Systems Architecture

9. Concurrency in C

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2024/2025

uc3m | Universidad Carlos III de Madrid

BY NC SA

Table of contents

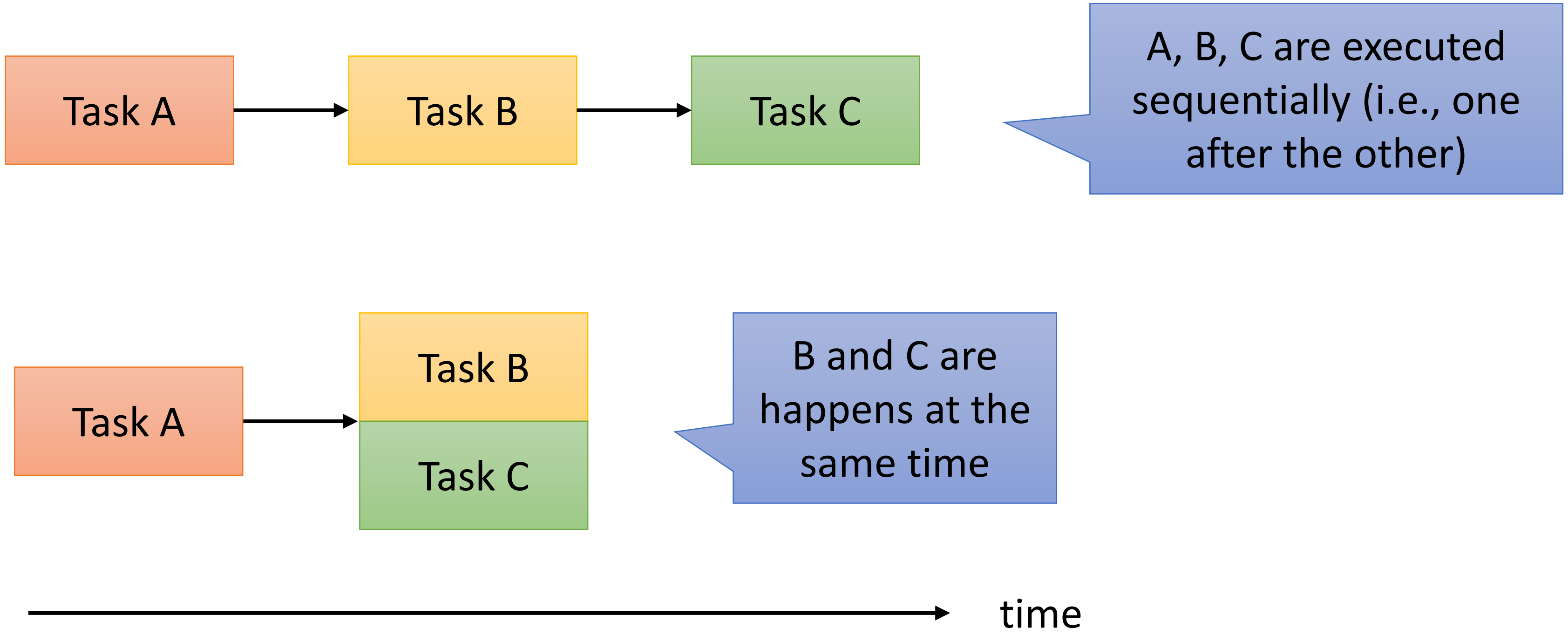
1. Introduction

- In software, **concurrency** means multiple computations are happening at the same time
- Concurrency is essential in modern programming to improve **performance**, for instance:
  - Web servers must handle multiple simultaneous users
  - Mobile apps need to do some of their processing on servers
  - Graphical User Interfaces (GUIs) require background work that does not interrupt the user
- In this unit, we learn the basics of concurrent programming in C:
  - How to manage *threads*
  - How to synchronize *threads*

Table of contents

2. Concurrency basics

- Generally speaking, **concurrency** is about two or more separate activities happening **at the same time**



Task A

Task B

Task C

A, B, C are executed sequentially (i.e., one after the other)

Task A

Task B

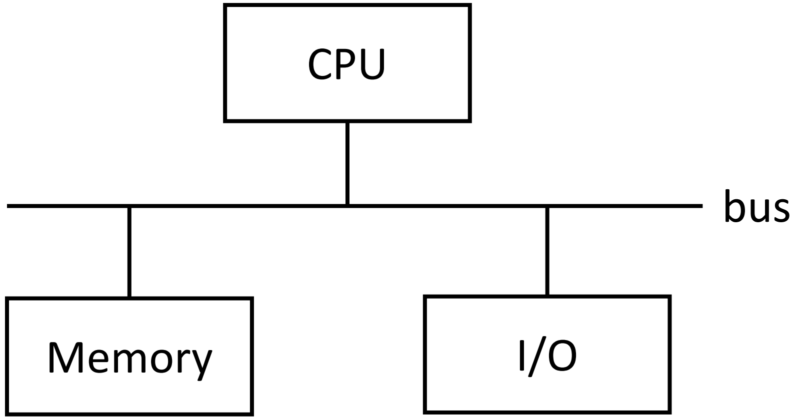
Task C

B and C are happens at the same time

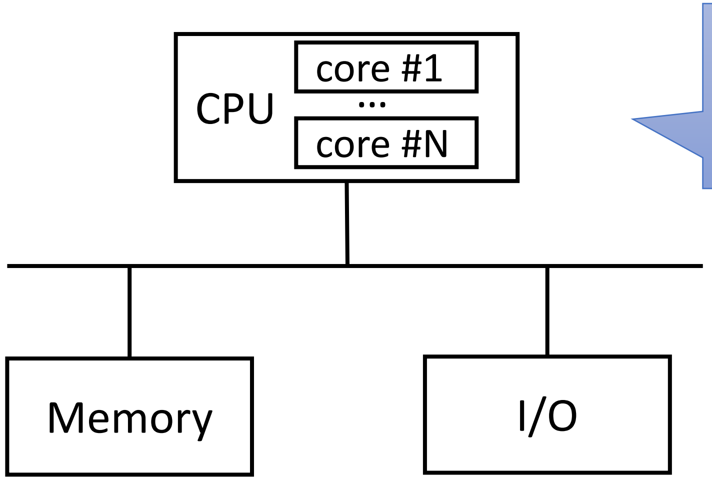
time

2. Concurrency basics

- Computer systems generally consist of the following parts:
  - The Central Processing Unit (**CPU**): executes instructions of computer programs
  - **Primary memory**: holds the programs and data to be processed
  - **I/O** (input/output) devices: peripherals that communicate with the outside world
  - **Bus**: communication system that transfers data between components



single-core system

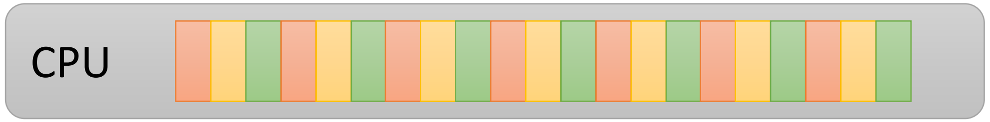


Each *core* is a single processing units (CPU)

multi-core system

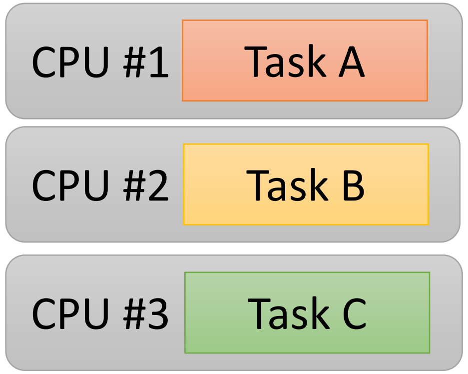
2. Concurrency basics

- In computer systems, we can distinguish between:
  - **Concurrency**: multiple tasks are performed in overlapping time periods with shared resources (e.g., *time slicing* on a single-core machine)



Logical concurrency

  - **Parallelism**: multiple tasks are tasks literally run at the same time (e.g., on a multicore processor)



Physical concurrency

2. Concurrency basics

- There are two main models for concurrent programming:

1. **Shared memory**: concurrent modules interact by reading and writing shared data in the same memory segment. For instance:
   - Two processes running in the same computer, reading and writing in the same filesystem
   - Two threads in the same process sharing the same variables
2. **Message passing**. In the message-passing model, concurrent modules interact by sending messages to each other through a communication channel. For instance:
   - Two processes running in different computers connected by the network
   - Two processes running in the same computer connected by a pipe

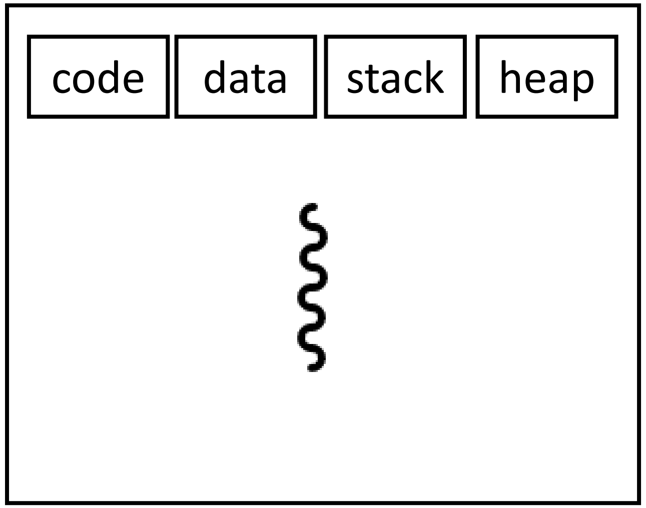
Table of contents

3. Processes vs threads - Processes

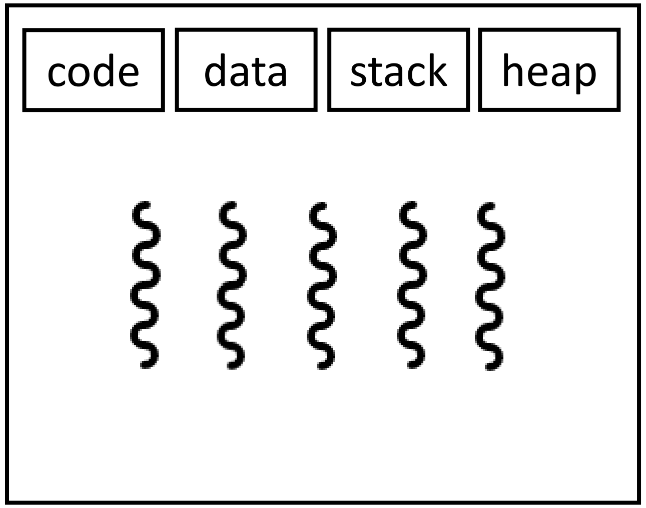
- A **process** is an executing program in a given operating system (e.g., Linux, Windows, macOS, etc.)
  - We use a separate system call (called `fork` in Unix-like systems) to create a process
  - Each process is independent and treated as an isolated entity in the operating system
  - Each process exists within its own address or memory space
  - Processes use some mechanisms called IPC (Inter-Process Communication) to communicate with each other, such as:
    - Files, sockets, message queue, pipes, or signals, among other mechanisms

3. Processes vs threads - Threads

- A **thread** is an execution unit that is part of a process
  - A process can have more than one thread
  - A thread is lightweight and can be managed independently
  - The thread takes less time to terminate as compared to the process but unlike the process, threads do not isolate



Single-threaded process



Multi-threaded process

3. Processes vs threads - Comparison

- The key differences between processes and threads are:

	Processes	Threads
Definition	Execution of a program	Segment of a process
Execution time	Processes takes more time for creation and termination	Threads takes less time for creation and termination
Memory	Process are totally isolated (don’t share memory)	Threads share memory
Communication	Processes must use inter-process communication (files, signals, pipes, etc.) to communicate with other processes	Threads can directly communicate with other threads of its process using shared memory
Controlled by	Process is controlled by the operating system	Threads are controlled by programmer in a program

3. Processes vs threads - Multitasking

- Concurrency at the operating system level is called **multitasking** (i.e., the concurrent execution of multiple processes over time)
  - To implement multitasking, the operating systems use a **process scheduler** to decide which process uses the CPU and for how long
- The most common process scheduler in Linux is called Completely Fair Scheduler (CFS)
  - CFS implements an algorithm to handle the scheduling of runnable entities (called tasks) which are either threads or (single-threaded) processes
  - CFS uses a technique called **context switching** to save and restore the state of the tasks being executed

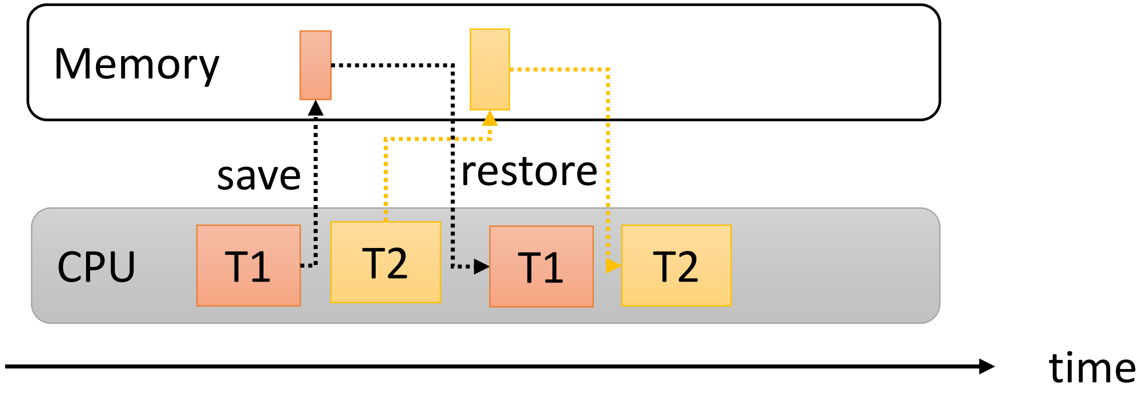


Table of contents

4. POSIX threads

- POSIX threads (*pthreads*) is a parallel execution model based on standards
  - POSIX (Portable Operating System Interface) is a family of IEEE standards to ensure compatibility between operating systems
- Pthreads are implemented in the C language in the `pthread.h` library
  - Once created, each thread progresses independently. This causes each of the threads can potentially travel at a different speed, running concurrently
- Programs that use pthreads need to be compiled with the option `-pthread` (to add support for multithreading)

```
gcc program.c -pthread
```

4. POSIX threads

- The **`pthread_create()`** function starts a new thread in the calling process. Its prototype is as follows:

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

  - Number of arguments: 4
    - 1st argument (**`thread`**): Pointer to the thread id (unique identifier)
    - 2nd argument (**`attr`**): Pointer to the configuration attributes for the new thread. If it is NULL, the thread is created with default attributes
    - 3rd argument (**`start_routine`**): Function executed as a new thread
    - 4th argument (**`arg`**): Pointer to the arguments of the function passed as the 3rd argument
  - Return value:
    - On success: 0
    - On error: error number (different than 0)

4. POSIX threads

- The **`pthread_join()`** function is used in order to wait for a thread to complete its execution. Its prototype is as follows:

```
int pthread_join(pthread_t thread, void **thread_return);
```

  - Number of arguments: 2
    - 1st argument (**`thread`**): Thread id that the current thread is waiting for
    - 2nd argument (**`thread_return`**): pointer that points to the location that stores the return status of the thread id that is referred to in the 1st argument
  - Return value:
    - On success: 0
    - On error: error number (different than 0)

4. POSIX threads

- The **`pthread_exit()`** function is used to terminate a thread

```
void pthread_exit(void *retval);
```

  - Number of arguments: 1
    - 1st argument (**`retval`**): Pointer to the return value (integer). This value has the return status of the thread

- The **`pthread_self()`** function obtain the id of the calling thread

```
pthread_t pthread_self(void);
```

  - Return value: calling thread’s id

4. POSIX threads

- The usual procedure to use POSIX threads in a C program is as follows:
  1. Declare a variable to identify the thread (type `pthread_t`)
  2. Define the thread function (`void* my_thread(void *data) {…}`)
     - Optionally, this function can control some input arguments
     - Also optionally, it can return some value
  3. Create thread (invoking function `pthread_create(…)`)
     - We can implement error handling here
  4. Synchronize threads. We typically wait for thread to finish (invoking function `pthread_join(…)`)
     - Returned values are handled with this function

Fork me on GitHub

4. POSIX threads

- Example 1: creating a thread without passing data and without return value

The `sleep` function pauses the execution a number of seconds

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void* thread_run(void *data) {
    pthread_t pt_id = pthread_self();
    printf("[PTHR: %ld]: New thread started\n", pt_id);

    sleep(3);

    printf("[PTHR: %ld]: Finishing new thread\n", pt_id);
    pthread_exit(NULL);
}

int main() {
    pthread_t thread_id;
    pthread_t main_id = pthread_self();

    printf("[MAIN: %ld]: Starting new thread from main\n", main_id);
    int thread_rc = pthread_create(&thread_id, NULL, thread_run, NULL);
    if (thread_rc != 0) {
        printf("Error creating thread %i\n", thread_rc);
        exit(1);
    }

    pthread_join(thread_id, NULL);
    printf("[MAIN: %ld]: Thread finished\n", main_id);

    return 0;
}
```

```
[MAIN: 140343014979392]: Starting new thread from main
[PTHR: 140343014975232]: New thread started
[PTHR: 140343014975232]: Finishing new thread
[MAIN: 140343014979392]: Thread finished
```

4. POSIX threads

- Example 2: creating a thread passing data (an integer value) and without return value

Fork me on GitHub

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void* thread_run(void *data) {
    pthread_t pt_id = pthread_self();
    printf("[PTHR: %ld]: New thread started\n", pt_id);

    int *th_data = (int*) data;
    printf("[PTHR: %ld]: Data received: %d\n", pt_id, *th_data);

    sleep(3);

    printf("[PTHR: %ld]: Finishing new thread\n", pt_id);
    pthread_exit(NULL);
}

int main() {
    pthread_t thread_id;
    pthread_t main_id = pthread_self();
    int data = 10;

    printf("[MAIN: %ld]: Starting new thread from main\n", main_id);
    int thread_rc = pthread_create(&thread_id, NULL, thread_run, &data);
    if (thread_rc != 0) {
        printf("Error creating thread %i\n", thread_rc);
        exit(1);
    }

    pthread_join(thread_id, NULL);
    printf("[MAIN: %ld]: Thread finished\n", main_id);

    return 0;
}
```

```
[MAIN: 139984335865664]: Starting new thread from main
[PTHR: 139984335861504]: New thread started
[PTHR: 139984335861504]: Data received: 10
[PTHR: 139984335861504]: Finishing new thread
[MAIN: 139984335865664]: Thread finished
```

4. POSIX threads

- Example 3: creating a thread passing data (a struct) and without return value

Fork me on GitHub

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

typedef struct thread_data {
    int a;
    int b;
} thread_data;

void* thread_run(void *data) {
    pthread_t pt_id = pthread_self();
    printf("[PTHR: %ld]: New thread started\n", pt_id);

    thread_data *th_data = (thread_data*) data;
    printf("[PTHR: %ld]: Data received: %d %d\n", pt_id, th_data->a,
            th_data->b);

    sleep(3);

    printf("[PTHR: %ld]: Finishing new thread\n", pt_id);
    pthread_exit(NULL);
}

int main() {
    pthread_t thread_id;
    pthread_t main_id = pthread_self();
    thread_data data = { .a = 10, .b = 20 };

    printf("[MAIN: %ld]: Starting new thread from main\n", main_id);
    int thread_rc = pthread_create(&thread_id, NULL, thread_run, &data);
    if (thread_rc != 0) {
        printf("Error creating thread %i\n", thread_rc);
        exit(1);
    }

    pthread_join(thread_id, NULL);
    printf("[MAIN: %ld]: Thread finished\n", main_id);

    return 0;
}
```

```
[MAIN: 140021651552064]: Starting new thread from main
[PTHR: 140021651547904]: New thread started
[PTHR: 140021651547904]: Data received: 10 20
[PTHR: 140021651547904]: Finishing new thread
[MAIN: 140021651552064]: Thread finished
```

4. POSIX threads

- Example 4: creating a thread passing data (an integer value) and with a return value (another integer value)

```
[MAIN: 140594746173248]: Starting new thread from main
[PTHR: 140594746169088]: New thread started
[PTHR: 140594746169088]: Finishing new thread
[MAIN: 140594746173248]: Thread finished, returning 32764
```

Fork me on GitHub

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void* thread_run(void *data) {
    pthread_t pt_id = pthread_self();
    printf("[PTHR: %ld]: New thread started\n", pt_id);

    sleep(3);

    printf("[PTHR: %ld]: Finishing new thread\n", pt_id);

    int ret = 42;
    pthread_exit(&ret);
}

int main() {
    pthread_t thread_id;
    pthread_t main_id = pthread_self();

    printf("[MAIN: %ld]: Starting new thread from main\n", main_id);
    int thread_rc = pthread_create(&thread_id, NULL, thread_run, NULL);
    if (thread_rc != 0) {
        printf("Error creating thread %i\n", thread_rc);
        exit(1);
    }

    int *output;
    pthread_join(thread_id, (void**) &output);
    printf("[MAIN: %ld]: Thread finished, returning %d\n", main_id, *output);

    return 0;
}
```

4. POSIX threads

- Example 5: creating a thread passing data (an integer value) and with a return value (another integer value)

```
[MAIN: 139752538543936]: Starting new thread from main
[PTHR: 139752538539776]: New thread started
[PTHR: 139752538539776]: Finishing new thread
[MAIN: 139752538543936]: Thread finished, returning 42
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void* thread_run(void *data) {
    pthread_t pt_id = pthread_self();
    printf("[PTHR: %ld]: New thread started\n", pt_id);

    sleep(3);

    printf("[PTHR: %ld]: Finishing new thread\n", pt_id);

    int *ret = malloc(sizeof(int));
    *ret = 42;
    pthread_exit(ret);
}

int main() {
    pthread_t thread_id;
    pthread_t main_id = pthread_self();

    printf("[MAIN: %ld]: Starting new thread from main\n", main_id);
    int thread_rc = pthread_create(&thread_id, NULL, thread_run, NULL);
    if (thread_rc != 0) {
        printf("Error creating thread %i\n", thread_rc);
        exit(1);
    }

    int *output;
    pthread_join(thread_id, (void**) &output);
    printf("[MAIN: %ld]: Thread finished, returning %d\n", main_id, *output);
    free(output);

    return 0;
}
```

4. POSIX threads

- Example 6: creating a thread passing data (a struct) and with a return value (an integer value)

```
[MAIN: 140524621989696]: Starting new thread from main
[PTHR: 140524621985536]: New thread started
[PTHR: 140524621985536]: Data received: 10 20
[PTHR: 140524621985536]: Finishing new thread
[MAIN: 140524621989696]: Thread finished, returning 42
```

Fork me on GitHub

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

typedef struct thread_data {
    int a;
    int b;
    int result;
} thread_data;

void* thread_run(void *data) {
    pthread_t pt_id = pthread_self();
    printf("[PTHR: %ld]: New thread started\n", pt_id);

    thread_data *th_data = (thread_data*) data;
    printf("[PTHR: %ld]: Data received: %d %d\n", pt_id, th_data->a,
            th_data->b);

    sleep(3);

    printf("[PTHR: %ld]: Finishing new thread\n", pt_id);

    th_data->result = 42;
    pthread_exit(NULL);
}

int main() {
    pthread_t thread_id;
    pthread_t main_id = pthread_self();
    thread_data data = { .a = 10, .b = 20 };

    printf("[MAIN: %ld]: Starting new thread from main\n", main_id);
    int thread_rc = pthread_create(&thread_id, NULL, thread_run, &data);
    if (thread_rc != 0) {
        printf("Error creating thread %i\n", thread_rc);
        exit(1);
    }

    pthread_join(thread_id, NULL);
    printf("[MAIN: %ld]: Thread finished, returning %d\n", main_id, data.result);

    return 0;
}
```

4. POSIX threads

- Multitasking improves the operating systems performance, but it also adds complexity
  - Because threads can run simultaneously, there’s no inherent guarantee about the order in which parts of your code on different threads will run
- This can lead to problems, such as:
  - Race conditions, in which threads are accessing data or resources in an inconsistent order
  - Deadlocks, in which two threads are waiting for each other, preventing both threads from continuing
  - Bugs that only happen in certain situations and are hard to reproduce and fix reliably

Table of contents

5. Race conditions

- Concurrent programs can suffer from synchronization problems that make the program to exhibit an unexpected behavior
- For example, consider the following program:

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

#define MAX 1000

int counter = 0;

void* count(void *arg) {
    for (int i = 0; i < MAX; i++) {
        counter++;
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("counter: %d\n", counter);

    return 0;
}
```

What is the value of the variable counter in this line?

Fork me on GitHub

5. Race conditions

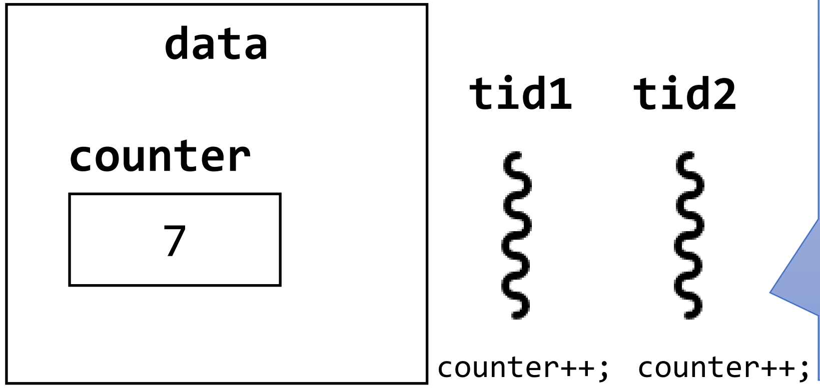
- Now, consider the following alternative (it only changes the value of **MAX** to **1000000**):

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

#define MAX 1000000

int counter = 0;

void* count(void *arg) {
    for (int i = 0; i < MAX; i++) {
        counter++;
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("counter: %d\n", counter);

    return 0;
}
```

What is the value of the variable counter in this line?

5. Race conditions

- A **race condition** is a situation on concurrent programming where two concurrent threads (or processes) compete for a resource and the resulting final state depends on who gets the resource first
- A **data race** is specific type of race condition that occurs when several threads access a shared variable and try to modify it at the same time
  - In the previous example, a data race happens since two threads try to modify the global variable `counter` at the same time



Each thread does the following (*read-modify-write*):

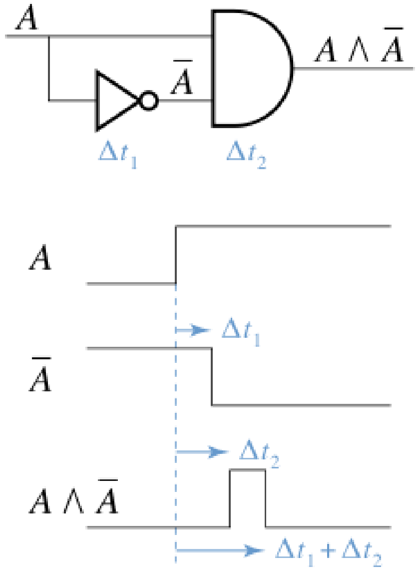
a. Retrieve the value of **`counter`**
b. Add **1** to this value
c. Store this value to **`counter`**

The scheduling algorithm can swap between threads at any time, e.g.:

`tid1`: reads **`counter`**, value is **7**
`tid1`: add **1** to **`counter`**, value is now **8**
`tid2`: reads **`counter`**, value is **7**
`tid1`: stores **8** in **`counter`**
`tid2`: adds **1** to **`counter`**, value is now **8**
`tid2`: stores **8** in **`counter`**

5. Race conditions

- The term *race condition* in programming has been borrowed from the hardware industry
- The term was coined with the idea of two signals racing each other to influence the output first (e.g., a race condition in a logic circuit):



Source:
https://en.wikipedia.org/wiki/Race_condition

In software, instead of signals, we have processes/threads competing for the same resource

5. Race conditions

- Race conditions can be avoided by employing some sort of **locking** mechanism before the code that accesses the shared resource
- For instance, in the previous example

```
void* count(void *arg) {
    for (int i = 0; i < MAX; i++) {
        // lock counter
        counter++;
        // unlock counter
    }
    pthread_exit(NULL);
}
```

We need to create a called **critical section** here, i.e., a protected region that only one process/thread can enter into at a time

Table of contents

6. Mutexes

- A **mutex** (short from *mutual exclusion*), also called lock, is a synchronization mechanism that enforces limits on access to a resource when there are many threads of execution
  - Synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment
- When a mutex is set, no other thread can access the locked region (**critical section**)
- Mutexes are used to protect shared resources, preventing inconsistencies due to race conditions
- If a mutex is already locked by one thread, the other threads **wait** for the mutex to become unlocked
  - In other words, only one process/thread can enter into critical section at a time

6. Mutexes

- In C, a mutex is an special variable of type **`pthread_mutex_t`** that can take two states: *locked* or *unlocked*
- The procedure to use a mutex in C is the following:
  1. Declare a mutex (variable with type `pthread_mutex_t`)
  2. Initialize the mutex
  3. Lock a mutex (creating a critical section)
  4. Unlock the mutex (releasing the critical section)
  5. Destroy the mutex

6. Mutexes

- To declare a mutex (1), we simply use:

```
pthread_mutex_t mutex;
```

- There are two ways of initializing (2) a mutex. In this course, we are going to use mutexes with the default attributes, therefore the initialization is as follows:
  a) Using the macro `PTHREAD_MUTEX_INITIALIZER`:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

  b) Calling to the function `pthread_mutex_init`:

```
pthread_mutex_init(&mutex, NULL);
```

Manual page:
https://linux.die.net/man/3/pthread_mutex_init

6. Mutexes

- To lock a mutex (3), we need to invoke:

```
pthread_mutex_lock(&mutex);
```

- To unlock a mutex (4), we need to invoke:

```
pthread_mutex_unlock(&mutex);
```

The piece of code between these two statements is the **critical section**

- When the unlock is not required anymore, we need to destroy it (5):

```
pthread_mutex_destroy(&mutex);
```

6. Mutexes

- This program prevents the race condition (caused for the concurrent access in the variable `counter`) by creating a critical section using a mutex

Fork me on GitHub

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

#define MAX 1000000

int counter = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* count(void *arg) {
    for (int i = 0; i < MAX; i++) {
        pthread_mutex_lock(&mutex);
        counter++;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("counter: %d\n", counter);

    pthread_mutex_destroy(&mutex);

    return 0;
}
```

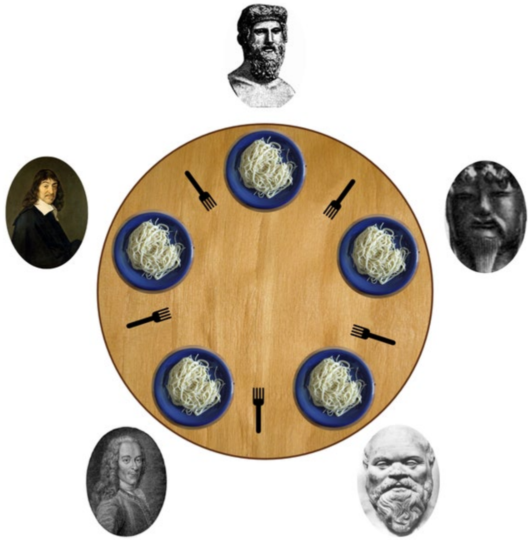
```
counter: 2000000
```

Table of contents

7. Deadlocks

- The bad use of mutex can lead to undesired behavior in our programs
- **Deadlock** is a situation where a set of threads are blocked because each one is holding a resource (e.g. a mutex) and waiting for another resource acquired by some other thread

A classic problem to model deadlocks is “The Dining Philosophers Problem” (originally formulated in 1965 by Edsger Dijkstra)

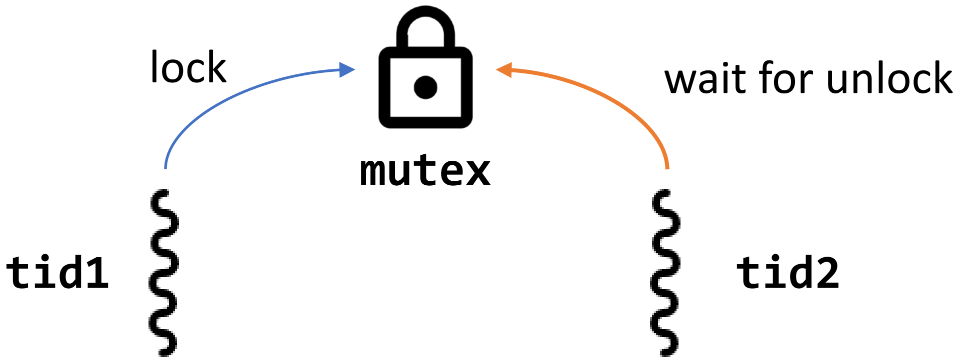


> *Five silent philosophers sit at a round table with bowls of food. Forks are placed between each pair of philosophers. All day the philosophers take turns eating and thinking. A philosopher must have two forks in order to eat, and each fork may only be used by one philosopher at a time. At any time a philosopher can pick up or set down the fork on their right or left, but cannot start eating until picking up both forks.*

Source: https://en.wikipedia.org/wiki/Dining_philosophers_problem

7. Deadlocks

- Deadlock example #1:



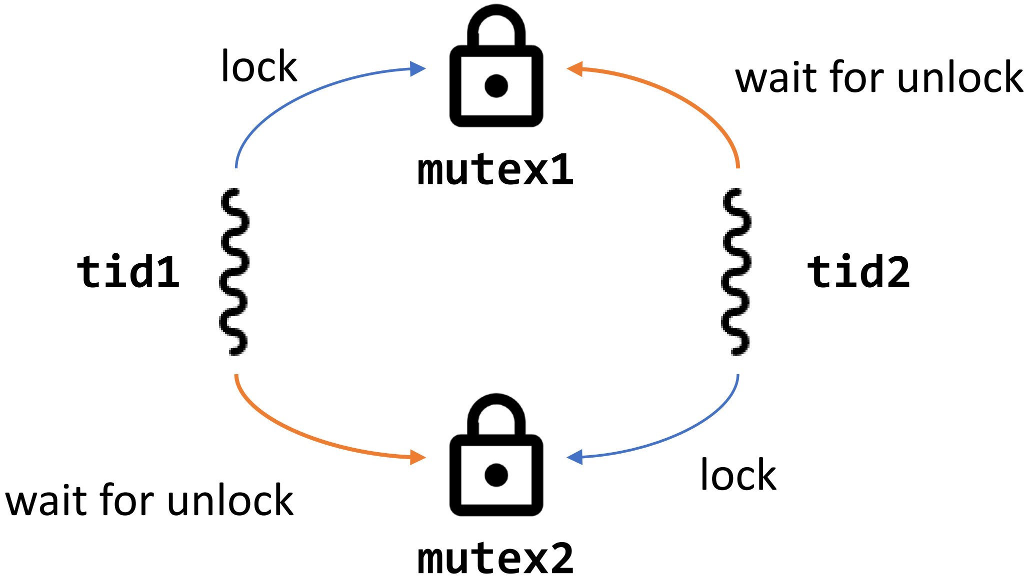
```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* thread_1(void *data) {
    pthread_mutex_lock(&mutex);
    printf("[PTHR: %ld]: Thread 1 started\n", pthread_self());

    pthread_exit(NULL);
}

void* thread_2(void *data) {
    pthread_mutex_lock(&mutex);
    printf("[PTHR: %ld]: Thread 2 started\n", pthread_self());
    pthread_mutex_unlock(&mutex);

    pthread_exit(NULL);
}

int main() {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, thread_1, NULL);
    pthread_create(&tid2, NULL, thread_2, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Fork me on GitHub

Fork me on GitHub

7. Deadlocks

- Deadlock example #2:



```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void* thread_1(void *data) {
    pthread_mutex_lock(&mutex1);
    sleep(1);
    pthread_mutex_lock(&mutex2);
    printf("[PTHR: %ld]: Thread 1 started\n", pthread_self());
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);

    pthread_exit(NULL);
}

void* thread_2(void *data) {
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(&mutex1);
    printf("[PTHR: %ld]: Thread 2 started\n", pthread_self());
    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex2);

    pthread_exit(NULL);
}

int main() {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, thread_1, NULL);
    pthread_create(&tid2, NULL, thread_2, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    pthread_mutex_destroy(&mutex1);
    pthread_mutex_destroy(&mutex2);

    return 0;
}
```

Table of contents

8. Helgrind

- Concurrent programs can be very difficult to debug
  - It is hard to make them happen the same way twice
- Concurrency bugs exhibit very poor reproducibility
  - Each time you run a program containing a race condition, you may get different behavior
  - These kinds of bugs are sometimes called *heisenbugs*, since they are nondeterministic and hard to reproduce

The term *heisenbugs* is coined from the Heisenberg Uncertainty Principle (quantum mechanics) which states that the act of observing a system inevitably alters its state

8. Helgrind

- To detect this kind of problems, the tool **Helgrind** (contained in Valgrind) might help
- Helgrind is a Valgrind tool for detecting synchronization errors in C programs that use the pthreads
- Helgrind can detect three classes of errors:
  - Bad use of the pthreads API
  - Potential deadlocks
  - Race conditions (accessing memory without adequate locking or synchronization)

Helgrind manual:
https://valgrind.org/docs/manual/hg-manual.html

8. Helgrind

- To use Helgrind in Valgrind we need to do the following:

1. Compile our program with the debug and pthread options:

```
gcc -g -pthread my_program.c -o my_program
```

2. Invoke Valgrind passing the executable as argument:

```
valgrind --tool=helgrind ./my_program
```

8. Helgrind

- For instance, as we have seen, this program has an specific type of race condition called data race
- So, let’s analyze it with Helgrind

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

#define MAX 1000

int counter = 0;

void* count(void *arg) {
    for (int i = 0; i < MAX; i++) {
        counter++;
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("counter: %d\n", counter);

    return 0;
}
```

8. Helgrind

```
gcc -pthread -g race_condition_1.c -o race_condition_1
```

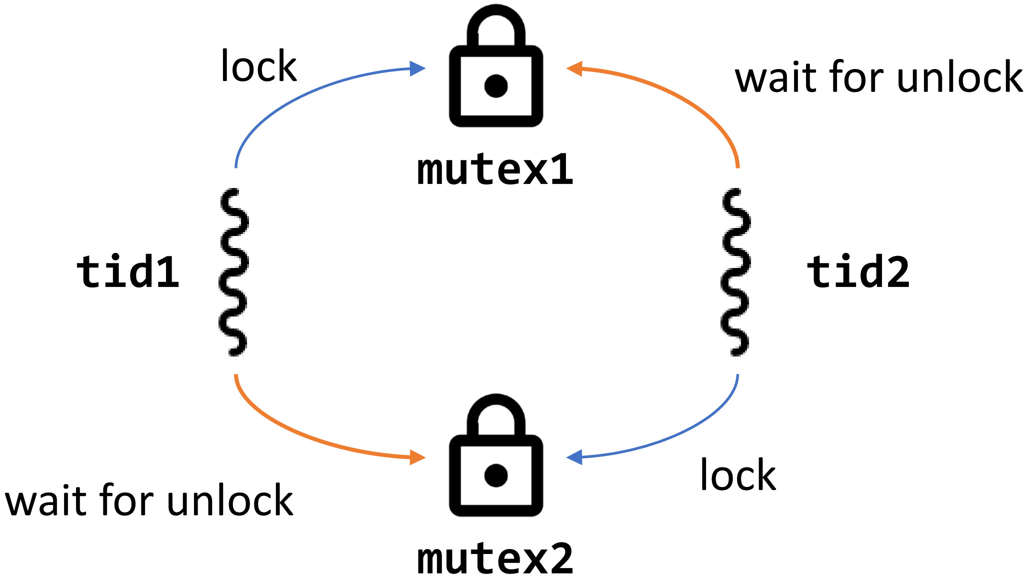
```
valgrind --tool=helgrind ./race_condition_1
```

```
==224== Helgrind, a thread error detector
==224== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==224== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==224== Command: ./race_condition_1
...
==224== Possible data race during read of size 4 at 0x10C014 by thread #3
==224== Locks held: none
==224==    at 0x1091E2: count (race_condition_1.c:10)
==224==    by 0x4842B1A: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgrind-amd64-linux.so)
==224==    by 0x4861608: start_thread (pthread_create.c:477)
==224==    by 0x499D102: clone (clone.S:95)
==224==
==224== This conflicts with a previous write of size 4 by thread #2
==224== Locks held: none
==224==    at 0x1091EB: count (race_condition_1.c:10)
==224==    by 0x4842B1A: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgrind-amd64-linux.so)
==224==    by 0x4861608: start_thread (pthread_create.c:477)
==224==    by 0x499D102: clone (clone.S:95)
==224==  Address 0x10c014 is 0 bytes inside data symbol "counter"
...
counter: 2000
==224==
==224== Use --history-level=approx or =none to gain increased speed, at
==224== the cost of reduced accuracy of conflicting-access information
==224== For lists of detected and suppressed errors, rerun with: -s
==224== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 44 from 14)
```

Fork me on GitHub

8. Helgrind

- Also, the following program contained a deadlock, although most of the times, the problem does not happen in runtime



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

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void* thread_1(void *data) {
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    printf("[PTHR: %ld]: Thread 1 started\n", pthread_self());
    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);

    pthread_exit(NULL);
}

void* thread_2(void *data) {
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(&mutex1);
    printf("[PTHR: %ld]: Thread 2 started\n", pthread_self());
    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex2);

    pthread_exit(NULL);
}

int main() {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, thread_1, NULL);
    pthread_create(&tid2, NULL, thread_2, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    pthread_mutex_destroy(&mutex1);
    pthread_mutex_destroy(&mutex2);

    return 0;
}
```

8. Helgrind

Fork me on GitHub

```
gcc -pthread -g deadlock_3.c -o deadlock_3
```

```
valgrind --tool=helgrind ./deadlock_3
```

```
==531== Helgrind, a thread error detector
==531== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==531== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==531== Command: ./deadlock_3
==531==
[PTHR: 86443776]: Thread 1 started
...
==227== Thread #3: lock order "0x10C040 before 0x10C080" violated
==227==
==227== Observed (incorrect) order is: acquisition of lock at 0x10C080
==227==    at 0x483FEDF: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgrind-amd64-linux.so)
==227==    by 0x1092C7: thread_2 (deadlock_3.c:18)
==227==    by 0x4842B1A: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgrind-amd64-linux.so)
==227==    by 0x4861608: start_thread (pthread_create.c:477)
==227==    by 0x499D102: clone (clone.S:95)
...

[PTHR: 99030784]: Thread 2 started
==531==
==531== Use --history-level=approx or =none to gain increased speed, at
==531== the cost of reduced accuracy of conflicting-access information
==531== For lists of detected and suppressed errors, rerun with: -s
==531== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 55 from 25)
```

8. Helgrind

- As usual, we aim to get zero errors in the Valgrind report:

```
gcc -pthread -g deadlock_3_sol.c -o deadlock_3_sol
```

```
valgrind --tool=helgrind --history-level=approx ./deadlock_3_sol
```

```
==593== Helgrind, a thread error detector
==593== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==593== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==593== Command: ./deadlock_3_sol
==593==
[PTHR: 86443776]: Thread 1 started
[PTHR: 99030784]: Thread 2 started
==593==
==593== For lists of detected and suppressed errors, rerun with: -s
==593== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 51 from 21)
```

Table of contents

9. Takeaways

- In programming, **concurrency** means multiple tasks are happening **at the same time** (run in parallel in multiple processor or by time slicing on a single processor)
- These tasks can be **processes** (program in execution in an operating system) or **threads** (piece of code within a process)
- In C, we use the POSIX threads (**pthreads**) API to manage threads
- A **race condition** happens when concurrent threads (or processes) compete for a shared resource and the resulting final state depends on who gets the resource first
- To avoid race conditions we can use **mutexes** (*mutual exclusion*) to protect shared resources. If a mutex is locked by a thread (critical section), other threads wait for the mutex to become unlocked
- **Deadlock** is a situation where a set of threads are blocked because each one is holding a resource (e.g. a mutex) and waiting for another resource acquired by some other thread
- **Helgrind** is a Valgrind tool for detecting synchronization errors (such as deadlocks or race conditions) in concurrent C programs