Lab 6

CS-335a

Fall 2012
Computer Science Department

Manolis Surligas
surligas@csd.uoc.gr

Summary

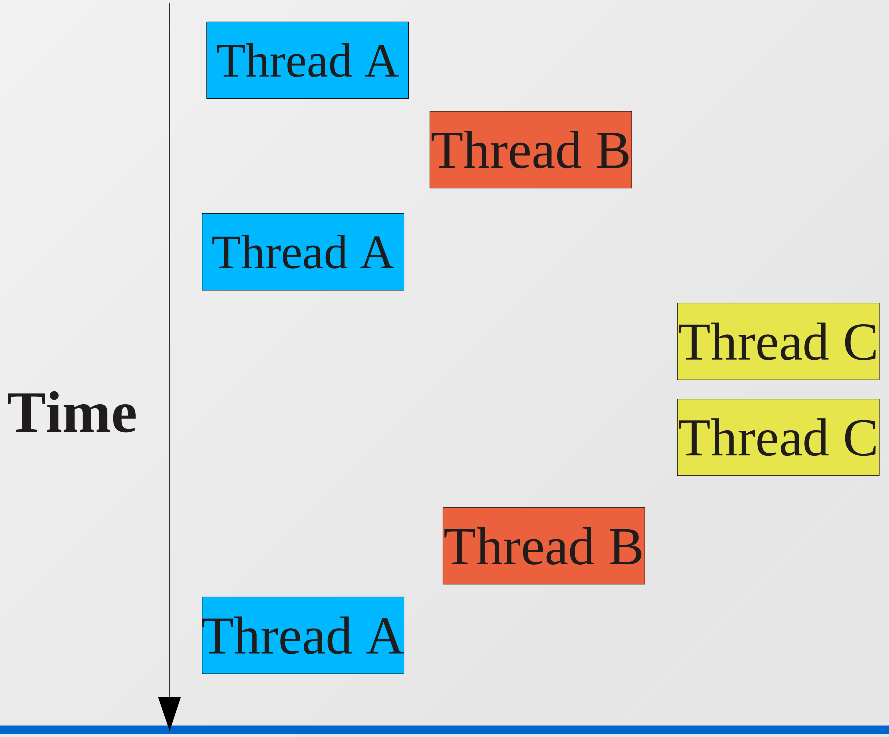
- What is a thread?
- Parallel ecexution
- Creating threads
- Passing parameters to threads with pthread_create()
- A multithreaded TCP server
- Avoiding race conditions

What is a thread?

- A thread is a lightweight process that is handled by the sheduler of the OS
- A process may own several threads
- A process may share with its threads resources, like a common memory address space
- Threads can communicate with other threads
- One thread may perform a task, while another performs another, in **parallel**

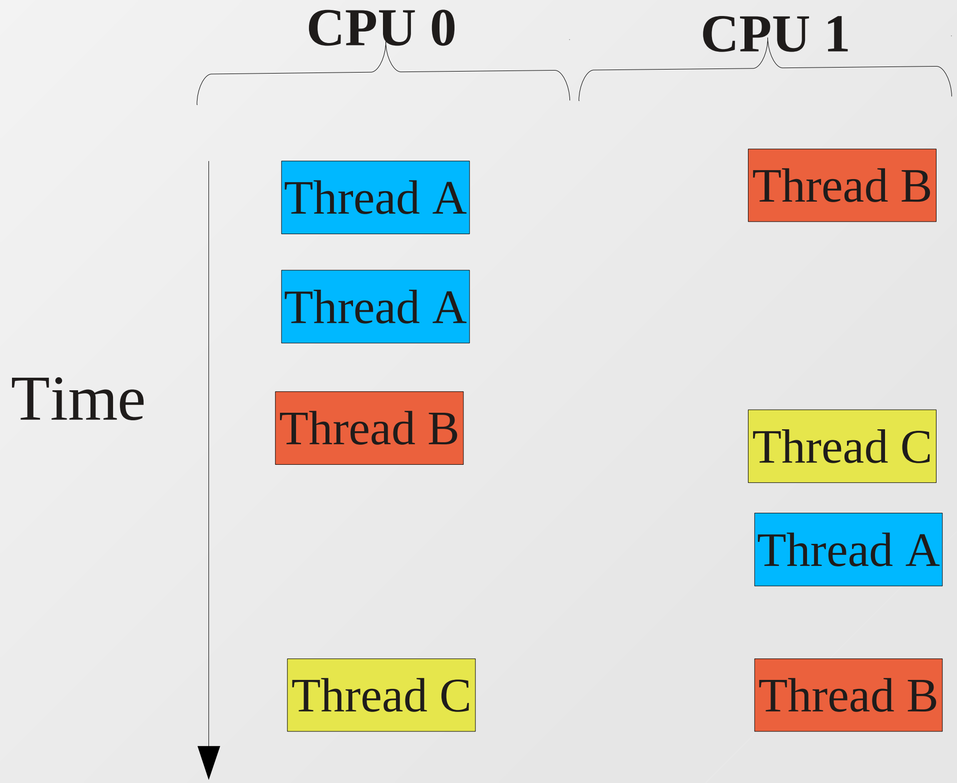
Parallel ecexution

- But how threads allow parallelism?
- In a single processor machine, the processor switches between different threads
- This transition is very fast, so the user has the feeling that threads run in parralel



Parallel ecexution

- The same it is done in multi-core systems
- The difference is that every core may run a different thread at the same time

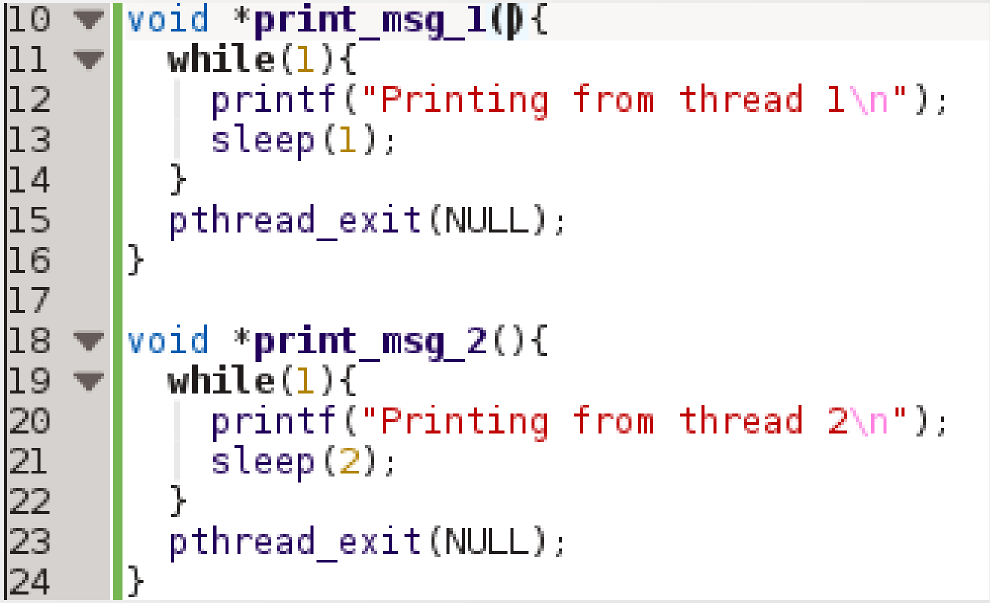


Creating threads

- For creating and manipulating threads, we are going to use the POSIX standard, usually known as *pthreads*
- Pthreads are implemented in all modern Linux distributions
- They come with very discriptive man pages
- A list with all available pthread functions, can be found by typing:
  - man pthread.h

Creating threads

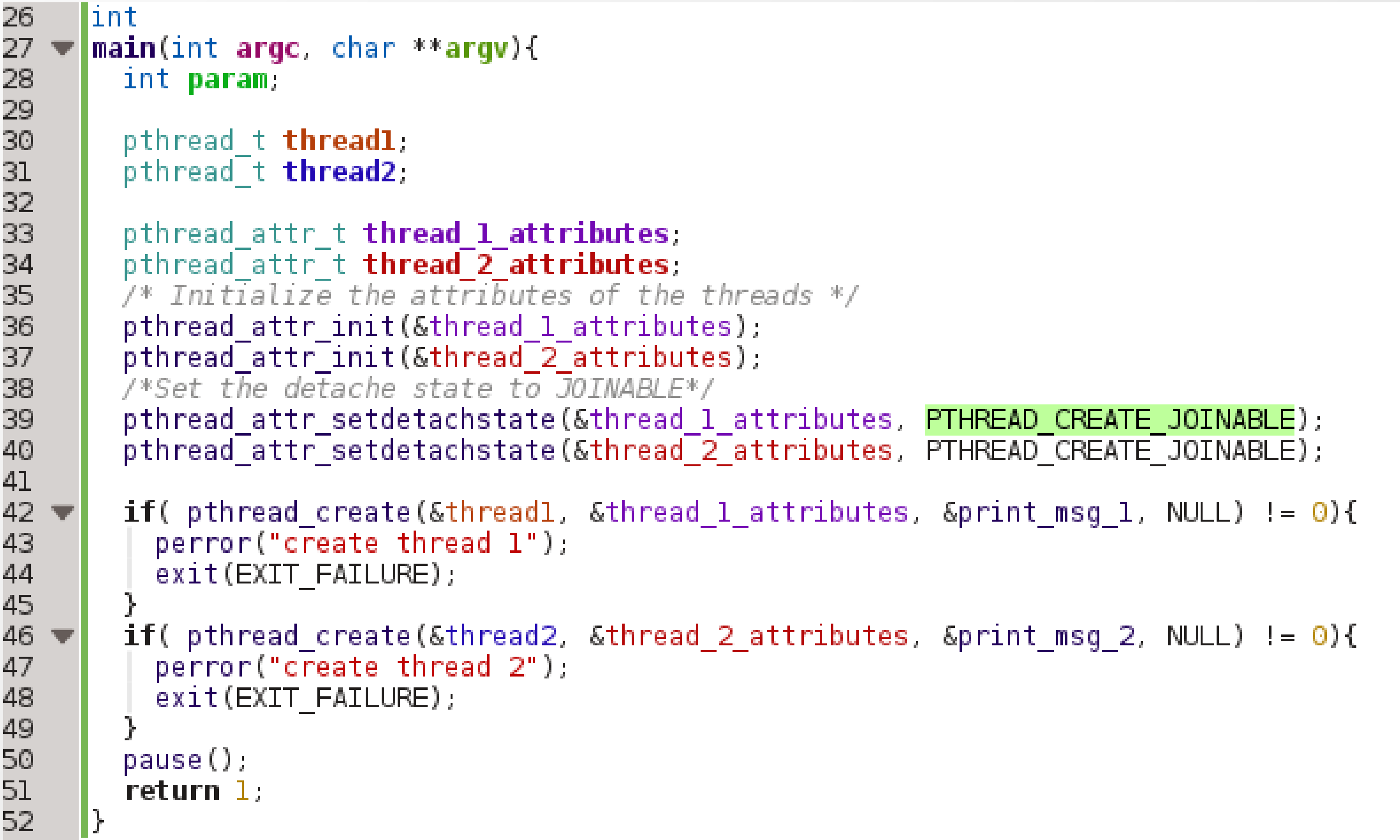
- Lets create our first simple threads
- Each thread will execute a simple function, that prints a different message



```
void *print_msg_1(){
  while(1){
    printf("Printing from thread 1\n");
    sleep(1);
  }
  pthread_exit(NULL);
}

void *print_msg_2(){
  while(1){
    printf("Printing from thread 2\n");
    sleep(2);
  }
  pthread_exit(NULL);
}
```

Creating threads



```c
int
main(int argc, char **argv){
  int param;

  pthread_t thread1;
  pthread_t thread2;

  pthread_attr_t thread_1_attributes;
  pthread_attr_t thread_2_attributes;
  /* Initialize the attributes of the threads */
  pthread_attr_init(&thread_1_attributes);
  pthread_attr_init(&thread_2_attributes);
  /*Set the detache state to JOINABLE*/
  pthread_attr_setdetachstate(&thread_1_attributes, PTHREAD_CREATE_JOINABLE);
  pthread_attr_setdetachstate(&thread_2_attributes, PTHREAD_CREATE_JOINABLE);

  if( pthread_create(&thread1, &thread_1_attributes, &print_msg_1, NULL) != 0){
    perror("create thread 1");
    exit(EXIT_FAILURE);
  }
  if( pthread_create(&thread2, &thread_2_attributes, &print_msg_2, NULL) != 0){
    perror("create thread 2");
    exit(EXIT_FAILURE);
  }
  pause();
  return 1;
}
```

Creating threads

- Lines 36,37: Initialize the variables that will hold the attributes of each thread
- Lines 39,40: Set the detached state to JOINABLE
- Lines 42,46: Create the thread by running the corresponding functions
- Line 50: Without *pause()* the process would terminate. Another solution may be the use of *pthread_join()* for every thread

Passing parameters to threads with pthread_create()

- At the previous example, the threads were not have any parameters

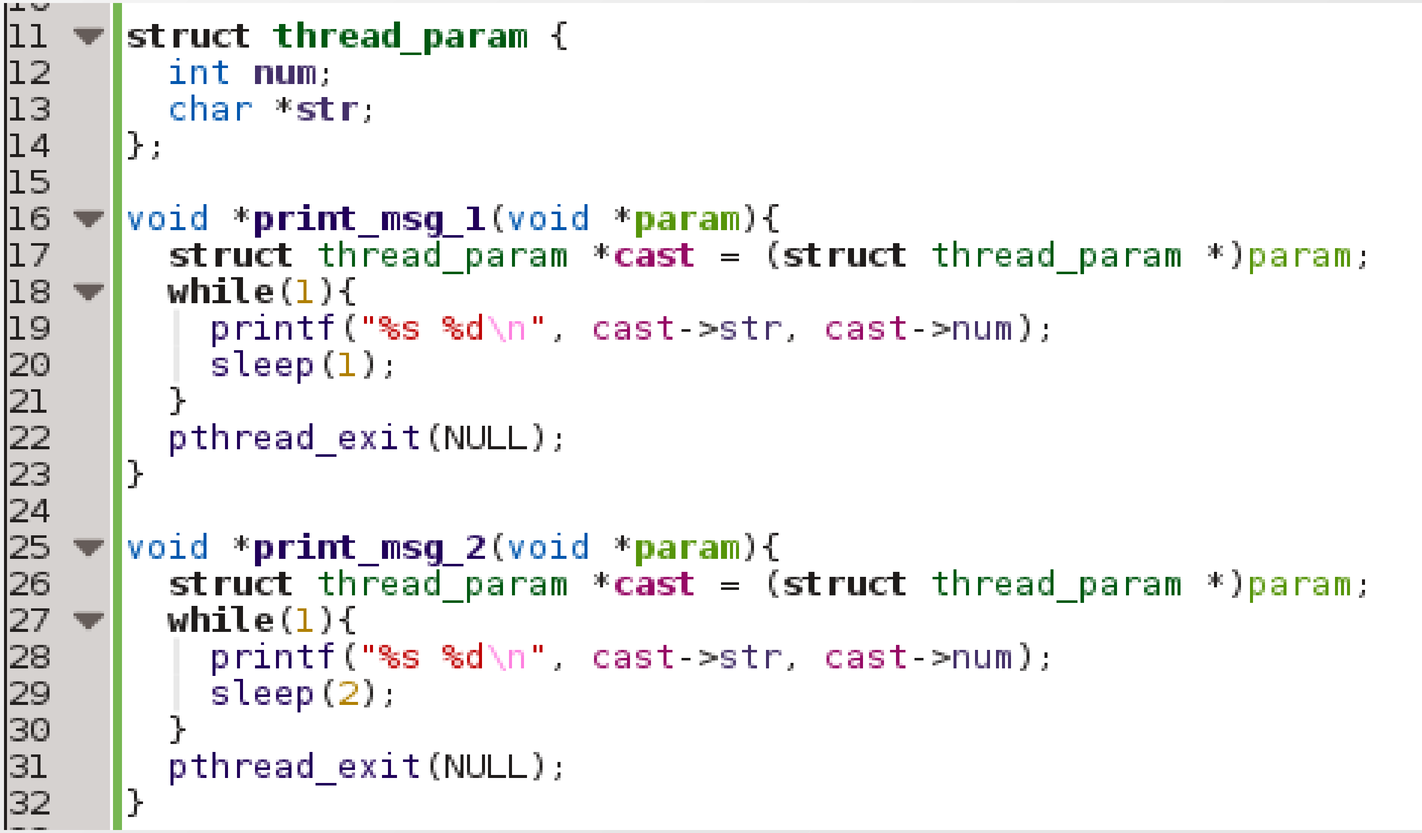
- This is not the general case, as we frequently pass many parameters to our functions

- The problem is that pthread_create(), restrict us to use only one parameter

- Not a problem! Declare an appropriate struct and perfom the necessary type casts

Passing parameters to threads with pthread_create()

- Assume that we want our threads to take as parameters an integer and a string
- We create the appropriate struct and perfom the cast at the functions, that are taking only a *void ** parameter



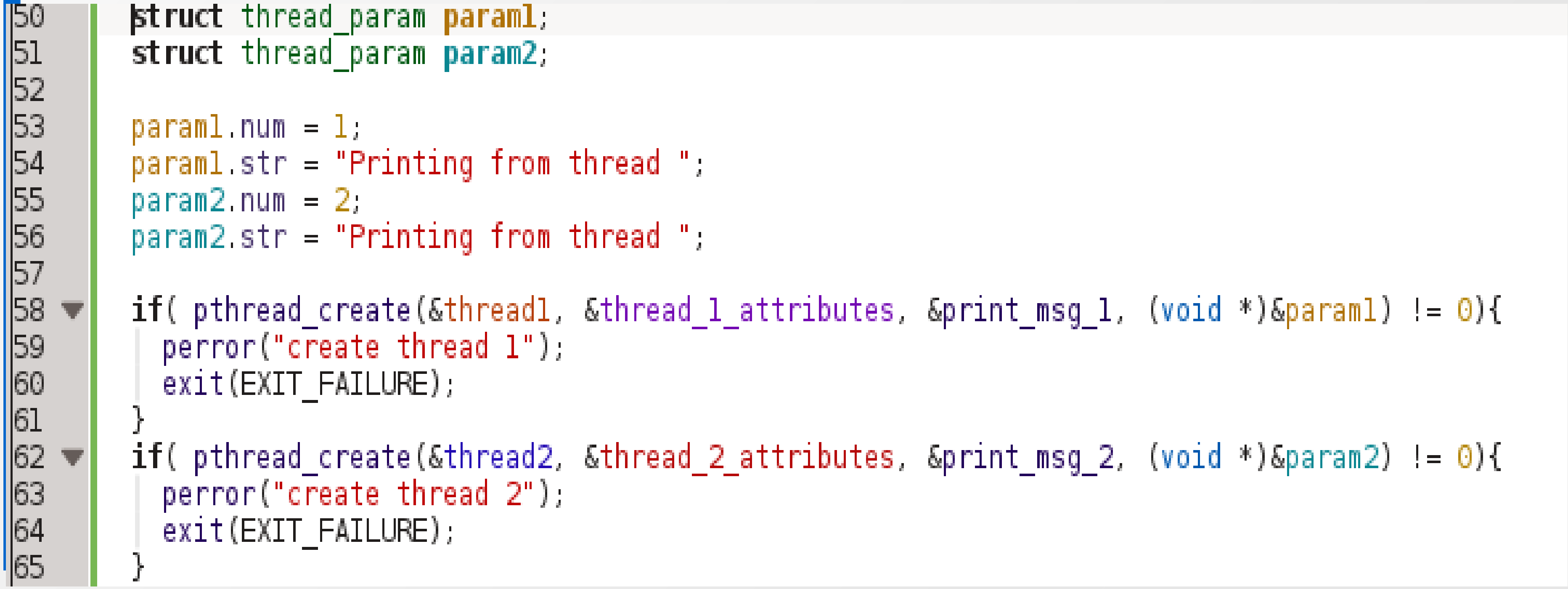
```c
struct thread_param {
  int num;
  char *str;
};

void *print_msg_1(void *param){
  struct thread_param *cast = (struct thread_param *)param;
  while(1){
    printf("%s %d\n", cast->str, cast->num);
    sleep(1);
  }
  pthread_exit(NULL);
}

void *print_msg_2(void *param){
  struct thread_param *cast = (struct thread_param *)param;
  while(1){
    printf("%s %d\n", cast->str, cast->num);
    sleep(2);
  }
  pthread_exit(NULL);
}
```

Passing parameters to threads with pthread_create()

- An make the necessary changes to the thread creators



```
struct thread_param param1;
struct thread_param param2;

param1.num = 1;
param1.str = "Printing from thread ";
param2.num = 2;
param2.str = "Printing from thread ";

if( pthread_create(&thread1, &thread_1_attributes, &print_msg_1, (void *)&param1) != 0){
    perror("create thread 1");
    exit(EXIT_FAILURE);
}
if( pthread_create(&thread2, &thread_2_attributes, &print_msg_2, (void *)&param2) != 0){
    perror("create thread 2");
    exit(EXIT_FAILURE);
}
```

- With this trick you can pass whatever parameter you want!

A multithreaded TCP server

- Remeber the simple TCP server of the previous Lab?
- Lets make him multithreaded!
- This means that our server will be able to handle multiple connections in parallel, as all modern servers do
- Can you imagine where and when the threads should be created?

A multithreaded TCP server

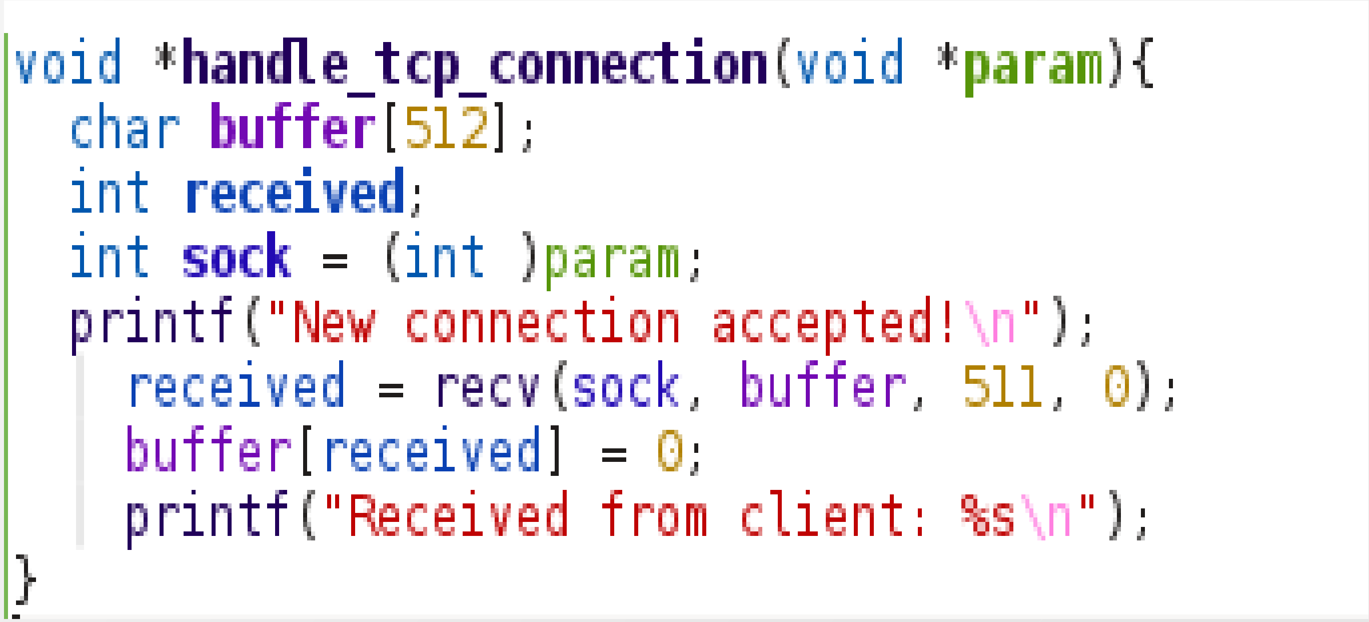
- Recall that *accept()* blocks until a new connection arrives and returns a new socket discriptor with the connected client

- Our goal is to create a new thread for every connection and pass the socket discriptor of this connection to the thread

- With this way the server is able to listen for new connections and we can serve all the established connections in parallel

A multithreaded TCP server

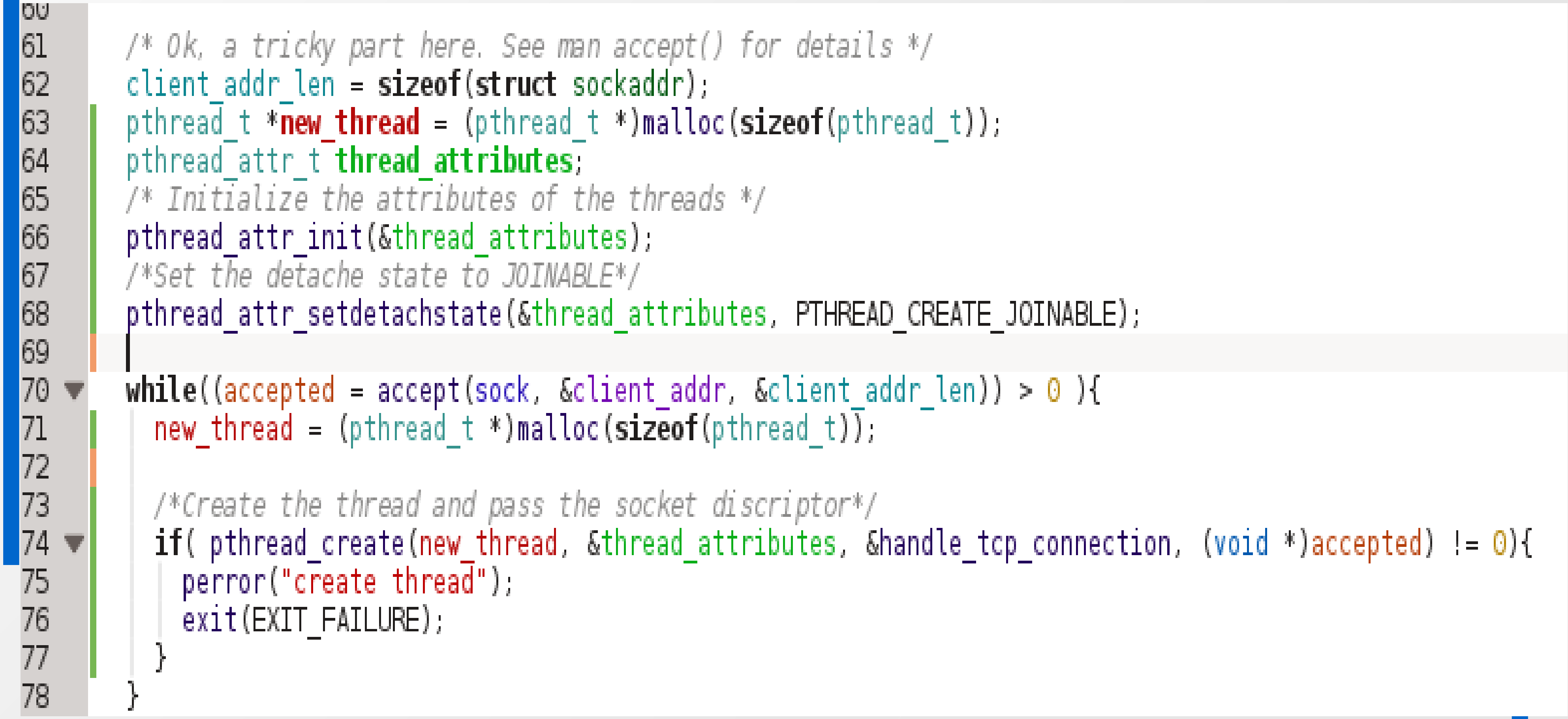
- We move all the code that hanldes the TCP connection with a client at a new thread that takes as parameter the socket discriptor



```
void *handle_tcp_connection(void *param){
  char buffer[512];
  int received;
  int sock = (int )param;
  printf("New connection accepted!\n");
    received = recv(sock, buffer, 511, 0);
    buffer[received] = 0;
    printf("Received from client: %s\n");
}
```

A multithreaded TCP server

- And after every accept() we create a thread...



```
/* Ok, a tricky part here. See man accept() for details */
client_addr_len = sizeof(struct sockaddr);
pthread_t *new_thread = (pthread_t *)malloc(sizeof(pthread_t));
pthread_attr_t thread_attributes;
/* Initialize the attributes of the threads */
pthread_attr_init(&thread_attributes);
/*Set the detache state to JOINABLE*/
pthread_attr_setdetachstate(&thread_attributes, PTHREAD_CREATE_JOINABLE);

while((accepted = accept(sock, &client_addr, &client_addr_len)) > 0 ){
  new_thread = (pthread_t *)malloc(sizeof(pthread_t));

  /*Create the thread and pass the socket discriptor*/
  if( pthread_create(new_thread, &thread_attributes, &handle_tcp_connection, (void *)accepted) != 0){
    perror("create thread");
    exit(EXIT_FAILURE);
  }
}
```

Avoiding race conditions

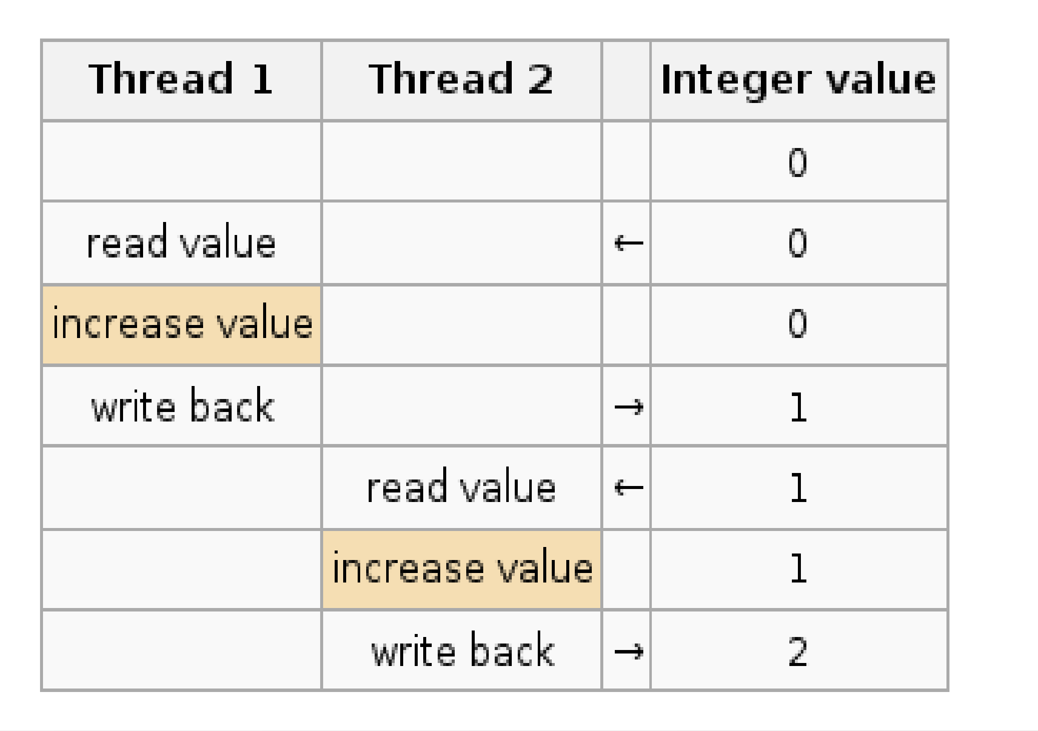
- The sheduler stops a thread and enables the execution of another in arbitary time slots

- This causes many problems in variables that are accesible by more than one threads

- Assume the following simple scenario
  - You have a variable i initialy 0 and two threads that increment the variable by one.

Avoiding race conditions

- The desirible result would be the following:

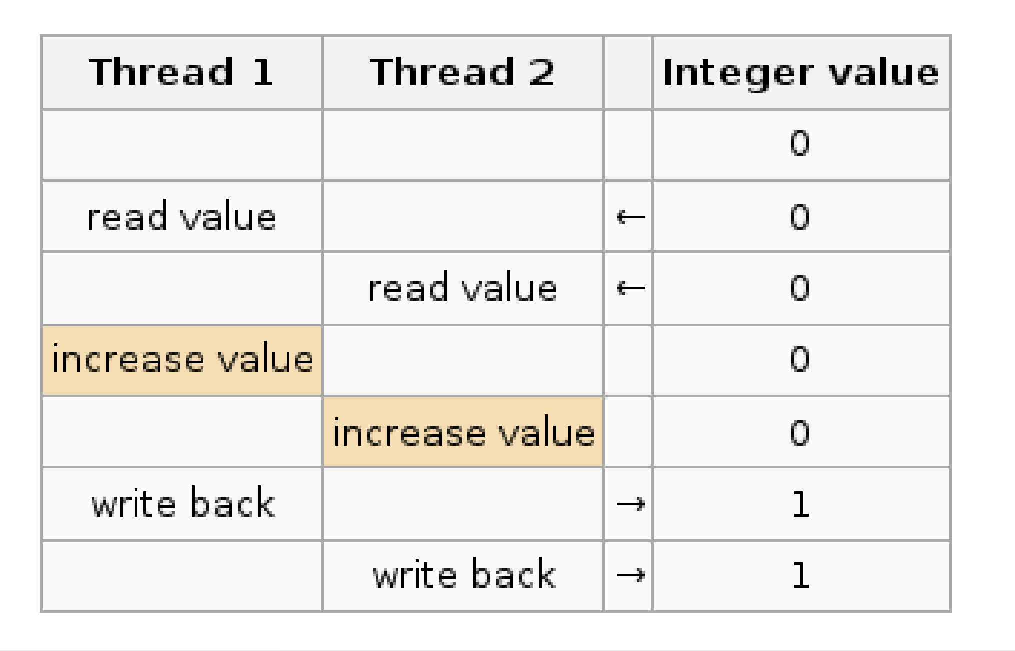


Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Avoiding race conditions

- But as we said before, the scheduler may stop a thread and enable another in an unpredictable way

- So there is a possibility that the following execution of the code happens



Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

- Which is not what we want...

Avoiding race conditions

- To avoid these situations we use locking
- The programmer has to lock, those variables that are accessed by more than a thread
- Before accessing the variable a lock() should be performed, in order to forbid other threads to access it
- After the variable access, the programmer should call unlock() to allow other threads to access it
- For locking pthread provides the pthread_mutex_t type and many pthread_mutex_* functions