Operating Systems 2016/17
Tutorial-Assignment 3

Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

Question 3.1: Process Switching

a. What data is stored in a process control block (PCB)? Where is it located?

Solution:
The PCB stores a process' saved execution context, including its instruction pointer (program counter), stack pointer, general purpose registers, and address space information. A PCB can also contain additional, implementation-dependent information, such as scheduling priorities, open files, and so on. The PCBs are stored as kernel objects to protect them against unauthorized modifications.

Note that, the information regarding the execution state (i.e., registers etc.) are only valid for currently NOT running processes.

b. Describe the actions taken by the kernel to perform a context-switch between processes.

Solution:
Basically, a context switch can be implemented by preserving the previous execution state (context) on the stack and in that process' PCB, exchanging the stack pointers, and loading the new state from the new stack and PCB. This is shown in Figure 1.

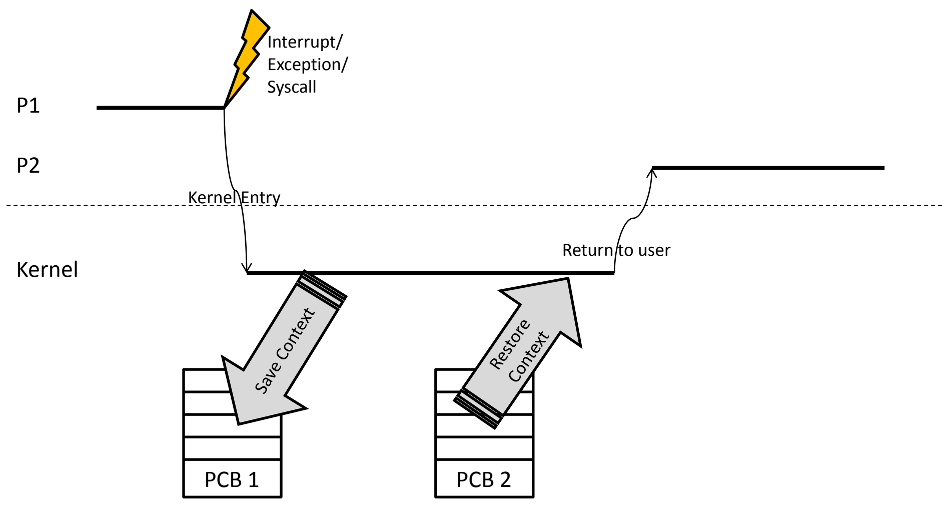

Abbildung 1: Switching from P1 to P2

The context of a process normally consists of all CPU registers, including the instruction pointer (program counter) and stack pointer of the process. In addition to these basic actions, hardware-dependent actions might be necessary: Switching address spaces can require to reload a special register, various caches might need to be flushed, ...

Note that the preempted application is not aware of the preemption.

Question 3.2: Threads

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

Solution:
A thread is an independent entity of execution, representing control flow. A thread resides within an address space. The combination of a thread and its address space is a (single-threaded) process.

b. Compare the three thread models: One-to-One threads (kernel-level threads), Many-to-One threads (user-level threads), and Many-to-Many threads (hybrid threads). Point out advantages and limitations of each thread model.

Solution:

One-to-One: *Also known as kernel level threads (KLTs). 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: *Many-to-One threads, also known as user-level threads (ULTs) 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 *: The Many-to-Many thread model (also known as* hybrid thread 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 that it violates the layered structure of the system: Applications call OS functions (system calls), but now the kernel also calls application functions (the scheduler). By these upcalls *from kernel land "up" into user land, the ULT property of never being preempted by the user-level scheduler is lost: The kernel can activate the user-level scheduler at any time, and the scheduler can then decide whether to preempt the currently running thread or not.*

c. Which types of events can trigger a One-to-One thread switch?

Solution:

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)** *Exception that cannot be handled immediately*
- **(e)** *Exception that leads to aborting the causing thread/task (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 Many-to-One thread switch?

Solution:
Most user level thread libraries only support cooperative scheduling: A Many-to-One thread (user level thread, ULT) is never preempted but must call `ult_yield()` *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. The problem here is that the kernel cannot simply return to user land whence it came, but needs to meddle with return addresses so as to "return" to the event handler, passing the original return address on to user space. Most systems do not support this kind of exception handling in user land, or support only a limited set of signals *(à la Unix* `kill`*).*

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

e. Discuss the following statement: "Jobs are either I/O-bound or compute-bound. In neither case would user-level threads be a win. Why would one go for pure user-level threads at all?"

Solution:
Comparing compute-bound applications implemented with pure user-level threads to a sequential implementation, no efficiency improvement can be observed: The thread library will execute the threads sequentially, maybe partially interleaved, on a single CPU.

An I/O-bound application with user-level threads will block as a whole when invoking a blocking system call (`read()`*,* `write()`*, etc.). A workaround would be to use non-blocking I/O system calls. This design decision, however, needs to be made explicitly as it does not automatically come with the thread model.*

The benefits of user-level threads are not improved performance, but significant improvements of the program structure: Instead of implementing a state machine to manage the various (potentially parallelizable) tasks/stages of a program, one can just let the thread library, which is likely to contain a state machine as well, manage them.

As an example, consider an application which comprises several pipelined stages such as a compiler (i.e., lexer, parser, optimizer, and code generator). An imperative design for the lexer/parser stage would repeatedly invoke the lexer from the parser to get the next tokens from the input file. The parser would then need to provide a context for the lexer (e.g., the current location in the input file) in each call, thus the parser needs to some degree manage or at least keep the state of a different stage.

A design with user-level threads would instead be based on two strictly separated stages with a communication channel in-between. In this approach the lexer continuously writes the next tokens into the channel, managing its state locally on the stack of the user-level thread. The parser, itself also driven by its own user-level thread, simply reads new tokens from the channel. A switch from the parser thread to the lexer thread could be performed voluntarily, for example when the channel is empty. The lexer would then write a certain amount of new tokens to the channel and yield back to the parser thread.

A multi-threaded design is also a good starting point for future real concurrency in a multiprocessor system: If one is able to produce a concurrent application with ULTs, the next step to KLTs and hence real parallelism may be less steep.

f. 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.

g. What is the motivation for and purpose of kernel-mode threads?

   Solution:
   An operating system kernel is only active when an application invoked a system call or when all applications are blocked (system idle). Certain kernel activities are, however, not directly related to a system call and thus should not be performed in the context of a system call. Doing so would reduce the application's performance due to unrelated activities. Instead, these activities are performed by threads inside the kernel that never execute in user mode. They are often subject to normal scheduling and perform background tasks such as garbage collection, freeing page frames and swapping them out to disk, calculating checksums, mirroring disks in a software RAID, or distributing load to other processors. Even the idle state can be modeled conveniently by a thread (running at the lowest priority), eliminating the need for checking for the idle condition in the scheduling code.

h. 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.

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;
}
```