#uni/semester3/Betriebssysteme/chapter26/chapter-notes

➜ the new abstraction for a single running process is that of a thread. A program doesn't have a single point of execution (a single PC where instructions are being fetched and executed from) anymore. A multi-threaded program has more than one point of execution (multiple PCs).
➜ thread is like a separate process, but threads share the same address space and thus can access the same data.
➜ state of a single thread has a program counter. Each thread has its own private registers it uses for computation. When two threads on a single processor and when switching from running one to running the other, a context switch must take place. The register state from the first thread must be saved and the one from second register must be restored. To store the state of each thread of a process we will need **one or more** thread control blocks (TCBs)
➜ During such a context switch the address space remains the same. There is no need to switch the page table.
➜ For a single threaded process there is only one stack. In a multi-threaded process each thread runs independently and may call various routines. Thus no more one stack per address space but one stack per thread.

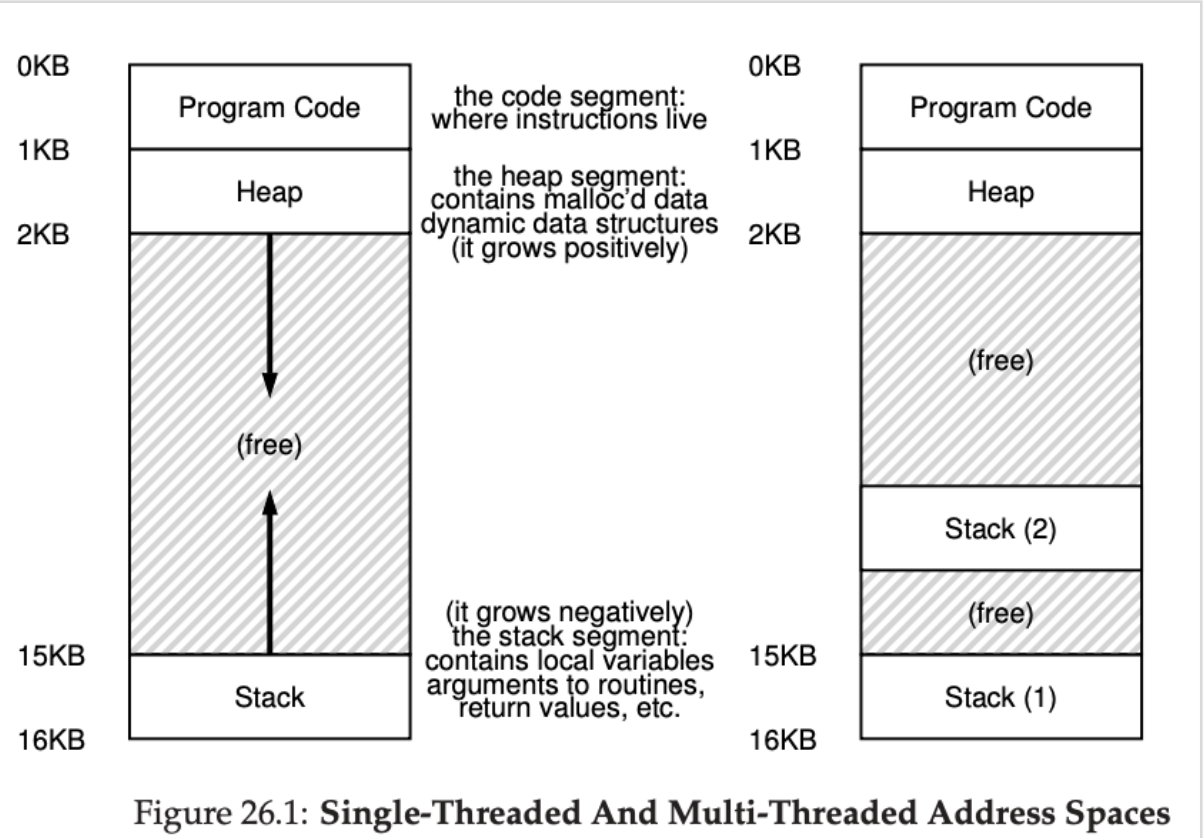

Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

➜ any stack allocated variables, parameters, return values will be put on **thread-local**

storage, namely on the stack of that thread.

➜ One problem is that now we have an ugly address space layout, with many stacks. Fortunately, this is ok because stacks don't need to be very large, with the exception being programs that use recursion intensely.

1. Why Use Threads?

   ➜ First reason is **parallelism**. Image a writing a program that performs operation on very large arrays. If running on a single processor, the you just perform each operation and be done. If running on a system with multiple processors, you can speed up this process considerably by using the processors to perform a different portion of work.

   ➜ **Parallelization** is the task of transforming a **single threaded** program into one that does this sort of work on multiple CPUs. Using a thread per CPU is a natural and typical way to make programs run fast.

   ➜ Second reason is to avoid blocking program progress due to slow I/O. Lets say you have a program waiting for I\O. Instead of waiting the program could do something else, like using CPU to perform computation or issuing further I\O requests.

   ➜ CPU scheduler can switch to other threads which can run and so something useful, thus avoiding getting stuck.

   ➜ Threading enables **overlap** of I\O with other actions within a single program. Multiprogramming did it for processes across programs. (Web servers, DBS make use of threads)

   ➜ You could also use multiple processes instead of threads. But threads share the same address space, so it is easy to share data. Processes are more useful for logically separate tasks, where little data sharing happens.

2. An Example: Thread Creation

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

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

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

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

➔ main program creates two threads; each will run the function mythread with a different string parameter. Once a thread is created, it may start running or put into a ready but not running state. On a multiprocessor both threads could run at the same time.

➔ After creating T1 and T2 phtread_join() waits for a particular thread to finish. By calling it twice we ensure that T1 and T2 both complete before allowing the main thread to run again. In total three threads were employed: the main thread, T1 and T2.

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

Figure 26.3: **Thread Trace (1)**

➜ In this diagram the time increases in the downwards direction. Each column shows when a different thread is running.

➜ This ordering is not deterministic, but depends on the scheduler and which thread it decides to run. Once a thread is created, it may run immediately:

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Figure 26.4: **Thread Trace (2)**

➜ There is no reason to assume that a thread that is created first will run first:

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Figure 26.5: **Thread Trace (3)**

➜ thread creation is like making a function call, but instead of first executing the function and then returning back to the caller, the system creates a new thread of execution for the routine that is called, and it runs independently of the caller. Maybe before returning from the creation or maybe later.
➜ What tuns next is determined bu the OS scheduler. It is hard to know what will run at a given moment in time. Thus threads make life complicated, because you don't know what will run when.

3. Why It Gets Worse: Shared Data
   ➜ Two threads update a global shared variable:

```c
#include <stdio.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

static volatile int counter = 0;

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

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

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

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

➜ the thread creation and join routines are wrapped to simply exit on failure. Difference between Pthread_create() and pthread_create() is that the former calls pthread_create() and makes sure the return code is 0. If it isn't Pthread_create() just prints a message and exits.

➜ 1e7 times in a loop, thus the desired final result is 20 000 000!

```
prompt> gcc -o main main.c -Wall -pthread; ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

➜ Sometimes we get the desired result, sometimes not. The results aren't deterministic!

➜ disassembler is a tool used to understand low level code in a c program. On linux run 'objdump -d main' (compile program with -g)

4. The Heart Of The Problem: Uncontrolled Scheduling

➜ Code sequence in x86 of the update to counter:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

➜ variable counter is located at address 0x8049a1c and has a start value of 50. Thread 1 runs first.

➜ First mov loads the value of counter in eax. Then adds 1 to max resulting to 51. Timer interrupt goes off, OS saves the currently running thread (its PC, its registers including eax) to the thread's TCB.

➜ Next Thread 2 is chosen to be run. Each thread has its own private registers, these are virtualised by the context switch code that saves and restores them. Thread 2 also executes the instructions, but it also saves the incremented value back. The global variable counter has now the value 51

➜ context switch happens again, thread 1 runs, and continues where it left off. It saves the value 51 to the global variable counter.

➜ 51 was saved twice, but the correct counter should have been 52.

➜ Here again the same, but in a diagram:

OS	Thread 1	Thread 2	(after instruction) PC	eax	counter
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

Figure 26.7: **The Problem: Up Close and Personal**

```
100 mov    0x8049a1c, %eax
105 add    $0x1, %eax
108 mov    %eax, 0x8049a1c
```

➜ This is a **race condition** (or **data race**) meaning the results depend on the timing execution of the code. Instead of getting a **deterministic** computation we get an **indeterminate** result, where is not known what the output will be.

➜ Because many threads executing this code can result in a race condition, we call this code a **critical section**. A critical section is a piece of code that accesses a shared variable (or shared resource) and must not be executed concurrently by more than one thread.

➜ We want for this code **mutual exclusion** which is a property that ensures if one thread is executing within critical section, the others will be prevented from doing so.

5. The Wish For Atomicity

➜ One solution is to have a more powerful instruction that in a single step does what we wanted and removed the possibility of an untimely interrupt.

```
memory-add 0x8049a1c, $0x1
```

➜ Assume this instruction adds a value to a memory location and the hardware ensures that it executes atomically. It could not be interrupted mid instruction, because this is the guarantee we get from the hardware. When an interrupt occurs, the instruction has not run at all or it has run to completion. There is not in between state. Atomically means 'as a unit' which we take as 'all or none'.

➜ We want to run this sequence atomically:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

➜ We don't necessary have one. Also the hardware can't just support all operations like an atomic update of B-tree.

➜ Thus we get a few useful instructions from hardware, upon which we can build a general set of what we call **synchronization primitives**.

➜ With hardware support and help from the OS we will be able to build multi-threaded code, that accesses critical sections in a synchronized and controller manner. This will produce the correct result despite the nature of concurrent execution.

➜ Atomic operations are important to file systems (journaling, copy on write), data base management systems, distributed systems, concurrent code, etc

➜ The grouping of actions into a single atomic action is called **transaction**. (Used in databases and transaction processing)

➜ We will be using synchronisation primitives to turn short sequences of instructions into atomic blocks of execution.

6. One More Problem: Waiting For Another

   ➜ First type of interaction between threads is that of accessing shared variables and the need to support atomicity for critical sections.

   ➜ Another type is when a thread must wait for another one to complete some action before it continues. For example when a process performs a disk I\O and goes to sleep. When is finished it must be awaked. (Condition variables it the topic of this type)

ASIDE: KEY CONCURRENCY TERMS
CRITICAL SECTION, RACE CONDITION, INDETERMINATE, MUTUAL EXCLUSION

These four terms are so central to concurrent code that we thought it worth while to call them out explicitly. See some of Dijkstra's early work [D65,D68] for more details.

- A **critical section** is a piece of code that accesses a *shared* resource, usually a variable or data structure.
- A **race condition** (or **data race** [NM92]) arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- An **indeterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not **deterministic**, something we usually expect from computer systems.
- To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.