Middle East Technical University Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2018-2019

Take Home Exam 2 - Mining Simulation

Due date: 28 04 2019, Sunday, 23:59

1 Objective

This assignment aims to familiarize you with the development of multi-threaded applications and synchronization using C. Your task is to implement a simulator for mining (Turkish; maden çıkarma) by simulating the different agents within the scenario using different threads. Towards this end, you will be using mutexes, semaphores and condition variables for synchronizing the different threads.

Keywords— Thread, Semaphore, Mutex, Condition Variable

2 Problem Definition

We want to simulate the processing of ores (Turkish; cevher) into ingots (Turkish: Külçe). Specifically, *iron*, *copper* and *coal ores* are mined and processed to produce *copper* and *steel ingots*

The processing is handled by four types of agents whose functions are described as below:

- **Miners,** produce ores at a certain rate and have three types: *iron*, *copper* and *coal miners*. A miner has a limited capacity to store the ores it produced. It sleeps when its storage gets full and wakes up, when ores are taken. There exists a maximum number of ores that a miner can produce. Once a miner reaches this number, it quits.

- **Smelters** produce copper or iron *ingots* at a certain rate. A smelter uses 2 ores to produce 1 ingot. Each smelter has a limited storage capacity for incoming ores. A smelter quits, if it cannot not produce ingots (due to the lack of incoming ores) for a certain duration.

- **Foundries:** produce *steel ingots* at a certain rate. A foundry uses *1 iron* and *1 coal ore* to produce *1 steel ingot*. Each foundry has a limited storage capacity for incoming iron and coal ores. The storage capacities for both ore types are the same. A foundry quits if it can not produce ingots (due to the lack of incoming ores) for a certain duration.

- **Transporters:** carry *ores* from *miners* to *smelters* or *foundries*. A transporter can carry one *ore* at a time. The transporter iterates over the miners, and can loads ores from them if the miner has ores in its storage.

Once an ore is loaded, the transporter carries it to producer (a smelter or a foundry) based on the type of the ore. Copper ores are carried to smelters, Iron ore can be carried either to smelters or foundries and, coal ores are carried to foundries. The transporter iterates over smelters or foundries, and can drop ores to them if the smelter or foundry has room in its storage. The transporters can drop ores to smelters or foundries while they are producing ingots. The transporter should only iterate over available producers with empty storage space or wait until one's storage becomes available.
Transporters works until all the miners stop producing ores and have no ore left in their storage. Transports should work miners in order and return to the first miner after reaching the end.

You shall implement the four types of agents as threads and synchronize their activities. The number of miner, transporter, smelter and foundry threads to be used will be given, and the threads will be created at the beginning of the simulation. Initially, all miners will be empty and transporters will have to wait for ores to be produced at the miners. After creation, your main thread should wait for all the agents to finish before stopping. The pseudo-code of simulation agent threads are given below:

Algorithm 1: Miner thread main routine

Data: ID, OreType, Capacity, Interval, TotalOre
CurrentOreCount $\leftarrow 0$
`FillMinerInfo(`*MinerInfo, ID, OreType, Capacity,* CurrentOreCount`)`
`WriteOutput(`*MinerInfo, NULL, NULL, NULL,* **MINER_CREATED**`)`
while *there are remaining ore in the mine* **do**
    `WaitCanProduce` ()
    `FillMinerInfo(`*MinerInfo, ID, OreType, Capacity,* CurrentOreCount`)`
    `WriteOutput(`*MinerInfo, NULL, NULL, NULL,* **MINER_STARTED**`)`
    Sleep a value in range of $Interval \pm (Interval \times 0.01)$ microseconds for production
    CurrentOreCount $\leftarrow$ CurrentOreCount $+ 1$
    `MinerProduced()`
    `FillMinerInfo(`*MinerInfo, ID, OreType, Capacity,* CurrentOreCount`)`
    `WriteOutput(`*MinerInfo, NULL, NULL, NULL,* **MINER_FINISHED**`)`
    Sleep a value in range of $Interval \pm (Interval \times 0.01)$ microseconds for the next round
end
`MinerStopped` ()
`FillMinerInfo(`*MinerInfo, ID, OreType, Capacity,* CurrentOreCount`)`
`WriteOutput(`*MinerInfo, NULL, NULL, NULL,* **MINER_STOPPED**`)`

The functions are explained below:

- `WaitCanProduce:` Wait until a storage space is cleared by a transporter and reserve a storage space for the next ore.
- `MinerProduced:` Informs available transporters that there is avaliable ores in the miners storage.
- `MinerStopped:` Signals the transporters waiting on the miners that this miner has stopped producing. Transporters can keep loading remaining ores in the storage. If this is the last miner, transporters waiting for a load wake up and terminate.

Algorithm 2: Smelter thread main routine

Data: ID, OreType, Capacity, Interval
ProducedIngotCount $\leftarrow 0$
WaitingOreCount is the number of waiting copper ores in storage
`FillSmelterInfo(`*SmelterInfo, ID, OreType, Capacity,* WaitingOreCount*,* ProducedIngotCount`)`
`WriteOutput(`*NULL, NULL, SmelterInfo, NULL,* **SMELTER_CREATED**`)`
while *True* **do**
    `WaitUntilTwoOres()` or timeout after 5 seconds
    if *timeout* **then**
        break
    end
    WaitingOreCount $\leftarrow$ WaitingOreCount $- 2$
    `FillSmelterInfo(`*SmelterInfo, ID, OreType, Capacity,* WaitingOreCount*,* ProducedIngotCount`)`
    `WriteOutput(`*NULL, NULL, SmelterInfo, NULL,* **SMELTER_STARTED**`)`
    Sleep a value in range of $Interval \pm (Interval \times 0.01)$ microseconds for production
    ProducedIngotCount $\leftarrow$ ProducedIngotCount $+ 1$
    `SmelterProduced()`
    `FillSmelterInfo(`*SmelterInfo, ID, OreType, Capacity,* WaitingOreCount*,* ProducedIngotCount`)`
    `WriteOutput(`*NULL, NULL, SmelterInfo, NULL,* **SMELTER_FINISHED**`)`
end
`SmelterStopped()`
`FillSmelterInfo(`*SmelterInfo, ID, OreType, Capacity,* WaitingOreCount*,* ProducedIngotCount`)`
`WriteOutput(`*NULL, NULL, SmelterInfo, NULL,* **SMELTER_STOPPED**`)`

The functions are explained below:

- `WaitUntilTwoOres`: Wait until two ores arrive at its storage and reserve the storage spaces until the production is finished. If storage of smelter already have two ores, thread will directly continue, otherwise it will block.
- `SmelterProduced`: Signals available transporters that two storage spaces have been opened in this smelter.
- `SmelterStopped`: Marks the smelter out of simulation so that transporters no more deliver to this smelter.

Algorithm 3: Foundry thread main routine

Data: ID, Capacity, Interval
ProducedIngotCount $\leftarrow 0$
WaitingIronCount is the number of waiting iron ores in storage
WaitingCoalCount is the number of waiting coal ores in storage
`FillFoundryInfo(`*FoundryInfo, ID, Capacity,* WaitingIronCount, WaitingCoalCount, ProducedIngotCount)
`WriteOutput(`*NULL, NULL, NULL, FoundryInfo,* **FOUNDRY_CREATED**)
while *True* **do**
    `WaitForOneIronOneCoal()` or timeout after 5 seconds
    if *timeout* **then**
        break
    end
    WaitingIronCount $\leftarrow$ WaitingIronCount $- 1$
    WaitingCoalCount $\leftarrow$ WaitingCoalCount $- 1$
    `FillFoundryInfo(`*FoundryInfo, ID, Capacity,* WaitingIronCount, WaitingCoalCount, ProducedIngotCount)
    `WriteOutput(`*NULL, NULL, NULL, FoundryInfo,* **FOUNDRY_STARTED**)
    Sleep a value in range of $Interval \pm (Interval \times 0.01)$ microseconds
    ProducedIngotCount $\leftarrow$ ProducedIngotCount $+ 1$
    `FoundryProduced()`
    `FillFoundryInfo(`*FoundryInfo, ID, Capacity,* WaitingIronCount, WaitingCoalCount, ProducedIngotCount)
    `WriteOutput(`*NULL, NULL, NULL, FoundryInfo,* **FOUNDRY_FINISHED**)
end
`FoundryStopped()`
`FillFoundryInfo(`*FoundryInfo, ID, Capacity,* WaitingIronCount, WaitingCoalCount, ProducedIngotCount)
`WriteOutput(`*NULL, NULL, NULL, FoundryInfo,* **FOUNDRY_STOPPED**)

The functions are explained below:

- `WaitForOneIronOneCoal`: Wait until one coal and one iron ore arrives at its storage and reserve the storage spaces until the production is finished. If storage of foundry already have these two ores, thread will directly continue, otherwise it will block.
- `FoundryProduced`: Signals available transporters that storage spaces for an iron and a coal ore have been opened in this foundry.
- `FoundryStopped`: Marks foundry stopped so that transporters no more deliver to this foundry.

Algorithm 4: Transporter thread main routine

```
Data: ID, Interval, Miners, Smelters, Foundries
FillTransporterInfo(TransporterInfo, CarriedOre)
WriteOutput(NULL, TransporterInfo, NULL, NULL,
 TRANSPORTER_CREATED)
while there are active miners or have ores in their storage do
    CarriedOre ← None
    Miner = WaitNextLoad()
    Transporter thread miner routine
    Producer = WaitProducer()
    if Producer == Smelter then
        Transporter thread smelter routine
    end
    else
        Transporter thread foundry routine
    end
end
FillTransporterInfo(TransporterInfo, CarriedOre)
WriteOutput(NULL, TransporterInfo, NULL, NULL,
 TRANSPORTER_STOPPED)
```

The functions are explained below:

- `WaitNextLoad`: Waits for the next miner with available ore in its storage. If none of the miners have ore available, it will wait. Otherwise assignment of the available ore will be based on a search. If the last miner that a transporter loaded an ore from has the ID k, it should start its search from the Miner with the ID $k + 1$ in cyclic manner. Reserves the storage space so that other transporters cannot take this ore.

- `WaitProducer`: Waits for the next producer with available storage. Producers have different priorities and they are from highest to lowest below:

  - Producers waiting for the next ore to start production. For smelters this means that, they already have one iron or copper ore in their storage and waiting for the second one to start production. For foundry this means that, they have one coal or iron ore in their storage and waiting for the other ore.
  - Producers with empty storage space.
  - If no producers have empty storage space for an ore type, thread will block until a suitable producer finishes production of an ignot and release storage.

  Once unblocked, storage is reserved so that no other transporters can fill that storage space.

Algorithm 5: Transporter thread miner routine

Data: ID, Interval, Miner
`FillMinerInfo(`*MinerInfo, Miner.ID, 0, 0, 0*`)`
`FillTransporterInfo(`*TransporterInfo,* CarriedOre`)`
`WriteOutput(`*MinerInfo, TransporterInfo, NULL, NULL,*
TRANSPORTER_TRAVEL`)`
Sleep a value in range of $Interval \pm (Interval \times 0.01)$ microseconds for travel
$Miner.CurrentOreCount \leftarrow Miner.CurrentOreCount - 1$
CarriedOre $\leftarrow Miner.OreType$
`FillMinerInfo(`*MinerInfo, Miner.ID, Miner.OreType, Miner.Capacity,*
Miner.CurrentOreCount`)`
`FillTransporterInfo(`*TransporterInfo,* CarriedOre`)`
`WriteOutput(`*MinerInfo, TransporterInfo, NULL, NULL,*
TRANSPORTER_TAKE_ORE`)`
Sleep a value in range of $Interval \pm (Interval \times 0.01)$ microseconds for loading
`Loaded(`*Miner*`)`

`Loaded:` Signals the miner to inform that new storage space is available.

Algorithm 6: Transporter thread smelter routine

Data: ID, Interval, Smelter
`FillSmelterInfo(`*SmelterInfo, Smelter.ID, 0, 0, 0*`)`
`FillTransporterInfo(`*TransporterInfo,* CarriedOre`)`
`WriteOutput(`*NULL, TransporterInfo, SmelterInfo, NULL,*
TRANSPORTER_TRAVEL`)`
Sleep a value in range of $Interval \pm (Interval \times 0.01)$ microseconds for travel
$Smelter.Waiting \leftarrow Smelter.Waiting + 1$
`FillSmelterInfo(`*SmelterInfo, Smelter.ID, Smelter.Capacity, Smelter.Waiting,*
Smelter.ProducedIngotCount`)`
`FillTransporterInfo(`*TransporterInfo,* CarriedOre`)`
`WriteOutput(`*NULL, TransporterInfo, SmelterInfo, NULL,*
TRANSPORTER_DROP_ORE`)`
Sleep a value in range of $Interval \pm (Interval \times 0.01)$ microseconds for unloading
`Unloaded(`*Smelter*`)`

`Unloaded:` Signals the foundry to inform that an ore has been unloaded to its storage.

Algorithm 7: Transporter thread foundry routine

Data: ID, Interval, Foundry
`FillFoundryInfo(`*FoundryInfo, Foundry.ID, 0, 0, 0, 0*`)`
`FillTransporterInfo(`*TransporterInfo,* CarriedOre`)`
`WriteOutput(`*NULL, TransporterInfo, NULL, FoundryInfo,*
TRANSPORTER_TRAVEL`)`
Sleep a value in range of $Interval \pm (Interval \times 0.01)$ microseconds for travel
if CarriedOre $==$ *IRON* **then**
    $Foundry.WaitingIron \leftarrow Foundry.WaitingIron + 1$
end
else
    $Foundry.WaitingCoal \leftarrow Foundry.WaitingCoal + 1$
end
`FillFoundryInfo(`*FoundryInfo, Foundry.ID, Foundry.Capacity, Foundry.WaitingIron, Foundry.WaitingCoal, Foundry.ProducedIngotCount*`)`
`FillTransporterInfo(`*TransporterInfo,* CarriedOre`)`
`WriteOutput(`*NULL, TransporterInfo, NULL, FoundryInfo,*
TRANSPORTER_DROP_ORE`)`
Sleep a value in range of $Interval \pm (Interval \times 0.01)$ microseconds for unloading
`Unloaded(`*Foundry*`)`

`Unloaded:` Signals the foundry to inform that an ore has been unloaded to its storage.

The simulation will be subjected to these constraints.

1. Initially, miners, transporters, smelters and foundries have no ores in their storage.
2. The duration a miner takes to produce an ore and the idle time between productions is given as input.
3. Each miner has limited capacity to store the ores it produced. If it becomes full, it should wait for a space in its storage to be opened.
4. Each miner has a limited amount of ore it can produce. When that number is reached, it should quit.
5. Transporters can load an ore from a miner, while the miner is mining ores. Similarly, they can unload an ore to a producers while the producer is producing ingots. Miner reserve a storage during mining for its output. This storage is considered empty until mining finishes. Similarly producers keep their ore input storage occupied until production of ignot finishes.
6. No two transporters can load an ore from a miner or drop an ore to a producer at the same time.
7. Transporters take a certain amount of time to travel between miner and producer agents in the simulation. They also take certain amount of time when loading/unloading. This duration is given to you as input.
8. Transporters should start their search from the first miner if it is the first call and next miner in ID order for the subsequent iterations.
9. Transporters should quit if there are no active miners or miners with ores in their storage.
10. If a transporter loads a copper ore from a miner, it should carry it to a smelter. If it is a iron ore, it should carry it to a smelter or foundry. There is no preference if they have the same priority. It is up to you how you want to do it as long as you respect priority of producers waiting for the second ore over producers with no ores. If the ore is coal, it should be carried to a foundry.
11. The smelters require two ores to produce a single ingot. They should wait for ores to be deposited without busy waiting.
12. Foundries require one iron and one coal to produce a steel ingot. They should wait for ores to be deposited without busy waiting.
13. Smelters/foundries can produce their ingots while a transporter is depositing ore into their storage. Similarly a transporter can deposit its ore into the storage while the smelter/foundry is producing ingots, as long as there is space in their input storage.
14. It takes a certain duration for the smelters/foundries to produce their ingots. This duration is given to you as input.
15. A foundry/smelter quits if it can not produce ingots (due to the lack of incoming ores) for a certain duration. Note that, they may still have have left-over ores in their input storage (e.g. one copper ore in a smelter or only coal/iron ores in a foundry).

3 Implementation Specifications

1. Each agent should be implemented as separate thread. When a agent thread created, following function call should be made for every agent:

   - **Miner:**
     `WriteOutput`(MinerInfo, NULL, NULL, NULL, **MINER_CREATED**)
   - **Transporter:**
     `WriteOutput`(NULL, TransporterInfo, NULL, NULL, **TRANSPORTER_CREATED**)
   - **Smelter:**
     `WriteOutput`(NULL, NULL, SmelterInfo, NULL, **SMELTER_CREATED**)
   - **Foundry:**
     `WriteOutput`(NULL, NULL, NULL, FoundryInfo, **FOUNDRY_CREATED**)

2. You should call InitWriteOutput function before creating threads.

3. You can use `usleep(time - (time*0.01) + (rand()%(int)(time*0.02)))` to sleep with a value in $[0.99 * time, 1.01 * time]$ microseconds range.

4. Main thread should wait for every thread to finish before exiting. Each agent should make the following call before exiting:

   - **Miner:**
     `WriteOutput`(MinerInfo, NULL, NULL, NULL, **MINER_STOPPED**)
   - **Transporter:**
     `WriteOutput`(NULL, TransporterInfo, NULL, NULL, **TRANSPORTER_STOPPED**)
   - **Smelter:**
     `WriteOutput`(NULL, NULL, SmelterInfo, NULL, **SMELTER_STOPPED**)
   - **Foundry:**
     `WriteOutput`(NULL, NULL, NULL, FoundryInfo, **FOUNDRY_STOPPED**)

5. Simulator should use WriteOutput function to output information, and no other information should be printed.

4 Input Specifications

Information related to simulation agents will be given through standard input. First line will contain number of mines (N_M) in the simulation. Following N_M lines contain the properties of the miner with i^{th} ID (All IDs start from 1) in the following format:
- I_M C_M T_M R_M where

- I_M is an unsigned integer representing the production and wait interval of the miner. It is given in microseconds. The miner will sleep this amount during and between production of each ore with slight deviation.
- C_M is an unsigned integer representing the storage capacity of the miner.
- T_M is an unsigned integer representing the oretype of the miner. Ores have corresponding values:
  - **IRON:** 0
  - **COPPER:** 1
  - **COAL:** 2
- R_M is an unsigned integer representing the total amount of ore that can be extracted from the mine.

Next line contains the number of transporters (N_T) in the simulation. Following N_T lines contain the properties of the transporters with i^{th} ID in the following format:
- I_T

- I_T is an unsigned integer representing the travel and load/unload time of the transporter. It is given in microseconds. The transporter will sleep this amount during travel or load/unload operation slight deviation.

Next line contains the number of smelters (N_S) in the simulation. Following N_S lines contain the properties of the smelters with i^{th} ID in the following format:
- I_S C_S T_S

- I_S is an unsigned integer representing the production interval of the smelter. It is given in microseconds. The smelter will sleep this amount during production of each ingot with slight deviation.
- C_S is an unsigned integer representing the storage capacity for ores of the smelter.
- T_S is an unsigned integer representing the oretype of the smelters. It can be IRON or COPPER with 0 and 1 values respectively.

Next line contains the number of foundries (N_F) in the simulation. Following N_F lines contain the properties of the foundries with i^{th} ID in the following format:
- I_F C_F

- I_F is an unsigned integer representing the production interval of the foundry. It is given in microseconds. The foundry will sleep this amount during production of each ingot with slight deviation.
- C_F is an unsigned integer representing the storage capacity for ores of the foundry.

5 Specifications

- Your code must be written in C or C++ on Linux. No other platforms and languages will be accepted.
- You are allowed to use pthread.h and semaphore.h libraries for the threads, semaphores, condition variables and mutexes. Your solution should not employ busy wait. Your Makefile should not contain any library flag other than -lpthread. It will be separately controlled.
- Submissions will be evaluated with black box technique with different inputs. Consequently, output order and format is important. Please make sure that calls to WriteOutput function done in the correct thread and correct step. Also, please do not modify "writeOutput.c" and "writeOutput.h" files as your submission for these files will be overwritten.
- There will be penalty for bad solutions. Non terminating simulations will get zero from the corresponding input.
- Your submission will be evaluated on lab computers (ineks).
- Everything you submit should be your own work. Usage of binary source files and codes found on internet is strictly forbidden.
- Please follow the course page on piazza for updates and clarifications.
- Please ask your questions related to homework through piazza instead of emailing directly to teaching assistants.

6 Submission

Submission will be done via COW. Create a tar.gz file named `hw2.tar.gz` that contains all your source code files together with your Makefile. Your tar file should not contain any folders. Your code should be able to compile and your executable should run using this command sequence.

```
> tar -xf hw2.tar.gz
> make all
> ./simulator
```

The name of your executable is not important as long as it executes with "make run" command. **If there is a mistake in any of the 3 steps mentioned above, you will lose 10 points.**

7 Cheating

We have zero tolerance policy for cheating. People involved in cheating will be punished according to the university regulations. Cheating Policy: Students may discuss the concepts among themselves or with the instructor or the assistants. However, when it comes to doing the actual work, it must be done by the student alone. As soon as you start to write your solution or type it, you should work alone. In other words, if you are copying text directly from someone else - whether copying less or typing from someone else's notes or typing while they dictate - then you are cheating (committing plagiarism, to be more exact). This is true regardless of whether the source is a classmate, a former student, a website, a program listing found in the trash, or whatever. Furthermore, plagiarism even on a small part of the program is cheating. Also, starting out with code that you did not write, and modifying it to look like your own is cheating. Aiding someone else's cheating also constitutes cheating. Leaving your program in plain sight or leaving a computer without logging out, thereby leaving your programs open to copying, may constitute cheating depending upon the circumstances. Consequently, you should always take care to prevent others from copying your programs, as it certainly leaves you open to accusations of cheating. We have automated tools to determine cheating. Both parties involved in cheating will be subject to disciplinary action. [Adapted from `http://www.seas.upenn.edu/cis330/main.html`]