NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4186 - Operating Systems

Solution 2

Scheduling, Processes, and Threads

Professor:
Di Liu
Teaching Assistant:
Roman K. Brunner
Student Assistants:
Eik Hvattum Røgeberg
Halvor Linder Henriksen
Ole Ludvig Hozman

Handout: 26.01.2023

Introduction

This weeks exercise is focussing on processes, threads, and how to schedule them. While we are not covering all topics in this exercise, you see a few different types of exercises. We provide more exercises than we expect you to solve this week, but we suggest you can use them to prepare for the exam in the end of the semester.

1 Processes

We first take a look at processes, their states and abstractions they provide.

1. Which abstraction do processes provide? Why don't we just run an executable directly on the processor and why do we keep all this additional state that comes along with a process?

   Answer: *Current processors only provides us with a small set of hardware threads. Thus, if we want to execute more programs than we have hardware threads simultaneously, we need a way to share the hardware threads. For that reason the OS needs to know which programs are currently running and in which state they are (Scheduling). Additionally, processes serve as entity to which an OS can map resources, so it serves the purpose of enforcing security guidelines (Process isolation).*

2. You learned about the fork() + exec() way of starting a new process under Unix like systems. Give two advantages and two disadvantages of handling process creation using the fork/exec paradigm. *Sidenote: Not all systems opted for the fork/exec paradigm. Windows for example has a CreateProcess function that creates a new process from scratch, loading the specified binary.*

   Answer: *Advantages:*

   - *Makes passing shared memory very easy. If we mapped shared memory before, both, parent and child can now access that location and use it to communicate with each other. All variables stay valid, so whatever has been accessible by the parent is now also accessible by the child.*
   - *It is very easy to implement on a system level. You don't need to set up a new process, you just copy the thing that already exists and continue execution.*

   Disadvantages:

   - *Even with copy-on-write implementations, copying the entire process to then just throwing it away and replace it with another binary can be very inefficient. That's also why more recent systems have other ways of creating processes (e.g.* `vfork` *and* `posix_spawn`*).*
   - *While we can change the behaviour entirely in code, we need to think about everything. If we do not want that the child has access to a certain file descriptor we need to manually clean up before executing the new program. Creating a new process from scratch and then passing the information that we want it to have might be easier and more secure from a user's program perspective.*

3. Which processor modes do exist? Why do we want them and what are the differences between a normal program and a kernel with respect to those modes?

   Answer: *Most processors have at least a privileged (kernel) and a restricted mode (user). In OS design, we then often refer to these two different contexts as kernel and user space. The kernel running in privileged mode has full control over the bus and thus sees all the device-interfaces that the processor sees, including full control over the physical memory space. A userspace program on the other side has no possibility to access the bus or any of the attached devices directly. It has to ask the kernel (using a Syscall) to execute actions on its behalf. That also means that the kernel can check if a program is actually allwed to access a certain resource and if not, it can kill the faulty process to enforce system security.*

4. How can we invoke functionality only the OS can provide? How does such an invocation differ from a normal procedure call?

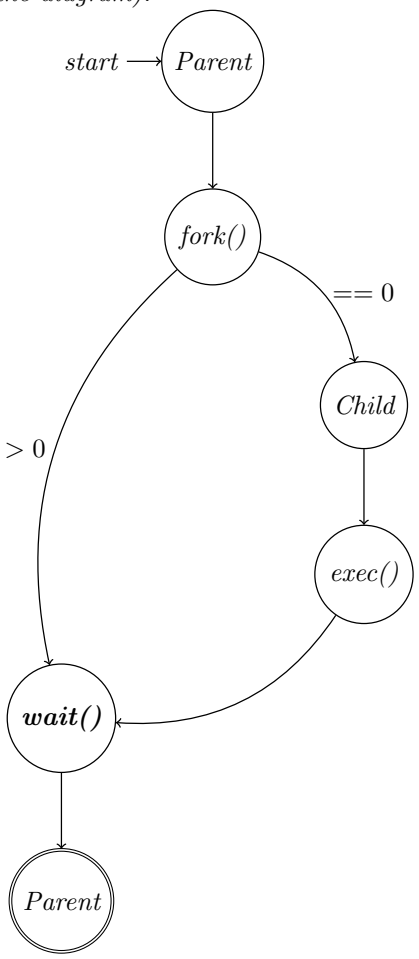
   Answer: *The program issues a so-called Syscall. Syscalls are not normal method invocations, as the user program running in userspace has no possibility to access the resources needed, hence we need to switch contexts to the kernel space. Thus, they are implemented using a special instruction, causing the hardware to cause an interrupt, giving control to the OS. To provide arguments to the syscall, the corresponding values are put into a predefined location, often on the top of the stack in predefined order. So when the OS takes over control, it knows where to look for the arguments to the Syscall. Returning a result is also more difficult, as we are executing in different contexts. There are multiple options to return data:*

   (a) *The userspace program allocates memory for the return value and passes the pointer as an argument. The OS then has to map the memory into its memory space, such that it can write to it.*

   (b) *The kernel allocates memory that is used internally and then is mapped into the memory space of the process.*

5. Draw a state diagram of the fork+execute creation of processes as seen from the User Space processes.

   Answer: *Note: even though it is looking like a finite state machine, there is one state that does not behave as in a FSM. The wait() node cannot accept immediately if the parent process ends up there immediately after fork. It needs to wait until the child process also ends up there and only then can continue the execution of the parent code (=the accepting state in the diagram).*



6. Which possibilities do we have to switch between multiple processes? What are the advantages and disadvantages of the different approaches?

   Answer:

(a) *Direct execution: We directly call the main function of a program and run it directly on the CPU until it completes. This means we can not preempt the process and we will run it to completion.*

(b) *Limited Direct Execution: We still want to directly run the program on the CPU but restrict its access to the system to enforce policies. We create a process and start it, but to access certain resources, the process needs to issue a syscall. The operating system might use those to also schedule other processes. Simple approach, but if a process never makes a syscall it will continue executing on the CPU "forever".*

(c) *Cooperative approach: We let the process execute until the process returns control to us (e.g. through a normal syscall). That also means we rely on all programs behaving nicely (no bugs and not malicious intentions). On the other hand, this makes process switching very simple and efficient.*

(d) *Preemptive approach: We start a hardware timer, that interrupts execution in predefined time intervals. That allows the OS to regain control of the hardware after a predefined time unit and make a scheduling decision. This introduces a lot of overhead but ensures a process cannot capture the CPU.*

7. What option does an OS (using the cooperative scheduling approach) have to deal with a process that does not behave? Is process, memory, and I/O isolation still guaranteed?

   Answer: *The OS cannot do anything until it regains control. The only way to regain control is for the currently misbehaving process to make a syscall. At that point, the OS can take any measure. However, if the process never makes a syscall, the OS has no possibility to stop the process. However, all guarantees with respect to the isolation of processes, memory and I/O are still guaranteed, as the currently running process can only touch memory that has been mapped in its own virtual memory space.*

8. Describe a timer interrupt-induced context switch as seen by a) process A that is executed before the interrupt, b) process B that will be executed after the interrupt, and c) the OS kernel. What is different for process A after it gets to run again after being scheduled out?

   Answer:

Process A: *The process is being executed. When the timer interrupt occurs, the hardware looks up the interrupt handler that the OS has set up at boot time. It then jumps directly to the corresponding interrupt handler. Thus the process does not notice anything (it does not know about the interrupt at all). When process A is scheduled again, the only thing it could use to detect that something happened in between is some notion of time (which either comes from the system or some external service).*

Process B: *Process B needs to be in the ready state when the time interrupt arrives, otherwise it won't be a candidate to be scheduled next. It doesn't know about the interrupt and once it starts/continues running on the core, the system looks the same as when B left.*

OS: *The OS is not running in the beginning, but once the timer interrupt occurs, the interrupt handler kicks in. It saves the current state of Process A (such that it can be reinstalled later, and the switching is transparent to all processes) before calling the scheduler. The scheduler makes a decision which process to scheduler next and reinstalls the state of the process that will run next. It sets the registers and any other state that might have been stored. Finally, it enables interrupts again before setting the PC register to the address of the next instruction to be executed in program B.*

2 Scheduling

For the following exercises, we will discuss scheduling. Please refer to the information in Table 1 for all the tasks.

Job ID	Arrival Time	Execution Time
1	0	50
2	5	250
3	7	3
4	10	50
5	150	50

Table 1: All jobs to be scheduled

1. Assume all jobs arrive at time 0, which of the scheduling algorithms presented in class (FIFO, SJF, STCF, RR) is the best with respect to the average response time? If strict ordering is required, use the job id as tie breaker. Assume 10 for every constant that hasn't been specified otherwise.

   Answer: *We calculate the response time for each of the algorithms:*

FIFO: *Since all arrive at the same time, we use the ID to order them in FIFO order (so from Job 1 to Job 5). This gives the following response times for each job:*

Job ID	Response Time
1	0
2	50
3	300
4	303
5	353
Sum	1006

*Since we are interested in the average we take the sum and divide it by the number of jobs (5 in this case), and thus end up with an average response time of **201.2**.*

SJF: *We order the jobs by execution time and execute them in that order. This gives us the following response time table:*

Job ID	Response Time
3	0
1	3
4	53
5	103
2	153
Sum	312

*Since we are interested in the average we take the sum and divide it by the number of jobs (5 in this case), and thus end up with an average response time of **62.4**.*

STCF: *As we assume that all arrive at zero, STCF and SJF are the same policy, as the shortest job will also be the job that completes first. This means that we get the same result for STCF and SJF.*

RR: *Assuming slots of length 10, and again using the Job ID as a tie breaker we get the following response time table:*

Job ID	Response Time
1	0
2	10
3	20
4	23
5	33
Sum	86

*Since we are interested in the average we take the sum and divide it by the number of jobs (5 in this case), and thus end up with an average response time of **17.2**.*

This concludes that RR is also in this example the best scheduling algorithm if we are interested in response time optimality.

2. Taking the arrival time into account, which of the scheduling algorithms presented in class is the best with respect to response time?

   Answer: *We calculate the response time for each of the algorithms:*

FIFO: *We execute each job in arrival order and to completion. That results in the following response times:*

Job ID	Response Time
1	0
2	45
3	293
4	293
5	203
Sum	834

*Since we are interested in the average we take the sum and divide it by the number of jobs (5 in this case), and thus end up with an average response time of **166.8**.*

SJF: *Every time a job finishes, we execute the shortest job that has arrived in the mean time. This gives us the following response times (in execution order):*

Job ID	Response Time
1	0
3	43
4	43
2	98
5	203
Sum	387

*Since we are interested in the average we take the sum and divide it by the number of jobs (5 in this case), and thus end up with an average response time of **77.4**.*

STCF: *Also called preemptive SJF, we make the decision which job to execute not after finishing a job, but every time a job arrives. We only look at the remaining duration. This leaves us with the following response times:*

Job ID	Response Time
1	0
3	0
4	43
2	98
5	0
Sum	141

*Since we are interested in the average we take the sum and divide it by the number of jobs (5 in this case), and thus end up with an average response time of **28.2**.*

RR: Assuming slots of length 10, we run round robin on all jobs that arrived at that point in time. We assume we always go through the jobs in ascending order of their ID, so a new job might need to wait until it is its turn. This gives us the following response time table:

Job ID	Response Time
1	0
2	5
3	13
4	13
5	13
Sum	44

*The reason why job 5 also sees a response time of 13 is that once it is added to the queue, job 2 is already in the queue. Since we are interested in the average, we divide the sum by the number of jobs (5 in this case), and thus end up with an average response time of **8.8**.*

This concludes that RR is also in this example the best scheduling algorithm if we are interested in response time optimality.

3. What would be the average turnaround time when we run jobs 1-4, using the STCF scheduling strategy? Is it optimal?

   ***Answer:** Turnaround time measures how long it took too execute the job to completion after it arrived ($T_{turnaround} = T_{completion} - T_{arrival}$). We schedule job 3 first as it is the first to complete. Since it only arrives at timestamp 7, it will complete at timestamp 10, thus job 1 has to wait until 10 to start running. Using this, we can calculate all the turnaround times for jobs 1-4:*

Job ID	Turnaround
1	53
2	348
3	3
4	93
Sum	497

   *Dividing the sum by the number of jobs (4) yields an average turnaround time of **124.25**. It is optimal in the sense that whatever job has the shortest time to completion runs first. This means that only long jobs have high turnaround times, while shorter jobs can complete quickly (In contrast to FIFO, where jobs 3 & 4 would have to wait for job 2 to finish first, thus adding to the turnaround for both shorter jobs).*

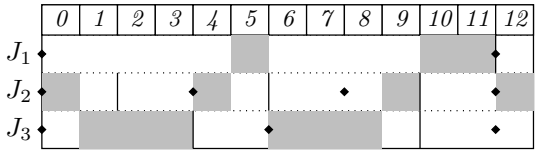
4. Assume you are designing a system that has three jobs that it needs to execute. The jobs are recurring, so they are executed again after a certain time interval. Assume all jobs arrive

at the same time in the beginning. Can you come up with a schedule that allows all the jobs to finish before their respective deadline? Can you develop a new algorithm on how to come up with such a schedule (Think about the ones you already know)? Can you reuse some of the ideas of the scheduling strategies presented in class?

Job ID	Execution Time	Deadline	Interval
1	3	12	12
2	1	2	4
3	3	4	6

Table 2: Jobs with deadline and interval

Answer: *We can extend STCF by inserting the recurring jobs multiple times and draw a schedule until we find a point in time where the state corresponds to a previous state. From there on we can repeat the scheduler indefinitely. Instead of the shortest time to completion, we now take the shortest time to deadline to decide which job gets run. And every time a new job is scheduled we evaluate if we need to change the current job running. That yields the below schedule (dots mark the entry of a job, vertical lines mark the deadline of a job). Note that at timestep 12 we are in the same state again as in timestep 0, so from there on we can just repeat the scheduling pattern as drawn in the time interval [0,11].*



Note: *This type of question is considered more difficult, as you need to develop a new technique, drawing from what you learned in the course.*