CSE 30341 – Lecture 6

Scheduling – Proportional, Multiprocessor, Concurrency

February 2^{nd}, 2023

Overview

- Chapter 9 – Proportional Scheduling
  - Probabilistic vs. Deterministic
- Chapter 10 - Multiprocessor
  - Challenges
- Chapter 26 – Introduction to Concurrency
  - Intro to pThreads / Vocabulary

Project 1

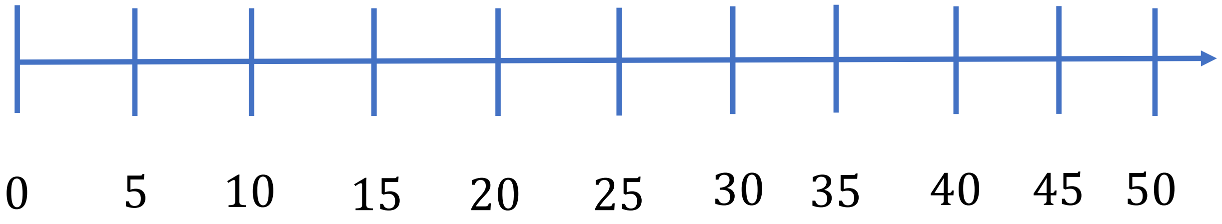
- **Help / discussion on-line**
  - Video / narration is posted
- **Recommendation**
  - Assume good arguments first
  - Increase complexity after core
- **If you have not started**
  - Have Level 1 functionality (assume good arguments) working by Thursday night
  - Have Level 2 functionality working by Friday at 5 PM
- **Split – Partner**
  - One partner – write the bit flip / reverse code
  - One partner – write the argument handling code

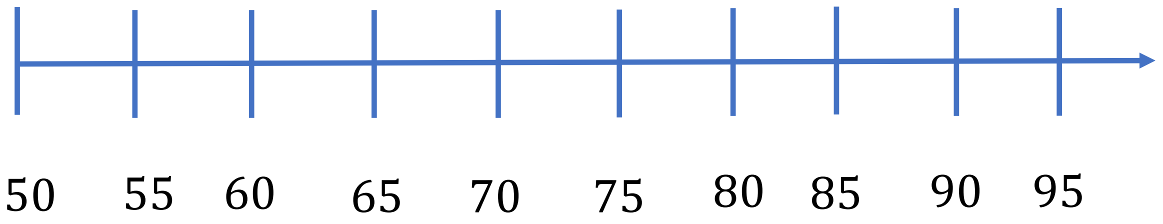
Key Points – Lecture 6 – Scheduling / Concurrency

- **Scheduling** – Chapter 8
  - Practice - MLFQ
- **Scheduling** – Chapters 9, 10
  - Describe **lottery** scheduling, **stride** scheduling
  - Compare / contrast: **deterministic**, **probabilistic**
  - Describe the challenges of **multiprocessor scheduling**
- **Concurrency** – Chapter 26
  - Compare / contrast: **thread**, **TCB**, **PCB**.
  - What is a **race condition**?
  - What is a **critical section**?
  - What is **atomicity**?
  - What is **mutual exclusion**?

Scheduling

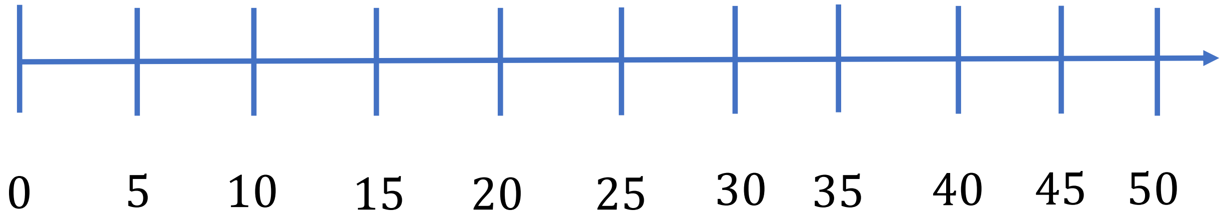
- Consider the following scenario
  - Four jobs – P_0 .. P_3
    - P_0 -> { 0, 30 }
    - P_1 -> { 10, 20 }
    - P_2 -> { 5, 30 }
    - P_3 -> { 30, 10 }
  - What does the schedule look like?
    - STCF
  - Compute
    - Turnaround Time
    - Response Time





STCF

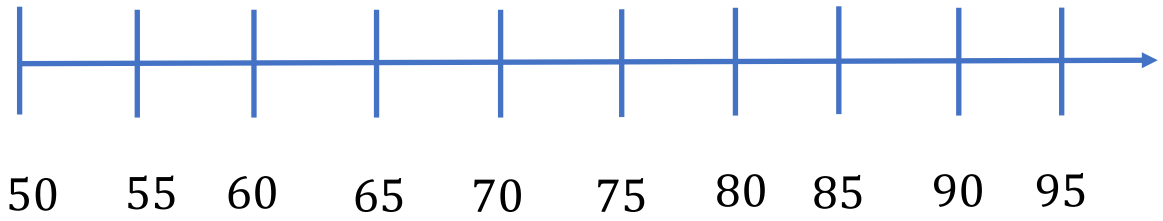
P_0 -> { 0, 30 }
P_1 -> { 10, 20 }
P_2 -> { 5, 30 }
P_3 -> { 30, 10 }



Could you do this on an exam?

FIFO, SJF, STCF, RR?

Compute TT, RT?

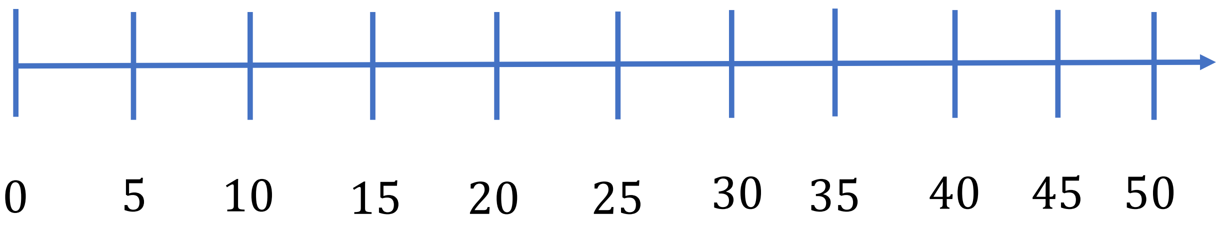


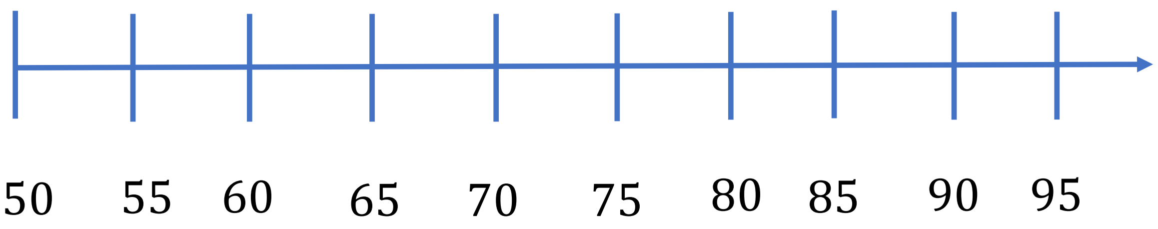
MLFQ

Slice = ??
Priority Levels = 3

P_0 -> { 0, 30 }
P_1 -> { 10, 20 }
P_2 -> { 5, 30 }
P_3 -> { 30, 10 }

Boost (S) of 50



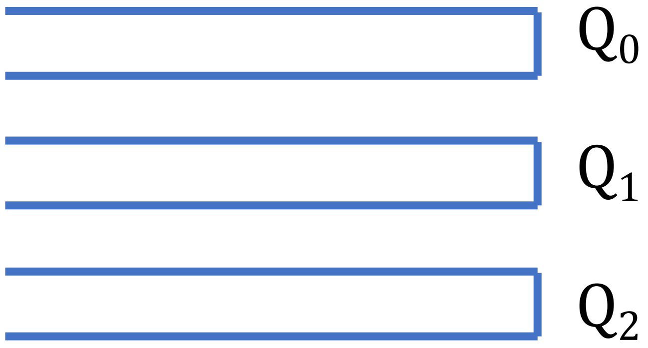


Let's Do This

Slice = 5
Priority Levels = 3

P_0 -> { 0, 30 }
P_1 -> { 10, 20 }
P_2 -> { 5, 30 }
P_3 -> { 30, 10 }

Boost (S) of 50



Working the example
→ separate recording

<u>Measure</u>

Pre-emptions
TT

Chapter 9 – Proportional Scheduling

- **Fairness**
  - MLFQ?
- **Two approaches**
  - **Lottery** scheduling
    - Probabilistic
  - **Stride** scheduling
    - Deterministic

Lottery Scheduling

- Tickets

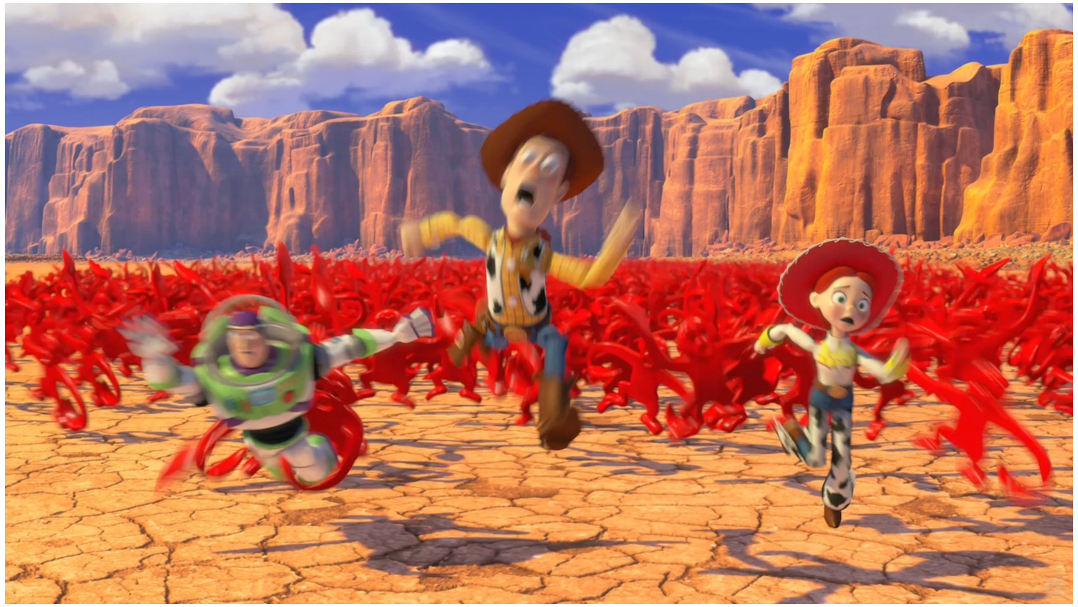
Over what time window is it fair?

- Probabilistic / Random
- Example
  - A = 75 tickets
  - B = 25 tickets

75% chance to get picked

25% chance to get picked

100 total tickets



Infinite window = Perfectly fair
Short period = TBD

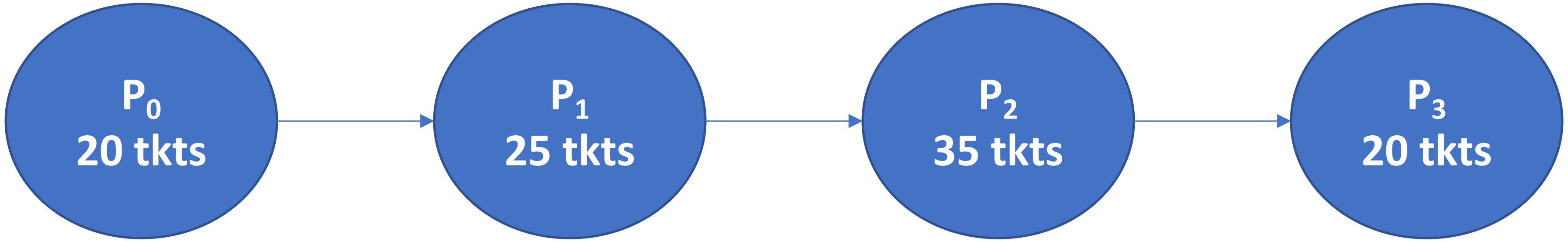
Code It

```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
//         get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// 'current' is the winner: schedule it...
```

Stop at the end of the linked list

Figure 9.1: **Lottery Scheduling Decision Code**

Example



<u>**Current**</u>			
0	**0+20 = 20**	**20+25 = 45**	**45+35 = 80**
	20 < 53	**45 < 53**	**Schedule**

```
// loop until the sum of ticket values is > the winner
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// 'current' is the winner: schedule it...
```

Figure 9.1: **Lottery Scheduling Decision Code**

TotalTickets = 20 + 25 + 35 + 20
→ 100

Roll the “dice” for a number from 0 to 100

Example: Winner → 53

Stride Scheduling

- **Deterministic** vs. **Probabilistic**
- **Stride** vs. Tickets
  - **Large # / Tickets = Stride**
  - Pass -> Counter, Start at zero
  - Schedule / increment Pass
  - Use pass to prioritize

```
current = remove_min(queue);       // pick client with minimum pass
schedule(current);                 // use resource for quantum
current->pass += current->stride;  // compute next pass using stride
insert(queue, current);            // put back into the queue
```

Stride Scheduling – Example

P0 → 100 tickets
P1 → 50 tickets

Use 1,000 for our "big number"

1000 / 100 → 10
P0 Stride

1000 / 50 → 20
P1 Stride

P0.Pass = 0, P1.Pass = 0

Pick Min of [0, 0] → P0
P0.Pass = P0.Pass + P0.Stride
0 + 10 = 10

Pick Min of [10, 0] → P1
P1.Pass = P1.Pass + P1.Stride
0 + 20 = 20

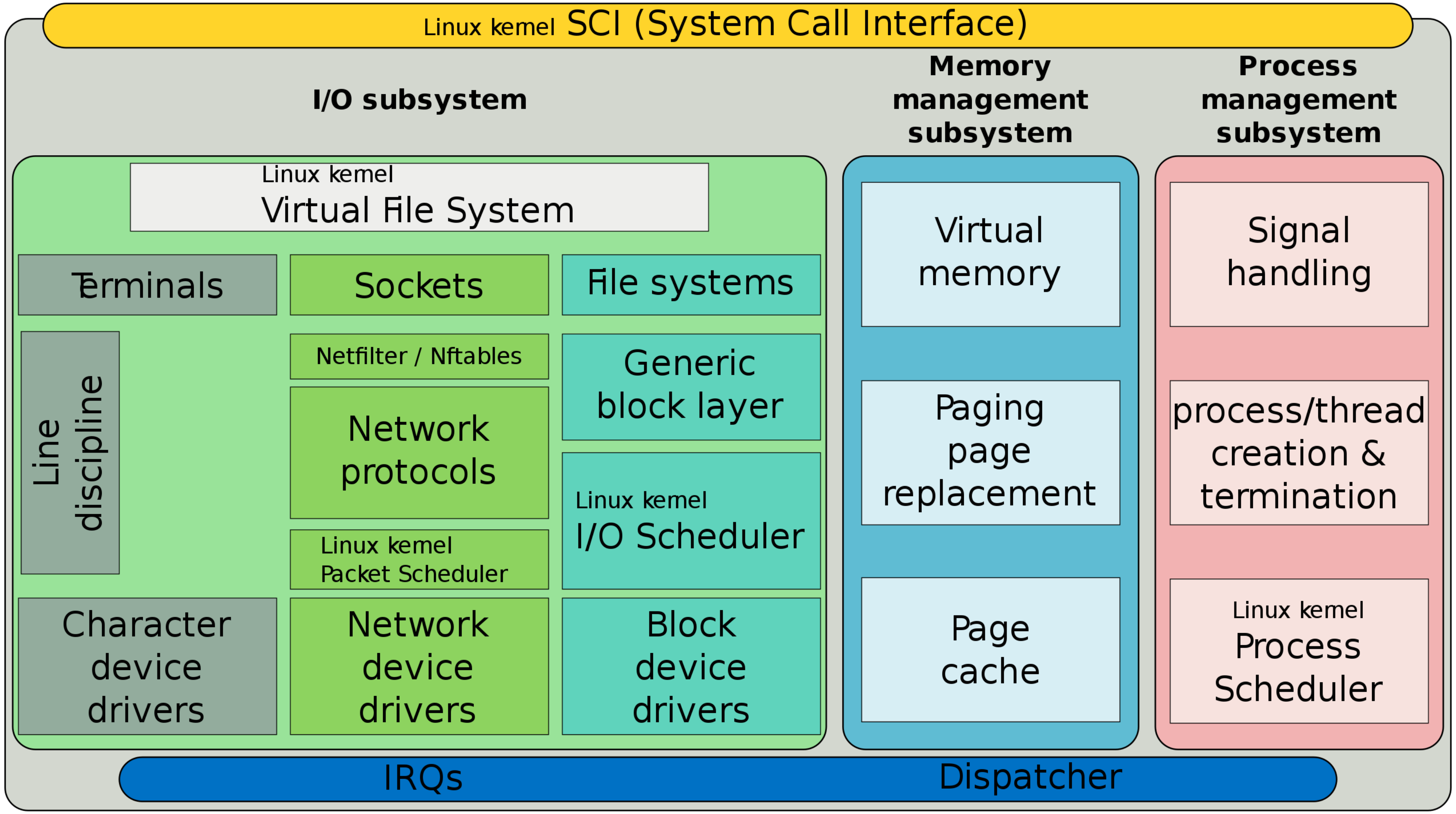
Pick Min of [10, 20] → P0
P0.Pass = P0.Pass + P0.Stride
10 + 10 = 20

Pick Min of [20, 20] → P0
P0.Pass = P0.Pass + P0.Stride
20 + 10 = 30

Takeaways

- Random is not always bad
  - Better than you think
- Random draws -> not free
  - Magic to generate random numbers
- Deterministic
  - Worse can be way, way worse
  - Better if we can manage it

Linux – CFS – Completely Fair Scheduler



Linux – CFS – Completely Fair Scheduler

- Introduced in 2007

Each per-CPU run-queue of type `cfs_rq` sorts `sched_entity` structures in a time-ordered fashion into a red-black tree (or 'rbtree' in Linux lingo), where the leftmost node is occupied by the entity that has received the least slice of execution time (which is saved in the `vruntime` field of the entity). The nodes are indexed by processor "*execution time*" in nanoseconds.[3]

A "*maximum execution time*" is also calculated for each process to represent the time the process would have expected to run on an "ideal processor". This is the time the process has been waiting to run, divided by the total number of processes.

When the scheduler is invoked to run a new process:

1. The leftmost node of the scheduling tree is chosen (as it will have the lowest spent *execution time*), and sent for execution.
2. If the process simply completes execution, it is removed from the system and scheduling tree.
3. If the process reaches its *maximum execution time* or is otherwise stopped (voluntarily or via interrupt) it is reinserted into the scheduling tree based on its newly spent *execution time*.
4. The new leftmost node will then be selected from the tree, repeating the iteration.

Linux - CFS

Con Kolivas's work with scheduling, most significantly his implementation of "fair scheduling" named Rotating Staircase Deadline, inspired Ingo Molnár to develop his CFS, as a replacement for the earlier O(1) scheduler, crediting Kolivas in his announcement.[4] CFS is an implementation of a well-studied, classic scheduling algorithm called weighted fair queuing.[5] Originally invented for packet networks, fair queuing had been previously applied to CPU scheduling under the name stride scheduling. CFS is the first implementation of a fair queuing process scheduler widely used in a general-purpose operating system.[5]

Stride = Deterministic selection

If your process does not run for its full "slice"
the stride that gets added is proportional to its actual run duration

When your interactive process wakes up, it gets a higher priority.

Chapter 10 – Multi-processor

- Multiprocessor / multicore challenges
  - Briefly cover now – revisit later
  - Caching
  - Affinity
  - Locking
  - Consistency

Caching

- Cache on each processor
  - Populated based on running code
    - Temporal locality
    - Spatial locality
  - What happens if we ignore this relationship?
- Issues
  - Coherence
  - Affinity

Affinity

- SQMS
  - One queue – multiple processors
  - Affinity
    - Run on same processor
- MQMS
  - Multiple queues – multiple processors
  - Issue
    - Load balancing

Key Points – Lecture 6 – Scheduling

- Scheduling – Chapters 9, 10
  - Describe lottery scheduling, stride scheduling
  - Compare / contrast: deterministic, probabilistic
  - Describe the challenges of multiprocessor scheduling

Chapter 26 – Intro to Concurrency

- **Sharing**
  - **CPU** – Discuss now
  - Memory – Discuss later
- **Concept**
  - Thread
  - Multi-Threaded
  - Context Switch
  - TCB
  - PCB
  - Thread-Local Storage

Threads vs. Processes

- **Process**
  - Major unit of organization for a running program
  - OS schedules a process
  - Parallelism
    - Run multiple instances of the program
    - Fork / run multiple children
      - Each child has its own ”independent” memory space

Example: Apache web server

- **Threads**
  - Exist within a process
    - Allows for intra-process parallelism
    - Memory is shared amongst threads in a process

Example: One thread to handle user input (scanf), another to wait for network data

Pthread example code – What happens?

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

void * is back!

Figure 26.2: **Simple Thread Creation Code (t0.c)**

Take it up a notch

```
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

A nice, friendly global variable

static = local to this file only

volatile = Please do not optimize me compiler

Is volatile helpful for multi-threaded code?

Take it up a notch

```c
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

Each thread should add 10M to the global variable counter

Take it up a notch – Part 2

```
//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Figure 26.6: **Sharing Data: Uh Oh (t1.c)**

Take it up a notch - What happens?

```c
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

Why?

- **Shared data**
  - Global or reference
- **Scheduler variability**
  - Who knows what runs when?
- **Atomic operations**
  - Test and set

```
% objdump -d main

theTally = theTally + 5;

/* alternatively */
theTally += 5;
```

```
if (isLocked == 0)
{
   isLocked = 1;
   /* Do atomic operation */
   isLocked = 0;
}
```

Assembly Decomposition

```
theTally = theTally + 5;

/* alternatively */
theTally += 5;


if (isLocked == 0)
{
   isLocked = 1;
   /* Do atomic operation */
   isLocked = 0;
}
```

Vocabulary

- Concurrency
  - Critical section
  - Race Condition
  - Indeterminate program
  - Deterministic
  - Mutual exclusion
  - Deadlock

Key Points – Lecture 6 – Scheduling / Concurrency

- Scheduling – Chapter 8
  - Practice - MLFQ
- Scheduling – Chapters 9, 10
  - Describe lottery scheduling, stride scheduling
  - Compare / contrast: deterministic, probabilistic
  - Describe the challenges of multiprocessor scheduling
- Concurrency – Chapter 26 (if time allows)
  - Compare / contrast: thread, TCB, PCB.
  - What is a race condition?
  - What is a critical section?
  - What is atomicity?
  - What is mutual exclusion?