Assignment 2: Tesla Factory Production Line Control System

Deadline: 23:55, Dec. 5, 2020. You get 5 bonus marks if you submit on or before 23:55, Nov. 28, 2020

Weighting: 15% of the course assessment

Full marks: 100 + 20 competition bonus + 5 early bird bonus

Table of Contents

Objectives

In this assignment, you need to write a multithreaded program in C with **pthreads** simulating the production process of Tesla electric cars. Upon the completion of this assignment, you should be able to:

- Use pthread library to write multithreaded program
- Use semaphores to handle thread synchronization
- Use semaphores to manage limited resources
- Solve producer and consumer problem

Prerequisites

Before you start to work on this assignment, you should have known the concepts of thread, semaphore, as well as deadlock. A review of related lecture notes and Tutorial 3 and 4 slides is highly recommended. You are also required to be able to code in C.

Background Story

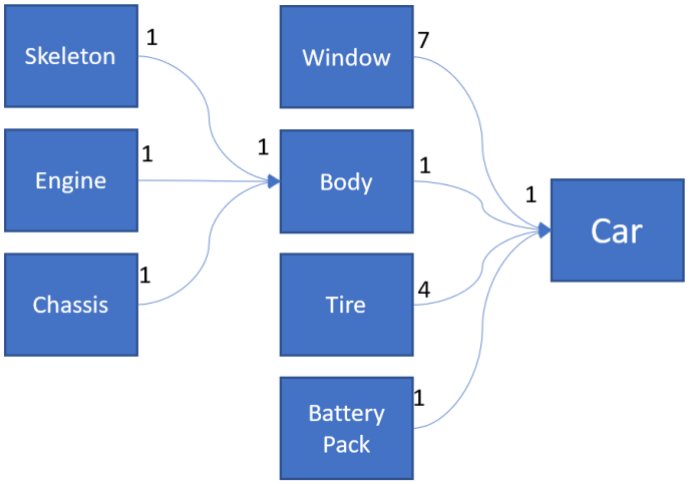
Tesla Motors is the leading company of electric cars and is also the first company who made electric cars so popular around the world. Tesla factory almost automated the whole production process by using a lot of smart trainable robot workers. There are some videos in the reference section you may check to learn more about the simplicity of the design and the complexity of making it possible.

Your job in this assignment is to write a program to simulate the production process of Tesla electric car and make the production process as efficient as you can with limited resources.

System overview

Dependency relationship

To simplify the process of manufacturing Tesla electric cars, 7 types of car parts need to be produced before the final assembly of a car. These parts are *skeleton*, *engine*, *chassis*, *body*, *window*, *battery*. The relationship among these parts can be found in the graph below:



Those parts which no arrow is pointing at depend on their own raw materials. We just assume that they are ready by default. The production rule is: 1 skeleton, 1 engine, and 1 chassis can make 1 car body; 7 windows, 1 body, 4 tires, and 1 battery pack can make 1 car. **17** parts in all are required for making 1 car.

File Structure

- lib *(contains tesla factory library files)*
  - libTeslaFactory.a
  - libTeslaFactoryDebug.a *(link against this lib to print debug info)*
- q1 *(question 1)*
  - include *(contains header files)*
    - production.h
    - robot.h
    - scheduler.h
    - utils.h
  - src *(contains source code)*
    - main.c
    - scheduler.c
  - makefile (take a look at it to know the program structure)
- q2 *(question 2)*
  - sampleTestCases.sh *(sample deadlock test cases)*
- q3 *(question 3)*

Please don’t change the directory structure.

System implementation

You need to read the source code to fully understand the program. Only the pivotal parts are introduced here.

libTeslaFactory.a and libTeslaFactoryDebug.a

This library provides basic functions you will use in this assignment. Function definitions and introductions are in the header files.

include/production.h: contains production related functions like make items, tracking production progress. When the tesla factory program starts, initProduction() will be called to initialize the factory environment. To make an item, the robot must reserve a unit of space first. Three types of item making functions are provided: make<item>, tryMake<item> and timedTryMake<item> which correspond to calling sem_wait(), sem_trywait(), sem_timedwait() for **acquiring the space**. tryMake<item> and timedTryMake<item> will return 0 only if successfully acquire a space and produce an item. Note that making a car doesn't require a space as the car will be shipped out of the factory once assembled. Once *makeBody() acquires a space or makeCar() starts, both functions will wait for all their dependent parts and the production process can't be stopped. If there's no enough dependent parts ready, they will wait infinitely.

include/robot.h: contains robot related functions robot structure and task structure. The robot structure contains 4 items: its id, type, associated pthread, and a pointer to Task_t.(Robot and Task have been defined as a pointer to Robot_t and Task_t respectively.) There are 3 types of robot, each has different production time for different parts:

	Type A	Type B	Type C	MIN	MAX
SKELETON	5	4	3	**3**	**5**
ENGINE	4	5	3	**3**	**5**
CHASSIS	3	4	4	**3**	**4**
WINDOW	1	2	3	**1**	**3**
TIRE	2	2	1	**1**	**2**
BATTERY	3	4	4	**3**	**4**
BODY	4	3	6	**3**	**6**
CAR	6	5	4	**4**	**6**
Total	40	47	49	**30**	**59**

Table 1: Production Time

You can make changes to the task structure according to your needs like adding more queues or semaphores. At the beginning of the program, initNumRobot() will be called to tell the system how many robots of each type there will be. Internally, the library will count how many robots of each type have been created, you need to use **createRobot()** and **releaseRobot()** to create and release robots. You can also query how many robots left of a certain type via function getNumFreeRobot().

include/utils.h: Contains a timer function and some thread-safe non-cached printing functions.

- **double getCurrentTime()**: a wrapper of omp_get_wtime().

You can use these printing functions when debugging.

- **flushed_printf(const char *msg, ...)**: use like printf except that this function will flush stdout immediately.
- **debug_printf(const char *msgHead, const char *msg, ...)**: this function will print the time, message head and message like the following:

```
[21:29:23]initNumRobot Initializing robots. TypeA: 3, TypeB: 2, TypeC: 1
[21:29:23]initProduction Production goal: 3 cars, number of storage space: 20
```

- **err_printf(const char *function, int error_num, const char *what, ...)**: this function will print error message:

```
[21:29:23]|ERROR!| createRobot(94): Creating TypeA robot failed. You are trying to create 4 TypeA robots, max: 3.
[21:29:23]|ERROR!| createRobot(94): Creating TypeB robot failed. You are trying to create 3 TypeB robots, max: 2.
[21:29:23]|ERROR!| createRobot(94): Creating TypeC robot failed. You are trying to create 2 TypeC robots, max: 1.
```

You can wrap the debugging functions between **#ifdef DEBUG** and **#endif**

```
#ifdef DEBUG
  debug_printf(__func__, "Num jobs: %d\n", task->jobQ->size);
#endif
```

so that you can control when to print these messy stuff by adding or removing -DDEBUG flag to the compiler(check the makefile). When you submit your code, make sure you don't print debugging messages when -DDBUG is removed.

If you link your code to **libTeslaFactoryDebug.a**, then it will print out each internal function call information.

The makefile provided has automated the debug and release compiling process. You can simply type *make* for release build and *make debug* for debug build.

Non library files

scheduler.h and scheduler.c: contains your robot production routine(pthread function) and runtime scheduling code. There's a simple example in these two file that all robot will get job IDs from a job queue, and produce items according to the job ids sequentially. You can create your own runtime routines here to make the production fast and deadlock free.

main.c: launches the whole production process. The main function will gather production information, initialize the production environment. When the production finishes, it will print out a production report. The function **startProduction()**: is where you put your preparation code, thread launching code and resource releasing code. There's no need to change code in the main() function except change the printf() line to print your own name and university ID.

Rules: You are not allowed to create more robots of each type than that the program arguments defined. You are not allowed to create more pthreads than the total number of robots (num_typeA + num_typeB + num_typeC).

Files and functions you are allowed and are necessary to modify: ***scheduler.h, scheduler.c, main.c:startProduction(), main.c:main() change to your name and UID, robot.h: definition of struct Task_t.***

Questions (85 marks + 20 bonus marks)

Q1 Complete the simple multithreaded version (20 marks)

The code you download is an incomplete program of the production line control system. What you need to do is to complete the multithreaded version with given simple function in scheduler.h and scheduler.c and get familiar with the system.

You need to:

1. Copy your thread-safe queue.c and queue.h from tutorial 4 exercise to directory q1/include and q1/src respectively.
2. In ***main.c***, you should complete the function startProduction() so that we can create all 3 types of robots and these robots can work concurrently. (10 marks for coding)
3. After you finish your code, you can compile your code by typing in command *make* in your Linux console (**makefile** has been provided). You can also type *make debug* to generate **tesla_factory_debug.out**. This executable will print out each step of the production process with a time stamp.
4. **Change the line in *main.c* to print out your name and your university ID at the beginning of your program**. Include a screenshot of your program executed with **./tesla_factory.out 3 20 1 2 3** in your report. You should get the same around 33 secs of production time as show in the sample screenshot below.
5. Then you need to run a few more times with different number of production goal and each type of robots. Then briefly answer the following questions given sufficient storage space (all robots can acquire a unit of storage space as they need immediately):
   - how many units of storage space can be considered as sufficient for sure?
   - Is it always true that as the number of robot increases, the production time decreases?
   - How can the number of cars and the number of robots of each type affect the production time(open-ended question, share your thoughts)?

   You can add charts to help you.(10 marks for screenshot and analysis)

Here's a sample screenshot of the output of q1 (debug mode off):

```
Name: Elon Mask  UID: 3035512345
Production goal: 3, num space: 20, num typeA: 1, num typeB: 2, num typeC: 3
====Final Report====
Num free space: 20
Produced Skeleton: 0
Produced Engine: 0
Produced Chassis: 0
Produced Body: 0
Produced Window: 0
Produced Tire: 0
Produced Battery: 0
Produced Car: 3
Production task completed! 3 cars produced.
Production time: 33.001176
```

Q2 Implement a deadlock free multithreaded program (25 marks)

In Q1, there's no limitation on the number of storage space. If we take storage space into consideration, we may encounter deadlock. Say we have 2 robots and only 2 storage spaces. Here's an example of deadlock(debug mode on):

```
[11:44:48]initProduction Production goal: 1 car, number of storage space: 2
[11:44:48]initNumRobot Initializing robots. TypeA: 1, TypeB: 1, TypeC: 0
Production goal: 1, num space: 2, num typeA: 1, num typeB: 1, num typeC: 0
[11:44:48]startProduction Num jobs: 17
[11:44:48]simpleRobotRoutine RobotA[0] starts...
[11:44:48]simpleRobotRoutine RobotA[0]: working on job 0...
[11:44:48]simpleWork RobotA[0] making skeleton...
[11:44:48]_requestSpace Requesting space, current space=2...
[11:44:48]_requestSpace Space requested, current space=1...
[11:44:48]simpleRobotRoutine RobotB[1] starts...
[11:44:48]simpleRobotRoutine RobotB[1]: working on job 1...
[11:44:48]simpleWork RobotB[1] making engine...
[11:44:48]_requestSpace Requesting space, current space=1...
[11:44:48]_requestSpace Space requested, current space=0...
[11:44:53]simpleWork RobotA[0] job 0 done! Time: 5.000184
[11:44:53]simpleRobotRoutine RobotA[0]: working on job 2...
[11:44:53]simpleWork RobotA[0] making chassis...
[11:44:53]_requestSpace Requesting space, current space=0...
[11:44:53]simpleWork RobotB[1] job 1 done! Time: 5.000177
[11:44:53]simpleRobotRoutine RobotB[1]: working on job 3...
[11:44:53]_requestSpace Requesting space, current space=0...
^C
 ~/comp3230_/TeslaFactory/q1  main !12 ?4   INT 1m 56s pfxu@workbench
```

./tesla_factory 1 2 1 1 0, 1m 56s passed still can't produce a single car

1. Robot 1 gets jobID=0, and requests 1 unit of storage space and make a skeleton.
2. Robot 2 gets jobID=1 and wants to build an engine. Robot 2 requests 1 unit of storage space and start production.
3. Robot 1 finishes and gets jobID=2 building a chassis. Robot 1 requests 1 unit of storage space but failed, because it's been taken to store the skeleton and the engine. Robot 1 stops and waits for space…
4. Robot 2 finishes and gets jobID=3 building a battery. Robot 2 requests 1 unit of storage space but failed as well. Robot 2 stops and waits for space…
5. Both robot 1 and robot 2 are waiting for a free space infinitely… Deadlock appears.

Copy your code from q1 to q2 and make it deadlock free. There are some sample test cases listed in **q2/sampleTestCases.sh**. The minimum number of storage space is **2**, and the minimum number of total number of robots(regardless of the type) is **2**. You should test your program with different combinations of input parameters (number of cars, number of spaces, number of robots) and make sure your program is deadlock free and bug free.

Hints:

1. You can add more stuff in Task_t like semaphores and more queues to help you coordinate threads solving deadlock.
2. In production.h, tryMake<item> and timedTryMake<item> are provided.
3. Deadlock handling strategies:
   a. Deadlock detection: you don't know whether you will encounter deadlock or not ahead. But once your program detects that certain thread has run for unreasonable amount of time, then you know there's deadlock. Your program needs to figure out a way to break the deadlock so that the production process can move on.
   b. Deadlock prevention: once the production goal is set, you know the total number of each car part to be produced. You also know how many unit of space you have. Your robots can make use of the known information and come up with an agreement(like the hungry philosophers problem in tutorial 4) so that there will be no deadlock.

A sample screenshot of a deadlock free test case:

```
Production goal: 3, num space: 2, num typeA: 1, num typeB: 0, num typeC: 1
====Final Report====
Num free space: 2
Produced Skeleton: 0
Produced Engine: 0
Produced Chassis: 0
Produced Body: 0
Produced Window: 0
Produced Tire: 0
Produced Battery: 0
Produced Car: 3
Production task completed! 3 cars produced.
Production time: 133.006202
```

Your program will be tested with a bunch of randomly generated test cases. If **any** of those test case fails, you will get 0 mark for this question. As mentioned before, the maximum time of sequentially producing one car is 59 seconds. If your program failed to complete production in **60s * number of cars**, it will be considered stuck in a deadlock situation. (15 marks for coding)

```
❯ timeout -k 60s -s 9 60s ./tesla_factory.out 1 2 1 1 1
Name: Elon Mask  UID: 3035512345
Production goal: 1, num space: 2, num typeA: 1, num typeB: 1, num typeC: 1
[1]    28431 killed     timeout -k 60s -s 9 60s ./tesla_factory.out 1 2 1 1 1
~/comp3230_2020/Assignment2/q1  main                                  ✘ KILL  1m 0s
```

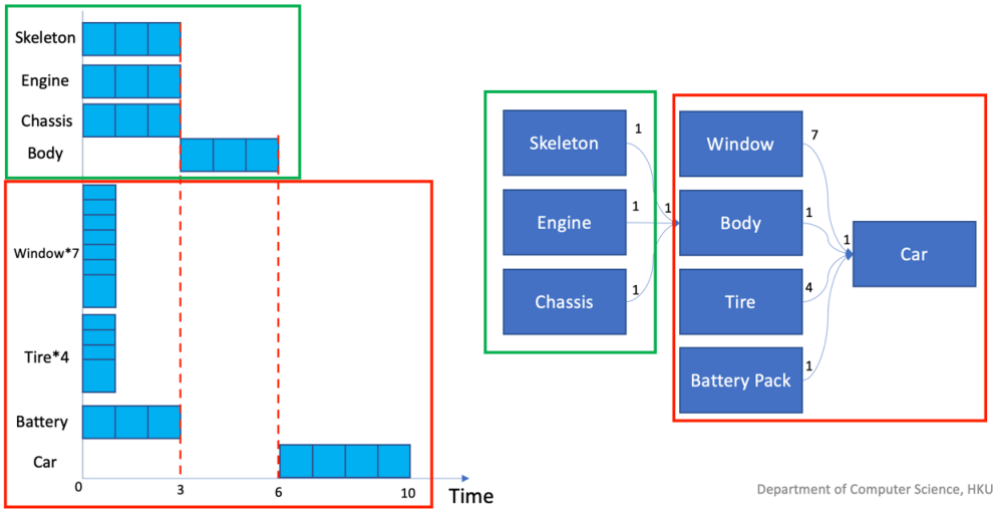
Use *timeout* command to test if there is a deadlock

Briefly introduce your deadlock solving algorithm in your report. Which strategy do you choose? You also need to explain how your algorithm can ensure that it's deadlock free. (10 marks for report)

Q3 Optimize Your Program (60 marks, 20 bonus marks included)

Copy your deadlock free code from q2 to q3. Given the production time of each type of robot for each job in Table 1, how can you make the production process faster while still make sure there's no deadlock?

The theoretical shortest production time for one car is 10 sec, if all parts are produced in parallel and each part is produced by the robot whose production time for that part is the shortest among these 3 types of robot and there is sufficient number of storage space.



How close your program can get to the theoretical shortest production time with constrains on the number of robots and storage space?

Briefly introduce your optimized/improved algorithm in the report. Please highlight what changes you've made to make your program run faster. You also need to include a scalability analysis of your program. Here are some points you may consider including in your scalability analysis:

How each factor (number of cars, number of storage space and number of robots) can affect your production time? For example, you can fix the number of cars (and number of storage space) and increase the number of robots to see how your production time changes. Or you can control other sets of variables and analyze how your program behaves. Will the production time converge? When does the number of storage space start to affect the production time? What is the theoretical best performance for given number of robots, number of storage space and number of cars to be made? What cause the performance difference between the theoretical best and your implementation? Or your implementation has reached the theoretical best and you can explain why your design is so good. You can also analyze with deadlock problem if you sacrifice performance to deal with deadlock. You need to explain your observations in conjunction with your own algorithm. You can also try to predict the performance(hard). Given the number of robots, number of storage space, and number of cars, can you predict the production time? What is(are) your prediction method(s)? How accurate can you predict? If there's prediction error, you may perform a quantitive analysis to evaluate the error. You can plot your predicted production time and the actual production time to help you explain. If time allowed (there's deadline on assignment 2), which part of your program can be further optimized to shorten the production time? Can your scheduling algorithm be applied to somewhere else (other applications)?

Marking scheme:

1. Speed contest: 30 marks (10 bonus marks included);

   Your total mark = $30 \times \frac{\sum_{i=1}^{N} T_{i\,min}}{\sum_{i=1}^{N} T_i}$, where T_i is your runtime of the ith test case, $T_{i\,min}$ is the minimum time among all your classmates. If there's deadlock, you get 0 mark and your performance won't be recorded either.
2. Scalability analysis: 30 marks (10 bonus marks included for high quality analysis and task scheduling scheme design).

General requirements (10 marks)

1. Code quality: write readable code, use meaningful variable names and function names, add necessary comments if possible to help others understand your code. No memory leak, free acquired resources… (we'll check with valgrind) (10 marks).
2. You must see the line "Production task completed! <n> car(s) produced." printed in the end indicating the production goal has been achieved. If the production goal is not achieved, you will get 0 mark for that question.
3. In the final report, the numbers of all produced parts should be 0 and produced cars should be equal to it is asked to produced. Otherwise, you program is producing wasted items and you will get only **half** of the marks for coding and won't gain any bonus marks in q3.
4. Make sure that your code for each question can be compiled and run without problem **on workbench**, or you will get **0** mark for that question no matter how well you explain your idea in the report. If your program can't run, we can't tell if your statements are true or not.
5. If we find you create more robot threads than required or storage space, **0 mark** will be given for that question.
   - When marking your program, we will intercept basic function calls to keep track of resource usage. For example, we will count how many robot threads of each type have been up and running before calling pthread_create().
6. **No plagiarism**. The minimal penalty for confirmed plagiarism is getting 0 mark for this assignment. The student who offers the solution to the others will receive double penalty (deducting 30 marks from the total course assessment).

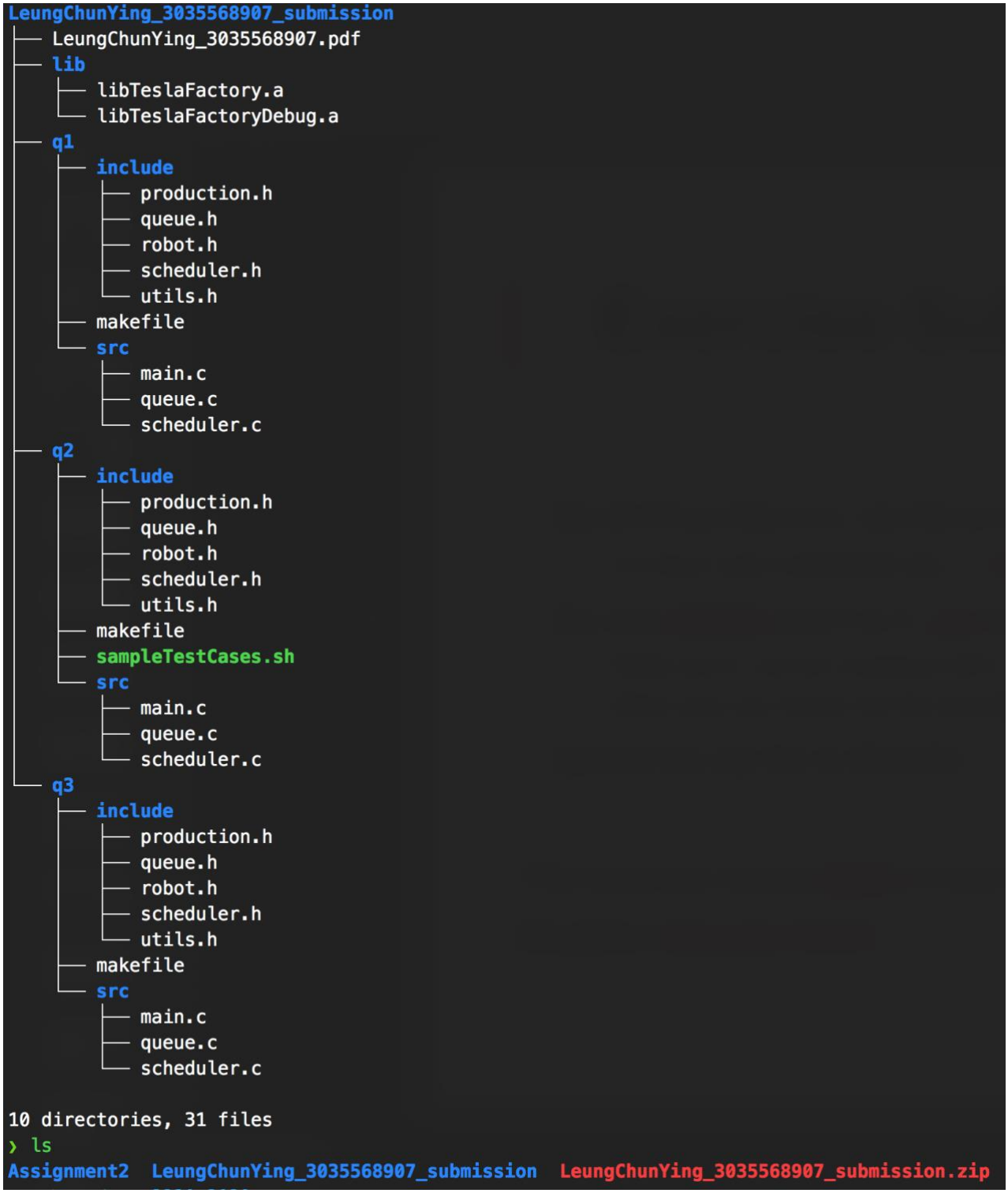
Reminder: Start working on your assignment ASAP. You are recommended to write code and debug your program on workbench directly.

Some questions are not allowed to be asked:

1. Ask for answers to compare with your own code or check your answer with tutors before you submit it
2. Ask if your idea/algorithm work or not. If you have an idea, you should find ways to proof/disproof it.
3. Questions related to C language. This is not a programming course, basic C programming skills should be enough for this assignment. If you don't know how to program in C, there are self-learning materials on Moodle, tons of great tutorials on YouTube…
4. Debug your program. Try to debug your program with printf or gdb **by yourself**.

Submission (5 marks)

Put your answer source code of question 1 in directory named q1, source code of question 2 in q2, and question 3 in q3; name your report <name>_<university ID>.pdf (Surname first). Then put q1, q2, q3 and your report in a directory named <name>_<university ID>_submission(Surname first), then zip it to a zip file <name>_<university ID>_submission.zip(Surname first) (5 marks).



Reference

1. How the Tesla Model S is Made | Tesla Motors Part 1 (4:54)
2. How Tesla Builds Electric Cars | Tesla Motors Part 2 (3:25)
3. Electric Car Quality Tests | Tesla Motors Part 3 (1:49)
4. Oh My Tmux! https://github.com/gpakosz/.tmux
5. A Gentle Introduction to tmux: https://medium.com/hackernoon/a-gentle-introduction-to-tmux-8d784c404340
6. tmux shortcuts & cheatsheet: https://gist.github.com/MohamedAlaa/2961058
7. Unix Nohup: Run a Command or Shell-Script Even after You Logout: http://linux.101hacks.com/unix/nohup-command/
8. nohup(1) - Linux man page: https://linux.die.net/man/1/nohup
9. nohup Execute Commands After You Exit From a Shell Prompt: https://www.cyberciti.biz/tips/nohup-execute-commands-after-you-exit-from-a-shell-prompt.html