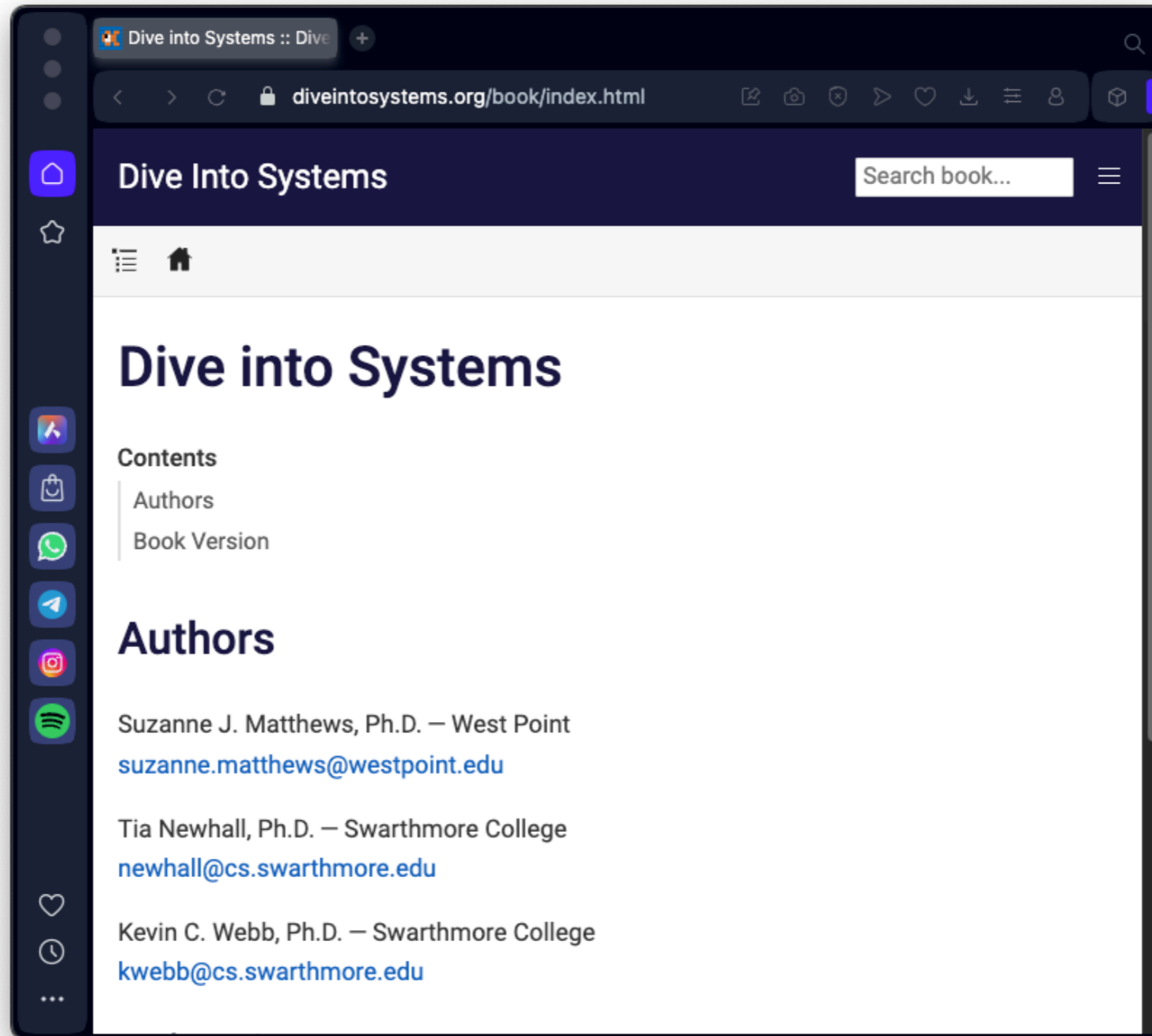


# **14a. Leveraging Shared Memory in the Multicore Era**

**For COMSC 142**

# Free online textbook



- <https://diveintosystems.org/book/index.html>

# Topics

## Ch 14a:

14.1. Programming Multicore Systems

14.2. POSIX Threads

14.3. Synchronizing Threads

## Ch 14b:

14.4. Measuring Parallel Performance

14.5. Cache Coherence

14.6. Thread Safety

14.7. Implicit Threading with OpenMP

# CPU, Processors, and Cores

- **Processor**
  - any circuit that performs computation
  - Central Processing Unit (CPU) is a processor
- **Multicore processor**
  - processor with multiple compute cores
- **Core**
  - a compute unit with an ALU, registers, and a bit of cache

# Performance walls

- **Memory wall**

- improvements in memory technology did not keep pace with improvements in clock speed
- memory is a bottleneck to performance
- speeding up the execution of a CPU no longer improves its overall system performance

- **Power wall**

- increasing the number of transistors on a processor necessarily increases that processor's temperature and power consumption
- increases cost to power and cool the system
- power is now the dominant concern in computer system design

# CPU v. GPU

- **Graphics Processing Unit (GPU)**
  - cores have even fewer transistors than CPU cores
  - are specialized for particular tasks involving vectors
  - A typical GPU can have 5,000 or more GPU cores
- Computing with GPUs is known as **manycore computing**
- In this chapter, we concentrate on **multicore computing**

```
$ lscpu

Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:  0-7
Thread(s) per core:   2
Core(s) per socket:   4
Socket(s):             1
Model name:            Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
CPU MHz:               1607.562
CPU max MHz:           3900.0000
CPU min MHz:           1600.0000
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
...
```

Total # physical cores: sockets x cores per socket = 1 x 4

# Hyperthreading

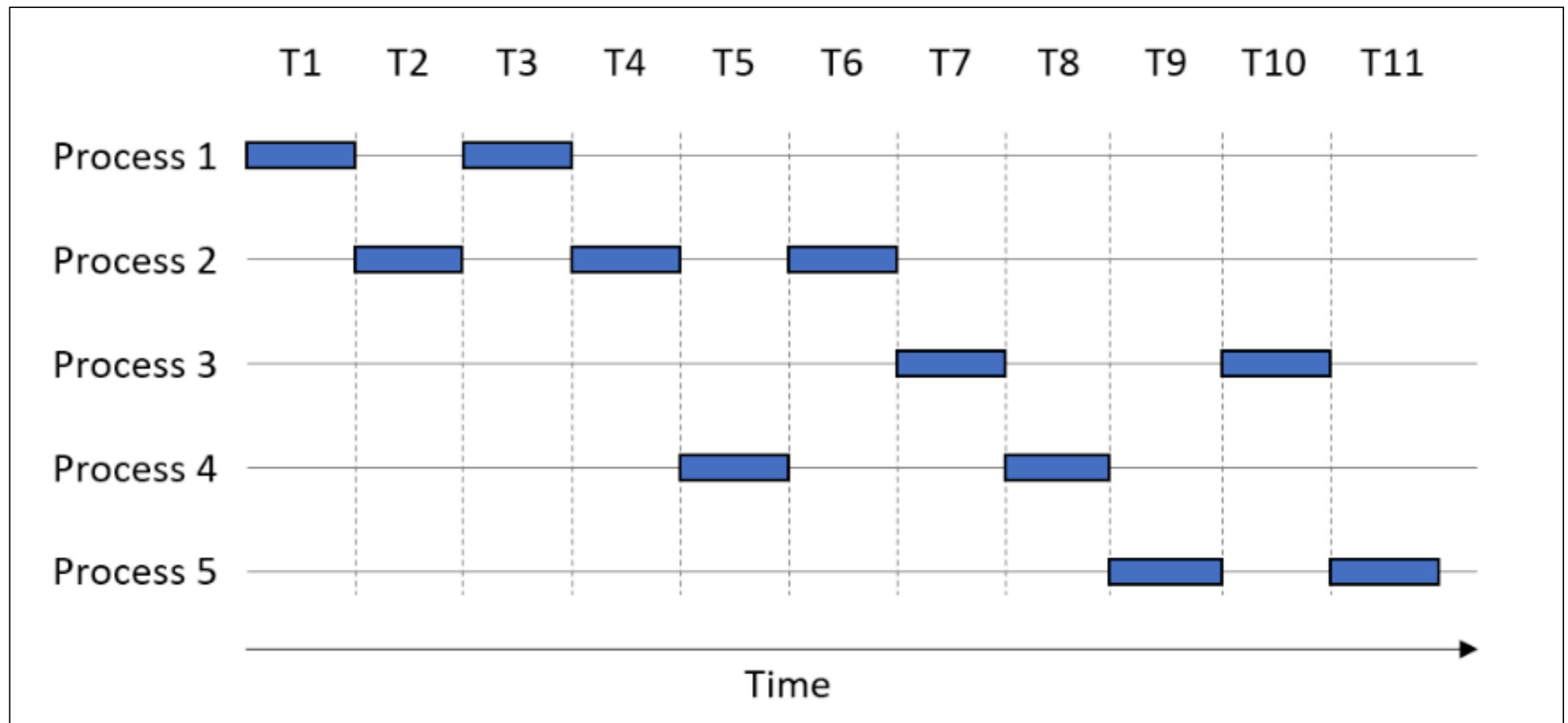
- Efficient processing of multiple threads on a single core
- So a processor that can run 2 threads per core
  - with 4 physical cores
  - has 8 logical cores
- **Performance Cores** and **Efficiency Cores**
  - E-cores consume less power
  - P-cores have higher clock speeds



# 14.1. Programming Multicore Systems

## 14.1.1. The Impact of Multicore Systems on Process Execution

- **Context switch**
  - occurs when the CPU changes which process it currently executes
- Diagram is for a single-core system

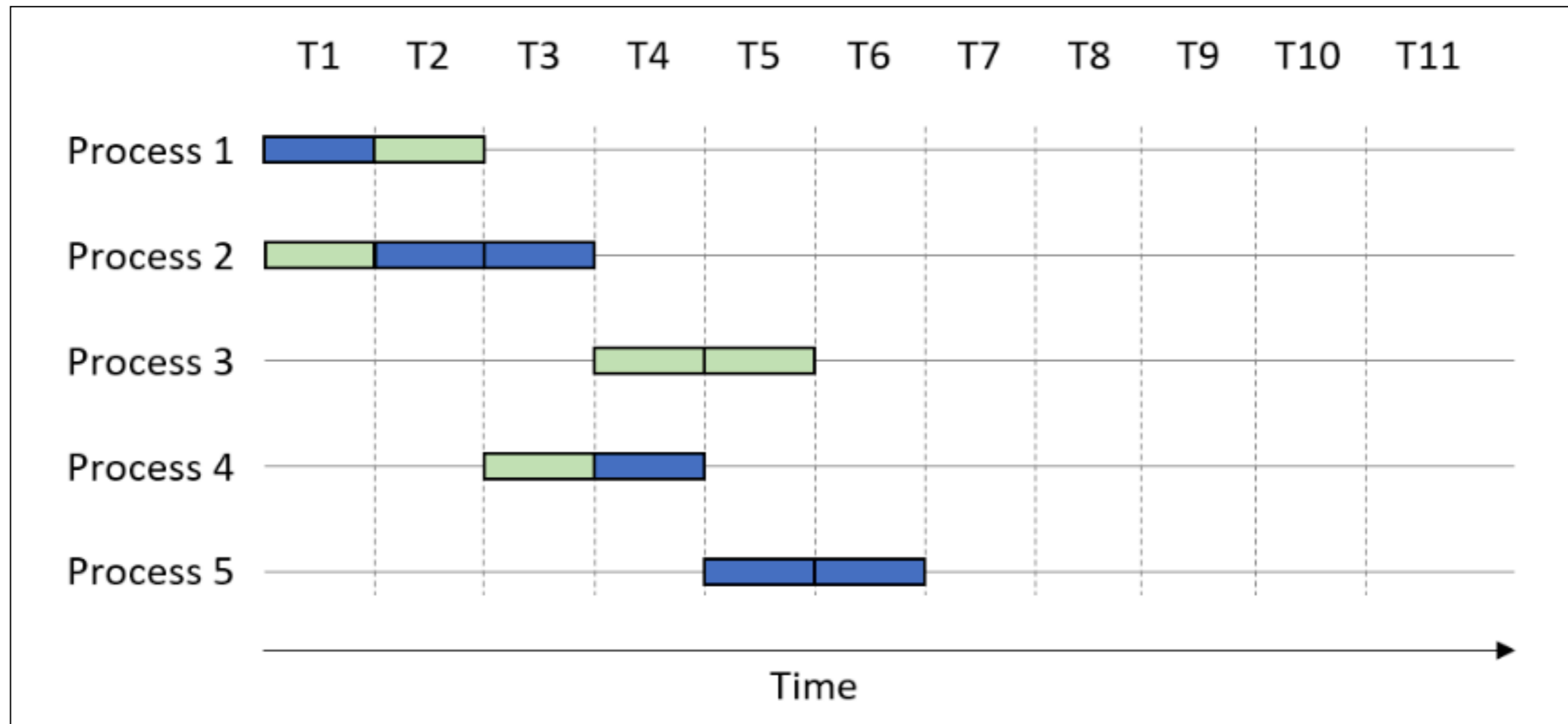


# CPU Time and Wall-Clock Time

- **CPU time**
  - **measures the** amount of time a process takes to execute on a CPU
- **Wall-clock time**
  - the amount of time a human perceives a process takes to complete.
  - often significantly longer than the CPU time, due to context switches

# Two-core system

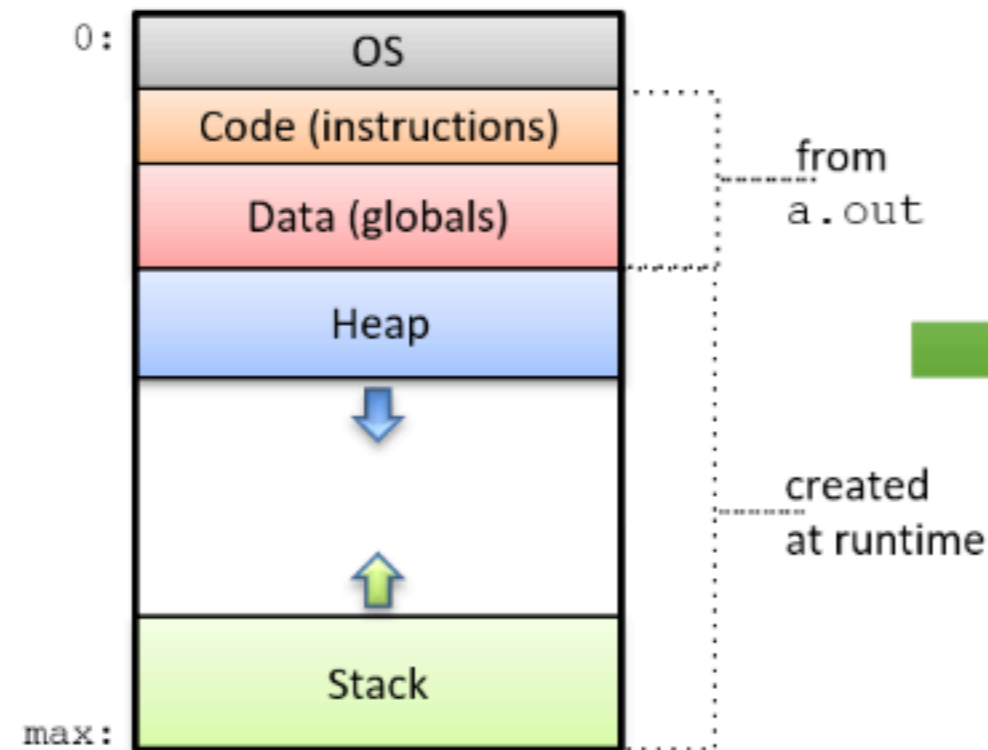
- CPU time is the same as single-core system
- But Wall Clock Time is decreased



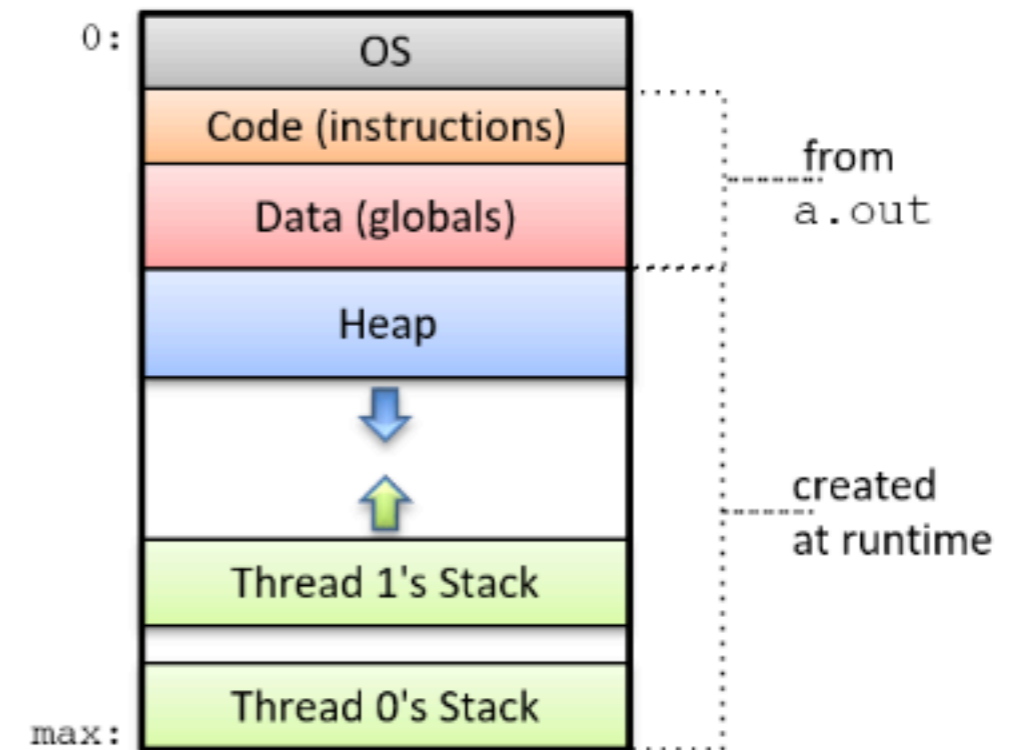
# 14.1.2. Expediting Process Execution with Threads

- **Threads**
  - lightweight, independent execution flows

Process's Virtual Address Space  
(1 Thread)



Process's Virtual Address Space  
(2 Threads)

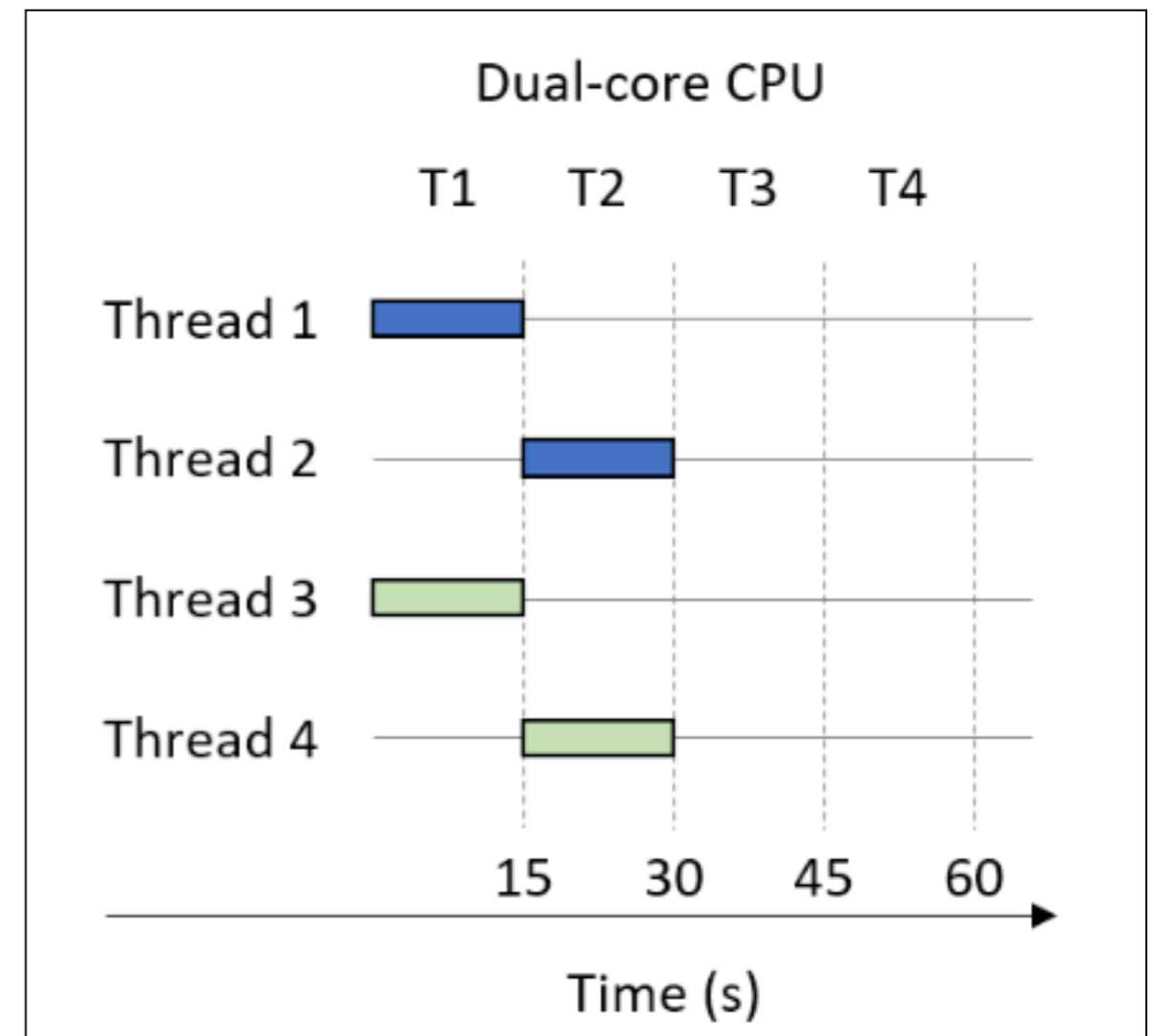
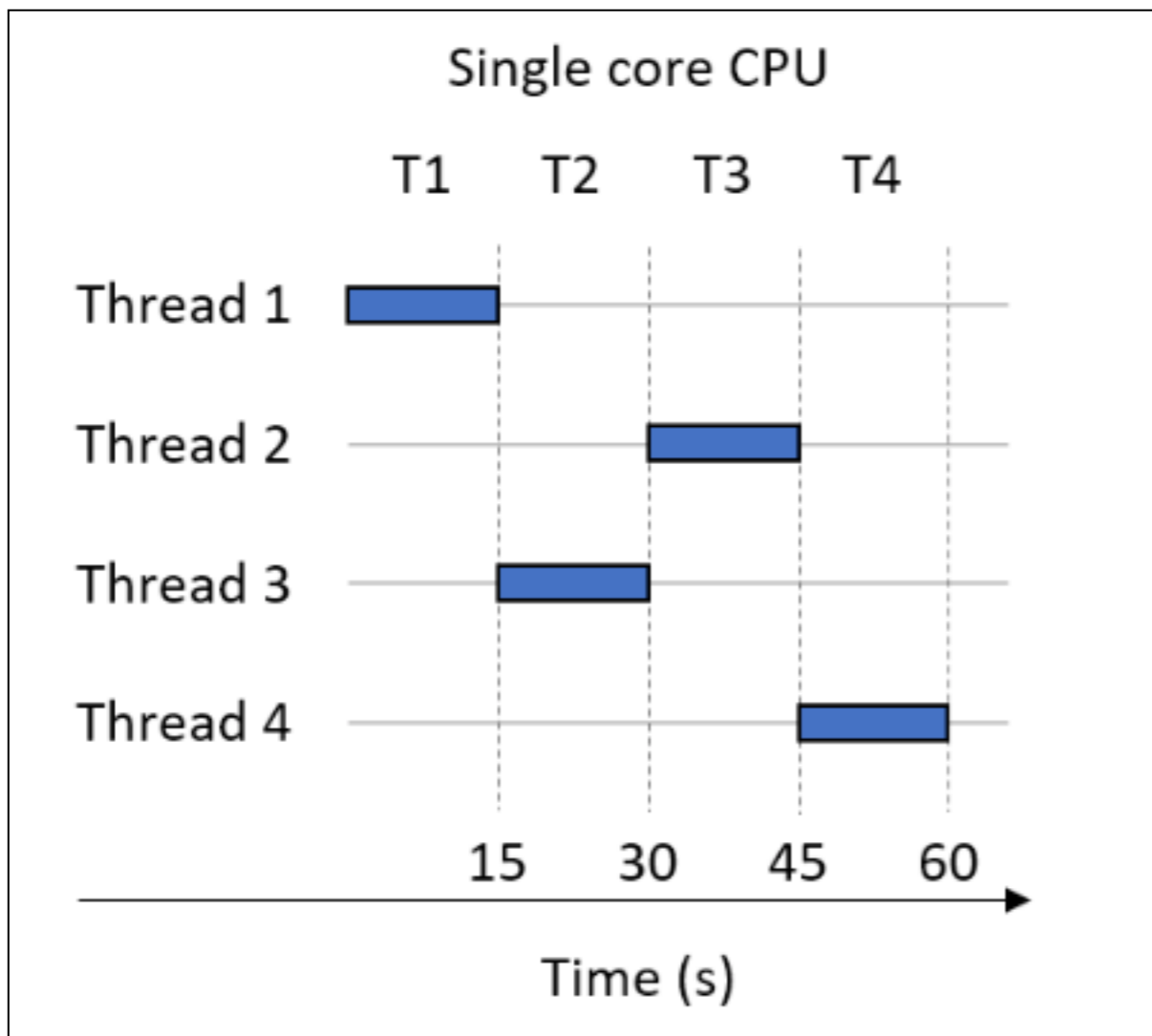


# An Example: Scalar Multiplication

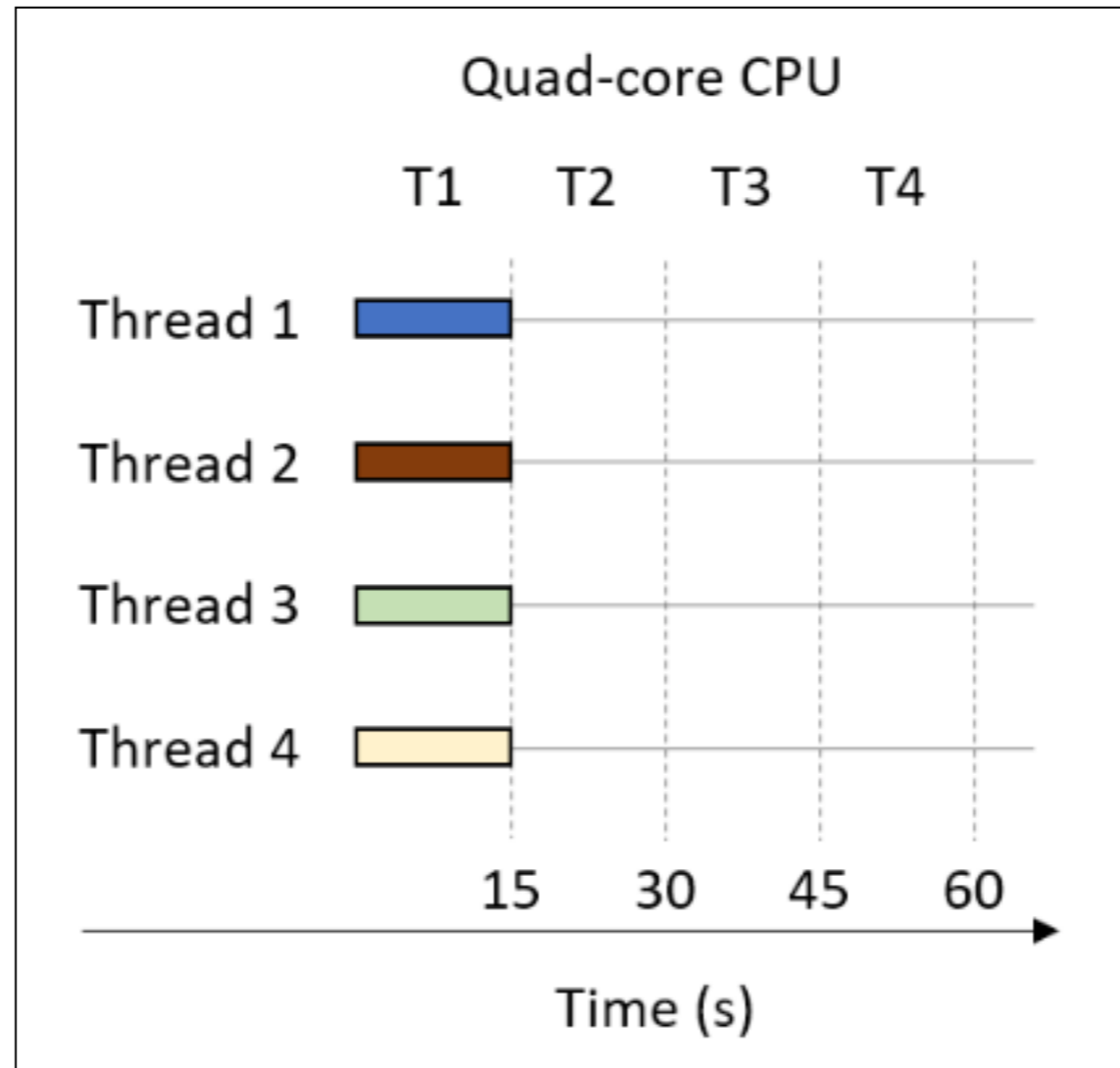
```
void scalar_multiply(int * array, long length, int s) {  
    int i;  
    for (i = 0; i < length; i++) {  
        array[i] = array[i] * s;  
    }  
}
```

- Suppose that array has **N** total elements. To create a multithreaded version of this application with **t** threads, it is necessary to:
  - Create **t** threads.
  - Assign each thread a subset of the input array (i.e., **N/t** elements).
  - Instruct each thread to multiply the elements in its array subset by **s**

# Four Threads



# Four Threads





# 14.2. POSIX Threads

# POSIX

- POSIX is an acronym for Portable Operating System Interface
- An IEEE standard that specifies how UNIX systems look, act, and feel
- Code using POSIX on a Linux machine will certainly work on other Linux machines, and it will likely work on machines running macOS or other UNIX variants

# Hello Threading! Writing Your First Multithreaded Program

- `wget https://samsclass.info/COMSC-142/proj/threads1.c`
- `gcc -o threads1 threads1.c`
- **#include <pthread.h>**
- Define a thread function that we later pass to **pthread\_create**
  - analogous to a main function
- The thread function is of type **void \***

```
sambowne — debian@debian: ~/COMSC-142/ch14 — ssh debian@192.168.121.173 — 80x17
[debian@debian:~/COMSC-142/ch14$ cat threads1.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* The "thread function" passed to pthread_create.  Each thread executes this
 * function and terminates when it returns from this function. */
void *HelloWorld(void *id) {

    /* We know the argument is a pointer to a long, so we cast it from a
     * generic (void *) to a (long *). */
    long *myid = (long *) id;

    printf("Hello world! I am thread %ld\n", *myid);

    return NULL; // We don't need our threads to return anything.
}
```

# Hello Threading! Writing Your First Multithreaded Program

- main allocates space for **thread\_array** and **thread\_ids**
- **thread\_array** contains the set of addresses for each thread
- **thread\_ids** array stores the set of arguments that each thread is passed

```
sambowne — debian@debian: ~/COMSC-142/ch14 — ssh debian@192.168.121.173 — 80x18
int main(int argc, char **argv) {
    int i;
    int nthreads; //number of threads
    pthread_t *thread_array; //pointer to future thread array
    long *thread_ids;

    // Read the number of threads to create from the command line.
    if (argc !=2) {
        fprintf(stderr, "usage: %s <n>\n", argv[0]);
        fprintf(stderr, "where <n> is the number of threads\n");
        return 1;
    }
    nthreads = strtol(argv[1], NULL, 10);

    // Allocate space for thread structs and identifiers.
    thread_array = malloc(nthreads * sizeof(pthread_t));
    thread_ids = malloc(nthreads * sizeof(long));
```

# Hello Threading! Writing Your First Multithreaded Program

- Creates threads with **pthread\_create**
- Threads execute independently
- **pthread\_join** waits for all the threads to complete
  - Terminates all the threads
  - Then proceeds as a single-thread process

```
sambowne — debian@debian: ~/COMSC-142/ch14 — ssh debian@192.168.121.173 — 80x18

// Assign each thread an ID and create all the threads.
for (i = 0; i < nthreads; i++) {
    thread_ids[i] = i;
    pthread_create(&thread_array[i], NULL, HelloWorld, &thread_ids[i]);
}

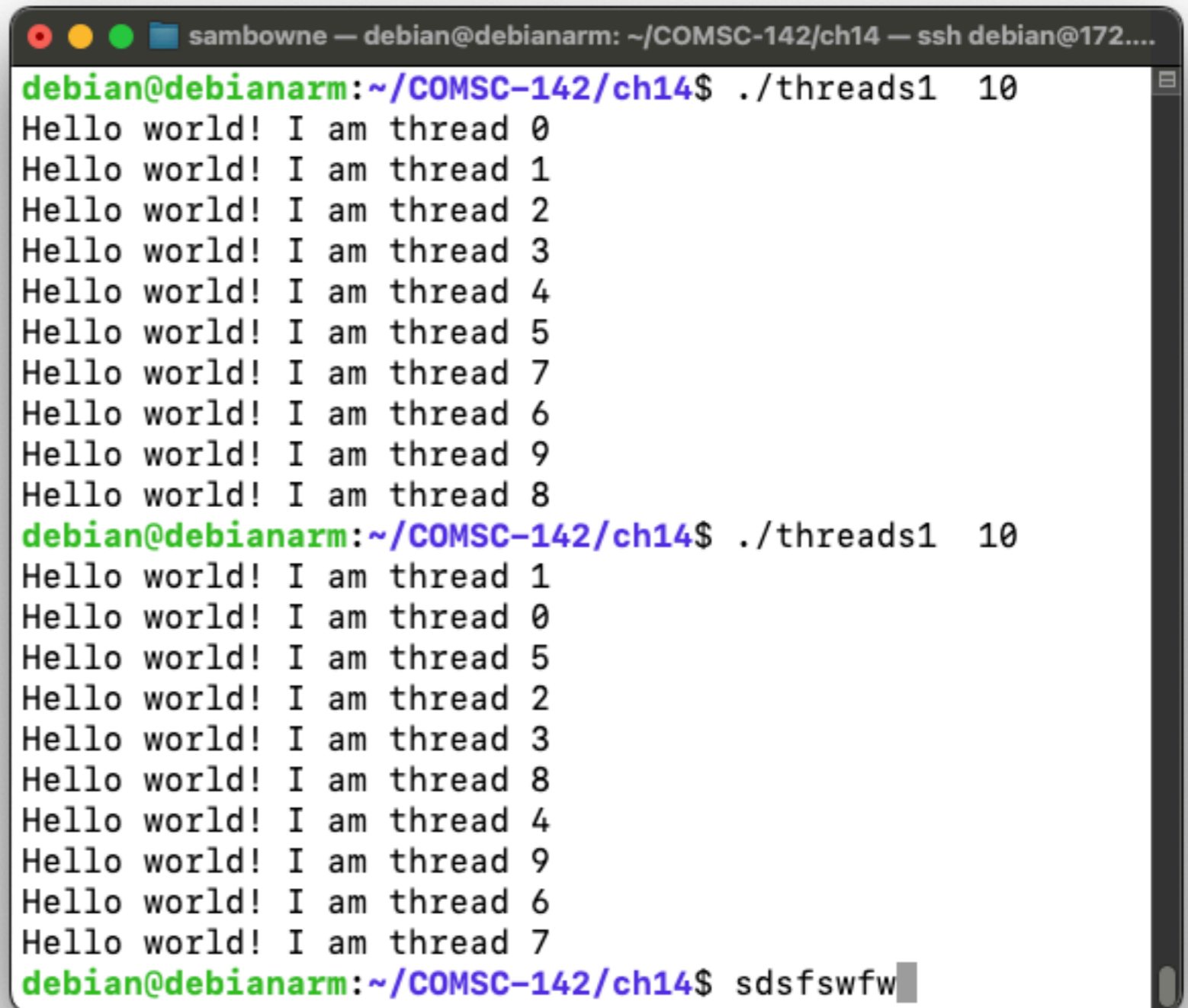
/* Join all the threads. Main will pause in this loop until all threads
 * have returned from the thread function. */
for (i = 0; i < nthreads; i++) {
    pthread_join(thread_array[i], NULL);
}

free(thread_array);
free(thread_ids);

return 0;
}
```

# Thread order

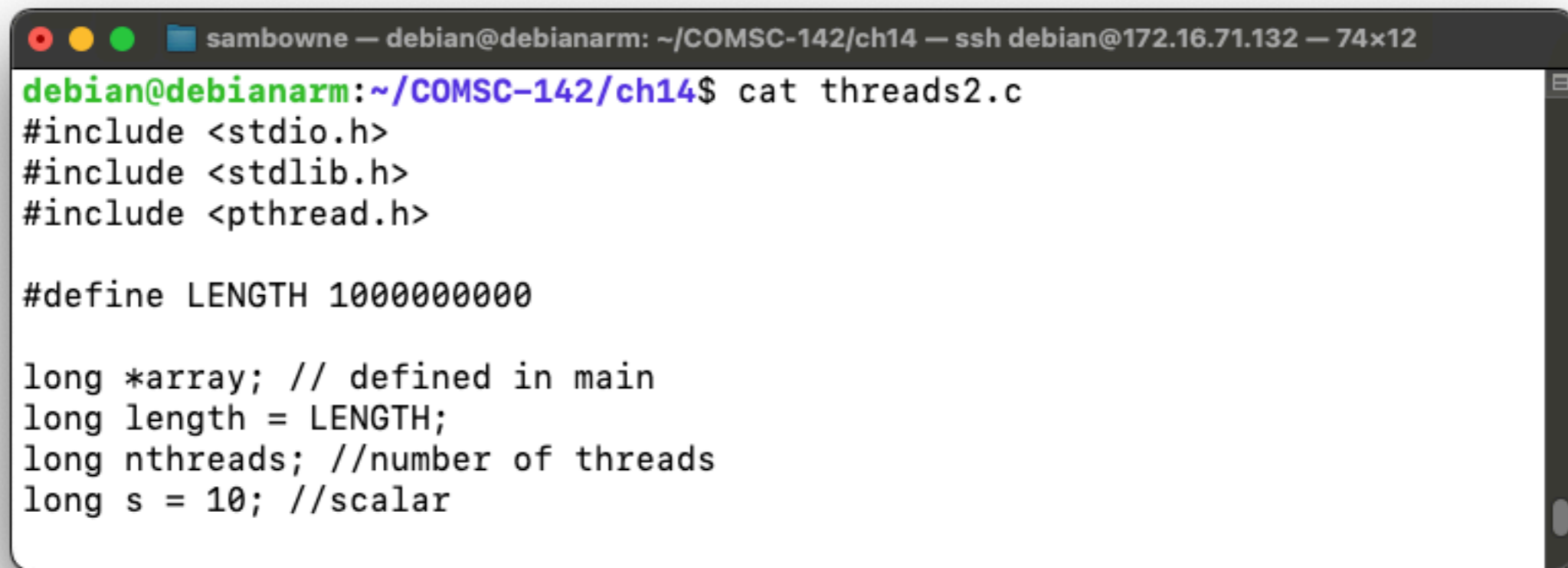
- Threads may not execute in order

A terminal window with a dark background and light text. The window title is "sambowne — debian@debianarm: ~/COMSC-142/ch14 — ssh debian@172...". The terminal shows two runs of a program. The first run shows threads 0 through 9 in sequential order. The second run shows threads 1, 0, 5, 2, 3, 8, 4, 9, 6, 7 in a non-sequential order, demonstrating that threads do not necessarily execute in the order they were created.

```
debian@debianarm:~/COMSC-142/ch14$ ./threads1 10
Hello world! I am thread 0
Hello world! I am thread 1
Hello world! I am thread 2
Hello world! I am thread 3
Hello world! I am thread 4
Hello world! I am thread 5
Hello world! I am thread 7
Hello world! I am thread 6
Hello world! I am thread 9
Hello world! I am thread 8
debian@debianarm:~/COMSC-142/ch14$ ./threads1 10
Hello world! I am thread 1
Hello world! I am thread 0
Hello world! I am thread 5
Hello world! I am thread 2
Hello world! I am thread 3
Hello world! I am thread 8
Hello world! I am thread 4
Hello world! I am thread 9
Hello world! I am thread 6
Hello world! I am thread 7
debian@debianarm:~/COMSC-142/ch14$ sdsfswfw
```

# 14.2.4. Revisiting Scalar Multiplication

- `wget https://samsclass.info/COMSC-142/proj/threads2.c`
- `gcc -o threads2 threads2.c`

A terminal window with a dark title bar. The title bar contains the text "sambowne — debian@debianarm: ~/COMSC-142/ch14 — ssh debian@172.16.71.132 — 74x12". The terminal content shows the command "cat threads2.c" and the output of the file's contents, which includes include statements for <stdio.h>, <stdlib.h>, and <pthread.h>, a macro definition for LENGTH, and variable declarations for array, length, nthreads, and s.

```
debian@debianarm:~/COMSC-142/ch14$ cat threads2.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define LENGTH 1000000000

long *array; // defined in main
long length = LENGTH;
long nthreads; //number of threads
long s = 10; //scalar
```

```
sambowne — debian@debianarm: ~/COMSC-142/ch14 — ssh debian@172.16.71.132 — 76x22

void *scalar_multiply(void *id) {
    long *myid = (long *) id;
    int i;

    //assign each thread its own chunk of elements to process
    long chunk = length / nthreads;
    long start = *myid * chunk;
    long end = start + chunk;
    if (*myid == nthreads - 1) {
        end = length;
    }

    //perform scalar multiplication on assigned chunk
    for (i = start; i < end; i++) {
        array[i] *= s;
    }

    // printf("Thread %d done\n", *myid);
    return NULL;
}
```



```
int main(int argc, char **argv) {
    int i;
    // int nthreads; //number of threads
    pthread_t *thread_array; //pointer to future thread array
    long *thread_ids;

    array = malloc(LENGTH * sizeof(long));

    // Read the number of threads to create from the command line.
    if (argc !=2) {
        fprintf(stderr, "usage: %s <n>\n", argv[0]);
        fprintf(stderr, "where <n> is the number of threads\n");
        return 1;
    }
    nthreads = strtol(argv[1], NULL, 10);

    // Allocate space for thread structs and identifiers.
    thread_array = malloc(nthreads * sizeof(pthread_t));
    thread_ids = malloc(nthreads * sizeof(long));
```



```
// Assign each thread an ID and create all the threads.
for (i = 0; i < nthreads; i++) {
    thread_ids[i] = i;
    pthread_create(&thread_array[i], NULL, scalar_multiply, &thread_ids[i]);
}

/* Join all the threads. Main will pause in this loop until all threads
 * have returned from the thread function. */
for (i = 0; i < nthreads; i++) {
    pthread_join(thread_array[i], NULL);
}

free(thread_array);
free(thread_ids);

return 0;
}
```

- My virtual system has 4 CPUs

```
sambowne — debian@debianarm: ~/COMSC-142/ch14 — ssh debian@172.16.71.132 — 78x9
debian@debianarm:~/COMSC-142/ch14$ { time(./threads2 1) 2>&1; } | grep real
real    0m2.055s
debian@debianarm:~/COMSC-142/ch14$ { time(./threads2 2) 2>&1; } | grep real
real    0m1.181s
debian@debianarm:~/COMSC-142/ch14$ { time(./threads2 4) 2>&1; } | grep real
real    0m0.786s
debian@debianarm:~/COMSC-142/ch14$ { time(./threads2 8) 2>&1; } | grep real
real    0m0.788s
debian@debianarm:~/COMSC-142/ch14$
```

```
sambowne — debian@debianarm: ~/COMSC-142/ch14 — ssh debian@172.1...
debian@debianarm:~/COMSC-142/ch14$ lscpu
Architecture:          aarch64
  CPU op-mode(s):      64-bit
  Byte Order:          Little Endian
CPU(s):                4
  On-line CPU(s) list: 0-3
Vendor ID:             Apple
  Model name:          -
  Model:               0
  Thread(s) per core:  1
  Core(s) per socket:  4
  Socket(s):           1
```

## 14.2.5. Improving Scalar Multiplication: Multiple Arguments

- To avoid global variables, define a struct

```
struct t_arg {  
    int *array; // pointer to shared array  
    long length; // num elements in array  
    long s; //scaling factor  
    long numthreads; // total number of threads  
    long id; // logical thread id  
};
```

# In main()

```
long nthreads = strtol(argv[1], NULL, 10); //get number of threads
long length = strtol(argv[2], NULL, 10); //get length of array
long s = strtol( argv[3], NULL, 10 ); //get scaling factor

int *array = malloc(length*sizeof(int));

//allocate space for thread structs and identifiers
pthread_t *thread_array = malloc(nthreads * sizeof(pthread_t));
struct t_arg *thread_args = malloc(nthreads * sizeof(struct t_arg));

//Populate thread arguments for all the threads
for (i = 0; i < nthreads; i++){
    thread_args[i].array = array;
    thread_args[i].length = length;
    thread_args[i].s = s;
    thread_args[i].numthreads = nthreads;
    thread_args[i].id = i;
}
```

# Pass struct as argument

- In main()

```
for (i = 0; i < nthreads; i++){  
    pthread_create(&thread_array[i], NULL, scalar_multiply, &thread_args[i]);  
}
```

# Kahoot!

**Ch 14a-1**

# **14.3. Synchronizing Threads**



# 14.3. Synchronizing Threads

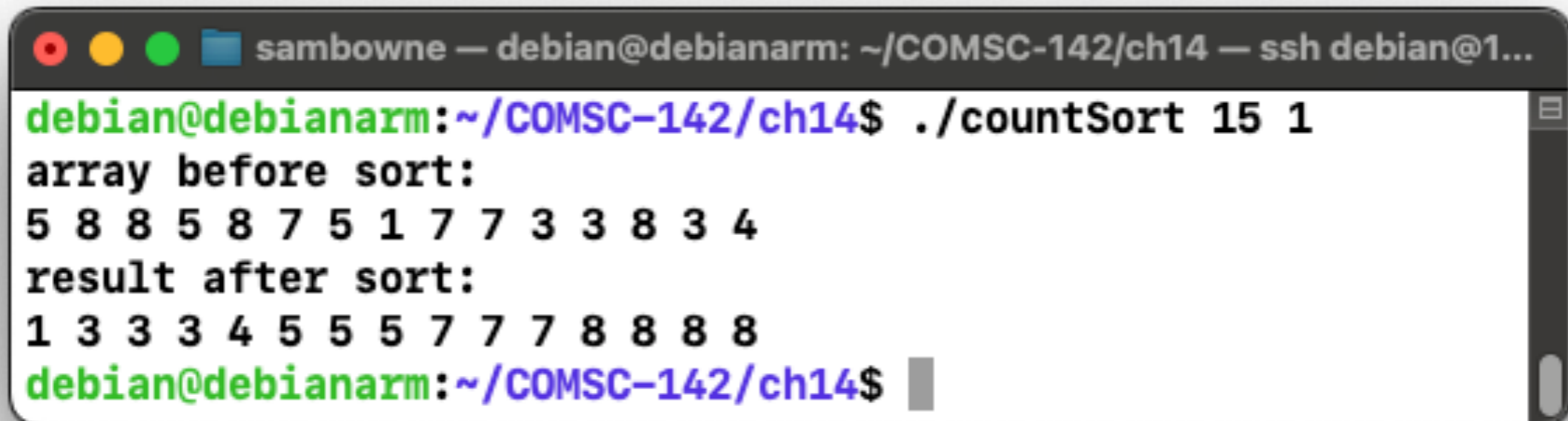
- **Thread synchronization**
  - Forcing threads to execute in a particular order
  - May slow execution, but may be necessary
- Four methods:
  - **Mutex**
  - **Semaphores**
  - **Barriers**
  - **Condition variables**
- All threads of a multithreaded process share the same heap

# CountSort

- Input is an array of  $N$  elements, with only  $R$  possible values
  - $R$  is much smaller than  $N$
- Input is an array  $A$  of 15 items:
  - **$A = [9, 0, 2, 7, 9, 0, 1, 4, 2, 2, 4, 5, 0, 9, 1]$**
  - Possible values are 0 through 9
- CountSort counts the number of times each value occurs
- Creates an array of counts
  - **$\text{counts} = [3, 2, 3, 0, 2, 1, 0, 1, 0, 3]$**
- Use those counts to build a sorted array
  - **$A = [0, 0, 0, 1, 1, 2, 2, 2, 4, 4, 5, 7, 9, 9, 9]$**

# CountSort

- `wget https://samsclass.info/COMSC-142/proj/countSort.c`
- `gcc -o countSort countSort.c`



```
sambowne — debian@debianarm: ~/COMSC-142/ch14 — ssh debian@1...
debian@debianarm:~/COMSC-142/ch14$ ./countSort 15 1
array before sort:
5 8 8 5 8 7 5 1 7 7 3 3 8 3 4
result after sort:
1 3 3 3 4 5 5 5 7 7 7 8 8 8 8
debian@debianarm:~/COMSC-142/ch14$
```

```
/*step 1:
 * compute the frequency of all the elements in the input array and store
 * the associated counts of each element in array counts. The elements in the
 * counts array are initialized to zero prior to the call to this function.
 */
void countElems(int *counts, int *array_A, long length) {
    int val, i;
    for (i = 0; i < length; i++) {
        val = array_A[i]; //read the value at index i
        counts[val] = counts[val] + 1; //update corresponding location in counts
    }
}
```

```
/* step 2:
 * overwrite the input array (array_A) using the frequencies stored in the
 * array counts
 */
void writeArray(int *counts, int *array_A) {
    int i, j = 0, amt;

    for (i = 0; i < MAX; i++) { //iterate over the counts array
        amt = counts[i]; //capture frequency of element i
        while (amt > 0) { //while all values aren't written
            array_A[j] = i; //replace value at index j of array_A with i
            j++; //go to next position in array_A
            amt--; //decrease the amount written by 1
        }
    }
}
```

# Parallel countElems

```
/*parallel version of step 1 (first cut) of CountSort algorithm:  
* extracts arguments from args value  
* calculates the portion of the array that thread is responsible for counting  
* computes the frequency of all the elements in assigned component and stores  
* the associated counts of each element in counts array  
*/  
void *countElems( void *args ) {  
    struct t_arg * myargs = (struct t_arg *)args;  
    //extract arguments (omitted for brevity)  
    int *array = myargs->ap;  
    long *counts = myargs->countp;  
    //... (get nthreads, length, myid)
```

# Parallel countless

```
//assign work to the thread
long chunk = length / nthreads; //nominal chunk size
long start = myid * chunk;
long end = (myid + 1) * chunk;
long val;
if (myid == nthreads-1) {
    end = length;
}

long i;
//heart of the program
for (i = start; i < end; i++) {
    val = array[i];
    counts[val] = counts[val] + 1;
}

return NULL;
}
```

# Results

- Error: different results for different number of threads
- More threads cause undercounts

```
$ gcc -o countElems_p countElems_p.c -pthread

$ ./countElems_p 10000000 1 1
Counts array:
999170 1001044 999908 1000431 999998 1001479 999709 997250 1000804 1000207

$ ./countElems_p 10000000 1 2
Counts array:
661756 661977 657828 658479 657913 659308 658561 656879 658070 657276

$ ./countElems_p 10000000 1 4
Counts array:
579846 580814 580122 579772 582509 582713 582518 580917 581963 581094
```



# Data race (race condition)

- Single-threaded version
  - **counts[val] = counts[val] + 1**
- Multithreaded version
  1. Read **counts[val]** and place into a **register**.
  2. Modify the **register** by incrementing it by one.
  3. Write the contents of the **register** to **counts[val]**.

# Two threads

*Table 1. A Possible Execution Sequence of Two Threads Running countElems*

Time	Thread 0	Thread 1
$i$	Read counts[1] and place into Core 0's register	...
$i+1$	Increment register by 1	Read counts[1] and place into Core 1's register
$i+2$	Overwrite counts[1] with contents of register	Increment register by 1
$i+3$	...	Overwrite counts[1] with contents of register

# Atomic operations

- An **atomic** operation is “all or none”
  - It either completes correctly, or fails completely
- We must isolate the **critical section** of code
  - And make it execute **atomically**

```
long i;
for (i = start; i < end; i++) {
    val = array[i];
    counts[val] = counts[val] + 1; //this line needs to be protected
}
```

# Sequence without error

*Table 2. Another Possible Execution Sequence of Two Threads Running countElems*

Time	Thread 0	Thread 1
$i$	Read counts[1] and place into Core 0's register	...
$i+1$	Increment register by 1	...
$i+2$	Overwrite counts[1] with contents of register	...
$i+3$	...	Read counts[1] and place into Core 1's register
$i+4$	...	Increment register by 1
$i+5$	...	Overwrite counts[1] with contents of register

# Using a Mutex

- Define the **mutex** as a global variable

```
pthread_mutex_t mutex; //global declaration of mutex, initialized in main()
```

- In countElem, only one thread at a time can get a **lock** on the **mutex**
  - Mutexes are unlocked by default

```
//heart of the program
pthread_mutex_lock(&mutex); //acquire the mutex lock
for (i = start; i < end; i++) {
    val = array[i];
    counts[val] = counts[val] + 1;
}
pthread_mutex_unlock(&mutex); //release the mutex lock
```

# Creating and destroying the mutex

```
//code snippet from main():  
  
pthread_mutex_init(&mutex, NULL); //initialize the mutex  
  
for (t = 0; t < nthreads; t++) {  
    pthread_create( &thread_array[t], NULL, countElems, &thread_args[t] );  
}  
  
for (t = 0; t < nthreads; t++) {  
    pthread_join(thread_array[t], NULL);  
}  
  
pthread_mutex_destroy(&mutex); //destroy (free) the mutex
```

# Counting is correct now

```
$ ./countElems_p_v2 10000000 1 1
```

```
Counts array:
```

```
999170 1001044 999908 1000431 999998 1001479 999709 997250 1000804 1000207
```

```
$ ./countElems_p_v2 10000000 1 2
```

```
Counts array:
```

```
999170 1001044 999908 1000431 999998 1001479 999709 997250 1000804 1000207
```

```
$ ./countElems_p_v2 10000000 1 4
```

```
Counts array:
```

```
999170 1001044 999908 1000431 999998 1001479 999709 997250 1000804 1000207
```

# Measuring performance

- More threads consume more time!

```
$ ./countElems_p_v2 100000000 0 1  
Time for Step 1 is 0.368126 s
```

```
$ ./countElems_p_v2 100000000 0 2  
Time for Step 1 is 0.438357 s
```

```
$ ./countElems_p_v2 100000000 0 4  
Time for Step 1 is 0.519913 s
```



# Locked for whole loop

- One thread must complete its loop before another thread can run
- Makes the program effectively serial

```
//code snippet from the countElems function from earlier  
//the heart of the program  
pthread_mutex_lock(&mutex); //acquire the mutex lock  
for (i = start; i < end; i++){  
    val = array[i];  
    counts[val] = counts[val] + 1;  
}  
pthread_mutex_unlock(&mutex); //release the mutex lock
```

# The Mutex: Reloaded

- Only lock for each write operation

```
/*modified code snippet of countElems function:  
 *locks are now placed INSIDE the for loop!  
 */  
 //the heart of the program  
 for (i = start; i < end; i++) {  
     val = array[i];  
     pthread_mutex_lock(&m); //acquire the mutex lock  
     counts[val] = counts[val] + 1;  
     pthread_mutex_unlock(&m); //release the mutex lock  
 }
```

# Operates correctly

```
$ ./countElems_p_v3 10000000 1 1
```

```
Counts array:
```

```
999170 1001044 999908 1000431 999998 1001479 999709 997250 1000804 1000207
```

```
$ ./countElems_p_v3 10000000 1 2
```

```
Counts array:
```

```
999170 1001044 999908 1000431 999998 1001479 999709 997250 1000804 1000207
```

```
$ ./countElems_p_v3 10000000 1 4
```

```
Counts array:
```

```
999170 1001044 999908 1000431 999998 1001479 999709 997250 1000804 1000207
```

# Performance

- Locking and unlocking are expensive operations
  - A lot of time overhead

```
$ ./countElems_p_v3 100000000 0 1  
Time for Step 1 is 1.92225 s
```

```
$ ./countElems_p_v3 100000000 0 2  
Time for Step 1 is 10.9704 s
```

```
$ ./countElems_p_v3 100000000 0 4  
Time for Step 1 is 9.13662 s
```

# The Mutex: Revisited

- Each thread has a private local array of counts
- Only uses mutex when adding totals to global counts array

```
//heart of the program
for (i = start; i < end; i++) {
    val = array[i];

    //updates local counts array
    local_counts[val] = local_counts[val] + 1;
}

//update to global counts array
pthread_mutex_lock(&mutex); //acquire the mutex lock
for (i = 0; i < MAX; i++) {
    counts[i] += local_counts[i];
}
pthread_mutex_unlock(&mutex); //release the mutex lock
```

# Performance

- More threads consume less wall time

```
$ ./countElems_p_v3 100000000 0 1  
Time for Step 1 is 0.334574 s  
  
$ ./countElems_p_v3 100000000 0 2  
Time for Step 1 is 0.209347 s  
  
$ ./countElems_p_v3 100000000 0 4  
Time for Step 1 is 0.130745 s
```

# Deadlock

- A **deadlocked** thread is blocked from execution by another thread, which itself is blocked on a blocked thread

```
struct account {  
    pthread_mutex_t lock;  
    int balance;  
};
```

```
void *Transfer(void *args){  
    //argument passing removed to increase readability  
    //...  
  
    pthread_mutex_lock(&fromAcct->lock);  
    pthread_mutex_lock(&toAcct->lock);  
  
    fromAcct->balance -= amt;  
    toAcct->balance += amt;  
  
    pthread_mutex_unlock(&fromAcct->lock);  
    pthread_mutex_unlock(&toAcct->lock);  
  
    return NULL;  
}
```

# Deadlock Condition

- Thread 0 sends funds from A to B
- Thread 1 sends funds from B to A

## Thread 0

```
Transfer(...) {  
    //acctA is fromAcct  
    //acctB is toAcct  
    pthread_mutex_lock(&acctA->lock);  
    Thread 0 gets here  
    pthread_mutex_lock(&acctB->lock);
```

## Thread 1

```
Transfer(...) {  
    //acctB is fromAcct  
    //acctA is toAcct  
    pthread_mutex_lock(&acctB->lock);  
    Thread 1 gets here  
    pthread_mutex_lock(&acctA->lock);
```

*Figure 1. An example of deadlock*



# Avoiding deadlock

- Only lock one mutex at a time

```
void *Transfer(void *args){  
    //argument passing removed to increase readability  
    //...  
  
    pthread_mutex_lock(&fromAcct->lock);  
    fromAcct->balance -= amt;  
    pthread_mutex_unlock(&fromAcct->lock);  
  
    pthread_mutex_lock(&toAcct->lock);  
    toAcct->balance += amt;  
    pthread_mutex_unlock(&toAcct->lock);  
  
    return NULL;  
}
```

# 14.3.2. Semaphores

- Can have many values
- **Counting semaphore**
  - Values: **0** through  $r$
  - Decrements each time a resource is used
  - When the counting semaphore reaches **0**, no more resources are available
    - Any other threads trying to acquire the resource are blocked
- Can be locked by default
- Any thread can unlock a semaphore
  - Only the calling thread can unlock a mutex

# Using semaphores

- **#include <semaphore.h>**
- There is no standard; function calls are different on different systems
- Declaration: **sem\_t semaphore**
- Initialize: **sem\_init(&semaphore, 1, 0)**
  - Parameters: address of semaphore, default state, whether to share with threads of a process (e.g., with value 0) or between processes (e.g., with value 1)
- Destroy: **sem\_destroy(&semaphore)**

# Using semaphores

- **sem\_wait** function indicates that a resource is being used, and decrements the semaphore
  - Blocks if semaphore reaches zero
- **sem\_post** indicates that a resource is being freed, and increments the semaphore

# 14.3.3. Other Synchronization Constructs

- **barrier**
  - part of the **pthread** library
  - forces all threads to reach a common point in execution before releasing the threads to continue executing
- Usage:
  - Declare a global variable **pthread\_barrier\_t barrier**
  - Initialize in main **pthread\_barrier\_init(&barrier)**
  - Destroy after use **pthread\_barrier\_destroy(&barrier)**
  - **pthread\_barrier\_wait** function creates a synchronization point

# Barrier example

```
void *threadEx(void *args){
    //parse args
    //...
    long myid = myargs->id;
    int nthreads = myargs->numthreads;
    int *array = myargs->array

    printf("Thread %ld starting thread work!\n", myid);
    pthread_barrier_wait(&barrier); //forced synchronization point
    printf("All threads have reached the barrier!\n");
    for (i = start; i < end; i++) {
        array[i] = array[i] * 2;
    }
    printf("Thread %ld done with work!\n", myid);

    return NULL;
}
```

# Condition Variables

- part of the **pthread**s library
- force a thread to block until a particular condition is reached
- always used in conjunction with a mutex

# Condition variables

- Usage:
  - initialize with **pthread\_cond\_init**
  - destroy with **pthread\_cond\_destroy**
  - **pthread\_cond\_wait(&cond, &mutex)**
    - causes the calling thread to block on the condition variable cond until another thread signals it (or "wakes" it up)
  - **pthread\_cond\_signal(&cond)**
    - causes the calling thread to unblock (or signal) another thread that is waiting



# Condition variable example

```
int main(int argc, char **argv){
    //... declarations omitted for brevity

    // these will be shared by all threads via pointer fields in t_args
    int num_eggs;           // number of eggs ready to collect
    pthread_mutex_t mutex; // mutex associated with cond variable
    pthread_cond_t  eggs;  // used to block/wake-up farmer waiting for eggs

    //... args parsing removed for brevity

    num_eggs = 0; // number of eggs ready to collect
    ret = pthread_mutex_init(&mutex, NULL); //initialize the mutex
    pthread_cond_init(&eggs, NULL); //initialize the condition variable
```

# Condition variable example

- main()

```
// create some chicken and farmer threads
for (i = 0; i < (2 * nthreads); i++) {
    if ( (i % 2) == 0 ) {
        ret = pthread_create(&thread_array[i], NULL,
                            chicken, &thread_args[i]);
    }
    else {
        ret = pthread_create(&thread_array[i], NULL,
                            farmer, &thread_args[i] );
    }
}
}
```

# Condition variable example

- main()

```
// wait for chicken and farmer threads to exit
for (i = 0; i < (2 * nthreads); i++) {
    ret = pthread_join(thread_array[i], NULL);
}

// clean-up program state
pthread_mutex_destroy(&mutex); //destroy the mutex
pthread_cond_destroy(&eggs);   //destroy the cond var

return 0;
```

# Condition variable example

```
void *chicken(void *args ) {
    struct t_arg *myargs = (struct t_arg *)args;
    int *num_eggs, i, num;

    num_eggs = myargs->num_eggs;
    i = 0;

    // lay some eggs
    for (i = 0; i < myargs->total_eggs; i++) {
        usleep(EGGTIME); //chicken sleeps

        pthread_mutex_lock(myargs->mutex);
        *num_eggs = *num_eggs + 1; // update number of eggs
        num = *num_eggs;
        pthread_cond_signal(myargs->eggs); // wake a sleeping farmer (squawk)
        pthread_mutex_unlock(myargs->mutex);

        printf("chicken %d created egg %d available %d\n",myargs->id,i,num);
    }
    return NULL;
}
```

# Condition variable example

```
void *farmer(void *args ) {
    struct t_arg * myargs = (struct t_arg *)args;
    int *num_eggs, i, num;

    num_eggs = myargs->num_eggs;

    i = 0;

    for (i = 0; i < myargs->total_eggs; i++) {
        pthread_mutex_lock(myargs->mutex);
        while (*num_eggs == 0 ) { // no eggs to collect
            // wait for a chicken to lay an egg
            pthread_cond_wait(myargs->eggs, myargs->mutex);
        }

        // we hold mutex lock here and num_eggs > 0
        num = *num_eggs;
        *num_eggs = *num_eggs - 1;
        pthread_mutex_unlock(myargs->mutex);

        printf("farmer %d gathered egg %d available %d\n", myargs->id, i, num);
    }
    return NULL;
}
```

# Broadcasting

- `pthread_cond_broadcast(&cond)`
  - wakes up all threads that are blocked on condition **cond**

```
// start barrier code
pthread_mutex_lock(&mutex);
*n_reached++;

printf("Thread %ld starting work!\n", myid)

// if some threads have not reached the barrier
while (*n_reached < nthreads) {
    pthread_cond_wait(&barrier, &mutex);
}
// all threads have reached the barrier
printf("all threads have reached the barrier!\n");
pthread_cond_broadcast(&barrier);

pthread_mutex_unlock(&mutex);
// end barrier code
```

# Kahoot!

**Ch 14a-2**