UNIVERSITY OF GOTHENBURG

CHALMERS

Lab 3: Batch Scheduling in Pintos

Noah Márquez
Madhav Goel

October 20, 2022

CONTENTS

1 Description of our solution

We have been asked to describe the implementation of our solution and for doing so we are going to briefly analyze how we implemented each of the following sections.

1 Data Structures

We added several things to **batch-scheduler.c** so we are going to list them with a brief description of their purpose:

1. A global variable called **START** which will make the first task that arrives decide which direction should the bus be at the start.
2. Two more global variables called **TASKDIR** & **TASKPRIO** that will make us easier access the direction and the priority of each task.
3. The last global variable called **TIMEINBUS**, which is the waiting sleep ticks for passing the bus.
4. Two semaphore structs in form of two arrays, one for normal priority tasks and the other for high priority tasks, called **high_prio_queue** & **norm_prio_queue**. This two semaphores will work as our condition variables for waking up or put into waiting the tasks that want to get on the bus in either direction.
5. We use a lock, called **lock**, to ensure mutual exclusion.
6. An integer to know the number of tasks in the bus (initialized to 0), called **slots_used**.
7. Two integer arrays, **waiting_high_prio** & **waiting_norm_prio**, which will be useful to know the number of tasks waiting to go in each direction, either for high or normal priority tasks.
8. An integer to store the current direction of the bus, **current_dir**.

2 Algorithms

In this section we will briefly describe how our code works, so it can be easier to understand our implementation. In the Synchronization section (3) we talk in detail of the most important parts of the code, which are related to synchronization, so here we will focus in giving a more general approach of how our solution works.

After initializing all the variables explained in the previous section, tasks start to ask for slots, transfer data and leaving the slots.

For getting a slot the task firs acquires the lock, then we check if the task is a high priority one so we can give some preference to it. If it is, we check whether the bus is full or if the task's direction doesn't match the current direction of the bus. If any of these condition applies, the number of waiting tasks in the direction of the task increases and the task releases the lock and waits. If it is waken up (it got a slot on the bus), it decreases the number of tasks waiting in its direction, increases the number of slots used in the bus and changes its direction and releases the lock.

A normal priority task works the same way, but it also checks to see if there are any high priority tasks waiting in any direction, if so it waits.

For leaving the bus we first acquire the lock and then we check what we can do:

1. We check if there are high priority tasks waiting in the current direction, if so, we wake them up.

2. If the bus is free and there are high priority tasks waiting in the opposite direction, we change the current direction despite the possibility of still having normal priority tasks waiting in the same direction of the task that just left the bus.
3. If there aren't any high priority tasks on the other side but there are normal priority tasks in the current direction (we checked previously if there were any high priority tasks), we wake them up.
4. Finally, if the bus is free, and arrived at this point, we change direction and start waking up tasks (if any) on the opposite direction.

3 Synchronization

Following the Narrow Bridge problem, to get a slot (or at least try) on the bus, the task first has to acquire a lock. After that, to guarantee that no more than 3 tasks are using the bus in the same direction we check if we have enough space in the bus and if the task has the same direction as the current one. While one of these conditions does not apply, we increase the number of tasks in the waiting list of its direction and we do a *down* in the semaphore struct of its direction to make it wait until it is possible to wake it up.

This applies both to high and normal priority tasks. But to avoid race conditions, when we are doing these for the normal priority tasks, we also check if any high priority task is waiting while this one tries to enter the bus, because we should prioritize high priority tasks.

To let data be transmitted in both directions but not at the same time we use a half-duplex communication bus. So as long as tasks on one direction are using the bus, the other direction cannot use the bus. We do this by simply checking if the task which is trying to enter the bus matches the current direction, if not, we make the task wait.

To grant priority to high priority tasks over the waiting tasks in the same direction we first check (in the *getSlot()* function) whether the task is a high priority one or not. In the case that it is a normal priority task, we make it wait if the list of high priority tasks waiting is not empty.

Moreover, when a task wants to leave its slot in the bus we check the following:

1. If there are high priority tasks waiting in the same direction, we wake them up.
2. If the bus is free and we have high priority tasks waiting in the other direction (although we could have normal priority tasks waiting in the current direction), we change direction and wake them up.

We also guarantee that despite having priority, high priority tasks do not start using the bus while there are other tasks using it in the opposite direction. We do this with simply checking if the task's direction matches the current one, if they do not match, the task is added to the waiting list despite its priority.

4 Rationale

With our solution we wanted to keep things simple and follow the basics, just to avoid unexpected twists and to write our code in a consistent style.

We considered this design based on the Narrow Bridge problem because as we discussed in the lectures, this problem applies to this lab in almost its entirety. The only thing we had to add and take care of is the high priority tasks. The Narrow Bridge problem and the solution we discussed in class were helpful and useful since they use the same concept in solving synchronization issues that applies to this lab.

The first approach that we implemented was with semaphores, pretty different from the Narrow Bridge problem. However, after talking with the TAs and realizing that we were over complicating things, we decided to follow the solution to the Narrow Bridge problem since we believe was the best design for our lab.