# Assignment 2: Tesla Factory Production Line Control System

**Deadline: Dec. 05, 2018. You get 5 marks bonus if you submit on or before Nov. 30, 2018**

**Weighting: 13% of the course assessment**

**Full marks: 100 + 20 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 limit the resource usage
- solve producer and consumer problem

## Prerequisite

Before you start to work on this assignment, you should know the concepts of thread, semaphore, as well as deadlock. A review of lecture notes and workshop 3 slides is highly recommended, or you will have trouble dealing with this assignment. You are also required to be able to code in C(as the course requirement states).

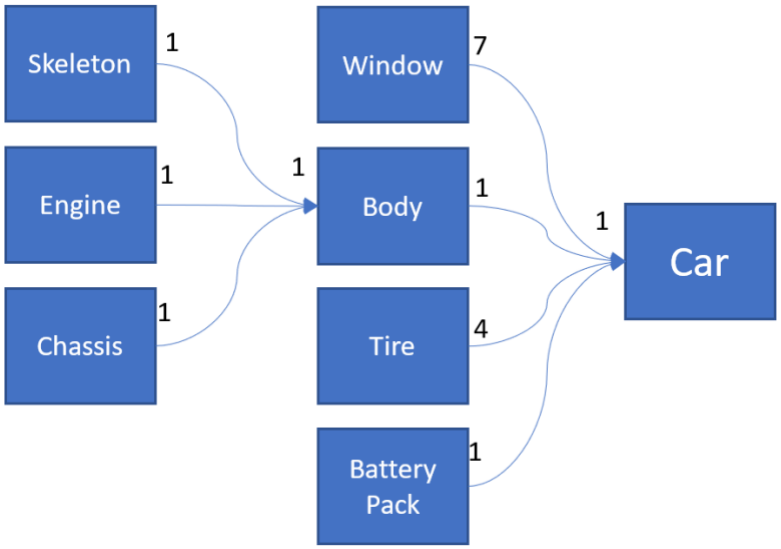
## Background Story

Tesla Motors is no doubt the leading company of electric cars and is also the first company who made electric cars so popular around the world. Tesla not only put a lot of high technologies in their cars but also how they manufacture electric cars. Tesla factory almost automated the whole production process by using a lot of smart trainable robot workers. Knowing the development of the world most advanced technology is even more important than just write this simple simulation program. 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

To simplify the process of manufacturing Tesla electric cars, 7 car parts need to be built 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:

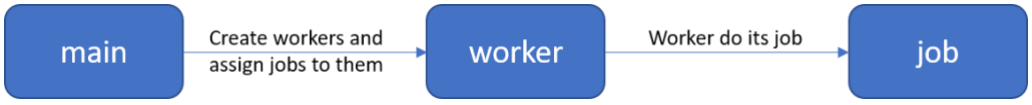



Those parts which no arrow is pointing at depend on their own raw materials. We just assume that they are ready by default. The production rule is: 1 skeleton, 1 engine, and 1 chassis make 1 car body; 7 windows, 1 body, 4 tires, and 1 battery pack make 1 car.

You will be given 9 files of this system. Below is a list of those files and their explanations:

| File name | Function |
|---|---|
| **definitions.h** | Defines system variables like production time. You are not allowed to change those variables |
| **main.h and main.c** | The main program, initiate factory status, manage and schedule workers, report results |
| **worker.h and worker.c** | Contains the worker(thread) function |
| **job.h and job.c** | Contains the manufacturing functions |

The relationships among these files:



## Implementation details

You need to also read the source code to fully understand the program. Only pivotal parts are introduced in this assignment sheet.

### Control of resources

Control of resources including number of robot workers, number of storage space in the factory as well as each car parts is implemented with counting semaphore. Initial values of worker and space are predefined while the initial value for each car parts is set to 0. The initiation process is done by the function ***initSem()***.

***main.c***:

```
 5 sem_t sem_worker;
 6 sem_t sem_space;
 7
 8 sem_t sem_skeleton;
 9 sem_t sem_engine;
10 sem_t sem_chassis;
11 sem_t sem_body;
12
13 sem_t sem_window;
14 sem_t sem_tire;
15 sem_t sem_battery;
16 sem_t sem_car;
17
18 int num_cars;
19 int num_spaces;
20 int num_workers;
```

```
151 int initSem(){
152 #if DEBUG
153         printf("Initiating semaphores...\n");
154 #endif
155         sem_init(&sem_worker,   0, num_workers);
156         sem_init(&sem_space,    0, num_spaces);
157
158         sem_init(&sem_skeleton, 0, 0);
159         sem_init(&sem_engine,   0, 0);
160         sem_init(&sem_chassis,  0, 0);
161         sem_init(&sem_body,     0, 0);
162
163         sem_init(&sem_window,   0, 0);
164         sem_init(&sem_tire,     0, 0);
165         sem_init(&sem_battery,  0, 0);
166         sem_init(&sem_car,      0, 0);
167 #if DEBUG
168         printf("Init semaphores done!\n");
169 #endif
170         return 0;
171 }
```

### Job assignment

Job is assigned to each robot worker(thread) via predefined struct ***work_pack***. Each work pack contains worker ID, job ID (defined in ***definition.h***), how many times should this worker do its job, and the resource package (semaphore package, struct *resource_pack*). You can find the above structures in ***work.h***:

```
 3 typedef struct resource_pack {
 4         int space_limit;
 5         int num_workers;
 6         sem_t *sem_space;
 7         sem_t *sem_worker;
 8
 9         sem_t *sem_skeleton;
10         sem_t *sem_engine;
11         sem_t *sem_chassis;
12         sem_t *sem_body;
13
14         sem_t *sem_window;
15         sem_t *sem_tire;
16         sem_t *sem_battery;
17         sem_t *sem_car;
18 } resource_pack;
```

```
20 typedef struct work_pack {
21         int tid;   // worker ID
22         int jid;   // job ID
23         int times; // how many times the job be run
24         resource_pack *resource;
25 } work_pack;
```

## Manufacture process

Since this is only a simulation of Tesla factory production line, there's no need to implement how these car parts are built or how a car is assembled. Instead those functions just sleep a certain amount of time for each production job. You can find these build functions in ***job.c***. Time lengths are defined in file ***definitions.h*** which you are not supposed to change. If a car part needs some other parts as its raw material like the car body, the worker must wait for the completion of building those parts. After a part of the car is built, the worker needs to put it in the storage space and notify its own semaphore that one piece has been made. Note that each part no matter what type it is will take only one unit of storage space and the final product electric car won't take any factory storage space as the car will be delivered to somewhere else.

Example of making car body in ***job.c***:

```
87 void makeBody(sem_t *sem_space, int space_limit, sem_t *sem_body,
88                 sem_t *sem_skeleton, sem_t *sem_engine, sem_t *sem_chassis) {
89         getItem(sem_space, space_limit, sem_skeleton);
90         getItem(sem_space, space_limit, sem_engine);
91         getItem(sem_space, space_limit, sem_chassis);
92         makeItem(sem_space, TIME_BODY, sem_body);
93 }
```

Get and make an item:

```
51 void makeItem(sem_t *space, int makeTime, sem_t* item) {
52         sleep(makeTime);
53         requestSpace(space);
54         sem_post(item);
55 }
56
57 void getItem(sem_t *space, int space_limit, sem_t *item) {
58         sem_wait(item);
59         releaseSpace(space, space_limit);
60 }
```

***definitions.h:*** you may change DEBUG to 1 to debug. The program will print more information to help you debug. Or you can just use gdb instead.

```
 6 #define DEBUG 0
 7
 8
 9 /*---------Do Not Change Anything Below This Line---------*/
10 //Job ID
11 #define SKELETON  0
12 #define ENGINE    1
13 #define CHASSIS   2
14 #define BODY      3
15 #define WINDOW    4
16 #define TIRE      5
17 #define BATTERY   6
18 #define CAR       7
19
20 /*-----Production time for each item-----*/
21 // Phase 1
22 #define TIME_SKELETON 5
23 #define TIME_ENGINE   4
24 #define TIME_CHASSIS  3
25 #define TIME_BODY     4
26
27 // Phase 2
28 #define TIME_WINDOW   1
29 #define TIME_TIRE     2
30 #define TIME_BATTERY  3
31 #define TIME_CAR      6
```

## Questions (90 marks + 20 bonus marks)

### Q1 Complete the single threaded 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 single threaded version and get familiar with the program.

Your tasks are:

1. In file ***job.c***, you should complete 2 functions: *makeBattery* and *makeCar*. Functions for making other parts have been implemented. You may go through those functions and have an idea of how they work before you implement these 2 functions.
2. Then you need to add lines to ***main.c*** to complete the rest of the program so that all parts will be made sequentially. To make it simple in this question, we just need to make 1 car. (15 marks for codding)
3. After you finish your code, you can compile your code by typing in command *make* in your Linux console (**makefile** has been provided). If there's no error, you can run your program by execute **./tesla_factory.out**.
4. Include a screenshot of your program. **Please add a line in *main.c* to print out your name and your university ID at the beginning of your program.** (5 marks for screenshot)

Since we only consider the situation that use one robot worker to make one car with sufficient space, these corresponding parameters will be pre-set in the main function.

Here's a sample screenshot of the output of question 1 (debug mode off):

```
Name: Elon Musk   UID: 6666688899
Job defined, 1 workers will build 1 cars with 20 storage spaces
-----Main: worker 0 doing 0...
-----Main: worker 0 doing 1...
-----Main: worker 0 doing 2...
-----Main: worker 0 doing 3...
-----Main: worker 0 doing 4...
-----Main: worker 0 doing 5...
-----Main: worker 0 doing 6...
-----Main: worker 0 doing 7...
=====Final report=====
Unused Skeleton: 0
Unused Engine: 0
Unused Chassis: 0
Unused Body: 0
Unused Window: 0
Unused Tire: 0
Unused Battery: 0
Production of 1 car done, production time: 40.001662 sec, space usage: 20
==========
```

## Q2 Implement a naïve multithreaded program (35 marks)

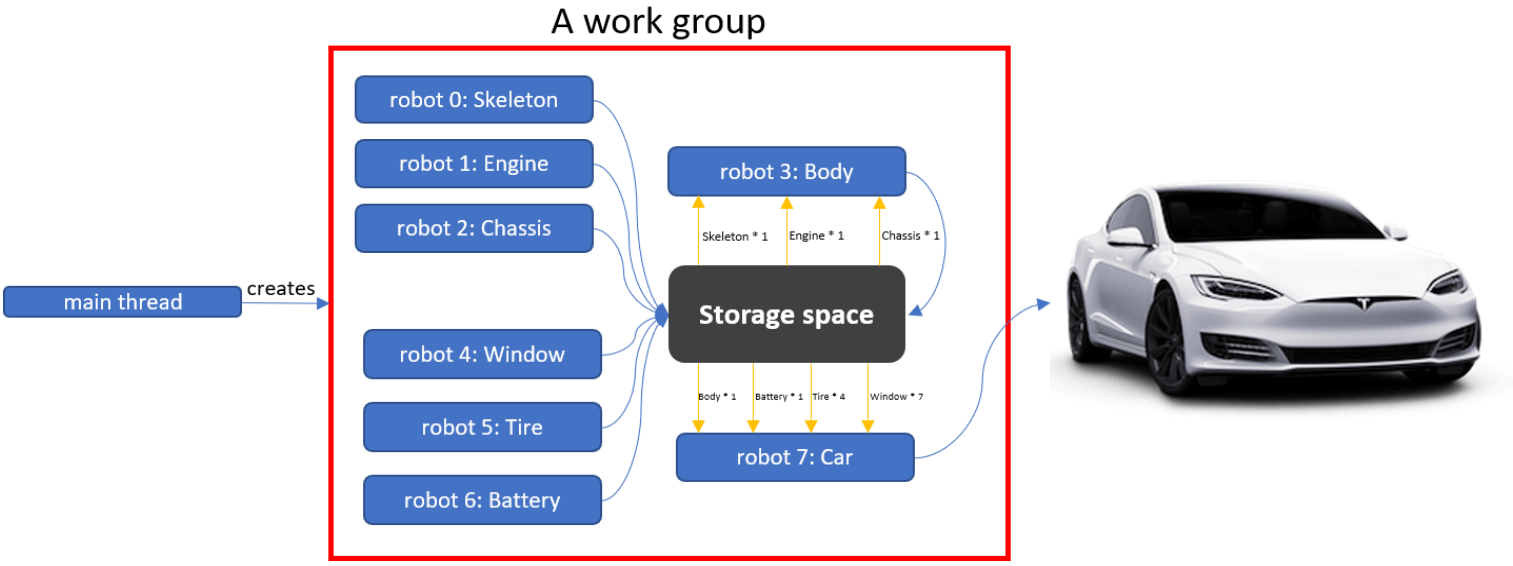
Now you have some basic ideas about how the program works. Let's copy the completed code from q1 to q2 and make it a simple multithreaded program that multiple workers will work simultaneously to speed up the production process. By the end of this question, your program should be able to produce more than one car with multiple threads and see speed-up from multithreading.

***main.c:***

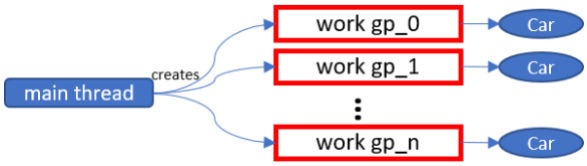
Number of cars to be made, number of storage space and number of workers are passed to the main function as parameters for the ease of testing later.

```
24        /*---------- For future use-------
25          if (argc < 4) {
26          printf("Usage: %s <number of cars> <number of spaces> <number of workers>\n",
27          argv[0]);
28          return EXIT_SUCCESS;
29          }
30          num_cars      = atoi(argv[1]);
31          num_spaces    = atoi(argv[2]);
32          num_workers   = atoi(argv[3]);                                    Enable
33          --------------------------------*/
34
35        // We only make one car with 1 thread and sufficient storage spaces
36          num_cars      = 1;
37          num_spaces    = 20;                    Delete
38          num_workers   = 1;
39          ////////////////////////////////////////////////////////
```

You may start with creating 8 threads to build 1 car and each thread corresponds to 1 job. Simply speaking, just assign these 8 jobs from q1 to 8 threads. You need to use *sem_worker* to keep track of how many workers are created and running at the same time. Make sure that the number doesn't exceed the limit of workers defined by *num_workers*. You also need to change the normal *work* function in **worker.c** to a thread start routine so that you can pass it as an argument to *pthread_create()* (10 marks for being able to run 8-thread version).



Once your 8-thread version works fine, you need to extend your code so that it will work with multiples of 8 threads. You may consider 8 workers as a work group. They work together simultaneously producing Tesla electric cars. Your program should have the ability to produce more than 1 car at this stage (20 marks).




You need to include a screenshot of you testing your code with different parameters by executing **run.sh** (5 marks). Sample output is shown below (debug mode disabled), you should see speedup as the number of work groups increases:

```
Name: Elon Musk   UID: 6666688899
Job defined, 8 workers will build 1 cars with 40 storage spaces
=====Final report=====
Unused Skeleton: 0
Unused Engine: 0
Unused Chassis: 0
Unused Body: 0
Unused Window: 0
Unused Tire: 0
Unused Battery: 0
Production of 1 car done, production time: 15.000558 sec, space usage: 40
==========
Name: Elon Musk   UID: 6666688899
Job defined, 8 workers will build 4 cars with 40 storage spaces
=====Final report=====
Unused Skeleton: 0
Unused Engine: 0
Unused Chassis: 0
Unused Body: 0
Unused Window: 0
Unused Tire: 0
Unused Battery: 0
Production of 4 cars done, production time: 60.002282 sec, space usage: 40
==========
Name: Elon Musk   UID: 6666688899
Job defined, 16 workers will build 4 cars with 40 storage spaces
=====Final report=====
Unused Skeleton: 0
Unused Engine: 0
Unused Chassis: 0
Unused Body: 0
Unused Window: 0
Unused Tire: 0
Unused Battery: 0
Production of 4 cars done, production time: 30.001211 sec, space usage: 40
==========
Name: Elon Musk   UID: 6666688899
Job defined, 32 workers will build 4 cars with 40 storage spaces
=====Final report=====
Unused Skeleton: 0
Unused Engine: 0
Unused Chassis: 0
Unused Body: 0
Unused Window: 0
Unused Tire: 0
Unused Battery: 0
Production of 4 cars done, production time: 15.001030 sec, space usage: 40
==========
```

8 workers take only 15 seconds (vs. 40 seconds in q1) to make a car

For making 4 cars, if we double the number of workers, the production time reduce by half. Good scalability for multiples of 8 threads.

## Q3 Implement your own job assignment scheme (35 marks + 20 bonus marks)

You may find that the naïve implementation in question 2 has some problems:

1. Naïve implementation may produce **wasted parts** that are not necessary.
2. Storage space is not considered as we just assign a large number of storage space making sure that it completes the whole process. **Deadlock** is not handled.

You should test many combinations of parameters see if your program has these problems above. Here are some sample outputs (your program may have different implementation, and some problems are not triggered by the same parameters below):

Example 1

Making 2 cars with 40 units of storage space and 13 workers: wasted parts

```
> ./tesla_factory.out 2 40 13
Name: Elon Musk   UID: 6666688899
Job defined, 13 workers will build 2 cars with 40 storage spaces
=====Final report=====
Unused Skeleton: 0
Unused Engine: 0
Unused Chassis: 0
Unused Body  2
Unused Window: 14
Unused Tire  0
Unused Battery: 0
There are waste car parts!
Production of 2 cars done, production time: 29.001668 sec, space usage: 40
==========
```

Example 2

Making 1 car with 1 unit of storage space and 3 workers: deadlock (debug on)

```
> ./tesla_factory.out 1 1 3
Name: Elon Musk   UID: 6666688899
Job defined, 3 workers will build 1 cars with 1 storage spaces
Initiating semaphores...
Init semaphores done!
Worker[0]: working on job 0 for 1 time...
Worker[1]: working on job 1 for 1 time...
Worker[2]: working on job 2 for 1 time...
Requesting free space, current space=1...
Space requested, current space=0...
Worker[2]: One chassis made!
Worker[2]: job 2 done!
Requesting free space, current space=0...
Requesting free space, current space=0...
```

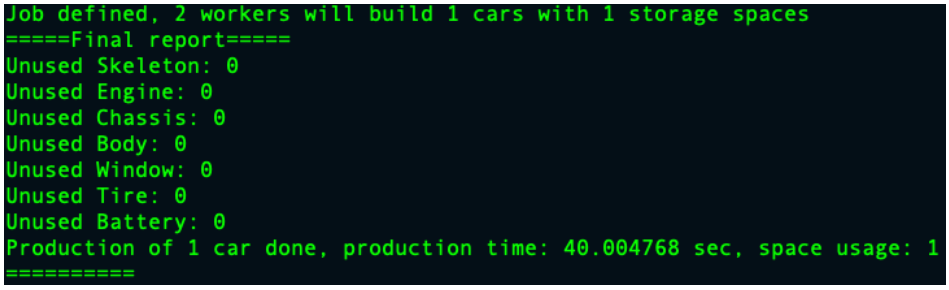
One worker placed a part that no other worker needs… Other workers can’t produce anything because there’s no space left. Deadlock appears.

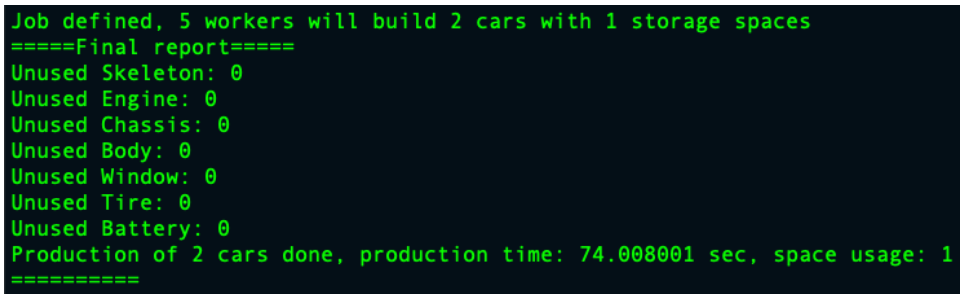
Your task is to tackle these problems above with your own dynamic job assignment implementation. You should write down your detailed implementation introduction in your report and run benchmark experiments with your program.

Requirements:

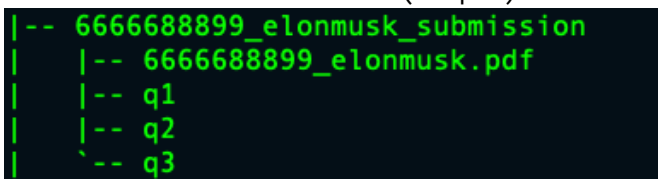
1. Your program should accept any numbers of workers to produce different number of cars. You should test your program with many combinations of input parameters (number of cars, number of spaces, number of workers) and make sure your program is bug free and easy to read. You need to run your program multiple times with different number of threads and collect their running time, then draw a graph to show the **scalability** of your program with the number of workers and run time. Put the graph into your report. You should analyse your output and explain the results you have. Other graphs like showing the relationship between producing different number of cars and the given number of threads are also welcome. Make sure your implementation won't create any wasted parts. (20 marks for no-wasted-part design, 15 marks for scalability analysis)
2. Bonus: Your program should also be able to handle deadlock when given small number of storage space. You may include some screenshots to help you prove that without your deadlock handling code there's deadlock and by adding your deadlock handling code there's no deadlock. Clearly introduce your deadlock handling algorithm in your report (10 bonus marks for program, 10 bonus marks for explanation and proof). There're some sample test cases in ***run.sh***, you are welcome to add more to test your code.
   Here are some hints:
   a. Deadlock detection: you don't know whether you will encounter deadlock or not ahead. But once your program detects that certain threads runs for unreasonable amount of time, you know that deadlock happens. Then you figure out a way to break the deadlock so that the production process can move on. *sem_trywait()* or *sem_timedwait()* may be useful here.
   b. Deadlock prevention: once the production goal is set, you know the number of each part to achieve the goal. You also know how many unit of space you have. Your program may analyse these data and assign jobs to workers properly so that deadlock will not happen for sure.

Random parameter combinations will be generated when marking Q3. Some sample screenshots with extreme test cases:



## General requirements (10 marks)

1. 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 as follows: <university ID>_<name>.pdf. Say your university ID is 6666688899 and your name is Elon Musk, so your report name is 6666688899_elonmusk.pdf. Then put q1, q2, q3 and your report in a directory named <university ID>_<name>_submission, then zip it to a zip file <university ID>_<name>_submission.zip (5 marks).
   Submission folder structure(sample):
   
   
2. Code quality: use meaningful variable names and function names, add necessary inline comments if possible to help others read your code (3 marks).
3. Report quality: we don't expect a long report. Express your idea briefly and logically by any means (graphs, charts, tables, etc.). Good structure, no obvious mistakes (2 marks).

**Make sure that your code for each question can be compiled and run without problem, or you will get 0 marks 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.

**Reminder**: Start working on your assignment ASAP. Try to avoid testing your code during the busy hours. You are recommended to write code and debug your program on workbench directly. If you want to keep your program running even if you terminate your ssh session on your local machine, here are two recommended command in Linux you can learn by yourself: *nohup* and *tmux*. There are many tutorials on Google and YouTube. Before you use nohup, make sure you know how to check your job (cmd: ***job***) and kill your program (cmd: ***kill***).

Google, Stack Overflow and GitHub are good places to help solve your problems. Try to find out solutions by yourself before you bring them to teaching stuffs. Some questions are never mean to be asked:

1. Ask for answers to compare with your own code or check your answer with TA before you submit it
2. Ask other people to implement your idea/algorithm, you should do your assignment completely by your own
3. Questions related to program in C. This is not a programming course, you should have known C before taking this course(course prerequisite). 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.
5. Solution or suggestion for bonus questions, hints have been given.

Be careful with your program resource usage. Use semaphores to control the creation of threads. Do not create many useless threads (there's limit in Linux of the number of running thread) or you will get 0 mark.

## 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. National Geographic: Tesla Motors Documentary (50:05)
5. A Gentle Introduction to tmux: https://hackernoon.com/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
10. vim + tmux - OMG!Code (1:17:20)