Operating Systems – spring 2023
Tutorial-Assignment 4

Instructor: Hans P. Reiser

Question 4.1: Segmentation

a. How does segmentation work?

Solution:
With segmentation, a virtual address space is regarded as a collection of segments. Each segment represents a separate part of a program, such as the code, the stack, or some library. Segments occupy contiguous parts of the virtual address space.

Virtual addresses are considered tupels, consisting of a segment number and an offset. The segment number is used as an index into a segment table. A segment table contains a number of entries, each of which consists of a segment base and a segment limit. The base describes the starting address of the segment in physical memory, the limit specifies the length of the segment.

A virtual address can be translated to a physical address with the following steps:

(a) The segment number of the virtual address is used as an index into the segment table to find the correct segment table entry.

*(b) The offset of the virtual address is compared against the limit of the segment table entry. If it is **larger or equal**, the virtual address refers to a location outside the segment, and an exception is raised.*

(c) If the address is valid (i.e., inside the segment), the base value of the entry is added to the offset of the virtual address. The resulting value is the desired physical address.

b. Assume a system with 16-bit virtual addresses that supports four different segments, which uses the following segment table:

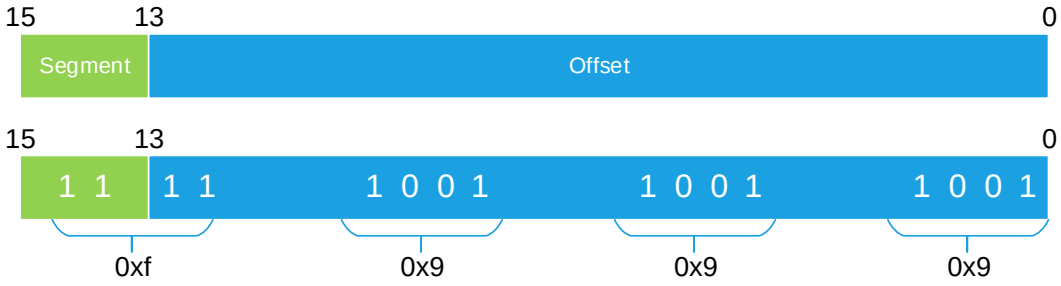
Segment Number	Base	Limit
0	0xdead	0x00ef
1	0xf154	0x013a
2	0x0000	0x0000
3	0x0000	0x3fff

Complete the following table and explain briefly how you derived your solution for each row in the table.

Virtual Address	Segment Number	Offset	Valid?	Physical Address
	3	0x3999		
0x2020				
		0x0204	yes	
			yes	0xf15f

Solution:
The segmentation address is split as follows:

Line 1:
To get the virtual address from the segment number and the offset, we can simply concatenate the bits, which gives us: `0xf999`



A lookup in the segment table reveals that the offset `0x3999` is smaller than the segment's limit of `0x3fff`. We can therefore take the physical base of the segment and add the offset to get the physical address.

Line 2:
We have to do the reverse operation and split the address `0x2020` into the segment index and the offset part. This will give us segment number `0` and offset `0x2020`. Since the offset `0x2020` is greater than segment 0's limit, the virtual address is not valid and cannot be translated to a physical address.

Line 3:
Line 3 constrains the virtual address to possess a valid translation with offset `0x0204`. We therefore must take segment `3`. The valid virtual address is `0xC204` and the physical address is `0x0204`.

Line 4:
The last line demands the virtual address that translates to the physical address `0xf15f`. We can find the segment by subtracting each segments base address from the physical address and check if the resulting offset is within the respective segment's limits. This is the case for segment `1`, where `0xf15f - 0xf154` = `0x000b` (smaller than `0x013a`). We can then use the method from line 1 to get the virtual address `0x400b`.

Note: Because segment 2's limit is 0, there cannot be a valid translation using this segment.

Virtual Address	***Segment Number***	***Offset***	***Valid?***	***Physical Address***
`0xf999`	`3`	`0x3999`	`yes`	`0 + 0x3999`
`0x2020`	`0`	`0x2020`	`no`	*Offset outside segment limit*
`0xC204`	`3`	`0x0204`	`yes`	`0x0204`
`0x400b`	`1`	`0x000b`	`yes`	`0xf15f`

Question 4.2: Threads

a. Explain the terms process, address space, and thread. How do they relate to each other?

Solution:
A process is a "program in execution". The address space is the set of all memory addresses a process can access, and it is part of the process state. A thread is an independent entity of execution. A process has at least one thread, but it is possible to have multiple threads for a single process. All threads of one process share the same address space.

b. Compare the three thread models one-to-one (kernel-level threads), many-to-one (user-level threads), and many-to-many (hybrid threads). Point out advantages and limitations

of each thread model.

Solution:

One-to-One: *Most commonly used model today. Each user thread is mapped to exactly one kernel thread. Creating new threads and switching between threads requires a kernel invocation (high overhead). Model can leverage multi-processor systems. Performing blocking system calls, blocks the whole thread in kernel (user- and kernel-part). Other threads are not affected.*

Many-to-One: *These are managed at user level. Their corresponding thread control blocks (TCBs) are located in user-land. User-level threads execute inside a process (or task) and only the latter is managed by the OS kernel. There is one process control block in the kernel, thus the kernel scheduler decides when to run this process as a whole, whatever user-level thread may be running. ULTs are not known within the kernel, only the single activity associated with the process ("user-level task") is known.*

Advantages of user-level threads *include extremely fast thread management operations as long as the kernel is not involved. A* `yield()` *at user level only requires action in user land. In contrast, operations on one-to-one threads always require costly (in terms of CPU cycles) crossings between user and kernel level. Furthermore, if the runtime offers an adaptive scheduling policy, an application programmer can establish a scheduling algorithm that optimally fits the particular multi-threaded application using ULTs. An additional benefit of user-level threads is that one can run a user-level thread application on each OS platform offering the needed user-level runtime system, whether the OS knows about kernel-level threads or not.*

However, there are also some drawbacks concerning user-level threads*: Each blocking system call blocks the whole application. In a multi-processor system, two user-level threads of the same application can never run concurrently. For specific scientific applications this fact can be very limiting. As the kernel-scheduler does not know about the internal behavior of the task, it may select a multi-threaded task of which only the idle thread can run, or it may de-schedule a task whose currently running thread has acquired a lock that is needed by other parts of the system. The scheduling policy of the kernel can interfere with the scheduling policy at user level, resulting in globally suboptimal performance.*

Many-to-Many *: This model tries to combine the advantages of both pure models. In a hybrid thread model, where* n *user-level threads are mapped to* m *kernel-level threads (*$n \geq m \geq 1$*), each kernel-level thread supports one or more user-level threads. This model is expected to require some interaction between the user-level and the kernel scheduler.*

The first advantage of user-level threads still holds: all thread operations that do not require a kernel entry are still fast, because they are implemented at user level (e.g., switching between user-level threads that are mapped to the same kernel-level thread). Additional kernel entries are required only for thread operations between user-level threads that are mapped to different kernel-level threads (e.g., switching between user-level threads that are mapped to different kernel-level threads).

Again, it is possible to implement a (user-level) scheduling policy that is tailored to the application's needs. However, in contrast to the ULT model, the user-level threads on different kernel-level threads can run in parallel on a multi-processor system.

Distinguishing the hybrid model from ULTs on top of KLTs, the kernel knows that there is a user-level scheduler in the hybrid model: If a thread executes a blocking system call, the kernel scheduler informs the user-level scheduler that the current thread must be blocked but allows the user-level scheduler to select and dispatch a different, runnable thread. With ULTs on KLTs, the kernel would block the KLT, thus

preventing even runnable ULTs that are assigned to this KLT from being dispatched. When the system call completes, the kernel-level scheduler again notifies the user-level scheduler, causing it to unblock (and possibly dispatch) the thread.

The kernel scheduler and the user scheduler may still work against each other, but the effects are slightly mitigated in the hybrid model due the two schedulers cooperating with each other.

A drawback of the hybrid thread approach is the higher complexity of that model.

c. Which types of events can trigger a thread switch in the one-to-one model?

Solution:
As we are using kernel-level threads here, basically all events that we previously discussed as sources of a context switch (process switch) apply here as well for a thread switch:

voluntary: *calling yield(), executing a blocking system call (e.g., `read()`)*

involuntary: *preemption, for example due to*

- **(a)** *End of time-slice*
- **(b)** *High priority thread becoming ready*
- **(c)** *Device interrupt*
- **(d)** *Exceptions (e.g., segmentation violation, privilege violation)*

Keep in mind that the OS does not always run in the background but needs to be invoked to run. Cases (a) and (b) can only be "detected" after invocation of the OS, which is caused by an event (e.g., the timer interrupt).

d. Which types of events can trigger a thread switch in the many-to-one model?

Solution:
Most user level thread libraries only support cooperative scheduling: A user level thread (ULT) is never preempted, but instead must call some `ult_yield()` function from time to time to allow other ULTs in the task to make progress.

Involuntary thread switches among ULTs are difficult to implement, because all of the above events are directed to the kernel, which then would need to return control to a user level event handler. For example, in Unix-like systems, this is to some extent possible with signal handlers, which are user space functions that can be triggered by the operating system in specific situations (ALARM signal based on timer interrupt, or some other signals in case of exceptions)

Blocking system calls, such as sleep, cannot be used to trigger a thread switch in the many-to-one model. The kernel is not aware of the user-level threads and simply blocks the whole process. There is no mechanism that would allow the other user-level threads to run in the meantime.

e. The Unix system call `fork()` creates a new process (child), which is identical to its parent in most parts. Would it make sense for the new process to also contain copies of all the parent's (other) threads?

Solution:
For the thread executing the `fork()`, the return value of `fork()` is the child's process id. The newly created child process will start running in the `fork()` system call. It sees a return value of 0, indicating that it is the child. The code for both parent and child already existed in the parent process. As such, parent and child are aware of the `fork()`.

If all other threads in the parent were to be duplicated into the child, what would their state be after the `fork()`? As they are unaware of the `fork()` invocation (they might, for

example, be running in a tight loop or be blocked in a system call), it is unclear what the duplicates' states should be: Continuing the duplicates where the originals were may lead to data corruption, because a thread is usually not well prepared for concurrency with (a copy of) its very self. Copying a thread which is running in kernel mode would be very difficult and is likely to crash the whole system. For example, consider a thread holding a global system lock while executing in a critical section.

Another use case of `fork()` *is to create a new process and replace its address space contents with a new program image using* `exec()` *or* `execve()`*. Then duplicating all threads would be of no use.*

Question 4.3: Programming with pthreads

a. Write a small program that creates five threads using the pthread library. Each thread should print its number (e.g., `Hello, I am 4`) and the main program should wait for each thread to exit.

Note: This part is not (yet) relevant for this week's programming assignments, but we will come back to using pthreads later (in another programming assignment).

Solution:
To build the program use `gcc` *with the following command line:*
`gcc pthread.c -lpthread -o pthread`
The `-lpthread` *links the pthread library to our sample program. The new threads will be in the ready state as soon as they are created. It is then up to the scheduler to decide when to dispatch which thread. The first thread thus might already be running while we are still creating new threads. You will see in which order the scheduler dispatched the threads in the output.*

```
#include <stdlib.h>
#include <inttypes.h>
#include <pthread.h>

void* greet(void *id)
{
    printf("Hello, I am %ld\n", (intptr_t)id);
    pthread_exit((void*)0);
}

int main()
{
#define NUM 5

    int i;
    // For each pthread, we need to have a pthread_t structure that allows us
    // to reference the pthread later.
    pthread_t threads[NUM];

    for(i = 0; i < NUM; ++i)
    {
        // Create new pthread with the greet() function as entry point.
        // We pass i as argument to the greet() function.
        int status = pthread_create(threads + i, NULL,
                                    greet, (void*)(( intptr_t ) i ));
        if(status != 0) {
```

```
            printf("Error creating thread");
            exit(1);
        }
    }

    // Wait for each thread to exit
    for(i = 0; i < NUM; ++i) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```