Distributed Computing

Autumn Term 2018

Computer Systems
Assignment 4: Scheduling and I/O

Assigned on: October 19, 2018

1 Scheduling

The following table describes tasks to be scheduled. The table contains the entry times of the tasks, their duration/execution times and their deadlines. All time values are given in ms.

<table>
<thead>
<tr>
<th>Task Number</th>
<th>Entry</th>
<th>Execution Time</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>30</td>
<td>70</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>20</td>
<td>90</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>30</td>
<td>120</td>
</tr>
</tbody>
</table>

Scheduling decisions are performed every 10ms. You can assume that scheduling decisions take about no time. The deadline values are absolute.

1.1 Creating schedules

In the following you are asked to create different types of schedules. Please visualize your schedule and also answer the questions below.

Types of schedules:

- **RR (Round Robin):** Assume, that a task which enters the system can be immediately scheduled and it is at the beginning of the scheduling ring.
- **EDF (Earliest Deadline first):** Assume that if two tasks have the same deadline, the first found task (the one longer in the run queue) is going to run.
- **SRTF (Shortest Remaining Time First):** Assume that if two tasks have the same remaining time, the first found task (the one longer in the run queue) is going to run.

Please answer the following questions for each of the schedules and according to the definitions given in the script:
a) How big is the wait time per task?

b) How big is the average wait time?

c) How big is the hold time per task?

d) How big is the time spent in the run queue but not scheduled per task?

e) How is the response time computed for this scheduler? If possible, calculate the response time per task.

Answer: Definitions of the terms “wait time”, “hold time”, and “response time” according to the lecture:

- The wait time or turnaround time of a job is the time taken to finish the job from the point where it entered the system.
- The hold time of a job is the time taken to start executing the job from the point where it arrives.
- The response time of an interactive program is the time taken to respond to a request for service.

a) RR (Round Robin) We assume, that a task which enters the system can be immediately scheduled and it is at the beginning of the scheduling ring. This leads to the following schedule:

<table>
<thead>
<tr>
<th>Task</th>
<th>00</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) How big is the wait time per task?

1: 90ms, 2: 70ms, 3: 60ms, 4: 10ms, 5: 60ms

(b) How big is the average wait time?

\[
(90ms + 70ms + 60ms + 10ms + 60ms) / 5 = 58ms
\]

(c) How big is the hold time per task?

1: 0ms, 2: 10ms, 3: 0ms, 4: 0ms, 5: 0ms

(d) How big is the time spent in the run queue but not scheduled per task?

1: 60ms, 2: 50ms, 3: 40ms, 4: 0ms, 5: 30ms

(e) The response time

In the worst case the task just lost its timeslice (is being preempted by the scheduler) when the user pressed a key. If the task is very fast, it can produce output immediately when it becomes running again. So with 5 jobs we have to wait for 4 jobs which are running in the meantime.

1: 5 jobs scheduled in RR with a timequanta of \((5 - 1) \times 10ms = 40ms\)
b) *EDF (Earliest deadline first)* The deadlines are absolute if the tasks are non-periodic. If two tasks have the same deadline, we assume that the first found task is going to run. This leads to the following schedule:

<table>
<thead>
<tr>
<th>Task</th>
<th>00</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) *How big is the wait time per task?*

1: 60ms, 2: 80ms, 3: 30ms, 4: 10ms, 5: 60ms

(b) *How big is the average wait time?*

\[
\frac{(60ms + 80ms + 30ms + 10ms + 60ms)}{5} = 48ms
\]

(c) *How big is the hold time per task?*

1: 0ms, 2: 60ms, 3: 0ms, 4: 0ms, 5: 30ms

(d) *How big is the time spent in the run queue but not scheduled per task?*

1: 30ms, 2: 60ms, 3: 10ms, 4: 0ms, 5: 30ms

(e) *The response time*

This is less obvious than in RR scheduling. Since the task are scheduled by their deadline we can say, that if a schedule is feasible, the response time is at most (deadline - entry time - execution time). This leads to:

1: 40ms, 2: 70ms, 3: 10ms, 4: 0ms, 5: 40ms

c) *SRTF (Shortest remaining time first)* The job with the shortest execution time is always chosen to be executed. This scheduling might lead to starvation as long jobs might never be scheduled due to short running ones. SRTF can lead to the following schedule:

<table>
<thead>
<tr>
<th>Task</th>
<th>00</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) *How big is the wait time per task?*

1: 80ms, 2: 20ms, 3: 20ms, 4: 20ms, 5: 60ms

(b) *How big is the average wait time?*

\[
\frac{(80ms + 20ms + 20ms + 20ms + 60ms)}{5} = 40ms
\]

(c) *How big is the hold time per task?*

1: 50ms, 2: 0ms, 3: 0ms, 4: 10ms, 5: 30ms

(d) *How big is the time spent in the run queue but not scheduled per task?*

1: 50ms, 2: 0ms, 3: 0ms, 4: 10ms, 5: 30ms
(e) The response time
There is no formalism that can be developed for SRTF for computing the response time. As SRTF can lead to starvation, it might happen that some jobs are never scheduled. As such, the response time can not be estimated.

1.2 General Questions

a) What is the problem with shortest job first - SJF (remember the old printing system@ETH)?
   Answer: Long jobs are potentially never scheduled, if short jobs are entering the system continuously and if there is no preemption.

b) What is the advantage of SJF?
   Answer: It minimizes the time a task is waiting in the run queue to be scheduled and the wait time.

c) What is the benefit of round robin?
   Answer: It is easy to implement, understand and analyze. It has a good response time.

d) What is the big conceptual difference between EDF and RR?
   Answer: EDF is a realtime-based scheduling strategy. Therefore tasks have priorities. RR treats all tasks the same. This is not a realtime scheduling strategy and there is no notion of priorities.

e) Why do hard realtime systems often don't have dynamic scheduling?
   Answer: In a dynamic setup it is not possible to guarantee the feasibility of a correct schedule. That means, if new tasks enter the system and the system allows that, it cannot be guaranteed that every task still meets the deadline. The other approach is to first compute whether there is enough time to allow a new task. If not, creation of a new task will fail. This guarantees that the already running task will always meet the deadlines, however it is possible that an important task cannot be created.

f) What algorithm would be good for interactive workloads?
   Answer: RR between the interactive workloads has the best response time.

1.3 Priority Inversion

Please explain in detail:

a) What is priority inversion?
   Answer: A lower-priority task is running while a high-priority task is not, but would be runnable. Those two tasks are independent and still the lower-priority task runs instead the high-priority task.

b) What is the problem with priority inversion?
   Answer: High-priority tasks cannot proceed, because lower-priority tasks are running. Blocking high-priority tasks is very bad in general.

c) What causes priority inversion?
   Answer: A low-priority task is holding a lock. Later, a high-priority task is trying to acquire the same lock which it can’t, because it is held by the low-priority task. In the meantime, a middle-priority task enters the system and therefore, the low-priority task is preempted, since it has lower priority. The middle-priority task and the high-priority task are completely independent and still the middle-priority task runs instead of the high-priority one. In fact it looks like the middle-priority task has higher priority than the high-priority one. So we have a priority inversion.
What precondition are required to achieve priority inversion?

Answer: Priority inversion can happen if there are at least three runnable tasks of different priorities and the lowest-priority task takes a lock which will be acquired later by the high-priority task.

d) How can this problem be solved?

Answer: This can be solved by a priority inheritance scheme. The task holding the lock temporarily inherits the priority of the task which wants to acquire the lock, if this priority is higher. That makes sure, that no middle-priority task can suspend the low-priority task holding the lock.

e) Priority inheritance

(a) How many levels of priority inheritance do you need?

Answer: In general you need as many levels as possible priority levels. If your tasks can have five levels of priorities (for example), then you need to have five levels of priority inheritance.

(b) Why?

Answer: We know now that the low-priority task holding the lock which a high-priority task wants to acquire has to inherit its priority. However, if for example the low-priority task holds two locks and a high-priority task wants to acquire one of them, it inherits this priority. If a very-high-priority task enters the system and wants to acquire the second lock, the low-priority task inherits the very high priority. After the low-priority task releases the second lock, its priority has to be reset. But we don’t want to reset it to the original low priority, since it is still holding the first lock which is also requested by a high-priority task. That means, we should reset the priority from very high to high. In general this can happen with all the possible levels of priorities. A low-priority task might hold many locks.

(c) How could you implement that?

Answer: This can be implemented by using a linked list. The linked list is kept sorted by priority. Whenever a even higher priority task tries to acquire a lock, the low-priority task inherits its priority and puts it at the front of the linked list. Whenever the low-priority task releases a lock, the corresponding entry is removed from the list. If the removed entry happens to be the head of the list, the task resets its priority to the one represented by the new head of the list. After releasing all the lock, it has its original low priority.

2 I/O Systems

2.1 General Questions

a) State three advantages and disadvantages of placing functionality in a device controller (hardware), rather than in the kernel (software).

Answer: Advantages:

• Bugs are less likely to cause an operating system crash.
• Performance can be improved by utilizing dedicated hardware and hard-coded algorithms.
• The kernel is simplified by moving algorithms out of it.

Disadvantages:

• Bugs are harder to fix - a new firmware version or new hardware is needed.
• Improving algorithms likewise require a hardware update rather than just a kernel or device driver update.
Embedded algorithms could conflict with application’s use of the device, causing decreased performance.

b) Why might a system use interrupt-driven I/O to manage a single serial port (character device), but polling I/O to manage a front-end processor, such as a terminal concentrator? A terminal concentrator is a piece of hardware that has multiple serial ports and one or more LAN ports.

**Answer:** Polling can be more efficient than interrupt-driven I/O. This is the case when the I/O is frequent and of short duration. Even though a single serial port will perform I/O relatively infrequently and should thus use interrupts, a collection of serial ports such as those in a terminal concentrator can produce a lot of short I/O operations, and interrupting for each one could create a heavy load on the system. A well-timed polling loop could alleviate that load without wasting many resources through looping with no I/O needed.

c) Polling for an I/O completion can waste a large number of CPU cycles if the processor iterates a busy-waiting loop many times before the I/O completes. But if the I/O device is ready for service, polling can be much more efficient than is catching and dispatching an interrupt. Describe a hybrid strategy that combines polling, sleeping and interrupts for I/O device service. For each of these three strategies (pure polling, pure interrupts, hybrid), describe a computing environment in which that strategy is more efficient than is either of the others.

**Answer:** A hybrid approach could switch between polling and interrupts depending on the length of the I/O operation wait. For example, we could poll and loop N times and if the device is still busy at N+1, we could set an interrupt and sleep. This approach would avoid long busy-waiting cycles. This method would be best for very long or very short busy times. It would be inefficient if the I/O completes at N+T (where T is a small number of cycles) due to the overhead of polling plus setting up and catching interrupts. Pure polling is best with very short wait times. Pure interrupts are best with known long wait times.

### 2.2 DMA

a) How does DMA increase system concurrency? How does it complicate hardware design?

**Answer:** DMA increases system concurrency by allowing the CPU to perform tasks while the DMA system transfers data via the system and memory buses. Hardware design is complicated because the DMA controller must be integrated into the system and the system must allow the DMA controller to be a bus master. Cycle stealing may also be necessary to allow the CPU and DMA controller to share use of the memory bus.

b) Although DMA does not use the CPU, the maximum transfer rate is still limited. Consider reading a block from disk. Name three factors that might ultimately limit the file transfer.

**Answer:** There are four ways that the maximum transfer rate can be limited:

- Limiting speed of the I/O device - in our case, the disk read throughput.
- Limiting speed of the bus. In this case the bus itself is the bottleneck.
- A disk controller with no internal buffers or too small buffer sizes could also limit the performance of the read file operation.
- Erroneous disk or block read (i.e, if the checksum is incorrect). In this case, an error is signaled and no transfer of the block happens. The block has to be retransmitted.

c) A DMA controller (or DMA engine) has multiple channels that can be used by device drivers to request a DMA transfer. The controller itself is capable of requesting a 32-bit word every 100 nsec. A response takes equally long. How fast does the bus have to be to avoid being a bottleneck?

**Answer:** Each bus transaction has a request and a response each taking 100 nsec, or 200 nsec per bus transaction. This gives 5 million bus transactions / sec. If each one is four bytes, the bus should be able to handle 20 MB/sec. The fact that these transactions may be distributed over four I/O devices (four channels) in round-robin fashion is irrelevant. A bus
transaction takes 200 nsec, regardless of whether consecutive requests are to the same device or different device, so the number of channels the DMA controller has does not matter.