Using Threads and Semaphores in C

Prof. Dr. Peter Braun

28. November 2024

1 Titlepage

Using Threads and Semaphores in C

Prof. Dr. Peter Braun

thws Technical University of Applied Sciences Würzburg-Schweinfurt

No Text – maybe some music :-)

2 Learning Goals

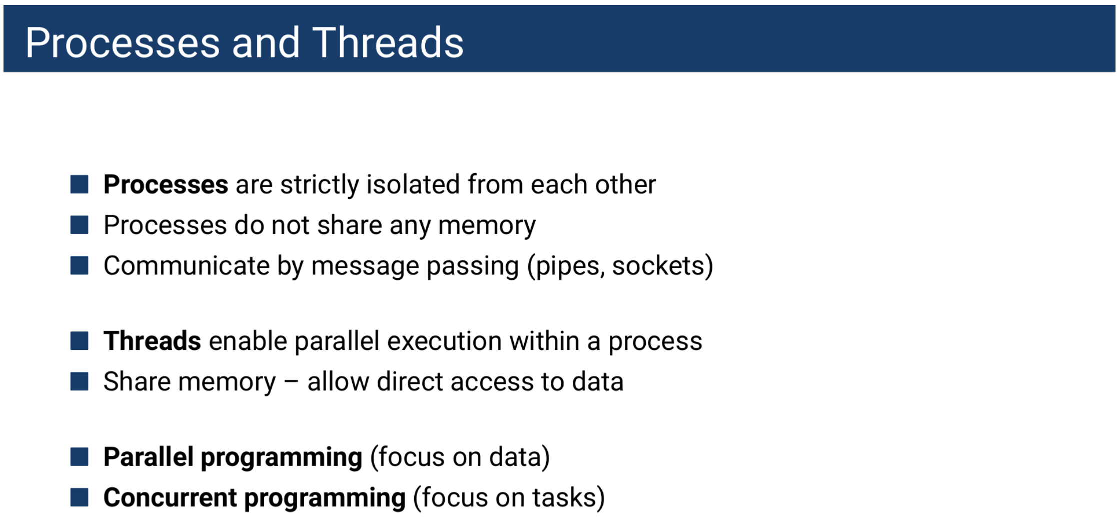
Learning Goals

- You know how to use threads in C
- You know about the Lost Update problem

- You can identify critical sections
- You learn how to use semaphores in C

Welcome to this unit, where we will study the concept of semaphores and their application in C programming. This unit is part of our series on C programming. We expect you already know about thread programming from Java or another programming language. While thread programming offers significant performance benefits by parallel programming, it also introduces challenges, particularly when accessing shared data. In this unit, we will begin by demonstrating thread programming in C and introduce the lost update problem, the most important example showing the challenges of thread programming. As a solution, we will discuss the concept of the critical section, a block of program instructions where shared data is accessed or modified. Our job as software developers is to identify these critical sections and manage them effectively. To address this, we will introduce semaphores, a synchronization tool that helps control access to critical sections. By using semaphores, we can ensure that only one thread accesses the critical section at a time, maintaining data integrity and preventing concurrency issues. After a theoretical introduction to semaphores and their mechanics, we will demonstrate their use with practical examples in C programming, giving you the tools to apply them in your projects.

3 Processes and Threads



Processes and Threads

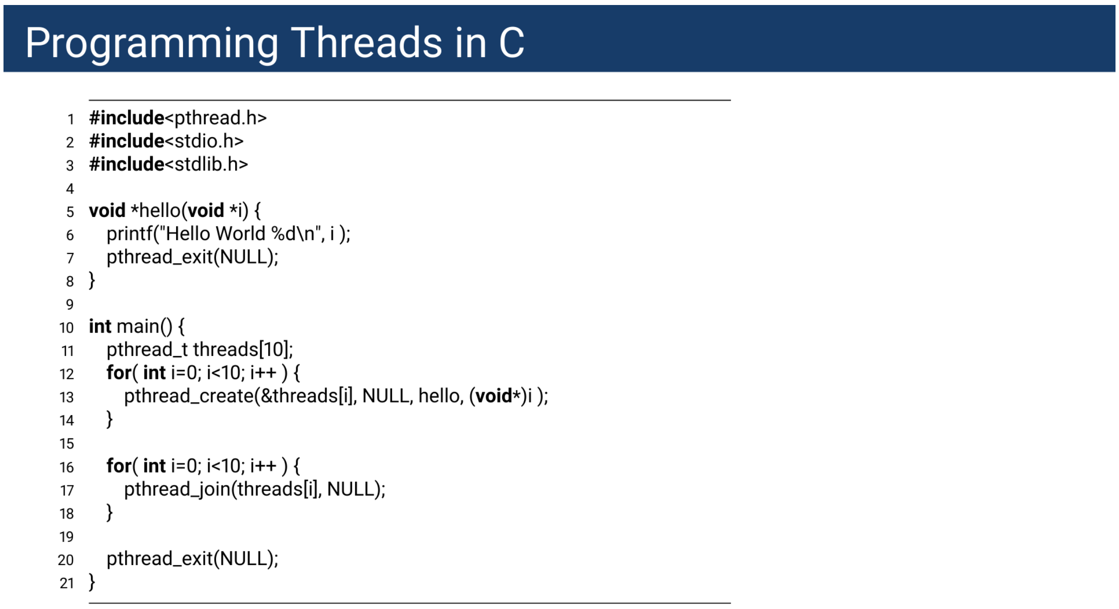
- **Processes** are strictly isolated from each other
- Processes do not share any memory
- Communicate by message passing (pipes, sockets)

- **Threads** enable parallel execution within a process
- Share memory – allow direct access to data

- **Parallel programming** (focus on data)
- **Concurrent programming** (focus on tasks)

Let's look at the differences between processes and threads and how they relate to parallel and concurrent programming. First, processes are completely isolated from one another. Each process has its own memory space and cannot directly access or modify the memory of another process. This isolation makes processes more secure and less prone to errors caused by other processes, but it also makes communication between them more complex. Processes typically communicate through message-passing mechanisms, such as pipes, sockets, or shared files, which can be slower and more resource-intensive than direct memory access. In contrast, threads exist within a single process and share the same memory space. This shared memory enables direct access to data, which makes communication between threads much faster and more efficient than between processes. However, it also introduces challenges, as multiple threads accessing the same data can lead to race conditions or data corruption if proper synchronization mechanisms, such as locks or semaphores, are not in place. A race condition is a situation in software where the result depends on the random execution order of commands. We will later study the lost-update problem as an example of a race condition. Let's distinguish between parallel and concurrent programming, which often involve processes or threads. Parallel programming focuses on splitting data across multiple threads or processes, allowing simultaneous computation to maximize CPU usage and speed up tasks. For example, dividing a large dataset for processing across multiple cores. On the other hand, concurrent programming deals with managing various tasks that may run in overlapping periods. The focus here is on handling tasks efficiently, even if they aren't running simultaneously, such as handling multiple user requests in a web server.

4 Programming Threads in C

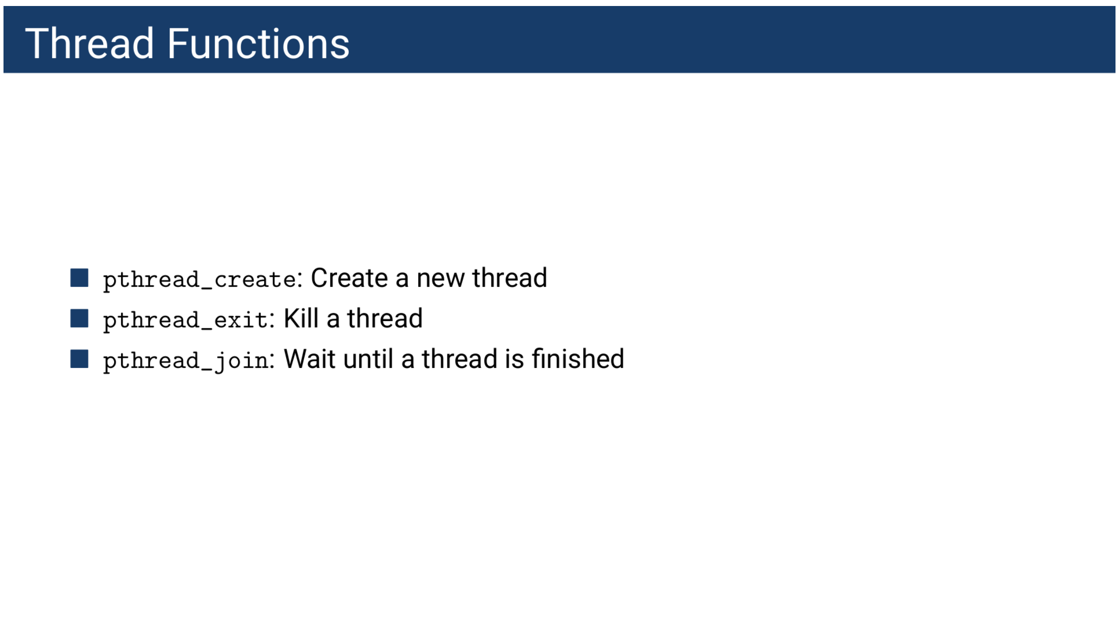


Programming Threads in C

```c
1  #include<pthread.h>
2  #include<stdio.h>
3  #include<stdlib.h>
4
5  void *hello(void *i) {
6    printf("Hello World %d\n", i );
7    pthread_exit(NULL);
8  }
9
10 int main() {
11   pthread_t threads[10];
12   for( int i=0; i<10; i++ ) {
13     pthread_create(&threads[i], NULL, hello, (void*)i );
14   }
15
16   for( int i=0; i<10; i++ ) {
17     pthread_join(threads[i], NULL);
18   }
19
20   pthread_exit(NULL);
21 }
```

Let’s go through this C code, which demonstrates using POSIX threads to create and manage multiple threads. We’ll break down the most important parts. The program starts by including the necessary headers: ‘pthread.h’ for working with threads. The ‘hello’ function is defined as the thread function. It accepts a single argument of type pointer to “void” and prints a message: ‘"Hello World"‘ followed by the value of ‘i.’ The datatype pointer to “void” is the most generic datatype in C, allowing all kinds of data to be passed. The function ends with ‘pthread_exit,’ ensuring the thread terminates cleanly. In the ‘main’ function, the program creates an array of ‘pthread_t’ in line 11 to hold the thread identifiers for up to 10 threads. The first ‘for’ loop iterates from 0 to 9, creating a new thread in each iteration using ‘pthread_create.’ This function takes four arguments: A pointer to the thread identifier. Thread attributes are set to ‘NULL’ when using the default settings. The third argument is the function to execute, which is ‘hello.’ The last argument is to be passed to the thread function. Here, you can see why this argument is a datatype pointer to void. This parameter must be passed to a standard C function where you cannot use specific datatypes. Each thread prints its message with the corresponding value of ‘i.’ When you run this code, you will notice that the output might not always be in the expected order due to the lack of synchronization. The second ‘for’ loop uses ‘pthread_join’ to wait for all threads to finish before the program exits. This ensures the main thread doesn’t terminate, leaving child threads incomplete. Finally, ‘pthread_exit’ ensures the main thread exits cleanly after completing all child threads. This is important in multi-threaded programs to allow all threads to finish properly.

5 Thread Functions

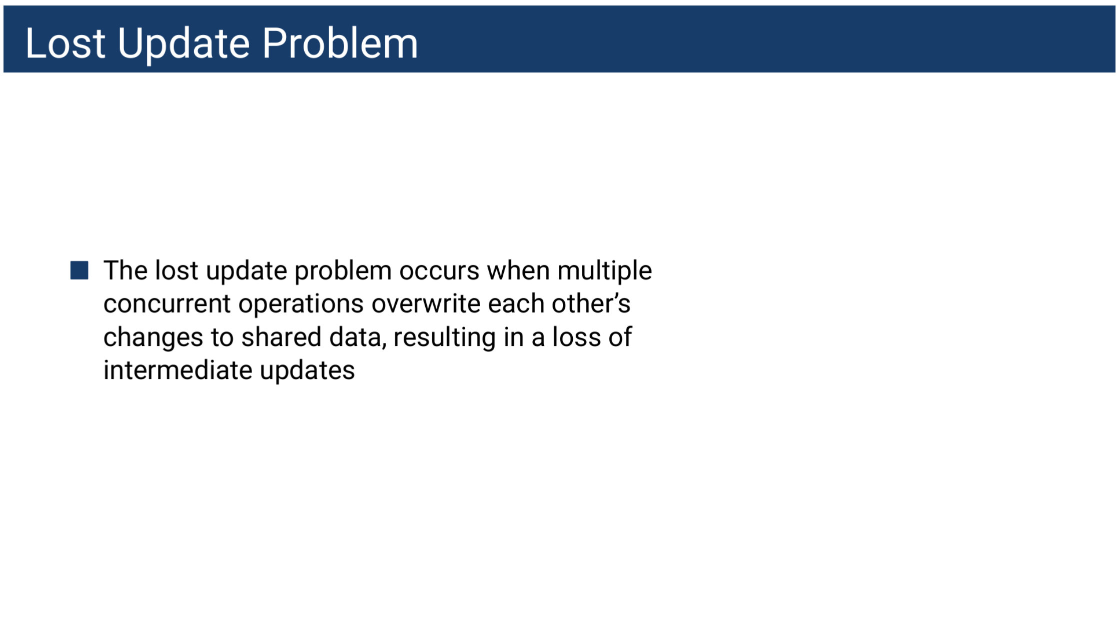


Thread Functions

- `pthread_create`: Create a new thread
- `pthread_exit`: Kill a thread
- `pthread_join`: Wait until a thread is finished

In this slide, we summarize the most important POSIX thread functions that are essential for creating and managing threads in C. The ‘pthread_create’ function is used to create a new thread. The ‘pthread_exit’ function is called when a thread finishes its execution. It allows the thread to return a value that the ‘pthread_join’ function can handle. This function ‘pthread_join’ ensures that the main thread or another thread waits for a specific thread to complete. This is crucial for synchronization, ensuring all threads finish before the program exits.

6 Lost Update Problem


Lost Update Problem

- The lost update problem occurs when multiple concurrent operations overwrite each other’s changes to shared data, resulting in a loss of intermediate updates

Now, we will study the lost update problem as one example of a race condition. The lost update problem arises in concurrent programming when multiple threads or processes simultaneously access and modify shared data without proper synchronization. Each thread reads the same initial value, performs its calculations, and writes back the result. However, the intermediate updates made by one thread are overwritten by another, leading to incorrect or inconsistent results. Race conditions are typically errors that only occur from time to time. They are very rare, which makes it hard for software developers to find them. Proper synchronization mechanisms, like mutexes or semaphores, are essential to prevent this issue and ensure data integrity in multi-threaded programs.

7 Example to Demonstrate the Lost Update Problem

Example to Demonstrate the Lost Update Problem

```
 1  int currentBalance = 300;
 2
 3  int getBalance(){
 4     return currentBalance;
 5  }
 6
 7  void deposit(int balance, int amount){
 8     int temp = balance + amount;
 9     currentBalance = temp;
10  }
11
12  void withdraw(int balance, int amount){
13     int temp = balance - amount;
14     currentBalance = temp;
15  }
```

Let's look at this code, which manages a simple bank account balance with functions to retrieve, deposit, and withdraw money. The 'getBalance' function returns the current balance stored in the global variable 'currentBalance.' The 'deposit' function takes the current balance and adds an amount to calculate a temporary value ('temp'), which is then stored back into the 'current balance'. Similarly, the 'withdraw' function subtracts an amount from the current balance and updates the 'current balance' with the result. We split one statement to update the variable 'currentBalance' in two lines, 8 to 9 and 13 to 14, only for demonstration. If we had written these statements in only one line, it would be more difficult to identify the problem.

8 Correct Execution Order

Correct Execution Order

Step	Thread A (deposit 100)	Thread B (withdraw 50)
1	`currentBalance` = 300	
2	`temp` = 300 + 100 = 400	
3	`currentBalance` = 400	
4		`currentBalance` = 400
5		`temp` = 400 - 50 = 350
6		`currentBalance` = 350

This table illustrates a correct execution order of two threads, Thread A performing a deposit and Thread B performing a withdrawal, on the shared variable ‘currentBalance’. Initially, ‘currentBalance’ is 300. Thread A starts, calculates ‘temp’ as 400 in step 2, and updates ‘currentBalance’ to 400 in step 3. Then, Thread B begins, reads the updated ‘currentBalance’ as 400, calculates ‘temp’ as 350 in line 5, and updates ‘currentBalance’ to 350. This sequential execution avoids race conditions, resulting in the correct final balance. However, such order is not guaranteed in multi-threaded environments without proper synchronization mechanisms.

9 Wrong Execution Order

Wrong Execution Order

Step	Thread A (deposit 100)	Thread B (withdraw 50)
1	`currentBalance` = 300	
2		`currentBalance` = 300
3	`temp` = 300 + 100 = 400	
4		`temp` = 300 - 50 = 250
5	`currentBalance` = 400	
6		`currentBalance` = 250

- The update to 400 is lost
- Only the update to 250 is visible

This table demonstrates a wrong execution order caused by concurrent access to the shared variable 'currentBalance' without synchronization. Initially, both Thread A and Thread B read the same value, 300. Thread A calculates 'temp' as 400 and updates 'currentBalance' to 400 in Step 5. However, Thread B independently calculates 'temp' as 250 based on the initial value, ignoring Thread A's update. In Step 6, Thread B overwrites 'currentBalance' with 250, effectively discarding the update made by Thread A. This is a classic example of the Lost Update Problem, where the final value only reflects one thread's operation (250), not the intended combination of both threads' actions. This race condition leads to data inconsistencies in multi-threaded programs without synchronization mechanisms, such as mutexes.

10 Terms and Keywords

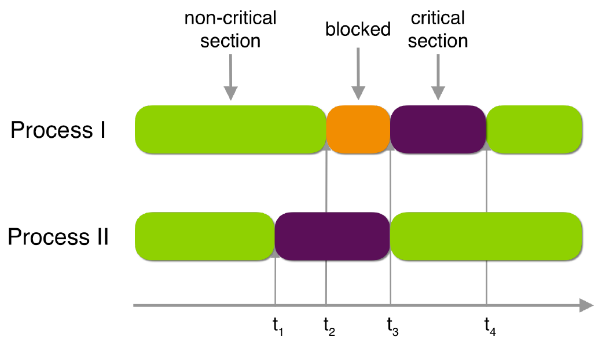
Terms and Keywords

- **Process Synchronisation**: Coordination of execution
- **Critical Section**: Section that uses shared resources
- **Mutual Exclusion**:
  - One process is allowed to enter a critical section
  - Other processes have to wait before entering

We continue by looking at solutions to this problem. Let's start by introducing some important terms. Under the term process synchronization, we summarize all types of measures that we, as developers, insert within a program to control the program's execution or the threads. Only we, as developers, have consciously decided to use parallel programming to solve a problem. Only we, as developers, can also choose at the level of the content or semantics of our program where issues will arise through parallel programming. We cannot hand over this task to the compiler, or at least not completely. It is, therefore, also our task as software developers to think about the synchronization of processes or threads. A critical section within our source code is a sequence of instructions that can cause problems if several processes or threads execute these instructions at the same time. Our synchronization measures relate to identifying critical regions and preventing simultaneous access at these points. One of the measures is called mutual exclusion. If we have identified a critical section, we use process synchronization to ensure that only one process enters this region at a time, and all other processes that also want to join this region must wait until the first process has left the section again. Mutual exclusion describes the solution in which only one process is allowed to work in the critical region at any time.

11 Critical Section

Critical Section



Let's first look at the concept of the critical section or critical region. I will use both terms from now on. When we talk about processes in the following, we also mean threads. All the problems and measures we are discussing today can be applied to both processes and threads. We will explicitly point this out again at the points where the difference is important. Please take a look at the picture next to me. You can see the time sequence of the two processes. Both processes must access a critical region during execution. In green, you can see the areas where instructions are executed outside the critical region. The purple color indicates the time ranges in which the process is in the critical region. The second process below reaches the critical region at time T one. As no other process has entered the critical region, the second process may start executing the instructions in the region. This lasts until time T three. The first process is ready to enter this critical region at time T two. However, as another process is already in the critical region, the first process must now wait. This is marked in orange in the image. During this time, the first process is blocked or waiting. At time T three, the second process leaves the critical region, and now the operating system automatically informs the first process that it can enter the critical region. Mutual exclusion means that only one process at a time works in the critical region.

12 Exercise 1: Find Critical Section

Exercise 1: Find Critical Section

```
1  int x = 5;
2
3  void foo(int s)
4  {
5    printf("Hello World\n");
6    int y = x;
7    y = y + s;
8    x = y;
9    printf("Good Bye\n");
10 }
```

Let’s look at some source code to understand better what a critical region means. The Foo function expects a parameter of type integer. Within the function, the text “hello world” is first displayed on the screen in line five. Three lines of instructions follow this. We first save the value of the global variable X in a local variable Y. Then we add the parameter s to the variable Y. Finally, we copy the value of Y into the global variable X. Another screen output follows in line nine. Initially, the global variable X is set to the value five. A function call with the value one will, therefore, set the variable X to the value six. Now, imagine that this function is executed not only by one process but two processes. For the sake of simplicity, however, let us assume that we are on a single-processor system. The execution of several parallel processes is only simulated by the operating system. In reality, only one instruction is executed at a time. Still, it could be that the operating system first executes instruction five from process one, for example, then passes control to process two, which then performs the instructions in lines five and six, then switches back to process one, which then executes instructions six and seven, and so on. Now, please take a few minutes and play through this case on paper or a whiteboard. What happens to the variables X and Y when two processes call this function simultaneously? To make it easier for you to distinguish the process, the first process should call the function with parameter one and the second process with parameter two. What do you notice now? Can problems occur here? You may already find the sequence of instructions that must be labeled as the critical region. When you are finished, or if you want to see the solution, please click the button.

13 Screencast

Screencast

- todo

We look at the function foo from our example here and we want to analyze this function with regards to critical regions or critical sections. So we check this function over time. And the first thread will start this function foo now and the parameter is 10. And now we execute or we copy here for the slide the statements. So y is set to x, y is then incremented by the value of s and then this global variable x is set to y. And the initial value of x should be 5. And now locally in this thread we have the variable y which is in the first step now set to value 5. And then we increment the value of y by the given parameter and the result is 15. And at the end we set the global variable x to this value of 15. So the second thread is called with parameter 20. And now we show one way to execute these two threads and let's say by chance in this example the second thread is executed after the first has been finished. So we start with y equals 15 now because this is the last value as the result of the first thread. Then we increase y to value 35 and finally we can set x to the correct value of 35. So this is only one way to execute this and now I would like to demonstrate where the problem comes from by choosing randomly another way to schedule these single statements. So of course in the second example the order is leading to the problems because I want to illustrate the problem here. And therefore we have to interleave these statements a little bit more compared to the first example. So first we had this assignment of y equals x and now in the second thread we execute this statement after the first thread has executed the same statement. So the value of this local variable y is set to the global variable x which is not 15 here but still 5. And we do not see this updated version of 15 already because this is set later by the first thread. And now there is a gap again this is randomly now. We increase the value to 25 and now we can see here the problem the expected result was 35 but because of this interleaving of assignments we have now the value of 25. So from this example again we can see the critical region are these three statements here. And the rule that you have to remind now is critical regions must not overlap. They must be executed in an atomic way but not overlap.

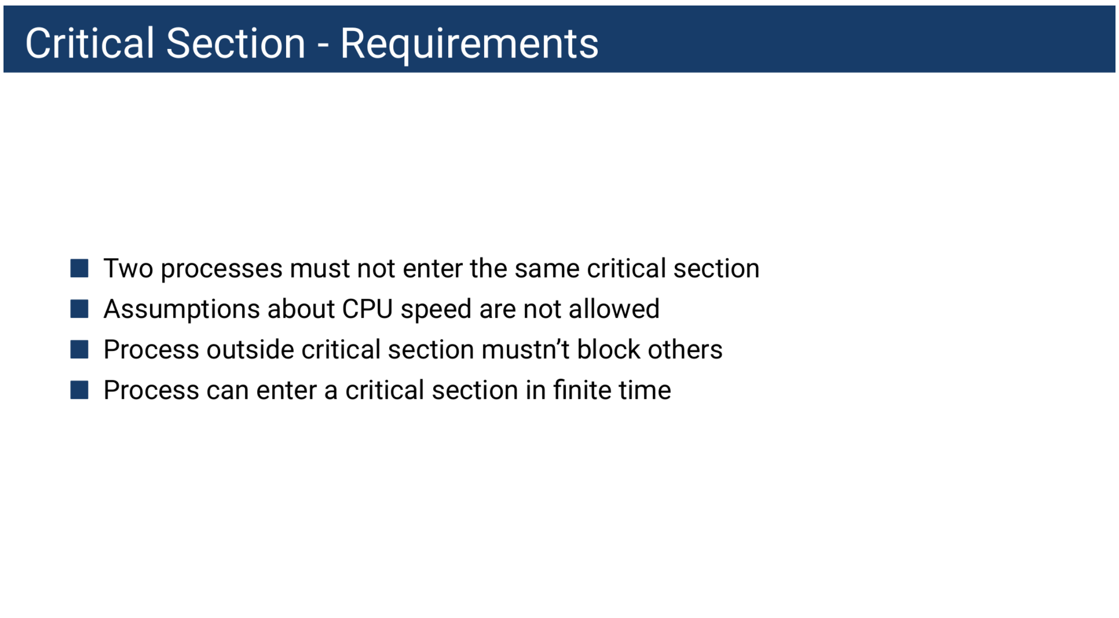
14 Solution: Critical Section

Solution: Critical Section

```
1  int x = 5;
2
3  void foo(int s)
4  {
5    printf("Hello World\n");
6
7    // Begin critical section
8    int y = x;
9    y = y + s;
10   x = y;
11   // End critical section
12
13   printf("Good Bye\n");
14 }
```

You have just seen how the simultaneous execution of two processes in our example can lead to a problem. The problem is the access to the variables X and Y. We marked the critical region in the source code with comments. It only includes these three instructions, now in the lines 8-10. When searching for critical regions, you always want to find the smallest region that leads to a solution. In this example, the two screen outputs are not part of the critical region because they do not access the data X or Y. Regarding program execution, including the two print statements within the critical region is not wrong. But it is unnecessary.

15 Critical Section - Requirements


Critical Section - Requirements

- Two processes must not enter the same critical section
- Assumptions about CPU speed are not allowed
- Process outside critical section mustn’t block others
- Process can enter a critical section in finite time

On the last slide, we marked the start and end of the critical region with comments. Now, we will look at the implementation of synchronization measures to control access to critical regions. Let’s first take a look at the requirements. Only one process may be authorized to access a critical region. Before the critical region, we must, therefore, insert an instruction instead of the first comment, allowing access to one process but denying access to other processes until the first process has left the critical region again. Accordingly, we must also replace the comment after the end of the critical region with an instruction that marks the end of the region. Processes that have to wait at the beginning of a critical region before they are granted access must not make any assumptions about the processor speed for this wait. For example, it is not permitted for such a waiting process to go to sleep for ten milliseconds, for example, and then enter the region without further testing. Processes waiting at the start of a critical region must not block any other processes. Finally, no process may be prevented from entering a critical region at any time. To put it another way, at some point, the time is reached when the last waiting process at the start of a critical region has the right to enter it. The operating system provides the control measures for dealing with critical regions. We will see in a moment that we only need to call a few functions in our program to control access to critical regions. Therefore, these four requirements must be addressed to the operating system. It is not our job as software developers to fulfill these requirements. We only use the functions provided by the operating system. Only if the operating system fulfills these four requirements can access to critical regions be allocated correctly and fairly.

16 Sleep and Wakeup as Mutual Exclusion Technique

Sleep and Wakeup as Mutual Exclusion Technique

- Processes wait to enter critical section passively
- When section can be entered, process is woken up
- The CPU will not be used while sleeping

As mentioned, implementing the functions for controlling access to critical regions is a task for the operating system. To realize this mutual exclusion technically, various solutions can be considered. We will not go into these different approaches in depth here, as this is part of another teaching unit that deals with scheduling processes. Perhaps just one sentence. Of course, it would not be a clever approach to implement the waiting of a process at the beginning of a critical region by an active wait, for example, within a loop. This would consume additional processor resources and would need to be more effective. Alternatively, you can imagine a possible solution in a simplified form as follows: Processes that have to wait at the start of a critical region are set to the WAITING state. This is the usual state that a process assumes when it waits for resources to be allocated by the operating system. When the operating system realizes that a process has left the critical region, it looks at a list of waiting processes. Then, it decides which process may enter the critical region according to certain criteria. Now, it checks again whether access to the critical region is possible.

17 Semaphore

Semaphore

- Data structure contains
  - A counter with initial value > 0
  - Operation `down` to decrement counter
  - Operation `up` to increment counter
  - A queue for waiting processes
- Operations are atomic and cannot be interrupted

Now, we come to the main topic of today's unit. We will introduce the function that we, as software developers, need to characterize critical regions. We use a data structure called semaphore, which consists of a counter and two operations. The name semaphore comes from the railway world and is a signal that indicates to trains whether they are allowed to use a certain section of the track or not. Using semaphores in computer science refers to the Dutch computer scientist Edsger Dijkstra, who introduced this concept in 1965. We will now explain how this data structure works. A semaphore contains a counter initialized with a number greater than zero. The counter describes the number of processes that may access a resource simultaneously. The down operation reduces the value of this counter by one. This function is executed when a critical region is entered. However, if the counter's value is already zero, the down function must put the calling process to sleep. To do this, it manages a queue of waiting processes. The UP function increases the value of the counter by one. This corresponds to leaving a critical region. This function also looks at the queue of waiting processes and activates the first process, i.e., the one waiting the longest. For a semaphore to function correctly, the two operations "down" and "up" must be implemented atomically. This means executing these two functions by a process must not be interrupted. These two functions also practically contain critical regions again because a common datum, the counter, is used. So, we are looking for a solution for managing critical areas, and we have another problem with critical regions of our solution. That sounds strange at first. But we have to distinguish between the different levels of abstraction here. The down and UP functions are implemented deep in the operating system. The operating system, in turn, has other mechanisms for managing critical regions.

18 Meaning of the Counter of a Semaphore

Meaning of the Counter of a Semaphore

- Number of processes to enter simultaneously
- Number of free slots in a data structure

A few more sentences about the meaning of this counter: according to our motivation for using semaphores, this counter means the number of processes allowed to enter a critical region simultaneously. In the vast majority of cases, the initial value of the counter will be one because we only want to let one process enter the critical region. We will go into this case in more detail in a moment. An initial value of the counter greater than one can be useful, for example, if we want to manage free spaces in a data structure. Imagine that we are working on a data field with ten elements. Several processes access this data field in parallel and use one of these ten elements. As soon as ten processes have each occupied a field simultaneously, another process must wait until it can use this data field. This means that another process must first release a field again. Instead of a data field, you can also imagine any other limited resources in a computer. The possible applications here are very diverse. Sometimes, the metaphor of a car park is used here. The car parks are the restricted resources, and the cars are the processes. The down and up functions are the barriers at the entrance and exit.

19 Initial Value of Counter == 1

Initial Value of Counter == 1

- Only one process can enter the critical section
- Mutual Exclusion between processes
- Also named binary semaphore, mutex or **lock**

We would now like to look at the special case where the counter is initialized with the value one. This case is common when we want to realize mutual exclusion between processes. A critical region, i.e., a sequence of instructions, is only guaranteed to be run through a single process. As this is a very common use case, other names have become established in software development for the case of an initial value of one. For example, such a semaphore is called binary because it either permits or prohibits access. As this involves mutual exclusion, the abbreviation MUTEX has also become widespread. And even simpler, this type of semaphore can also be called a lock.

20 Semaphore in C

Semaphore in C

```c
 1 #include<semaphore.h>
 2
 3 sem_t sem;
 4 int balance = 300;
 5
 6 int main() {
 7   sem_init(&sem, 0, 1);
 8   //...
 9   sem_destroy(&sem);
10 }
11
12 void deposit(int amount) {
13   sem_wait(&sem);
14
15   int temp = balance + amount;
16   balance = temp;
17
18   sem_post(&sem);
19 }
```

Let us now turn to the concrete functions of how the concept of semaphores is implemented in C under Linux. The corresponding file must be included in line one. A data structure named sem_T is defined there. You now need four functions to use semaphores. First, you must initialize the semaphore. This is done using the sem_init function in line seven. You enter the address of the data structure as the first parameter. With the second parameter, you specify whether you want to synchronize several processes or threads. The value 0 stands for the case of threads. The third parameter defines the initial value of the counter. As we are using the value one here, we are using a binary semaphore. The comment in line eight indicates that functions containing critical regions should now be called. You can see such a function from line twelve. The function deposit is used to realize a deposit of money into a bank account. You can see the current value in the account in a very simplified form in line four in the global variable balance. At the beginning of the function, the sem_wait function is called in line 13. A pointer to the data structure of the semaphore is passed as a parameter. This function corresponds to the operation down. If no other process has already entered the critical region, execution is then continued in line 15 within the critical region. Otherwise, the current process is set to the waiting state in line 13. After the critical region, the sem_Post function is called. A pointer to the data structure is also passed as a parameter. This corresponds to the UP operation. That's almost everything. How exactly these two functions for WAIT and Post are implemented is of no interest to us now. We can no longer control how the operating system selects any waiting processes. The SEM_DESTROY function in line nine in the main function is also important. This function deletes the semaphore to which we pass a pointer here. This variable SEM can then no longer be used.

21 How Long Do Processes Wait?

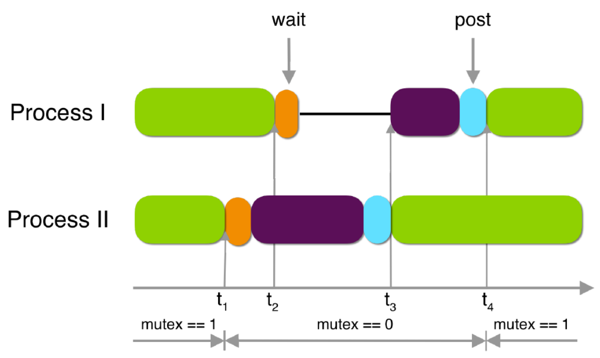
How Long Do Processes Wait?

```
 1  #include<semaphore.h>
 2  #include<time.h>
 3
 4  sem_t sem;
 5  struct timespec t;
 6  int balance = 300;
 7
 8  int main() {
 9    sem_init(&sem, 0, 1);
10    // set t to some absolute time in future
11  }
12
13  void deposit(int amount) {
14    int sw = sem_timedwait(&sem, &t);
15    // handle values of sw
16
17    int temp = balance + amount;
18    balance = temp;
19
20    sem_post(&sem);
21  }
```

Our list of requirements for implementing semaphores in an operating system includes the point that a process must not wait indefinitely for access to the critical region. At some point, every process has its turn. Nevertheless, the waiting time at the sem_WAIT function can be very long. Depending on the use case, it may even be too long. And from the process's point of view, it could make sense to limit this waiting time. Access to a resource or access to a critical region may no longer be of interest to a process after a certain waiting time, and it also has an alternative solution for this case. Therefore, a variant of this function can be given a maximum waiting time as a parameter. You can see a simplified example in the source code beside me. In line 14, the function sem_TIMEDWAIT is called. The second parameter is a pointer to a data structure named Timespec. In this data structure, the absolute time must be specified in seconds from the start date of 1 January 1970 up to the maximum wait time allowed. You do not see the calculation of this absolute time next to me. Please have a look at the examples in the literature. You must evaluate the return value of this function because you have to distinguish between two cases. Either, the waiting process gained regular access to the critical region before the waiting time expired. Or the function in line 14 was terminated because the wait time had expired. In this case, you are, of course, not allowed to continue with the execution of further instruction within the region.

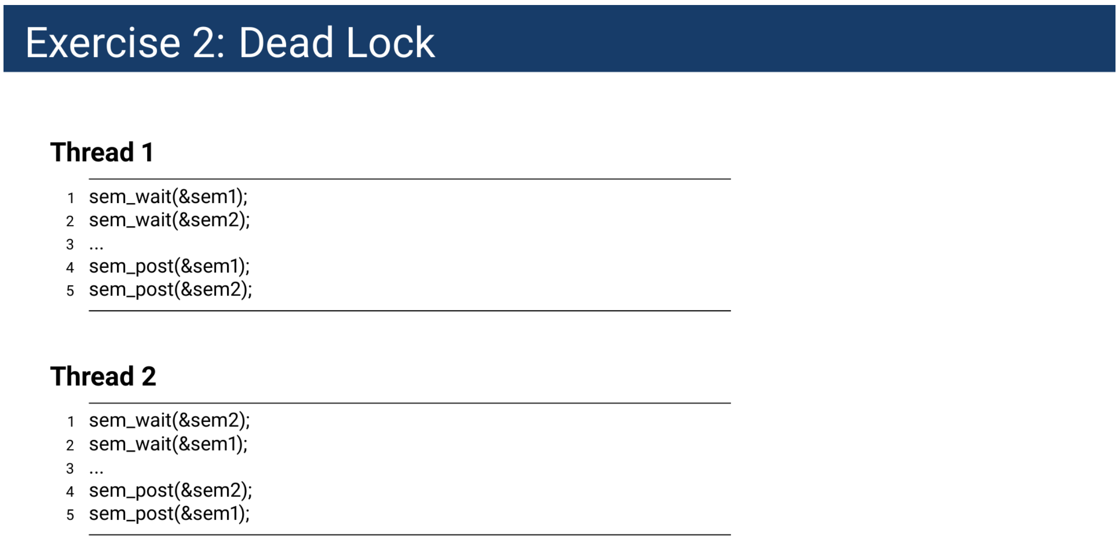
22 Binary Semaphore

Binary Semaphore



The following image illustrates the process of using a binary semaphore. Two processes compete for access to a critical region. A binary semaphore protects this critical region. You can see the current value of the counter below, labeled with the name MUTEX. At the beginning, this counter has the value one. Process two enters the critical region at time T one. The short waiting time to check whether another process has already entered the critical region is now marked in orange. As this is not the case, the value of MUTEX is now set to 0, and process two enters the critical region. As in the first image, the time within the critical region is marked in purple. At time T two, process one reaches the critical region and must first wait within the input check. This process is still active for the time of the check, also marked in orange here, but then switches to the waiting state. This waiting time is marked with a horizontal black line. Process two then reaches the end of the critical region at some point and informs the semaphore. This is marked in blue in the graphic. However, the counter's value remains zero because the semaphore memorized the waiting process one. This process is now informed and is allowed to enter the critical region. As soon as process one has reached the end of the critical region, the counter is now set to one.

23 Exercise 2: Dead Lock



Exercise 2: Dead Lock

Thread 1

```
1 sem_wait(&sem1);
2 sem_wait(&sem2);
3 ...
4 sem_post(&sem1);
5 sem_post(&sem2);
```

Thread 2

```
1 sem_wait(&sem2);
2 sem_wait(&sem1);
3 ...
4 sem_post(&sem2);
5 sem_post(&sem1);
```

In the simplest case, we use semaphores to control multiple processes' access to a critical region. So far, all the examples we have seen use this type of process synchronization. However, as soon as we use two or more semaphores in a program, a very serious problem can arise. The name for this problem is deadlock. Two processes are in a deadlock while waiting for each other to release resources. Neither process can continue working in this situation. The program comes to a standstill. We will look at the theory of deadlocks in another unit. Today, we will look at a relatively simple rule for recognizing or preventing the possibility of deadlocks in the source code. For this exercise, please look at the example next to me. We assume that two processes, or in this case, two threads, are working simultaneously with two semaphores. You can recognise the different semaphores by the numbers in the variable name. The dots in line three should represent the critical regions in both cases. Looking closely, you will recognize that the two semaphores are assigned in different sequences. Now think why this way of using semaphores could lead to a deadlock. You need to think about the order in which the respective instructions of the two threads could be executed. As a reminder, we, as software developers, do not influence how the operating system executes the threads. A deadlock can occur with this type of programming, but a deadlock does not always have to happen. If you have a solution or would like to have a look at our solution, please click on the button.

24 Screencast

Screencast

- Live am Rechner zeigen

Let's look at the execution order here. We start with the first thread. And what we have here are these statements to start a critical region by waiting for a semaphore to be free for us. And the order for the first thread is that we first wait for the first semaphore, then we wait for the second semaphore, and when we have both semaphores we can start the critical region, so to speak. The second thread does the same. It waits for the two semaphores but in a different order. So we first wait for the second semaphore, and then we wait for the first semaphore. And now what happens when we are in the second thread waiting for this second semaphore? This blocks because we assume that the first thread has already gained the second semaphore. And so in this case it would have been no problem. But now we write down just a different execution order to illustrate where the deadlock comes from. So if by any chance the processor or the operating system schedules the two threads in a different way, so that we first wait for the first semaphore in thread one, and now the second thread is scheduled, and the first statement there is to wait for the second semaphore. Since the second semaphore is free, execution goes further to the next statement, which is now wait for the first semaphore, and now here the second thread blocks because thread one has already gained the first semaphore. And thread one is now waiting for the second semaphore, but this already belongs to the second thread. So this is the deadlock situation now. Thread one and thread two are waiting, so to speak, for each other, and cannot go further, and there's no chance for them to resolve this deadlock situation. Once you are in a deadlock, there's no chance to get out of the deadlock, rather than killing, let's say, two threads. So what can we do as software developers to solve this solution? So it is our task to take care of the order in which the threads ask for the semaphore, or we wait for the semaphore, and to take this as a rule of thumb, it is absolutely necessary that in all threads, when you use more than one semaphore, the order in which you wait for the semaphores is the same. So when the first thread waits for the, or gets the first semaphore, then at this statement the second thread would wait, but it would not block the first thread from continuing to wait for the second semaphore. So by this, let's say, condition that you always ask for the semaphores in the same order, it is guaranteed that there is no deadlock.

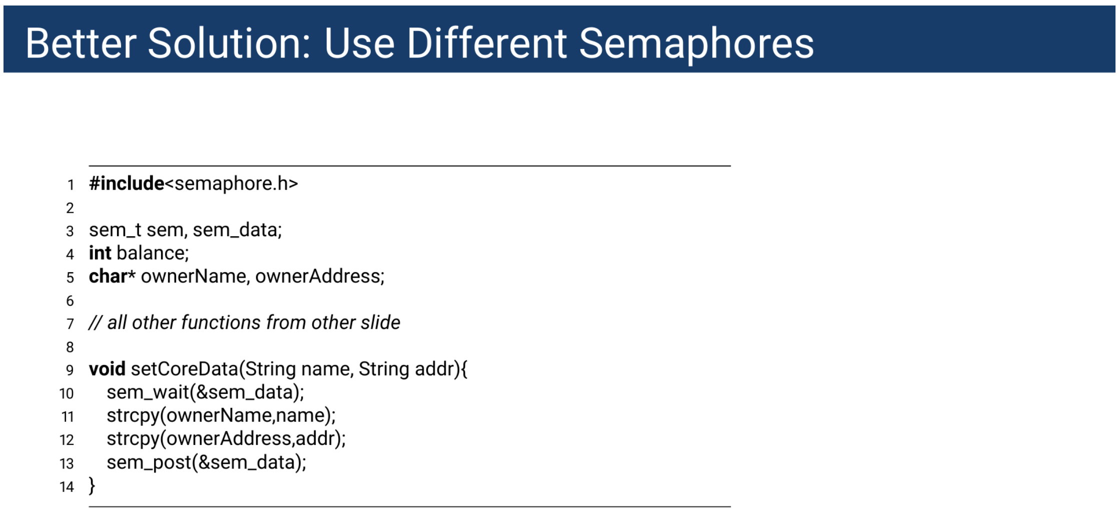
25 Using the Same Semaphore for Different Use-Cases

Using the Same Semaphores for Different Use-Cases

```
 1  #include<semaphore.h>
 2
 3  sem_t sem;
 4  int balance;
 5  char* ownerName, ownerAddress;
 6
 7  // all other functions from last slide
 8
 9  void setCoreData(String name, String addr){
10    sem_wait(&sem);
11    strcpy(ownerName,name);
12    strcpy(ownerAddress,addr);
13    sem_post(&sem);
14  }
```

Now, I briefly point out with an example that one semaphore should only protect one kind of critical region. In the source code beside me, we extend the source code from the last slides with another function now in line nine, named "setCoreData." Please remember the previous slides' functions for handling the bank account. In this new function from line nine onwards, there is again a critical region which, however, does not work with the credit balance but with the name of the owner of the bank account. Nevertheless, we use the same semaphore in lines 10 and 13 as in the other two functions. In terms of program execution, reusing the same semaphore is correct. You can do it that way. However, these are two critical regions that are better protected by different semaphores. In the current implementation, a process would be prevented from changing the bank account owner data because another process is working on the credit value.

26 Better Solution: Use Different Semaphores

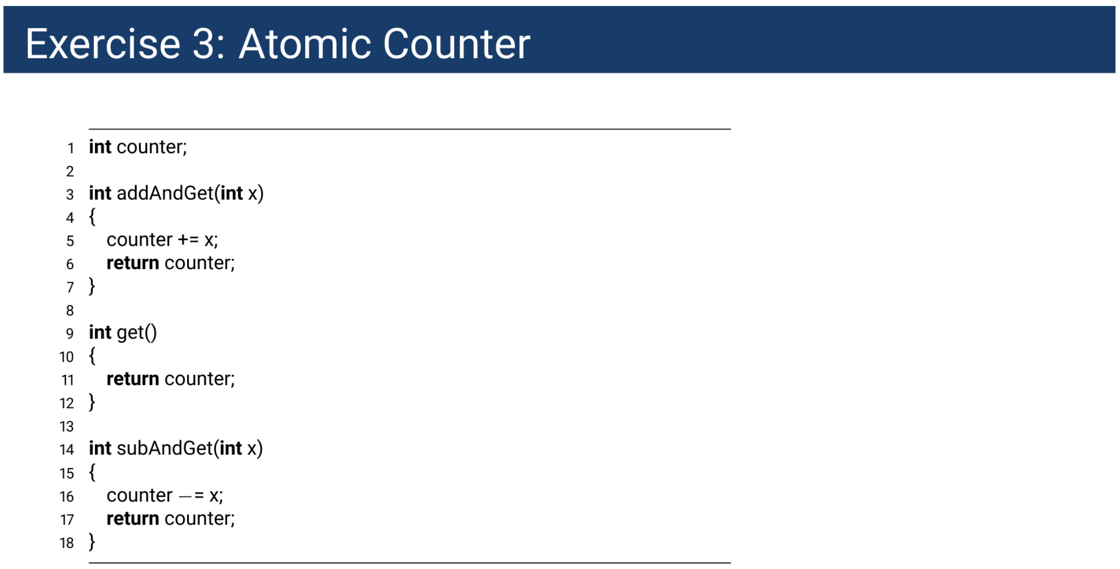


Better Solution: Use Different Semaphores

```
1  #include<semaphore.h>
2
3  sem_t sem, sem_data;
4  int balance;
5  char* ownerName, ownerAddress;
6
7  // all other functions from other slide
8
9  void setCoreData(String name, String addr){
10    sem_wait(&sem_data);
11    strcpy(ownerName,name);
12    strcpy(ownerAddress,addr);
13    sem_post(&sem_data);
14 }
```

The solution is, of course, very simple. We need two semaphores. You can now see in line three that we are creating two variables of the data type sem_T. We use the second semaphore in the added function to change the name. When implementing the function, please always ensure that you only protect those critical regions with the same semaphore that belong together in terms of content. This means that the same data must be changed. If completely different data is changed, you should also use different semaphores.

27 Exercise 3: Atomic Counter



Exercise 3: Atomic Counter

```
int counter;

int addAndGet(int x)
{
  counter += x;
  return counter;
}

int get()
{
  return counter;
}

int subAndGet(int x)
{
  counter -= x;
  return counter;
}
```

At the end of today’s unit, here is a short exercise to give you some practical experience in programming with semaphores. After everything you have learned today, this should be easy. You can see a global variable with the name counter in the example next to me. Three functions follow this. First, a function to increment the counter and return the new value; at the bottom, a function to decrement the counter and return it. In the center is another function that only returns the current value. Please identify the critical regions first. Then, please implement a solution on your computer using semaphores to protect the critical regions. When you are finished and would like to view the solution, please click on the button.

28 Screencast

Screencast

- todo

So we have prepared the template for the solution already and we restrict here ourselves to the function at and get. And after we have introduced the semaphores you can extend this to the other functions. We first need to include the semaphore file here, then we need a semaphore to protect the critical region. And we are going to initialize the semaphore in the main function. The first parameter is the address of the mutex, then the second parameter is set to zero because we are working with threads here. And the last parameter is the start value of the counter and this is one because we are working here with a mutex. Before we stop the program we destroy the semaphore just to clean up. And now we have to add the two statements to mark the critical region. So for the first statement when we wait for the semaphore this is quite easy. This must be the first statement here. But where do we put the release statement? So we could do this as the last statement here, but obviously this does not make sense because the function would terminate when the return statement is called. So the function called semphost is never reached. So this does not make sense here. So what we can do is we save the current value of the global variable counter in a local variable R. And by this, using the local variable, this is guaranteed not to be influenced by any other threads that also call this function because all threads have their own local stack, so to speak, of the local variables. And after we save this value of counter in R we can release the mutex and then we have to return the value R here. So now we have the calculation of the new value of counter in the critical region. But the return statement is of course outside the critical region, but this is safe as long as of course no other function uses this global variable counter without using our functions in the critical regions and the mutexes. So what I want to show you last is how to compile this program. Therefore we leave the editor now. And we call the compiler. And we have to add two flags here. With the first flag we inform the GCC compiler that we want to use pthreads. And with the second flag we inform the compiler to, we define a library where these pthread functions are implemented. And then we can call this function, but obviously the program is now too simple. We cannot see if it works. We had to add more threads and let the program run for a very long time to get a chance to see that there might be a problem.

29 Summary

Summary

- You are able to identify critical sections
- You learned about mutexes and semaphores
- You learned how to use semaphores in C

This is the end of today's unit. Using processes or threads for parallel programming offers extreme speed advantages under certain conditions. But even if several users can use your program simultaneously, you will come into contact with these mechanisms. It is then very important that you can identify the critical cases in which several processes access data simultaneously and have possible solutions to hand. Today, we first introduced the concept of the critical region, which we use to describe sections in our program that need to be protected against simultaneous access by multiple processes. We then took a close look at the semaphore data structure. You have seen how semaphores can be used to protect critical regions, and we have shown you examples of this in the C programming language. But, using semaphores is not entirely trivial because you have to make sure that you use the semaphores correctly so as not to run the risk of a deadlock.

30 Literature

Literature

- Sri Manikanta Palakollu: Practical System Programming with C. Apress, 2021.