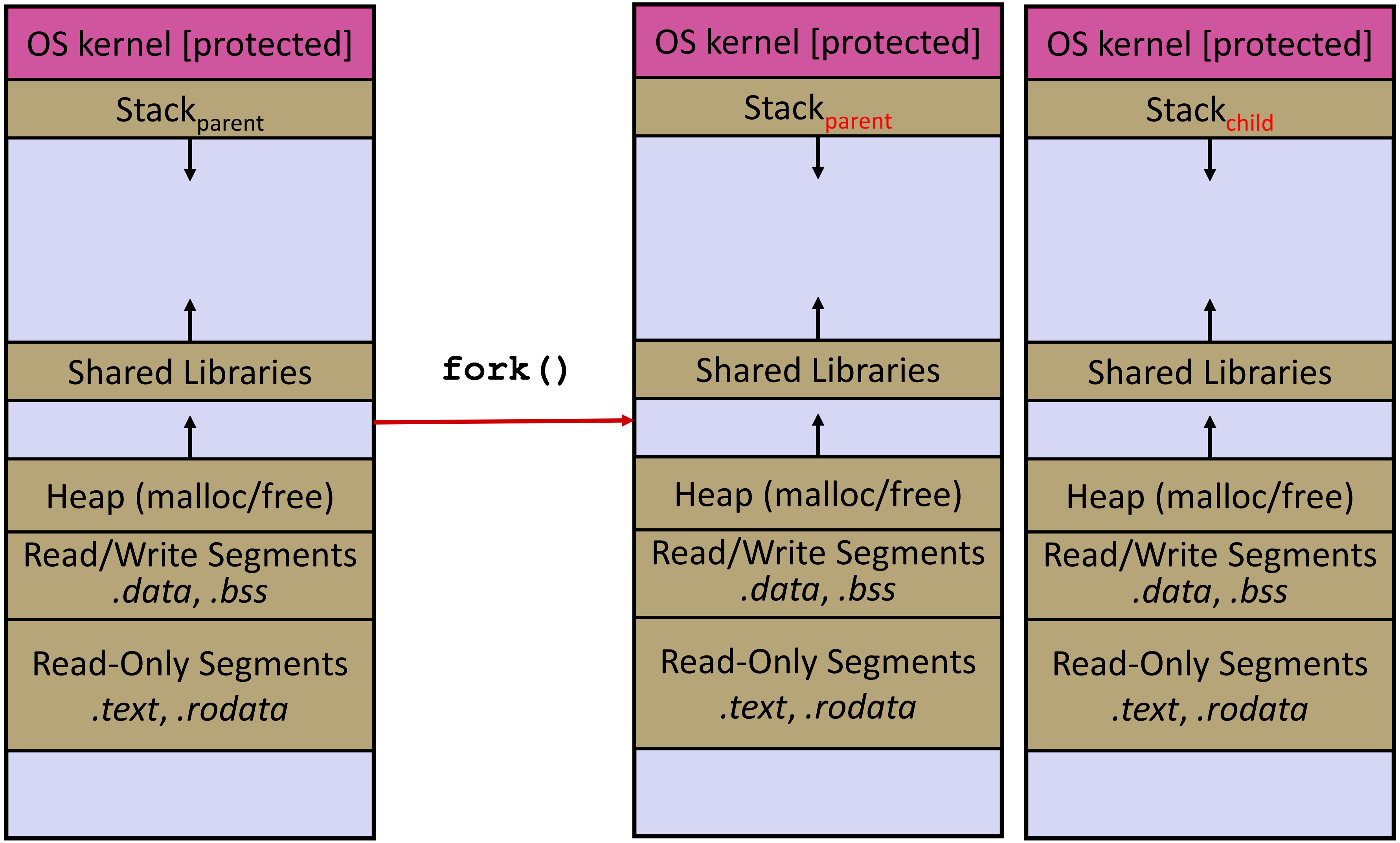


- In most modern OS's:
  - Threads are the *unit of scheduling.*

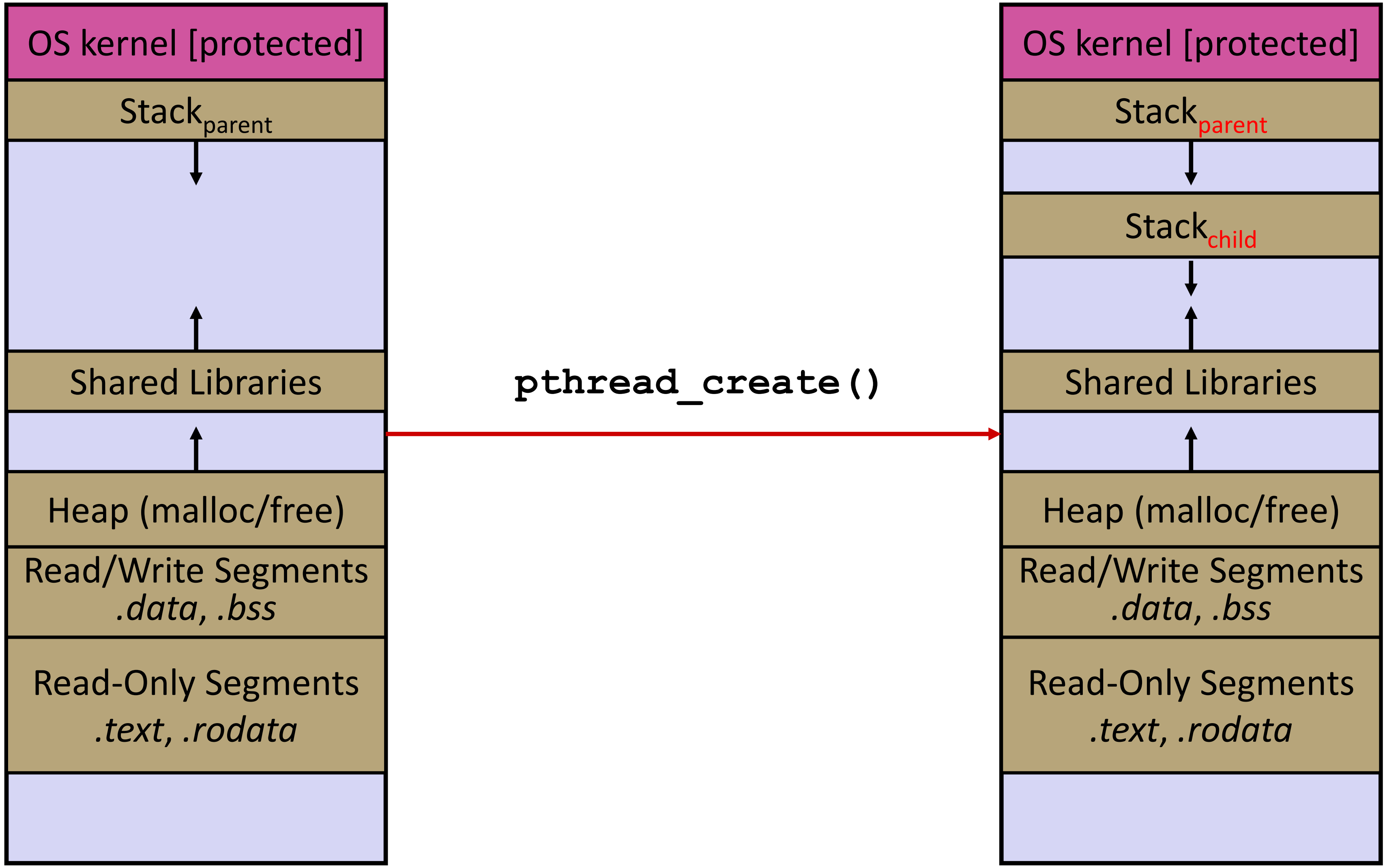
Threads vs. Processes

- In most modern OS's:
  - A Process has a unique: address space, OS resources, & security attributes
  - A Thread has a unique: stack, stack pointer, program counter, & registers
  - Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it

Threads vs. Processes



Threads vs. Processes



POSIX Threads (pthreads)

- The POSIX APIs for dealing with threads
  - Declared in `pthread.h`
    - Not part of the C/C++ language
  - To enable support for multithreading, must include `-pthread` flag when compiling and linking with `gcc` command
    - `gcc -g -Wall -pthread -o main main.c`
  - Implemented in C
    - Must deal with C programming practices and style

Creating and Terminating Threads

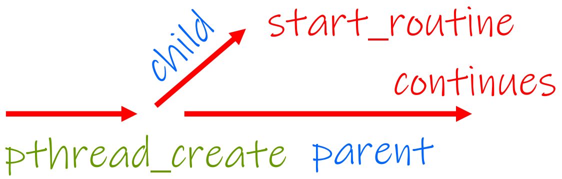
Output parameter.
Gives us a "thread_descriptor"

```
int pthread_create(
        pthread_t* thread,
        const pthread_attr_t* attr,
        void* (*start_routine)(void*),
        void* arg);
```

← Argument for the thread function

Function pointer!
Takes & returns void*
to allow "generics" in C

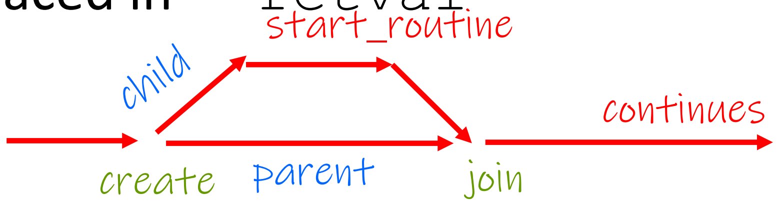
- Creates a new thread into `*thread`, with attributes `*attr` (`NULL` means default attributes)
- Returns **0** on success and an error number on error (can check against error constants)
- The new thread runs **`start_routine`**`(arg)`



What To Do After Forking Threads?

- ```
  int pthread_join(pthread_t thread, void** retval);
  ```
  - Waits for the thread specified by `thread` to terminate
  - The thread equivalent of **`waitpid()`**
  - The exit status of the terminated thread is placed in `**retval`

Parent thread waits for child thread to exit, gets the child's return value, and child thread is cleaned up


  ```

Why Threads?

- Advantages:
  - You (mostly) write sequential-looking code
  - Threads can run in parallel if you have multiple CPUs/cores
    - Takes advantage of the multiple cores
    - Can make progress on multiple tasks at once, even if only 1 core

- Disadvantages:
  - If threads share data, you need locks or other synchronization
    - Very bug-prone and difficult to debug
  - Threads can introduce overhead
    - Lock contention, context switch overhead, and other issues
  - Need language support for threads

Lecture outline

- Scheduler
  - Round robin variants
  - Linux Scheduler
- **Threads & Shared Data**
  - Thread Refresher
  - **Mutex**
  - TSL
  - Disable Interrupts
  - Petersons

Lecture ended right before TSL

Shared Resources

- Some resources are shared between threads and processes

- Thread Level:
  - Memory
  - Things shared by processes

- Process level
  - I/O devices
    - Files
    - terminal input/output
    - The network

Issues arise when we try to shared things

Data Races

- Two memory accesses form a data race if different threads access the same location, and at least one is a write, and they occur one after another
  - Means that the result of a program can vary depending on chance **(which thread ran first? When did a thread get interrupted?)**

Data Race Example

- If your fridge has no milk, then go out and buy some more
  - What could go wrong?

```
if (!milk) {

  buy milk

}
```

- If you live alone:

- If you live with a roommate:

Poll Everywhere

pollev.com/tqm

- Idea: leave a note!
  - Does this fix the problem?

```
if (!note) {
  if (!milk) {
    leave note
    buy milk
    remove note
  }
}
```

A. Yes, problem fixed

B. No, could end up with no milk

C. No, could still buy multiple milk

D. We're lost…

Poll Everywhere

pollev.com/tqm

- Idea: leave a note!
  - Does this fix the problem?

```
if (!note) {
  if (!milk) {
    leave note
    buy milk
    remove note
  }
}
```

We can be interrupted between checking note and leaving note ☹

A. **Yes, problem fixed**

B. **No, could end up with no milk**

C. **No, could still buy multiple milk**

D. **We're lost…**

*There are other possible scenarios that result in multiple milks

you	roommate
Check note	
	Check note
Check milk	
Leave note	
	Check milk
	Leave note
	Buy milk
Buy milk	

time ↓

Threads and Data Races

- Data races might interfere in painful, non-obvious ways, depending on the specifics of the data structure
- Example: two threads try to read from and write to the same shared memory location
  - Could get "correct" answer
  - Could accidentally read old value
  - One thread's work could get "lost"
- Example: two threads try to push an item onto the head of the linked list at the same time
  - Could get "correct" answer
  - Could get different ordering of items
  - Could break the data structure! ☠

Poll Everywhere

pollev.com/tqm

- What does this print?

```
#define NUM_PROCESSES 50
#define LOOP_NUM 100

int sum_total = 0;

void loop_incr() {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
}


int main(int argc, char** argv) {
  pid_t pids[NUM_PROCESSES];  // array of process ids

  // create processes to run loop_incr()
  for (int i = 0; i < NUM_PROCESSES; i++) {
    pids[i] = fork();
    if (pids[i] == 0) {
      // child
      loop_incr();
      exit(EXIT_SUCCESS);
    }
    // parent loops and forks more children
  }

  // wait for all child processes to finish
  for (int i = 0; i < NUM_PROCESSES; i++) {
    waitpid(pids[i], NULL, 0);
  }

  printf("%d\n", sum_total);

  return EXIT_SUCCESS;
}
```

Always prints 0, the global counter is not shared across processes, so the parent's global never changes

Poll Everywhere

pollev.com/tqm

- What does this print?

```
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

[redacted]

void* thread_main(void* arg) {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  return NULL;  // return type is a pointer
}


int main(int argc, char** argv) {
  pthread_t thds[NUM_THREADS];  // array of thread ids

  // create threads to run thread_main()
  for (int i = 0; i < NUM_THREADS; i++) {
    pthread_create(&thds[i], NULL, &thread_main, NULL);
  }

  // wait for all child threads to finish
  // (children may terminate out of order, but cleans up in order)
  for (int i = 0; i < NUM_THREADS; i++) {
    pthread_join(thds[i], NULL);
  }

  printf("%d\n", sum_total);

  return EXIT_SUCCESS;
}
```

Usually 5000

Demos:

- See `total.c` and `total_processes.c`
  - Threads share an address space, if one thread increments a global, it is seen by other threads
  - Processes have separate address spaces, incrementing a global in one process does not increment it for other processes

- NOTE: sharing data between threads is actually kinda unsafe if done wrong (we are doing it wrong in this example), **more on this NOW**

Increment Data Race

- What seems like a single operation is actually multiple operations in one. The increment looks something like this in assembly:

```
++sum_total
```

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

- What happens if we context switch to a different thread while executing these three instructions?

- **Reminder: Each thread has its own registers to work with. Each thread would have its own R0**

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 0

Thread 0 R0 = 0

```
LOAD sum_total into R0
```

Thread 1

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 0

Thread 0 **R0 = 0**

```
LOAD  sum_total into R0
```

Thread 1 **R0 = 0**

```
LOAD  sum_total into R0
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 0

Thread 0 **R0 = 0**

```
LOAD   sum_total into R0
```

Thread 1 **R0 = 1**

```
LOAD   sum_total into R0
ADD    R0 R0 #1
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 1

Thread 0 **R0 = 0**

```
LOAD  sum_total into R0
```

Thread 1 **R0 = 1**

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 1

Thread 0 **R0 = 1**

```
LOAD   sum_total into R0



ADD    R0 R0 #1
```

Thread 1 **R0 = 1**

```
LOAD   sum_total into R0
ADD    R0 R0 #1
STORE  R0 into sum_total
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 1

Thread 0 **R0 = 1**

```
LOAD  sum_total into R0



ADD   R0 R0 #1
STORE R0 into sum_total
```

Thread 1 **R0 = 1**

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

- With this example, we could get 1 as an output instead of 2, even though we executed ++sum_total twice

Synchronization

- Synchronization is the act of preventing two (or more) concurrently running threads from interfering with each other when operating on shared data
  - Need some mechanism to coordinate the threads
    - “Let me go first, then you can go”
  - Many different coordination mechanisms have been invented

- Goals of synchronization:
  - Liveness – ability to execute in a timely manner (informally, “something good eventually happens”)
  - Safety – avoid unintended interactions with shared data structures (informally, “nothing bad happens”)

Lock Synchronization

- Use a "Lock" to grant access to a *critical section* so that only one thread can operate there at a time
  - Executed in an uninterruptible (*i.e.* atomic) manner

- Lock Acquire
  - Wait until the lock is free, then take it

- Lock Release
  - Release the lock
  - If other threads are waiting, wake exactly one up to pass lock to

- Pseudocode:

```
// non-critical code

lock.acquire();    // block if locked
// critical section
lock.release();

// non-critical code
```

Lock API

- Locks are constructs that are provided by the operating system to help ensure synchronization
  - Often called a mutex or a semaphore

- Only one thread can acquire a lock at a time,
  No thread can acquire that lock until it has been released

- Has memory barriers built into it and usually uses TSL to ensure that acquiring the lock is atomic (more on TSL and memory barriers in a little bit)

Milk Example – What is the Critical Section?

- What if we use a lock on the refrigerator?
  - Probably overkill – what if roommate wanted to get eggs?

- For performance reasons, only put what is necessary in the critical section
  - Only lock the milk
  - But lock *all* steps that must run uninterrupted (*i.e.* must run as an atomic unit)

```
fridge.lock()
if (!milk) {
  buy milk
}
fridge.unlock()
```

↓

```
milk_lock.lock()
if (!milk) {
  buy milk
}
milk_lock.unlock()
```

pthreads and Locks

- Another term for a lock is a mutex ("mutual exclusion")
  - `pthread.h` defines datatype `pthread_mutex_t`

- ```
  int pthread_mutex_init(pthread_mutex_t* mutex,
                         const pthread_mutexattr_t* attr);
  ```
  - Initializes a mutex with specified attributes

- ```
  int pthread_mutex_lock(pthread_mutex_t* mutex);
  ```
  - Acquire the lock – blocks if already locked Un-blocks when lock is acquired

- ```
  int pthread_mutex_unlock(pthread_mutex_t* mutex);
  ```
  - Releases the lock

- ```
  int pthread_mutex_destroy(pthread_mutex_t* mutex);
  ```
  - "Uninitializes" a mutex – clean up when done

pthread Mutex Examples

- See `total.c`
  - Data race between threads
- See `total_locking.c`
  - Adding a mutex fixes our data race
- How does `total_locking` compare to sequential code and to `total`?
  - Likely *slower* than both– only 1 thread can increment at a time, and must deal with checking the lock and switching between threads
  - One possible fix: each thread increments a local variable and then adds its value (once!) to the shared variable at the end
    - See `total_locking_better.c`

Lecture Outline

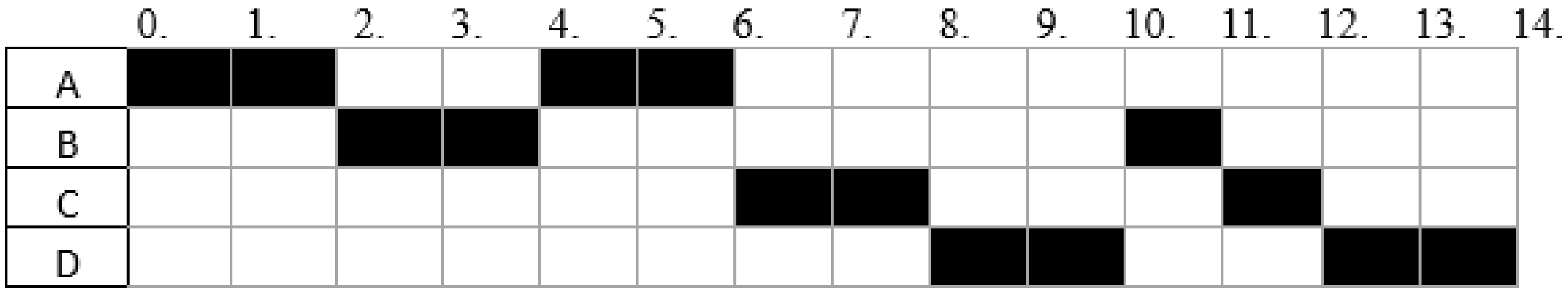
- Scheduler
  - Round robin variants
  - Linux Scheduler
- **Threads & Shared Data**
  - Thread Refresher
  - Mutex
  - **TSL**
  - Disable Interrupts
  - Petersons

Lecture ended right before TSL
We will cover it after break!

That’s all!

- See you next time!

More Round Robin Practice

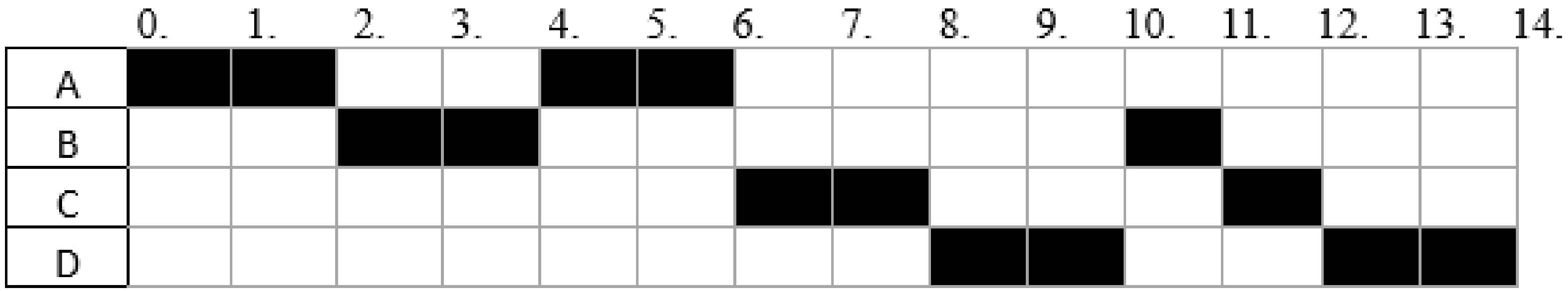


- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

❖ What is the earliest time that process C could have arrived?

- If C arrived at time 0, 1, or 2, it would have run at time 4
- C could have shown up at time 3 and come after A in the queue
- C showed up at time 3 at earliest

More Round Robin Practice

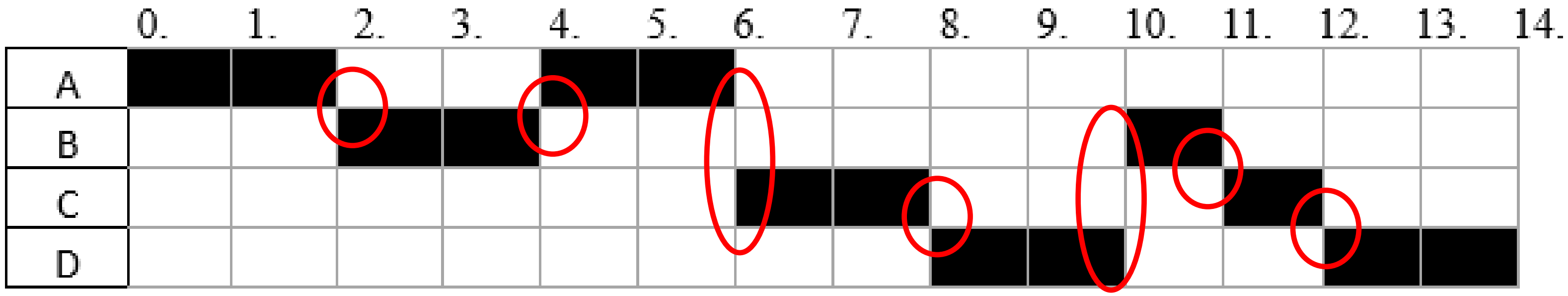


- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

❖ Which processes are in the ready queue at time 9?

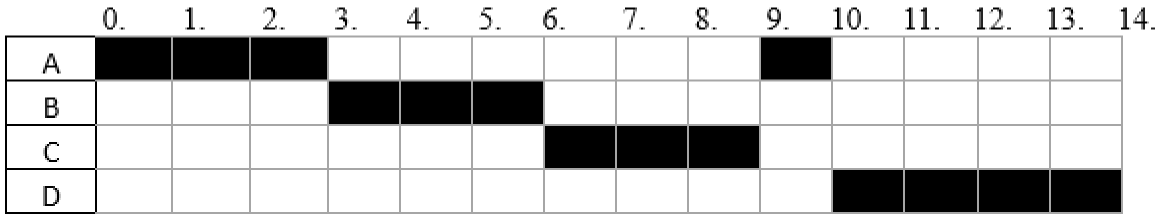
- D is running, so it is not in the queue
- A has finished
- B and C still have to finish, so they are in the queue.

More Round Robin Practice

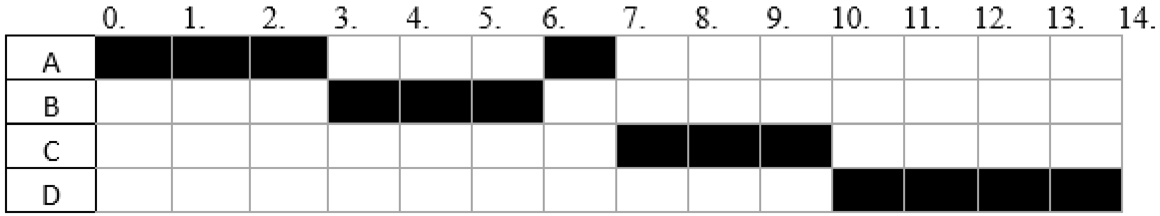


- If this algorithm used a quantum of 3 instead of 2, how many fewer context switches would there be?
  - Currently there are 7 context switches
  - If quantum was 3:

Depends on if C shows up at time 3 or 4



  - Or:



Either way, only 4 context switches, so 3 less than quantum = 2