Scheduler & Intro to Threads

Computer Operating Systems, Summer 2025

Instructors: Joel Ramirez Travis McGaha

TAs: Ash Fujiyama Maya Huizar Sid Sannapareddy

Poll Everywhere

pollev.com/tqm

- How are you doing? Any questions?

Administrivia

- Midterm is Wednesday **THIS WEEK Wednesday 7pm to 9pm**
  - Old exams and exam policies are posted on the course website
  - Review session in class tomorrow!!!!!!!!!!!!
  - What we get to in this lecture will be testable.

- Penn-shell is out (this shouldn't be news)!
  - ***Extension:*** Since autograder was down for 12 hours, and there is an exam this week
  - ***Full thing is due (This Friday!)***
    - ***Can only use ONE late token now: late due date is Sunday the 29th.***
  - ***Done in partners***
    - Everything was covered already that you would need…

Administrivia

- PennOS:
  - Specifications and team sign-up to be posted Thursday (day after exam)
  - Done in groups of 4
  - Partner signup due by end of day on Monday the 30th
    - Those left unassigned will be randomly assigned the next morning (Tuesday the 31st)
  - Lecture dedicated to PennOS in class on Tuesday the 31st. Highly recommend you go.

Lecture Outline

- **Scheduler**
  - **Round robin variants (Cont.)**
  - Linux Scheduler
- Threads Intro
  - Thread High Level
  - pthreads
  - Processes vs threads
  - Thread Interleaving & Sequential Consistency

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)

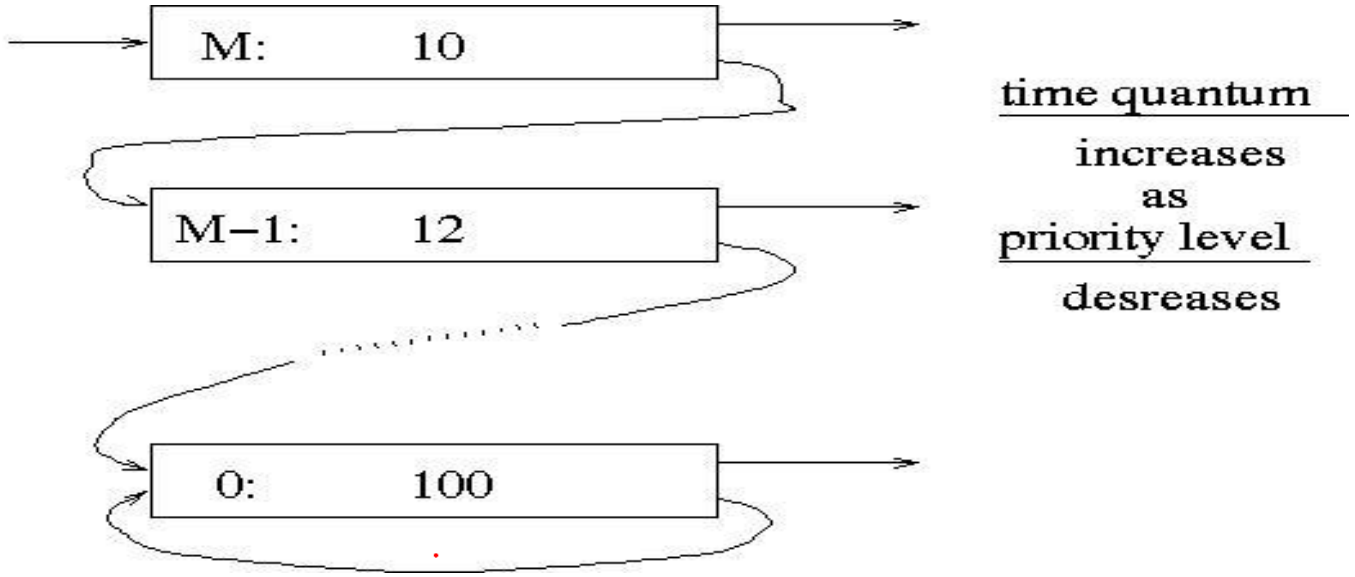
→ - **Preemptive:** the thread may be interrupted after a given time and/or if another thread becomes ready
  - Round Robin
  - Priority Round Robin
  - …

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.

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 (Cont.)
  - **Linux Scheduler**
- Threads Intro
  - Thread High Level
  - pthreads
  - Processes vs threads
  - Thread Interleaving & Sequential Consistency

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

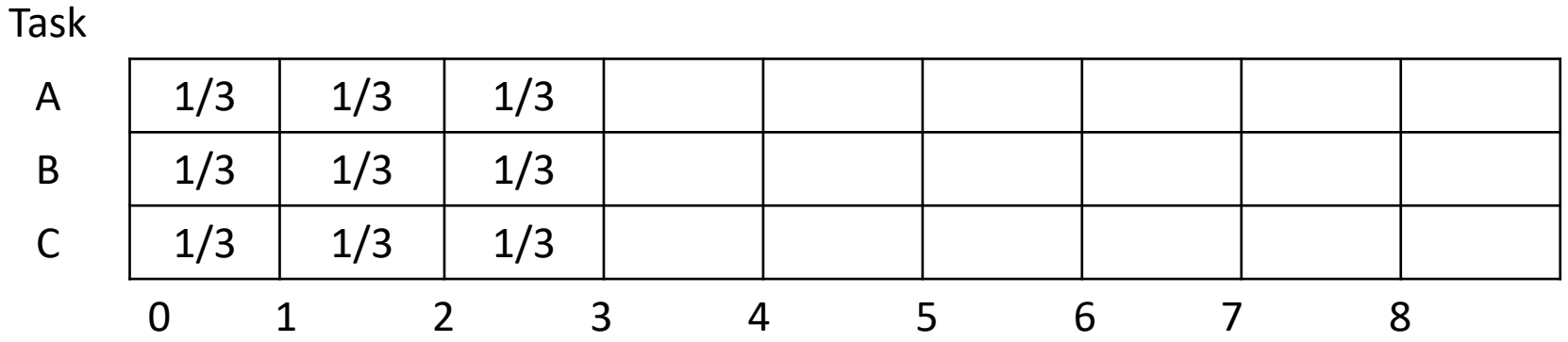
"Completely Fair Scheduling"

- "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



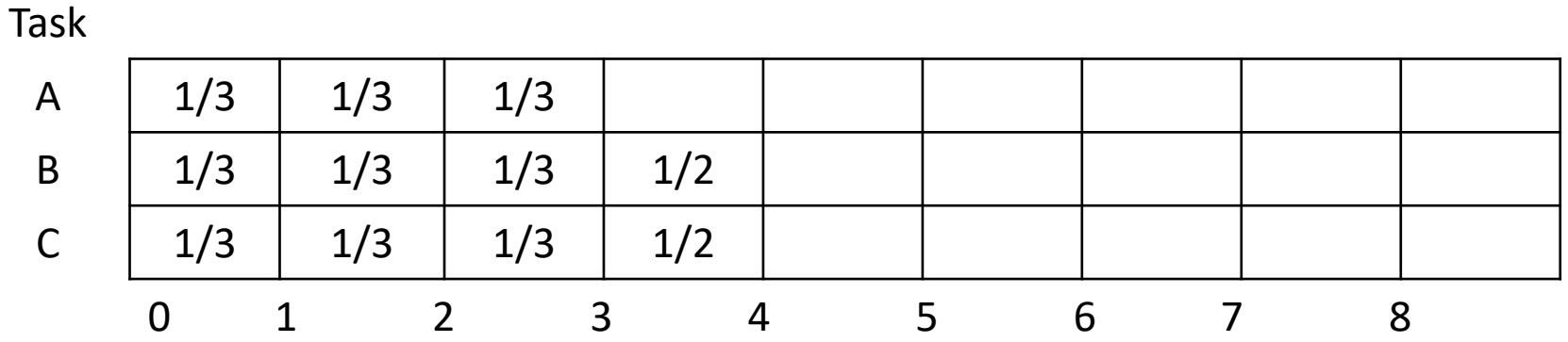
"Completely Fair Scheduling"

- "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



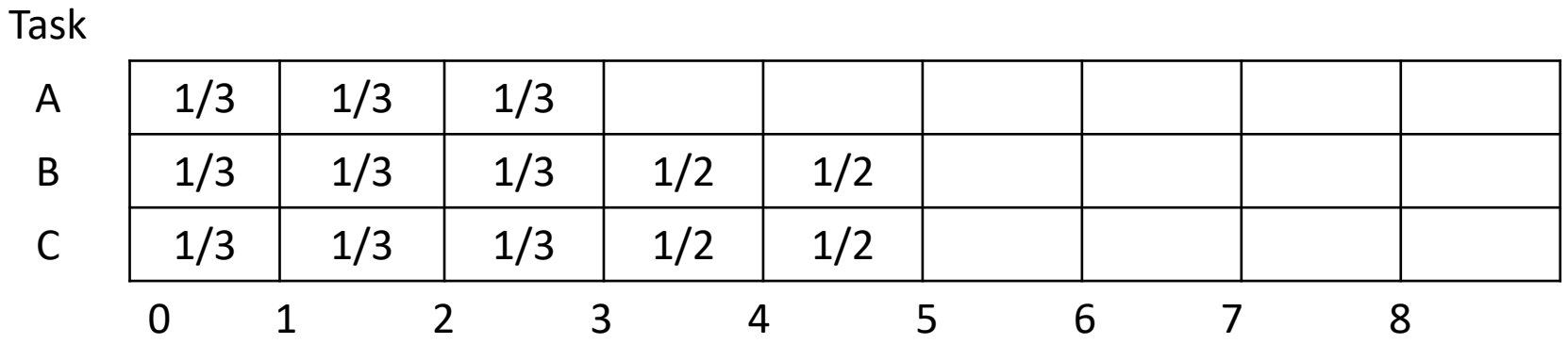
"Completely Fair Scheduling"

- "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



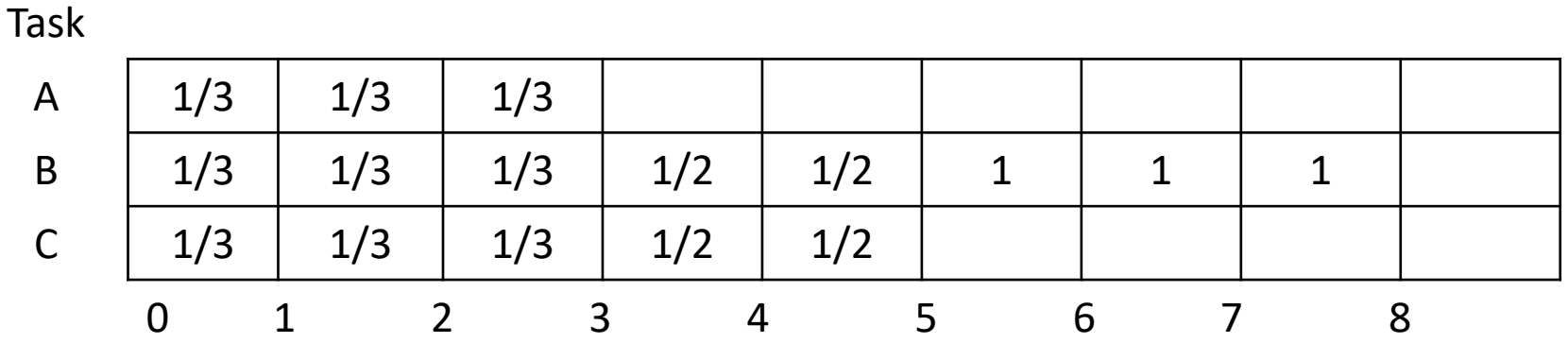
"Completely Fair Scheduling"

- "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



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 kernel that don't really belong to a single task
  - Task may not be pre-emptible sometimes and we need to wait for the task to become pre-emptible.
  - Etc.

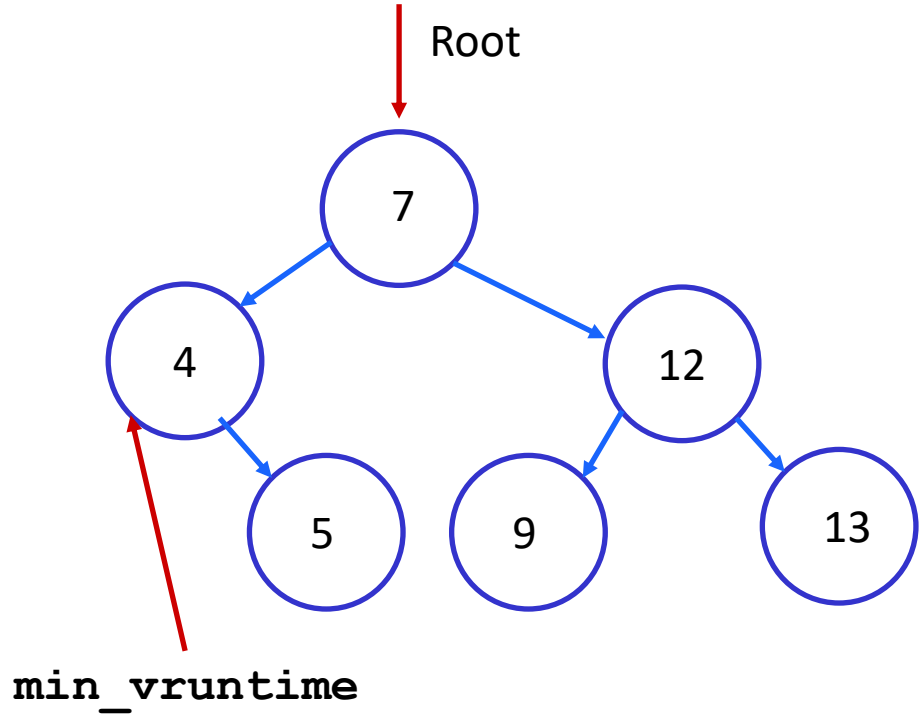
CFS – Implementation

- 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 milliseconds iirc)

CFS – Implementation Details

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

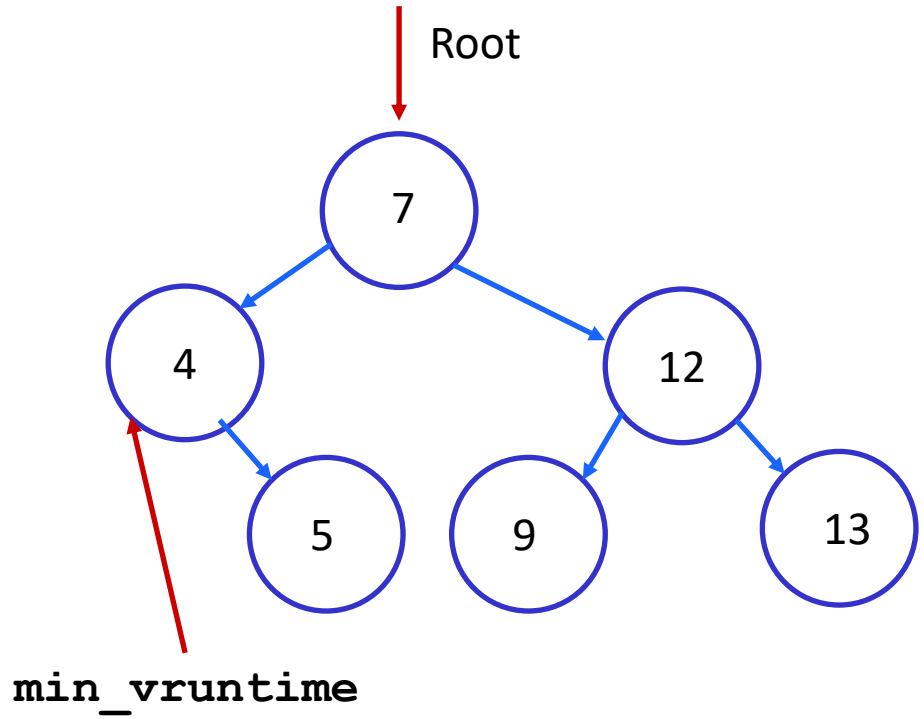
- The runtimes of all tasks are stored by the scheduler inside of a Red-Black Tree
  - Red-Black Tree is a Self balancing binary tree
  - Sorted on the vruntime for each task
  - Smallest vruntime task is the leftmost node



- 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 (e.g. not real runtime)
- You may have thought:
  - `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/

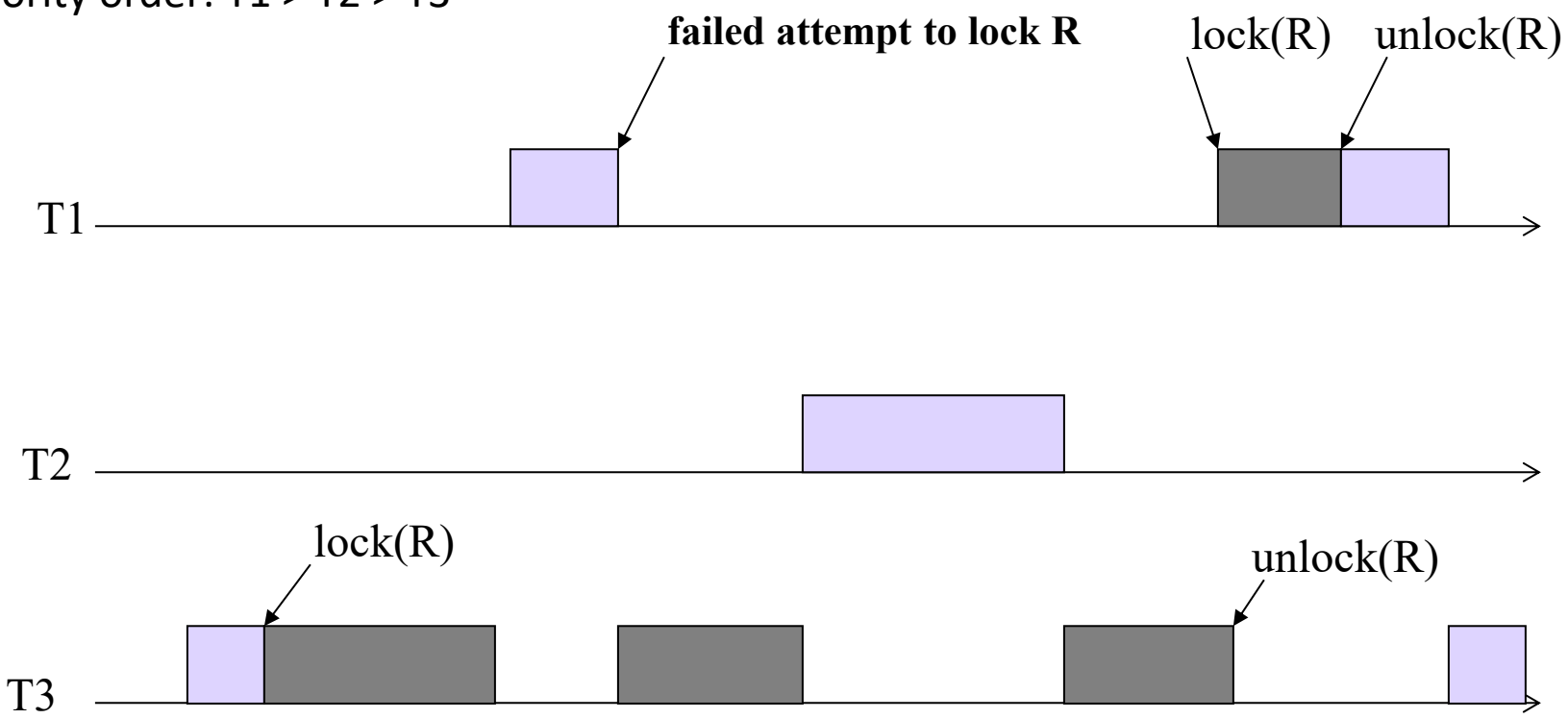
Consideration: Interactive Tasks

- There is still ongoing work to make schedulers that are better
  - "Better" either in general or to specific situations

- Example: People are already working on EEVDF upgrades. VARD scheduler for SteamOS
  https://youtu.be/xJjZ5tzlHOY?si=lgGNWaQe03qSgCP2&t=1682

Another Issue: 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

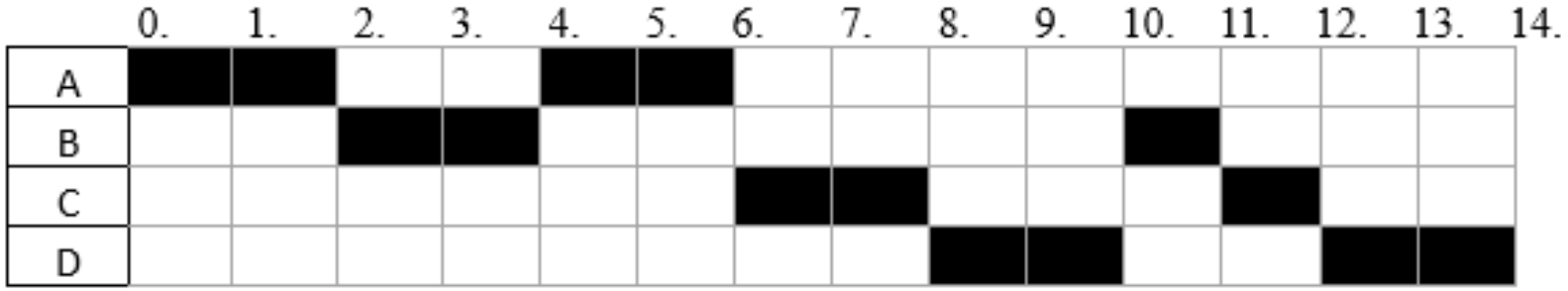
- RTOS (Real Time Operating Systems)
  - For real time applications
  - CRITICAL that data and events meet defined time constraints
  - Different focus in scheduling. Throughput is de-prioritized

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

- Etc.

More Round Robin Practice

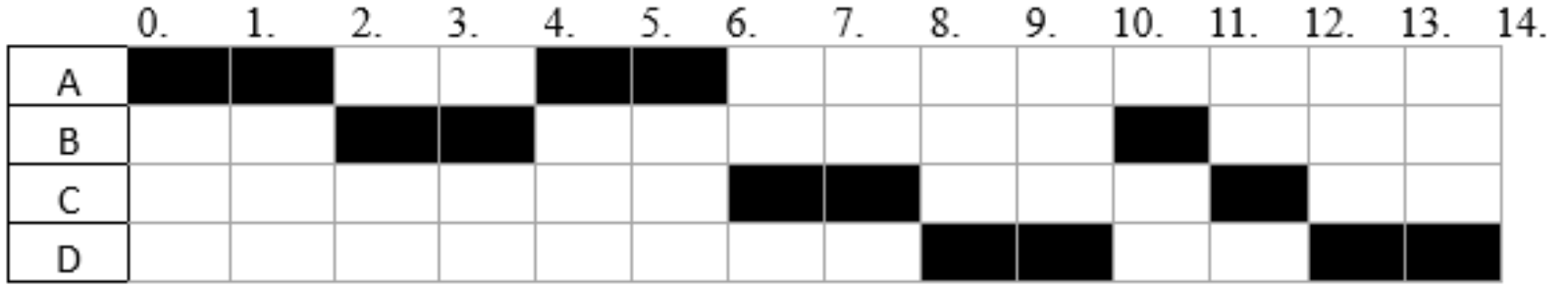
- Four processes are executing on one CPU following round robin scheduling:



- 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?

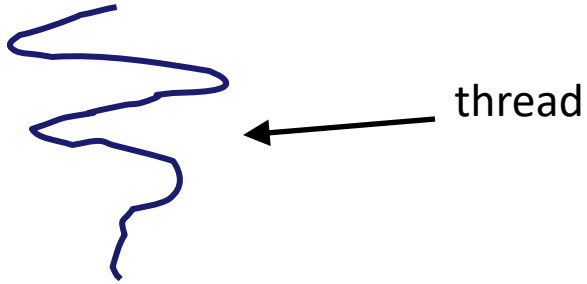
Solutions at end of slide deck

Lecture Outline

- Scheduler
  - Round robin variants (Cont.)
  - Linux Scheduler
- **Threads Intro**
  - **Thread High Level**
  - pthreads
  - Processes vs threads
  - Thread Interleaving & Sequential Consistency

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

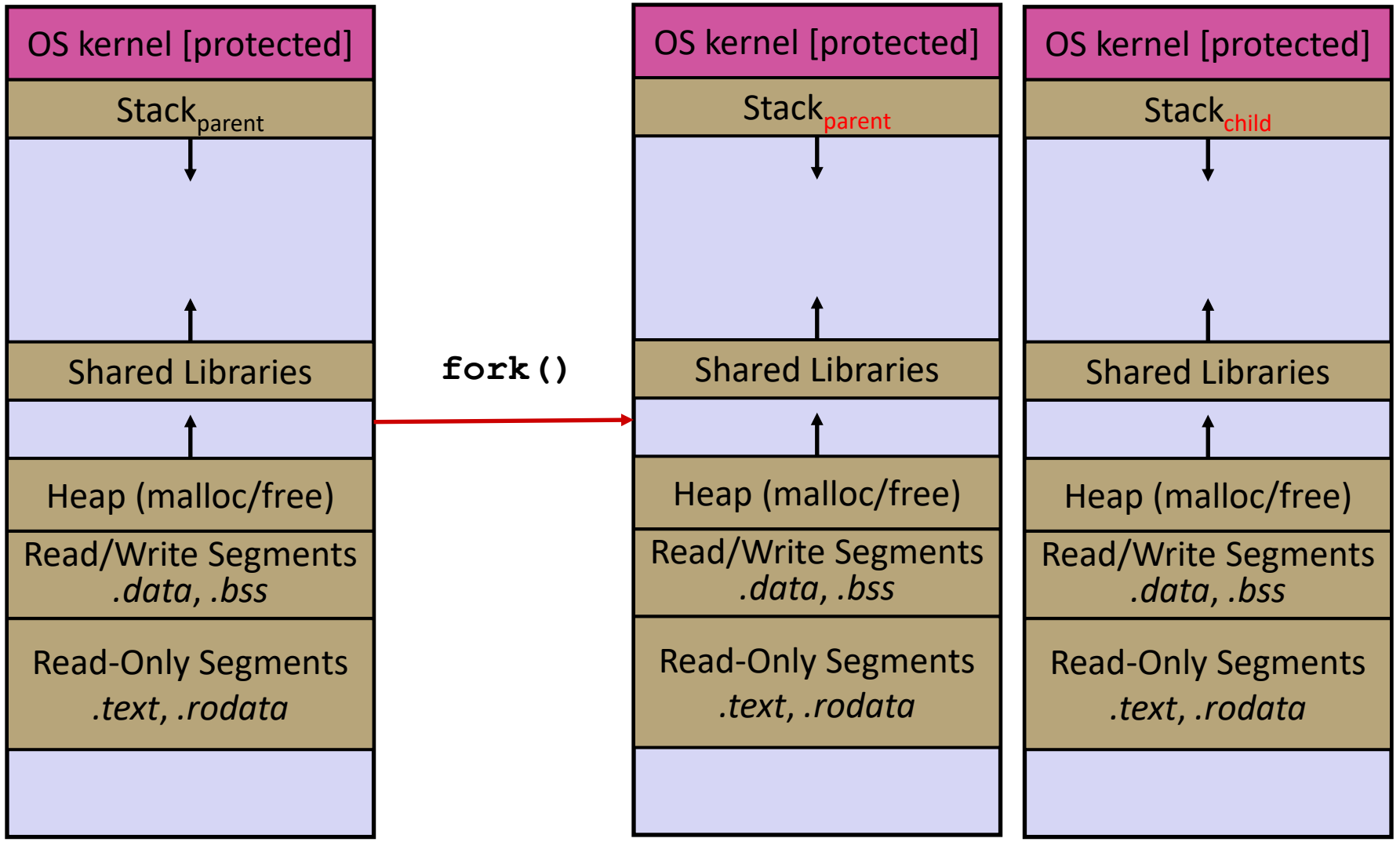


- In most modern OS's:
  - Threads are the *unit of scheduling.*

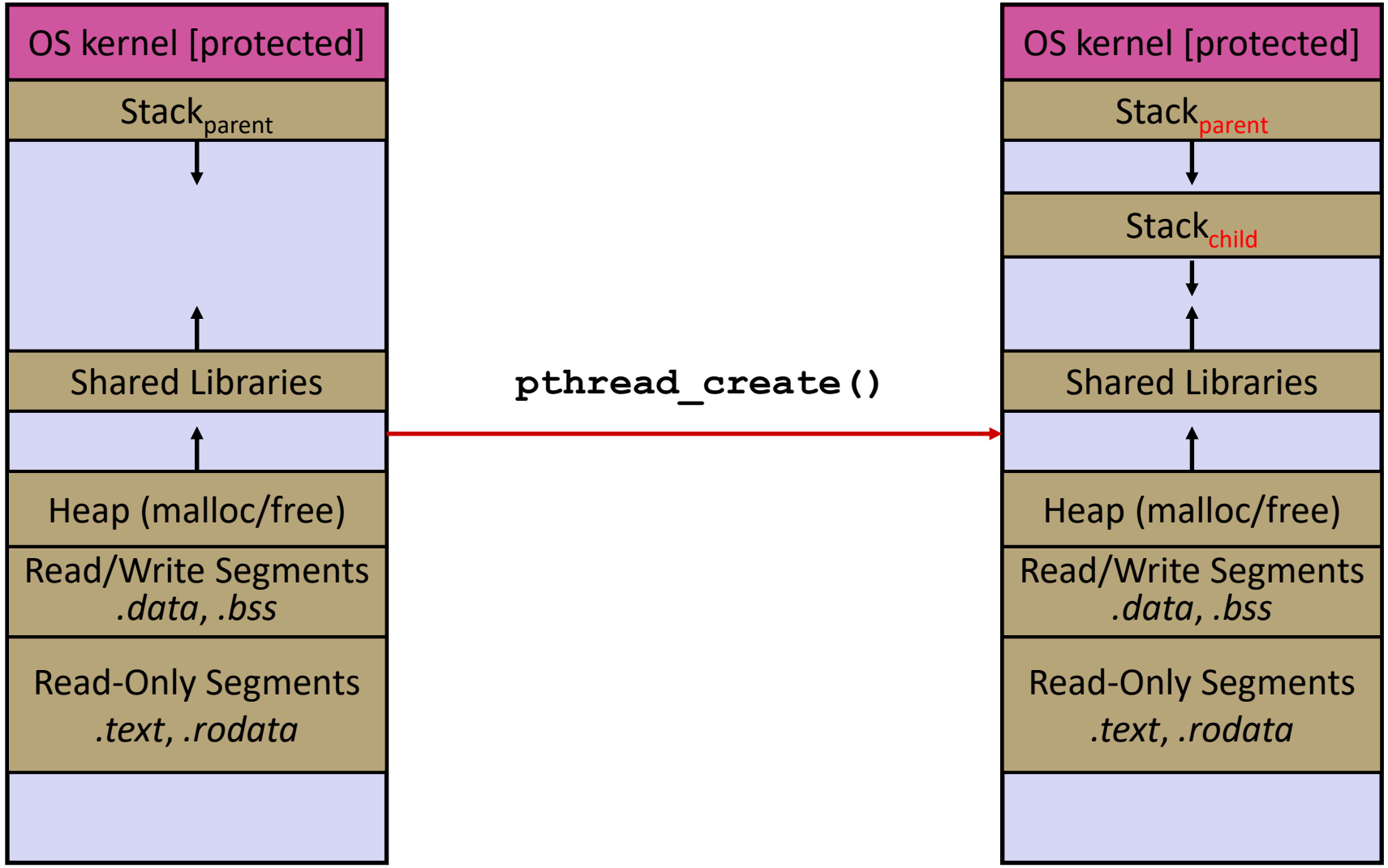
Threads vs. Processes

- 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

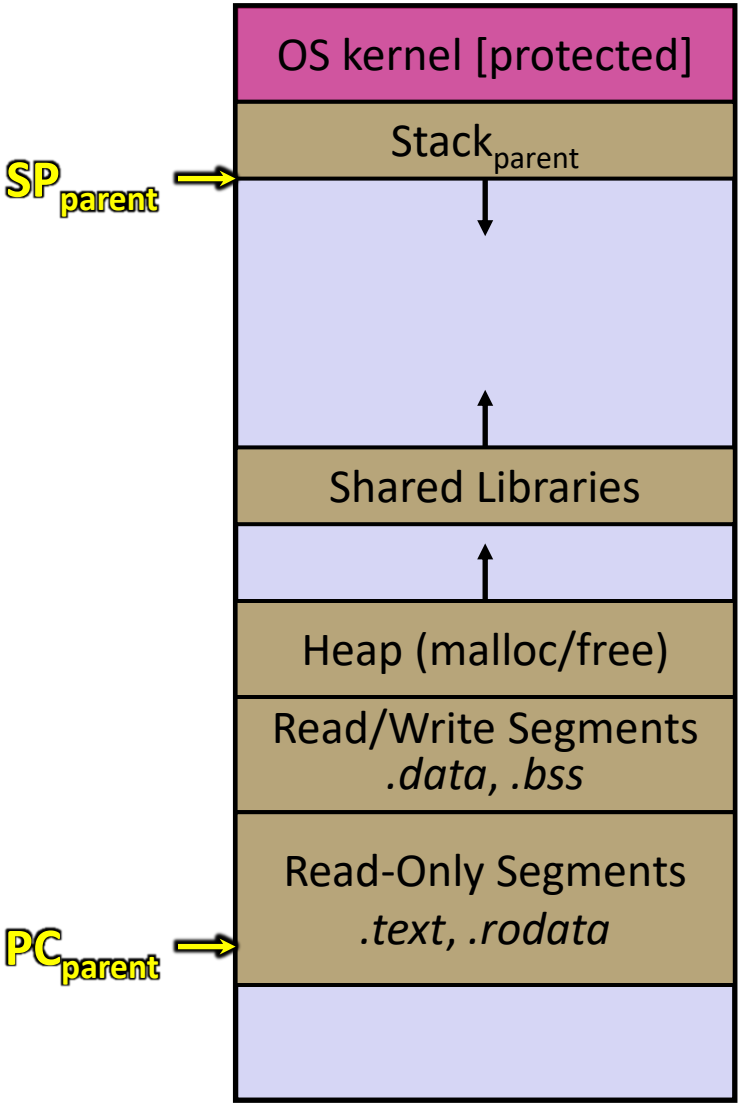


Threads

- Threads are like lightweight processes
  - **They execute concurrently like processes**
    - **Multiple threads can run simultaneously on multiple CPUs/cores**
  - Unlike processes, threads cohabitate the same address space
    - Threads within a process see the same heap and globals and can communicate with each other through variables and memory
      - But, they can interfere with each other – need synchronization for shared resources
    - Each thread has its own stack

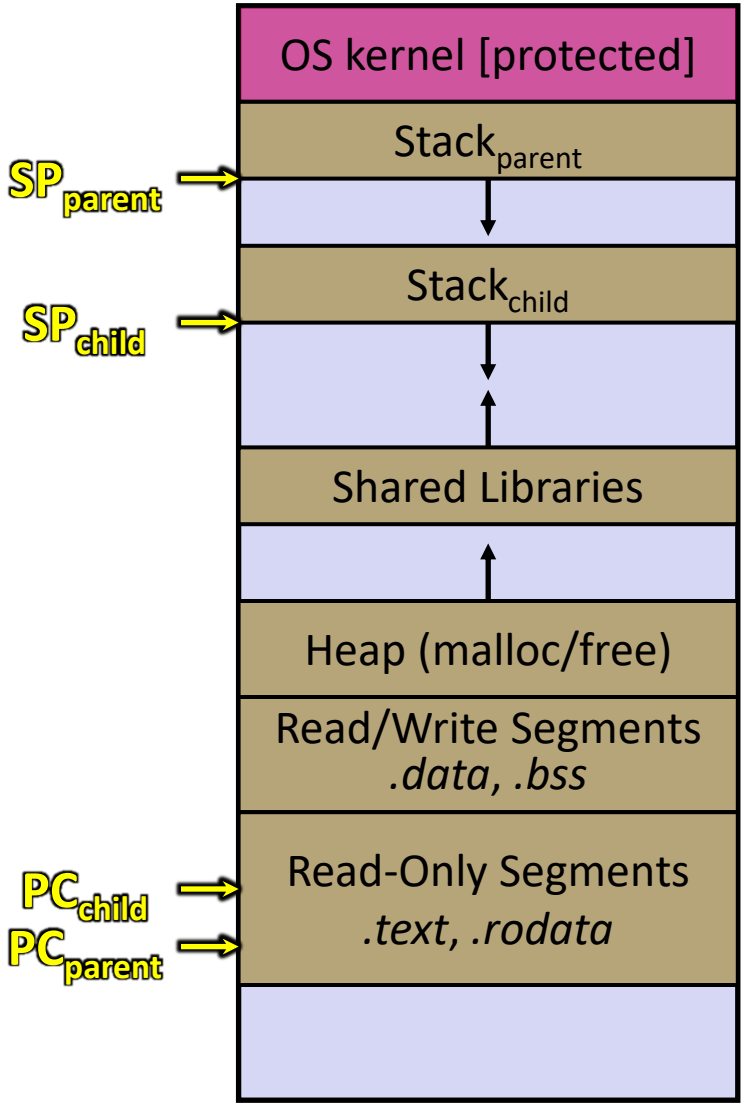
- Analogy: restaurant kitchen
  - Kitchen is process
  - Chefs are threads

Single-Threaded Address Spaces



- Before creating a thread
  - One thread of execution running in the address space
    - One PC, stack, SP
  - That main thread invokes a function to create a new thread
    - Typically `pthread_create()`

Multi-threaded Address Spaces



- After creating a thread
  - *Two* threads of execution running in the address space
    - Original thread (parent) and new thread (child)
    - New stack created for child thread
    - Child thread has its own *values* of the PC and SP
  - Both threads share the other segments (code, heap, globals)
    - They can cooperatively modify shared data

Lecture Outline

- Scheduler
  - Round robin variants (Cont.)
  - Linux Scheduler
- **Threads Intro**
  - Thread High Level
  - **pthreads**
  - Processes vs threads
  - Thread Interleaving & Sequential Consistency

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

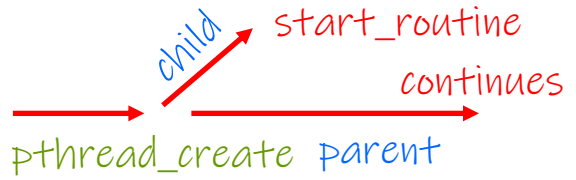
Output parameter.
Gives us a "thread_descriptor"

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

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

Argument for the thread function

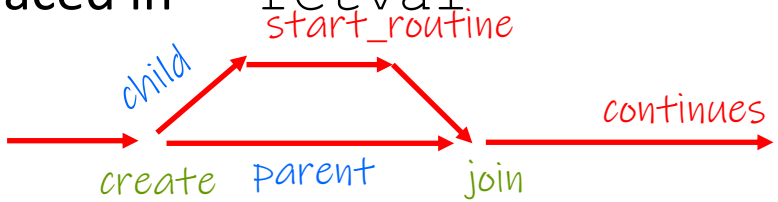
- Creates a new thread into `*thread`, with attributes `*attr` (`NULL` means default attributes)
- Returns **0** on success and an error number on error (can check against error constants)
- The new thread runs **`start_routine`**`(arg)`



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


  ```

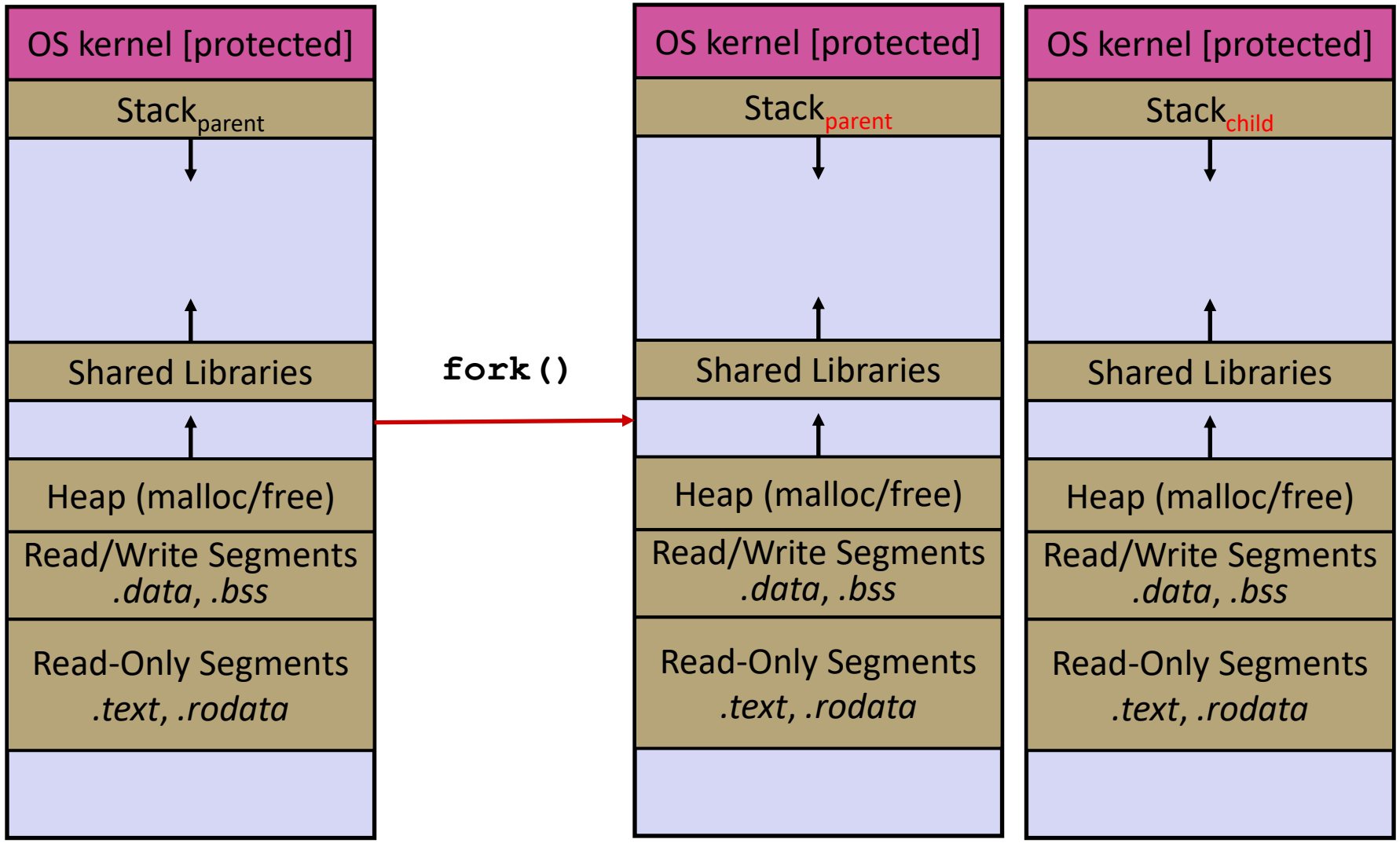
Thread Example

- See `cthreads.c`
  - How do you properly handle memory management?
    - Who allocates and deallocates memory?
    - How long do you want memory to stick around?
  - Threads execute in parallel

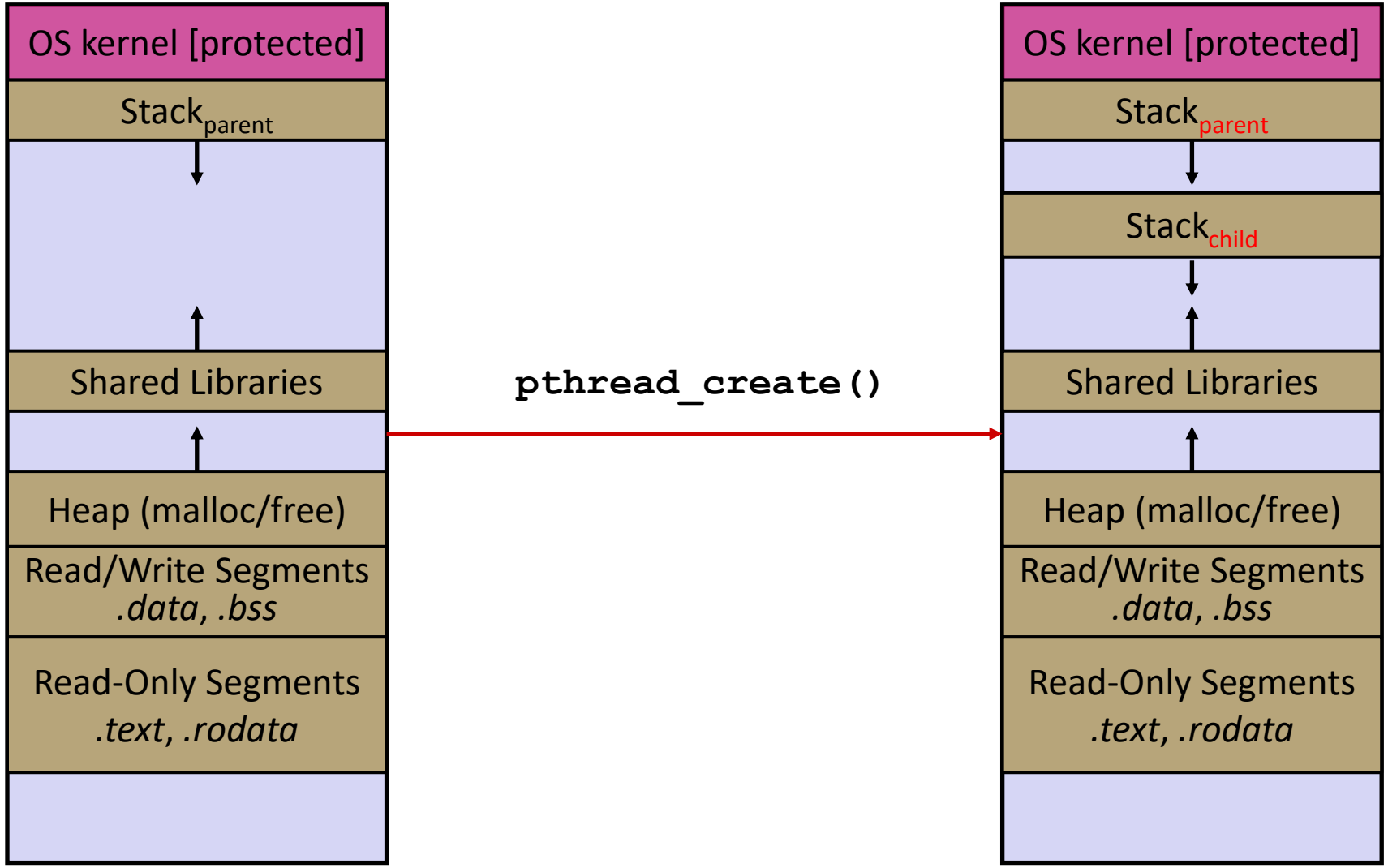
Lecture Outline

- Scheduler
  - Round robin variants (Cont.)
  - Linux Scheduler
- **Threads Intro**
  - Thread High Level
  - pthreads
  - **Processes vs threads**
  - Thread Interleaving & Sequential Consistency

Threads vs. Processes



Threads vs. Processes



Poll Everywhere

pollev.com/tqm

- What does this print?

```c
#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;
}
```

Poll Everywhere

pollev.com/tqm

- What does this print?

```c
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

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++) {
    if (pthread_create(&thds[i], NULL, &thread_main, NULL) != 0) {
      fprintf(stderr, "pthread_create failed\n");
    }
  }

  // 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++) {
    if (pthread_join(thds[i], NULL) != 0) {
      fprintf(stderr, "pthread_join failed\n");
    }
  }

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

  return EXIT_SUCCESS;
}
```

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 next week**

Process Isolation

- Process Isolation is a set of mechanisms implemented to protect processes from each other and protect the kernel from user processes.
  - Processes have separate address spaces
  - Processes have privilege levels to restrict access to resources
  - If one process crashes, others will keep running

- Inter-Process Communication (IPC) is limited, but possible
  - Pipes via pipe()
  - Sockets via socketpair()
  - Shared Memory via shm_open()

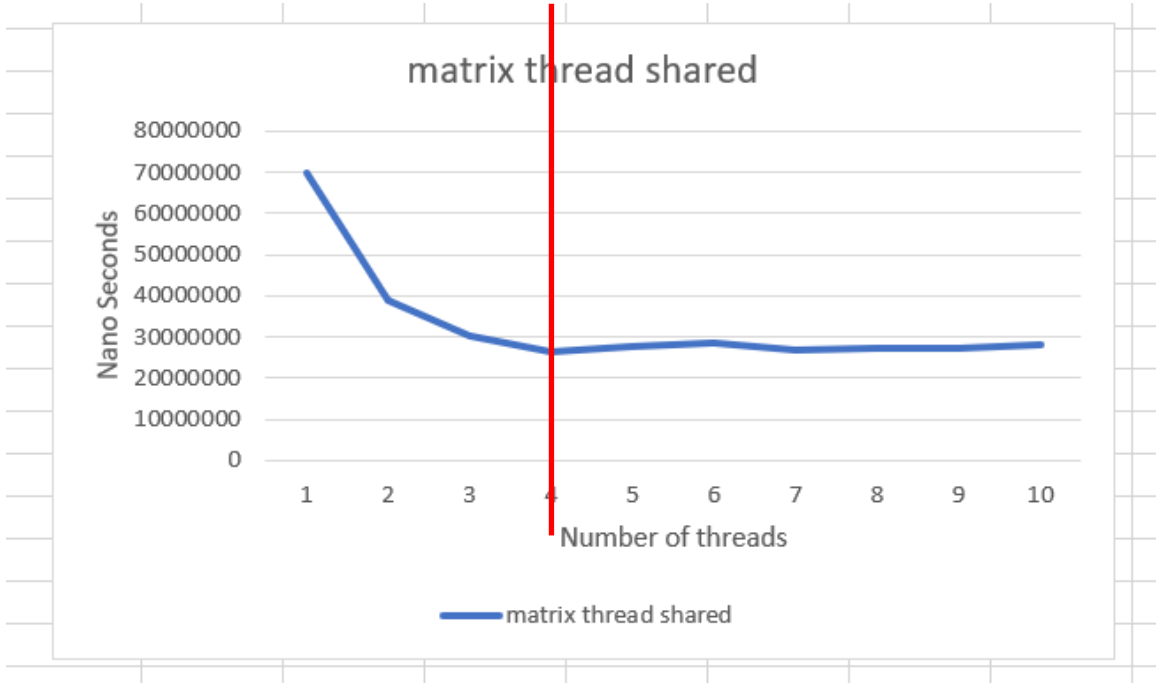
Parallelism

- You can gain performance by running things in parallel
  - Each thread can use another core

- I have a 3800 x 3800 integer matrix, and I want to count the number of odd integers in the matrix

Parallelism

- I have a 3800 x 3800 integer matrix, and I want to count the number of odd integers in the matrix
- I can speed this up by giving each thread a part of the matrix to check!
  - **Works with threads since they share memory**



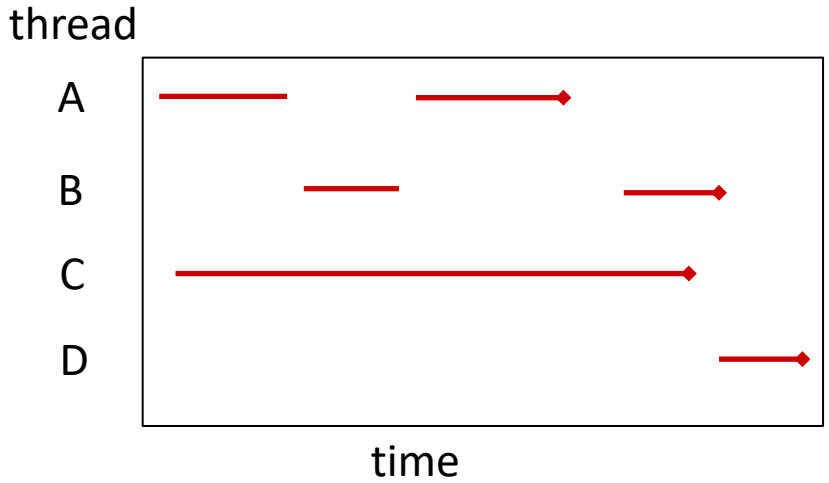
Diminishing returns

After 4 threads, no gain in speed

why? Machine run on only has 4 cores

Parallelism vs Concurrency

- Two commonly used terms (often mistakenly used interchangeably).
- Concurrency: When there are one or more "tasks" that have overlapping lifetimes (between starting, running and terminating).
  - That these tasks are both running within the same **period**.
- Parallelism: when one or more "tasks" run at the same **instant** in time.
- Consider the lifetime of these threads. Which are concurrent with A? Which are parallel with A?



How fast is fork()?

- ~ 0.5 milliseconds per fork*
- ~ 0.05 milliseconds per thread creation*
  - 10x faster than fork()

- *Past measurements are not indicative of future performance – depends on hardware, OS, software versions, …
  - Processes are known to be even slower on Windows

Context Switching

- Processes are considered “more expensive” than threads. There is more overhead to enforce isolation

- Advantages:
  - No shared memory between processes
  - Processes are isolated. If one crashes, other processes keep going

- Disadvantages:
  - More overhead than threads during creation and context switching
  - Cannot easily share memory between processes – typically communicate through the file system

Lecture Outline

- Scheduler
  - Round robin variants (Cont.)
  - Linux Scheduler
- **Threads Intro**
  - Thread High Level
  - pthreads
  - Processes vs threads
  - **Thread Interleaving & Sequential Consistency**

Poll Everywhere

pollev.com/tqm

- What are all possible outputs of this program?

```c
void* thrd_fn(void* arg) {
  int* ptr = (int*) arg;
  printf("%d\n", *ptr);
  return NULL;
}

int main() {
  pthread_t thd1;
  pthread_t thd2;
  int x = 1;
  pthread_create(&thd1, NULL, thrd_fn, &x);
  x = 2;
  pthread_create(&thd2, NULL, thrd_fn, &x);

  pthread_join(thd1, NULL);
  pthread_join(thd2, NULL);
}
```

For simplicity: assume that there is one CPU and that "printf" is an atomic/indivisible operation (we can't context switch mid printf)

Are these output possible?

1
2

2
2

1
1

2
1

Visualization

For simplicity: assume that there is one CPU and that “printf” is an atomic/indivisible operation (we can’t context switch mid printf)

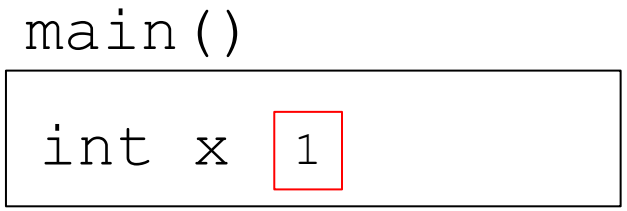
```c
int main() {
  int x = 1;
  pthread_create(...);
  x = 2;
  pthread_create(...);

  pthread_join(...);
  pthread_join(...);
}
```

```c
thrd_fn() {
  printf(*ptr);
  return NULL;
}
```

```c
thrd_fn() {
  printf(*ptr);
  return NULL;
}
```

Visualization: Memory

- The variable x is shared across all threads.




```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

  pthread_join(thd1);
  pthread_join(thd2);
}
```

Visualization: Memory

- The variable x is shared across all threads.


```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

  pthread_join(thd1);
  pthread_join(thd2);
}
```

Visualization: Memory

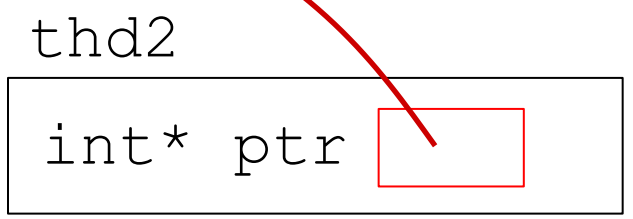
- The variable x is shared across all threads.


```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

  pthread_join(thd1);
  pthread_join(thd2);
}
```

Visualization: Memory

- The variable x is shared across all threads.


```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

  pthread_join(thd1);
  pthread_join(thd2);
}
```



Sequential Consistency

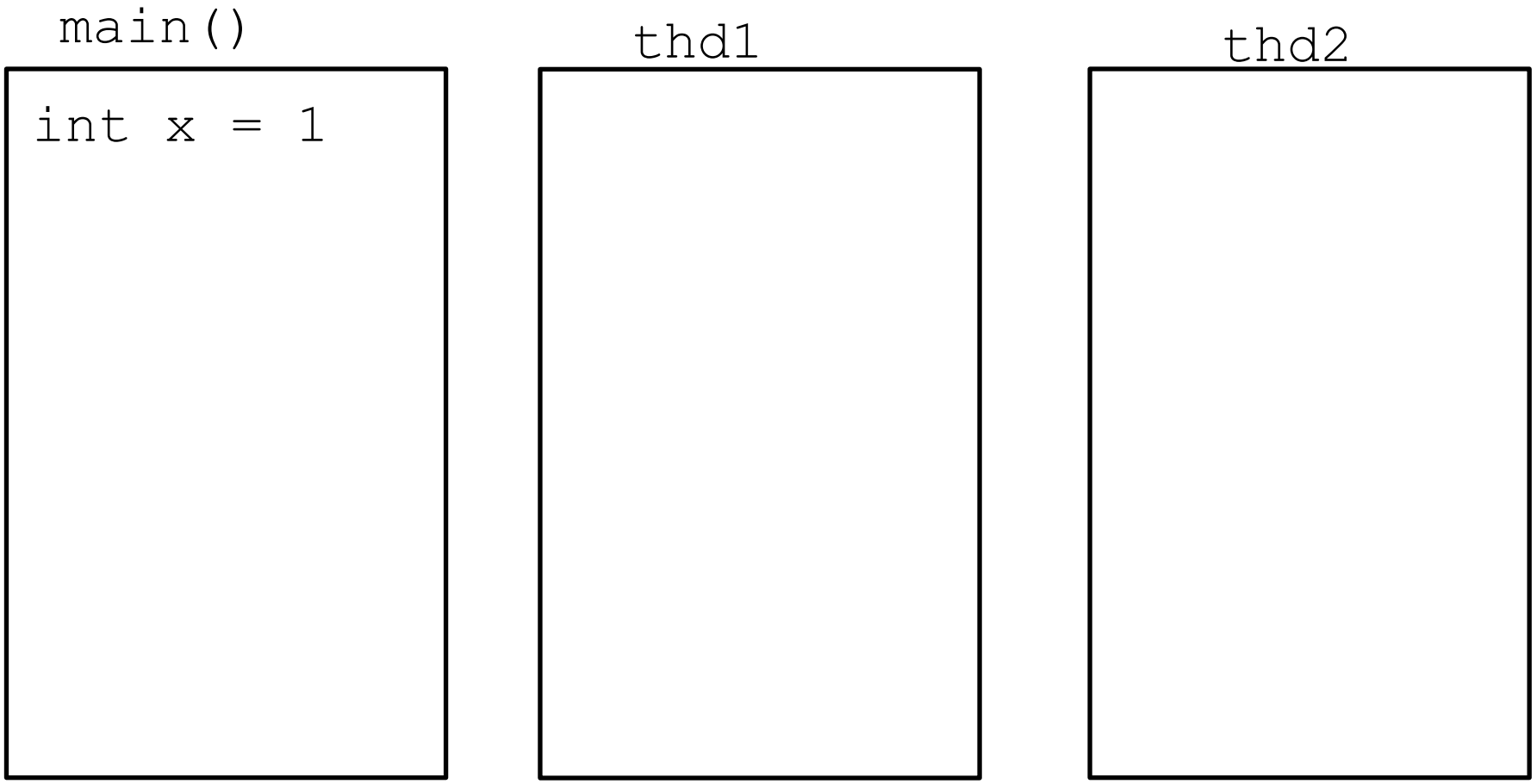
- Within a single thread, we assume* that there is sequential consistency. That the order of operations within a single thread are the same as the program order.

```
main()
int x = 1
create thd1
x = 2
create thd2
```

Within main(), x is set to 1 before thread 1 is created
then thread 1 is created
then x is set to 2
then thread 2 is created

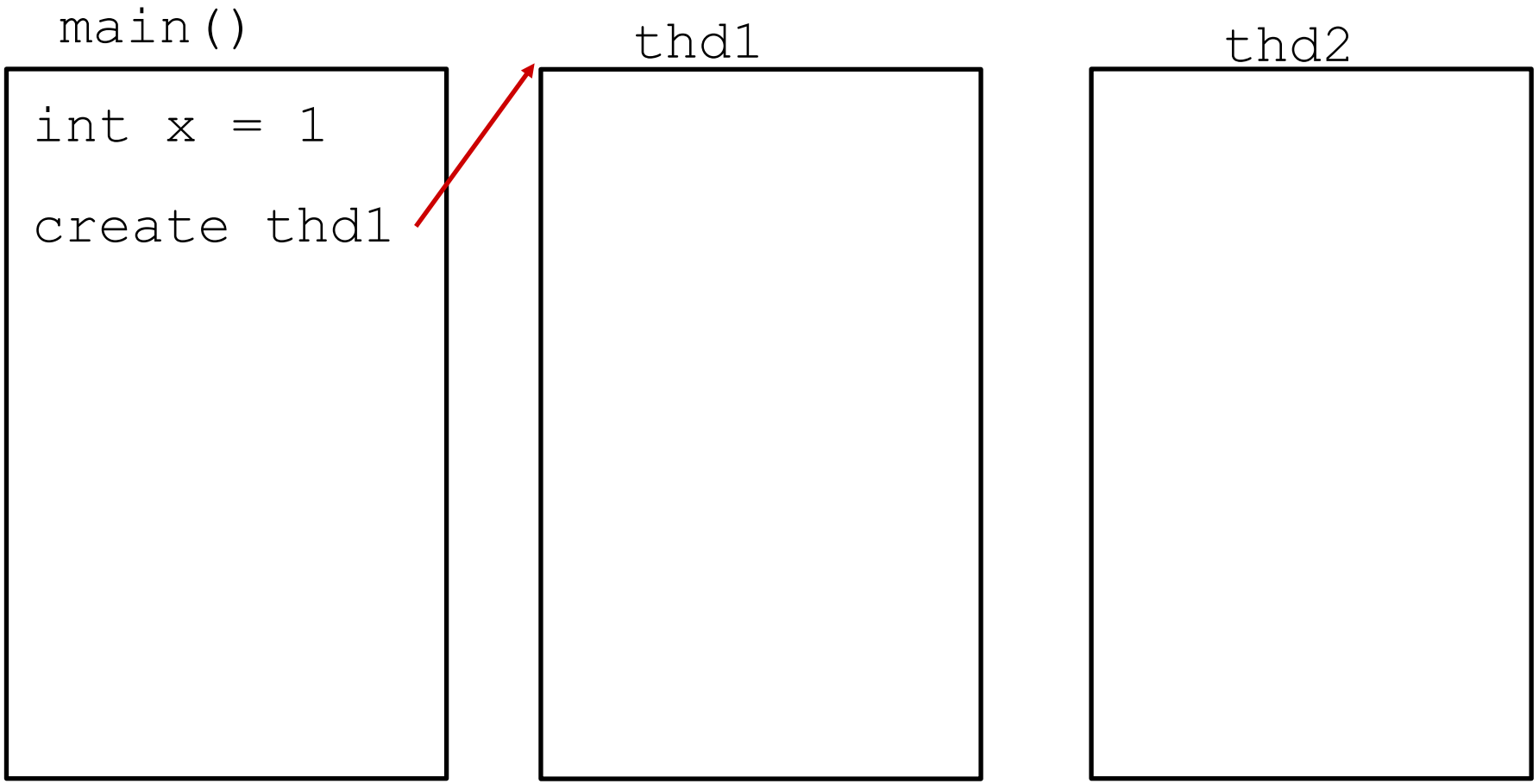
Visualization: Ordering

- Threads run concurrently; we can't be sure of the ordering of things across threads.



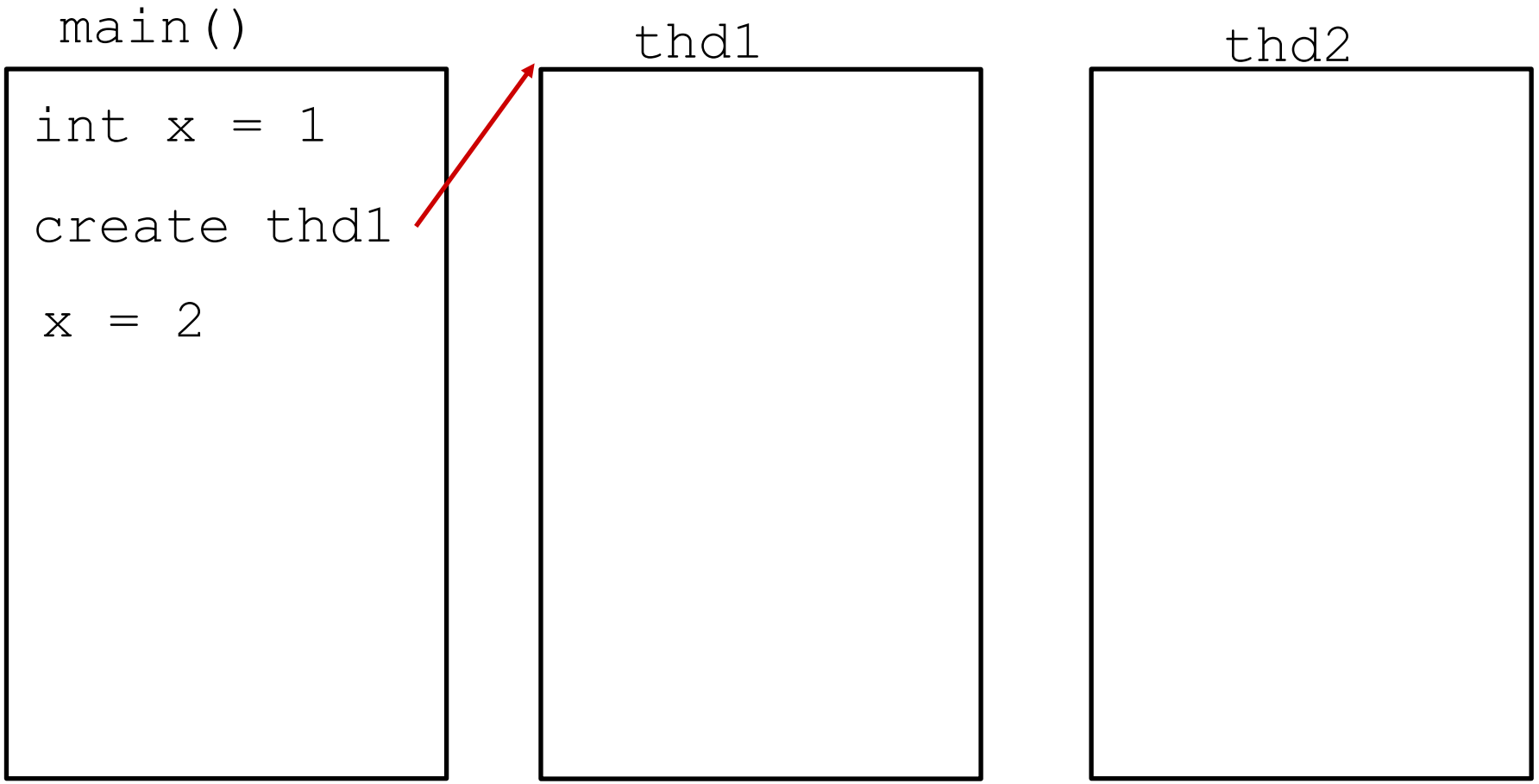
Visualization: Ordering

- Threads run concurrently; we can't be sure of the ordering of things across threads.



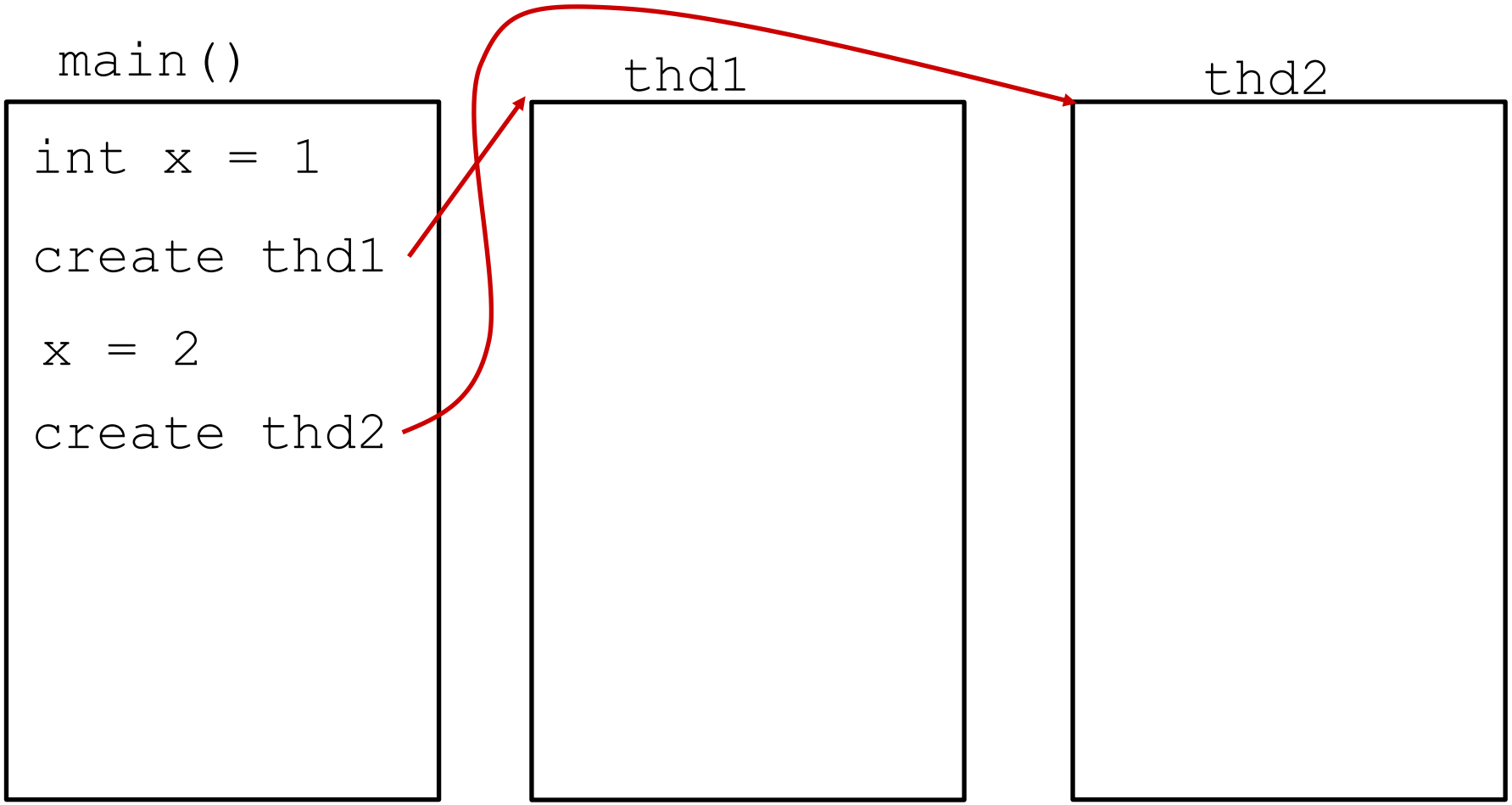
Visualization: Ordering

- Threads run concurrently; we can't be sure of the ordering of things across threads.



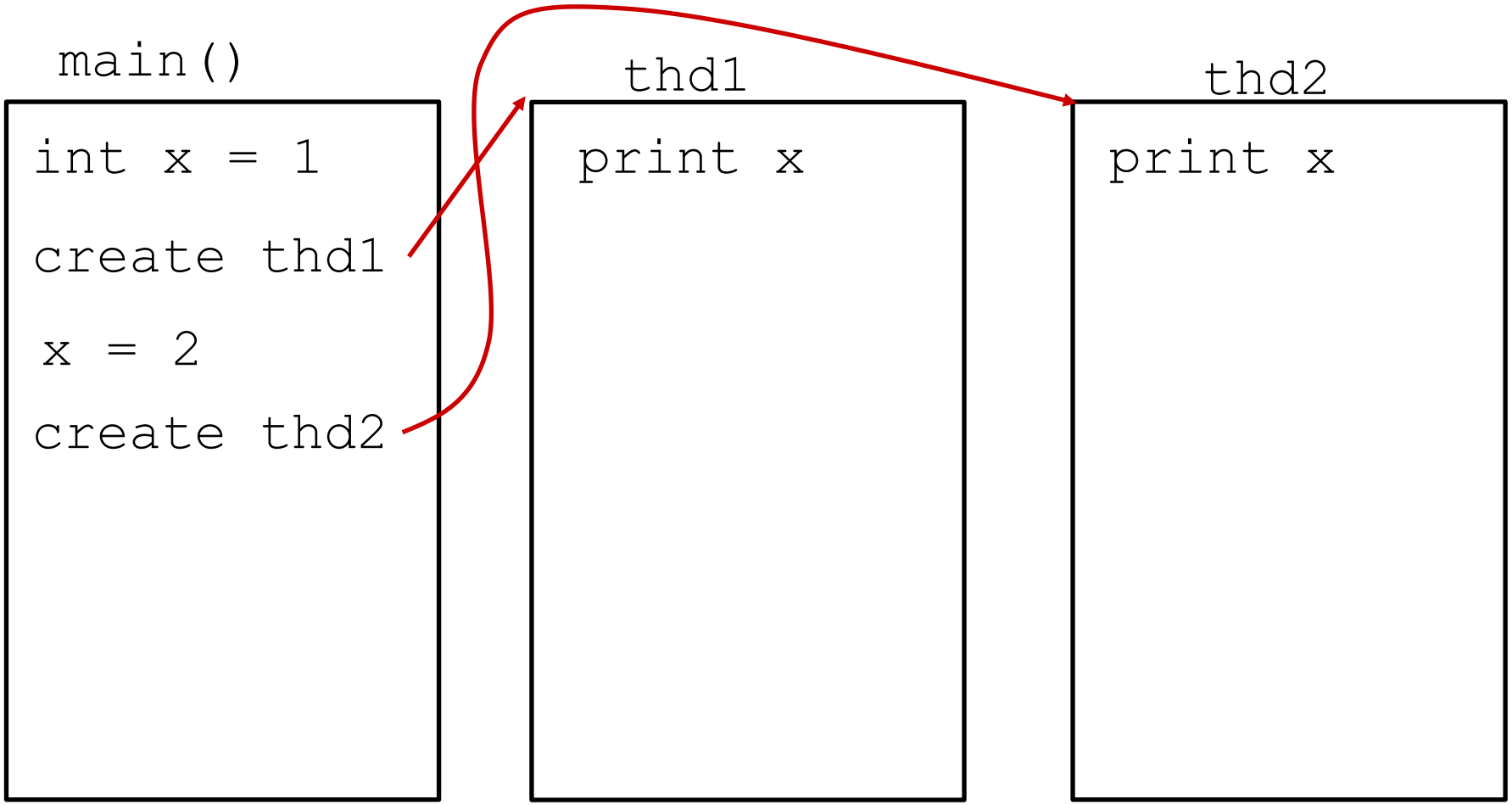
Visualization: Ordering

- Threads run concurrently; we can't be sure of the ordering of things across threads.



Visualization: Ordering

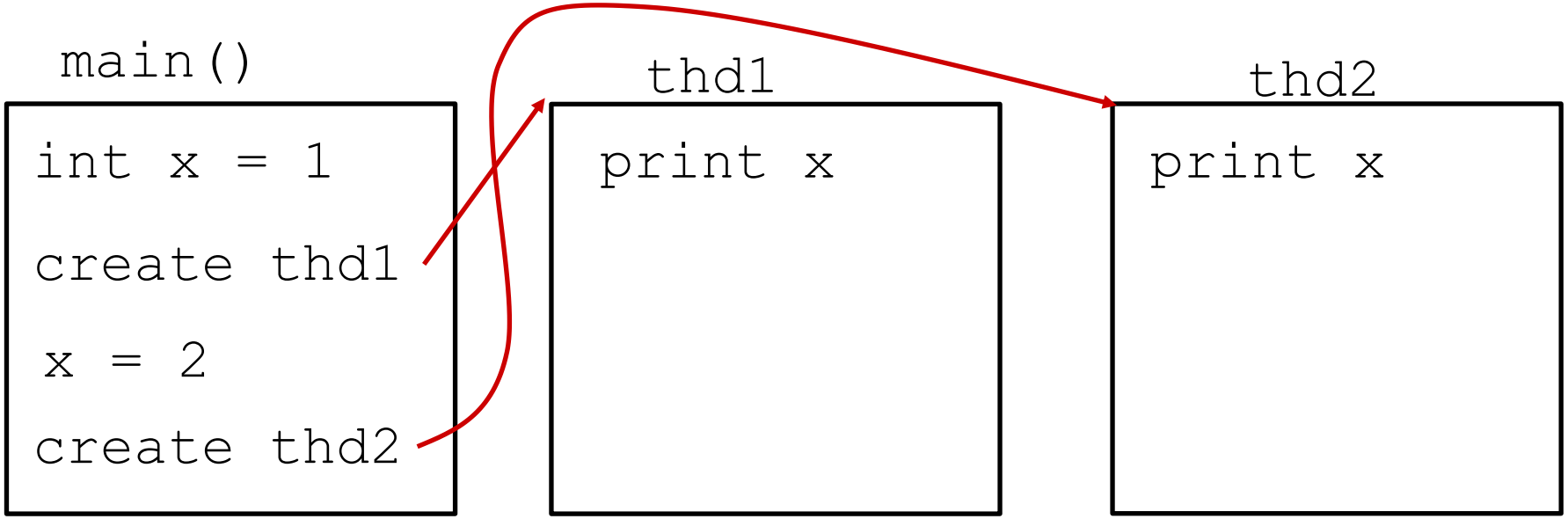
- Threads run concurrently; we can't be sure of the ordering of things across threads.



Visualization: Ordering

This is also why total.c malloc'd individual integers for each thread.
Though it could have also just made an array on the stack

- Threads run concurrently; we can't be sure of the ordering of things across threads.



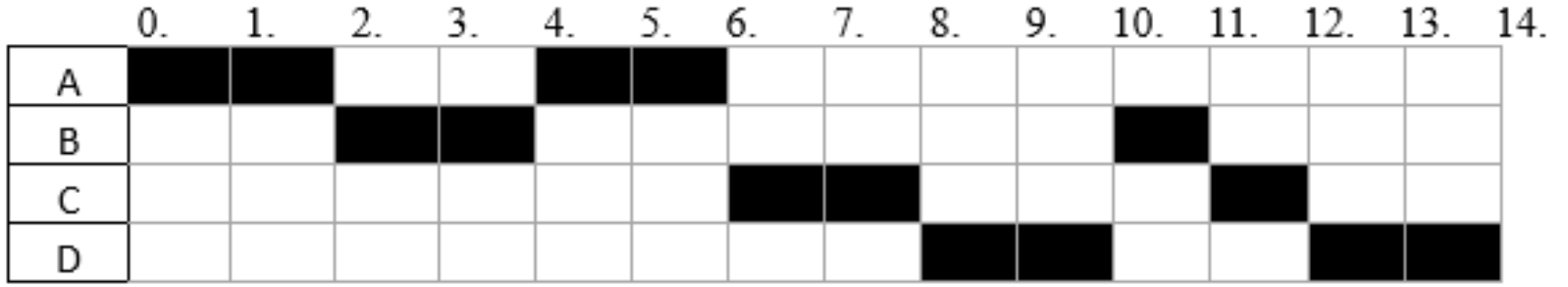
We know that x is initialized to 1 before thd1 is created
We know that x is set to 2 and thd1 is created before thd2 is created

Anything else that we know? **No**. Beyond those statements, we do not know the ordering of main and the threads running.

That's all!

- See you next time!

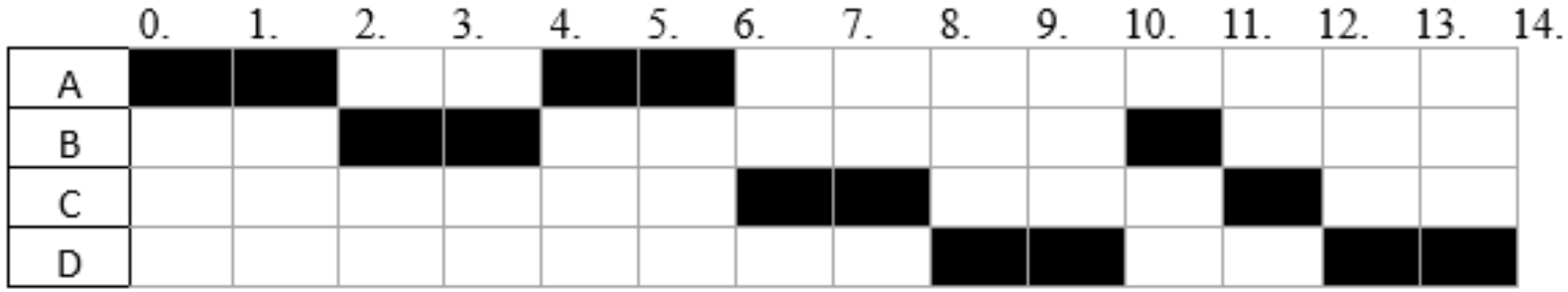
More Round Robin Practice



- 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

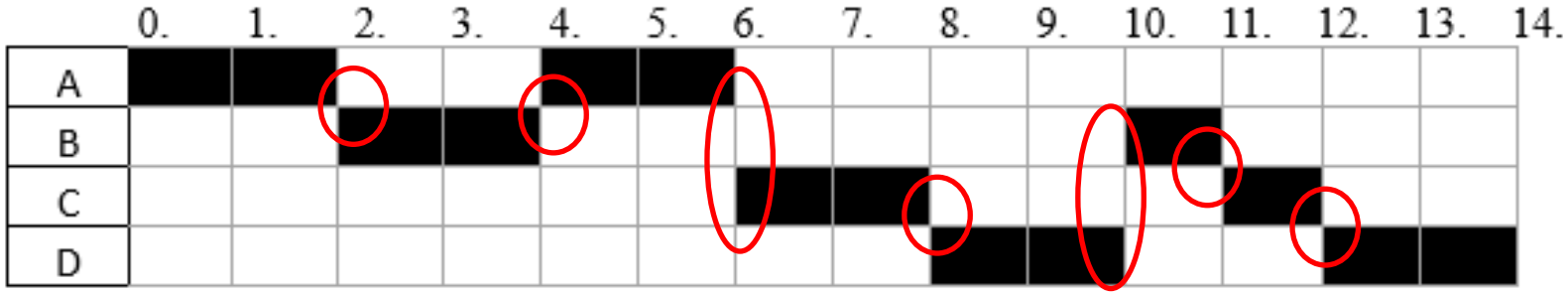
More Round Robin Practice



- 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.

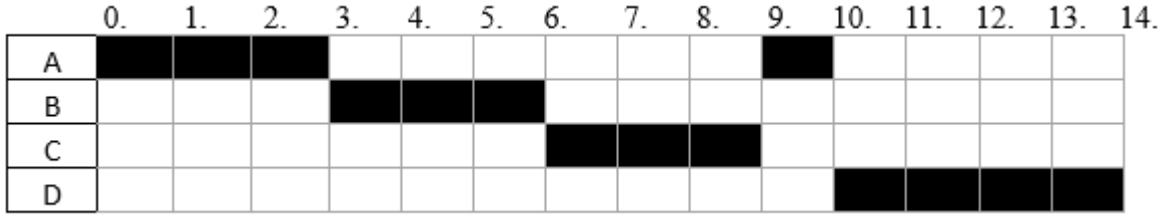
- Which processes are in the ready queue at time 9?
  - D is running, so it is not in the queue
  - A has finished
  - B and C still have to finish, so they are in the queue.

More Round Robin Practice

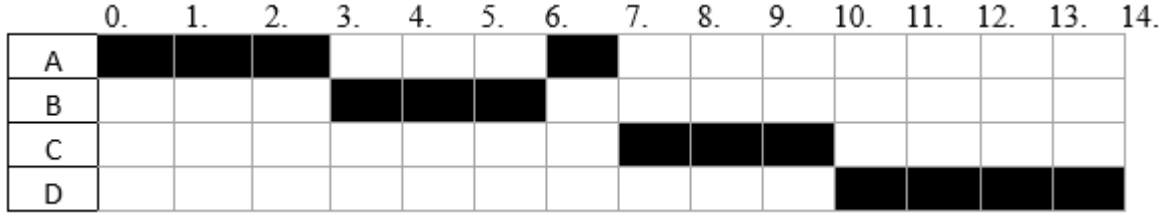


- If this algorithm used a quantum of 3 instead of 2, how many fewer context switches would there be?
  - Currently there are 7 context switches
  - If quantum was 3:

Depends on if C shows up at time 3 or 4



  - Or:



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