Assignment 2

Tesla Factory Production Line

Details of assignment 2 please refer to Assignment 2 document

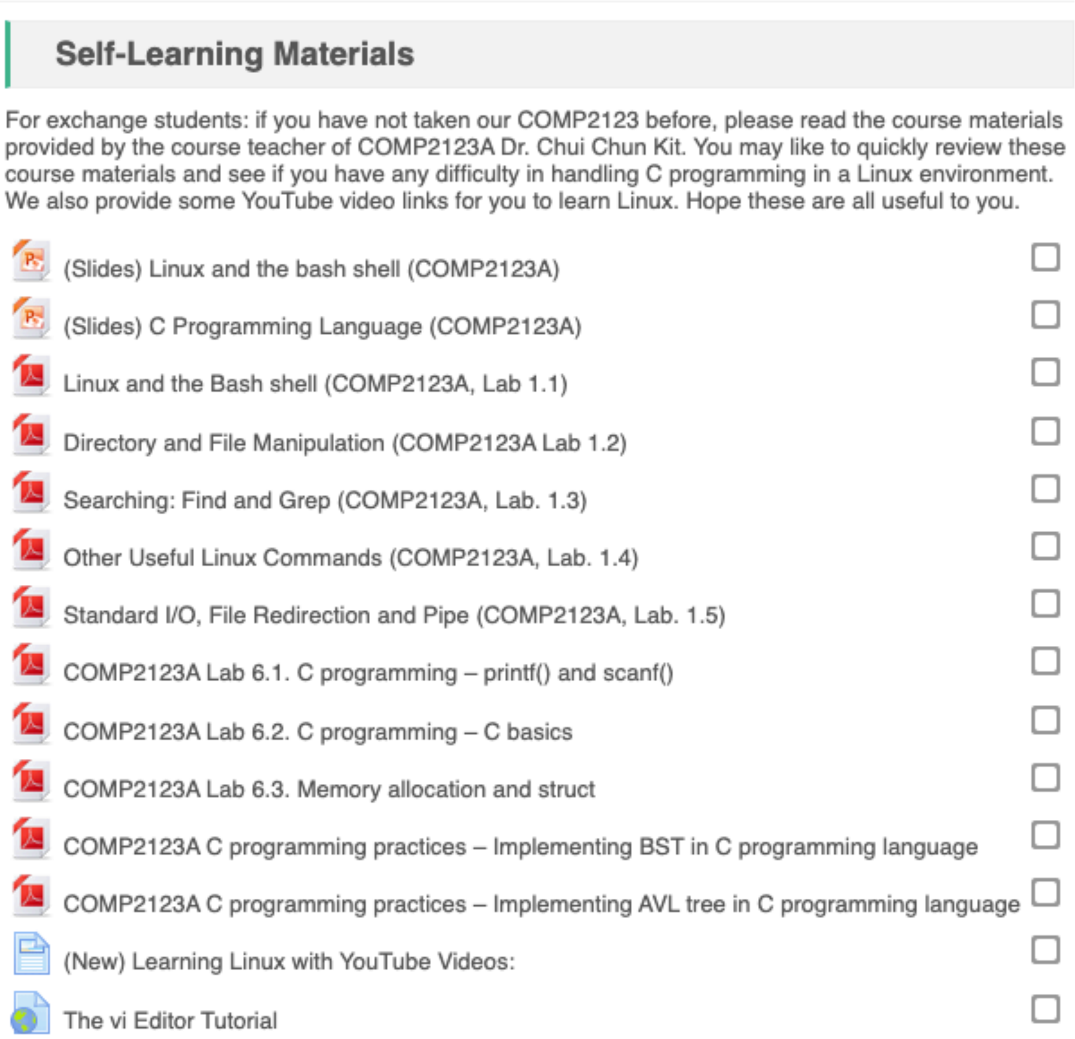
Objectives

- 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

Prerequisites

- Program in C (prerequisite of this course)
  - Review Tutorial 1
  - Self-learning materials on Moodle

- Tutorial 3
  - Multithread programming with Pthread
  - Thread synchronization with Semaphore



Self-Learning Materials

For exchange students: if you have not taken our COMP2123 before, please read the course materials provided by the course teacher of COMP2123A Dr. Chui Chun Kit. You may like to quickly review these course materials and see if you have any difficulty in handling C programming in a Linux environment. We also provide some YouTube video links for you to learn Linux. Hope these are all useful to you.

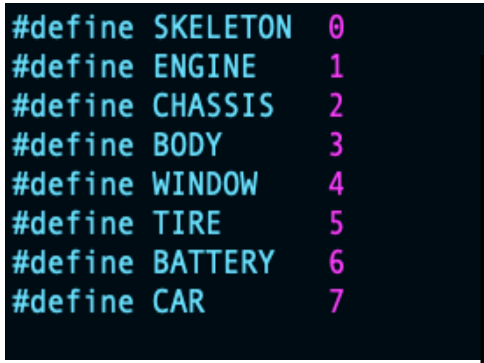
- (Slides) Linux and the bash shell (COMP2123A) □
- (Slides) C Programming Language (COMP2123A) □
- Linux and the Bash shell (COMP2123A, Lab 1.1) □
- Directory and File Manipulation (COMP2123A Lab 1.2) □
- Searching: Find and Grep (COMP2123A, Lab. 1.3) □
- Other Useful Linux Commands (COMP2123A, Lab. 1.4) □
- Standard I/O, File Redirection and Pipe (COMP2123A, Lab. 1.5) □
- COMP2123A Lab 6.1. C programming – printf() and scanf() □
- COMP2123A Lab 6.2. C programming – C basics □
- COMP2123A Lab 6.3. Memory allocation and struct □
- COMP2123A C programming practices – Implementing BST in C programming language □
- COMP2123A C programming practices – Implementing AVL tree in C programming language □
- (New) Learning Linux with YouTube Videos: □
- The vi Editor Tutorial □

Background Story

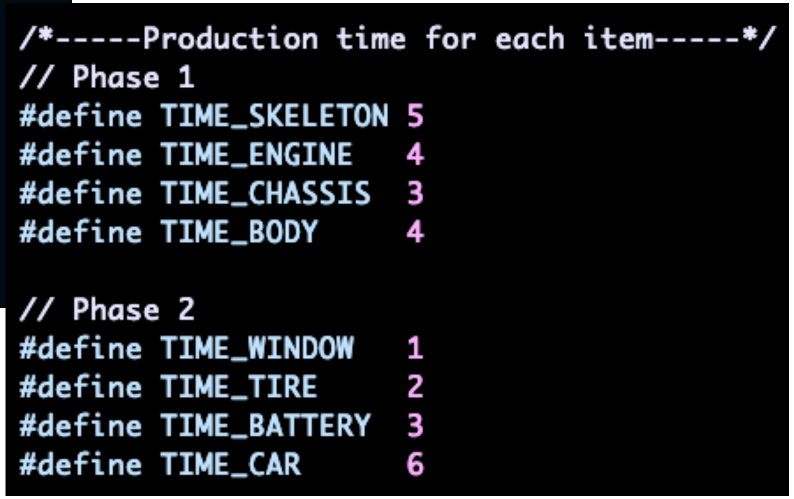
- Tesla Factory
  - Automated producing process with Robots

- YouTube Videos
  - How the Tesla Model S is Made | Tesla Motors Part 1 (4:54)
  - How Tesla Builds Electric Cars | Tesla Motors Part 2 (3:25)
  - Electric Car Quality Tests | Tesla Motors Part 3 (1:49)
  - National Geographic: Tesla Motors Documentary (50:05)

System Overview

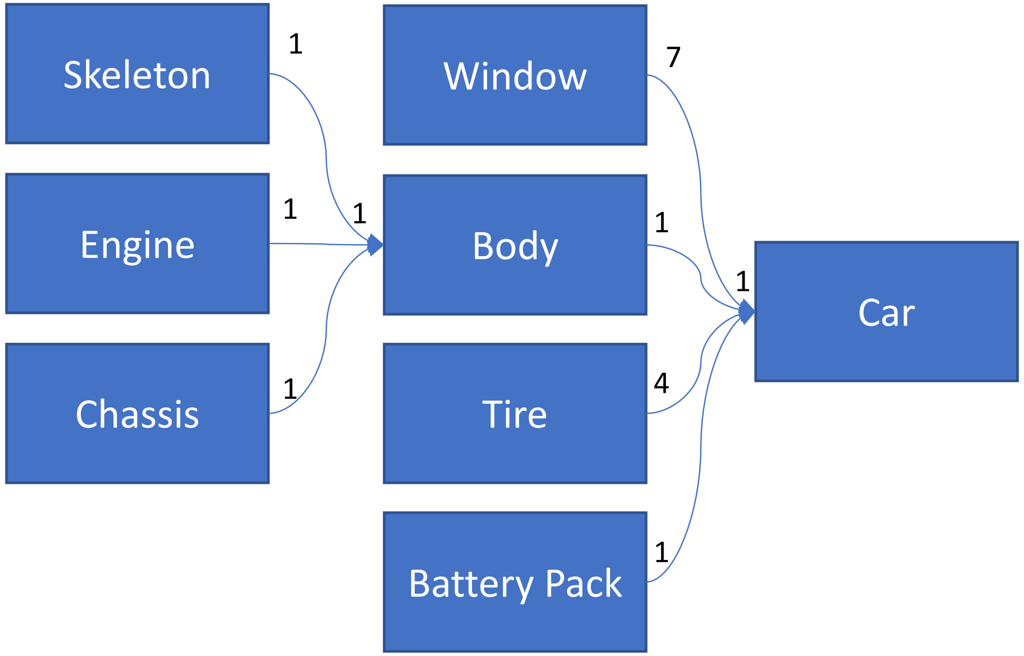


```
#define SKELETON  0
#define ENGINE    1
#define CHASSIS   2
#define BODY      3
#define WINDOW    4
#define TIRE      5
#define BATTERY   6
#define CAR       7
```



```
/*-----Production time for each item-----*/
// Phase 1
#define TIME_SKELETON 5
#define TIME_ENGINE   4
#define TIME_CHASSIS  3
#define TIME_BODY     4

// Phase 2
#define TIME_WINDOW   1
#define TIME_TIRE     2
#define TIME_BATTERY  3
#define TIME_CAR      6
```

- Simplified manufacturing process
  - 7 car parts need to be built for making a car
    - 1 skeleton
    - 1 engine
    - 1 chassis
    - 1 car body
    - 7 windows
    - 1 body
    - 4 tires
    - 1 battery pack

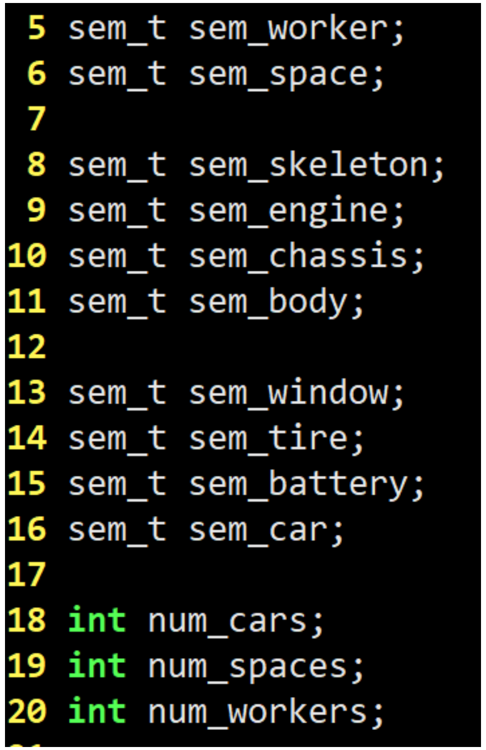


Source Code Files

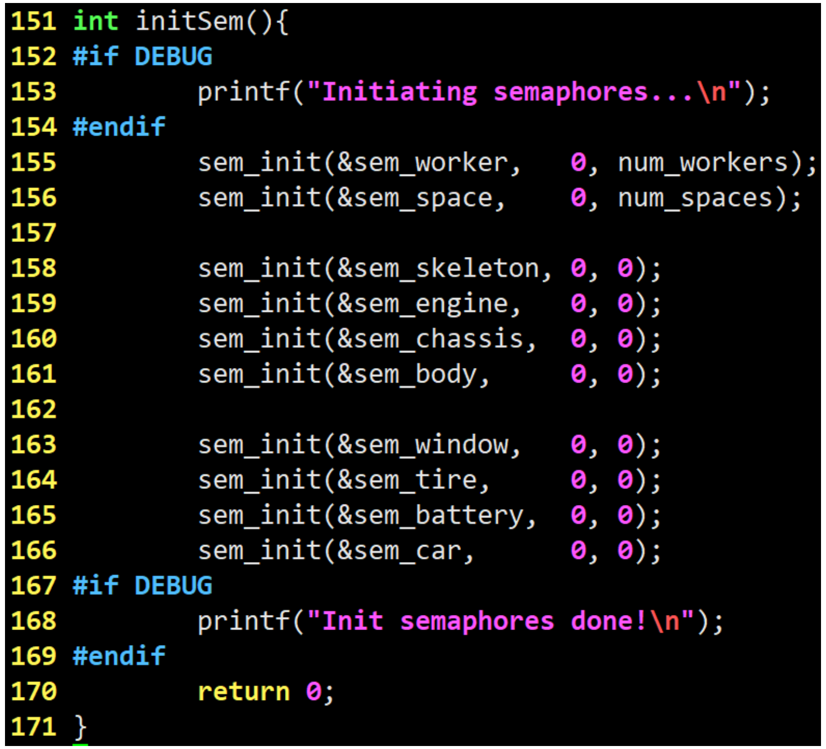
File name	Function
definitions.h	Defines system variables like production time. You are not allowed to change those variables
main.h/c	The main program, initiate factory status, manage and schedule workers, report results
worker.h/c	Contains the worker(thread) functions
job.h/c	Contains the manufacturing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

Implementation Details

- Control of resources
  - Semaphores are used to keep track of all resources and produced parts

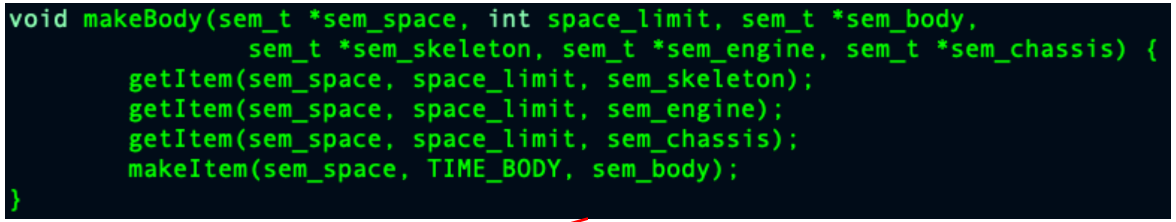


```
sem_t sem_worker;
sem_t sem_space;

sem_t sem_skeleton;
sem_t sem_engine;
sem_t sem_chassis;
sem_t sem_body;

sem_t sem_window;
sem_t sem_tire;
sem_t sem_battery;
sem_t sem_car;

int num_cars;
int num_spaces;
int num_workers;
```



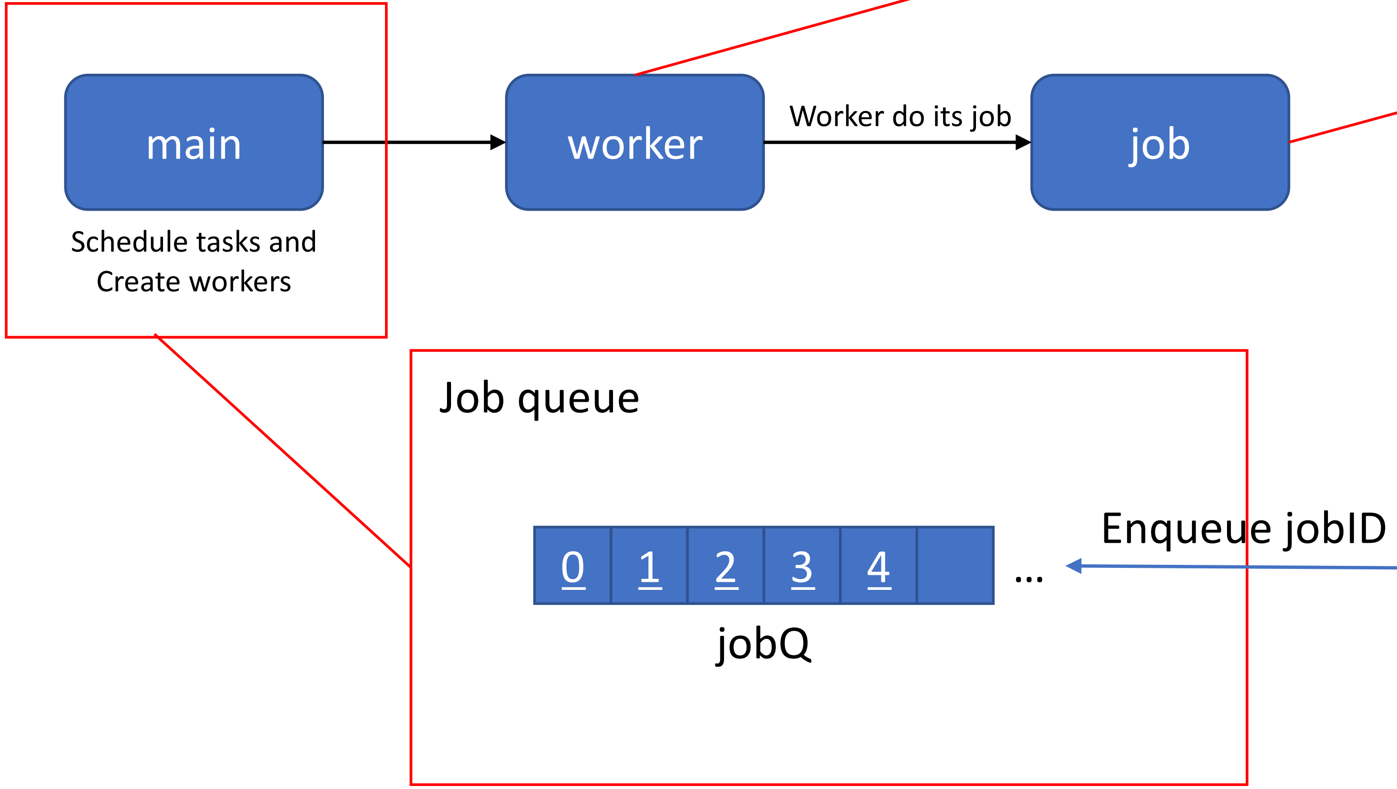
```
int initSem(){
#if DEBUG
        printf("Initiating semaphores...\n");
#endif
        sem_init(&sem_worker,   0, num_workers);
        sem_init(&sem_space,    0, num_spaces);

        sem_init(&sem_skeleton, 0, 0);
        sem_init(&sem_engine,   0, 0);
        sem_init(&sem_chassis,  0, 0);
        sem_init(&sem_body,     0, 0);

        sem_init(&sem_window,   0, 0);
        sem_init(&sem_tire,     0, 0);
        sem_init(&sem_battery,  0, 0);
        sem_init(&sem_car,      0, 0);
#if DEBUG
        printf("Init semaphores done!\n");
#endif
        return 0;
}
```

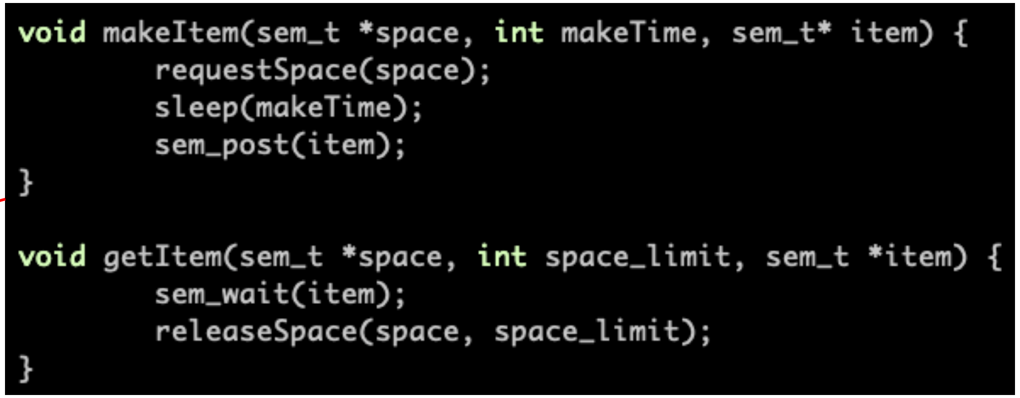
Implementation Details



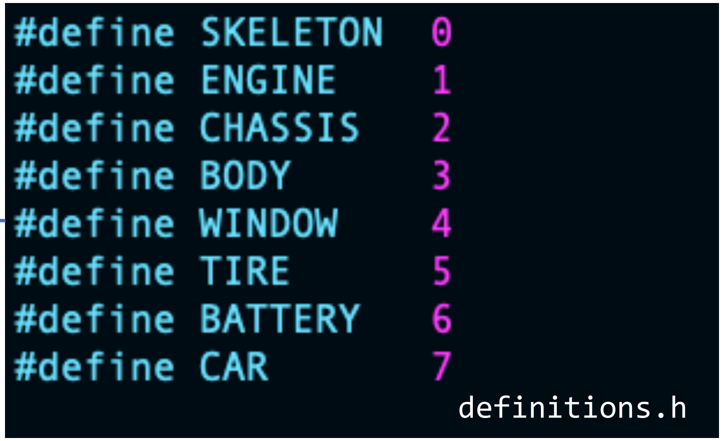
```
void makeBody(sem_t *sem_space, int space_limit, sem_t *sem_body,
                sem_t *sem_skeleton, sem_t *sem_engine, sem_t *sem_chassis) {
        getItem(sem_space, space_limit, sem_skeleton);
        getItem(sem_space, space_limit, sem_engine);
        getItem(sem_space, space_limit, sem_chassis);
        makeItem(sem_space, TIME_BODY, sem_body);
}
```

- Manufacture process



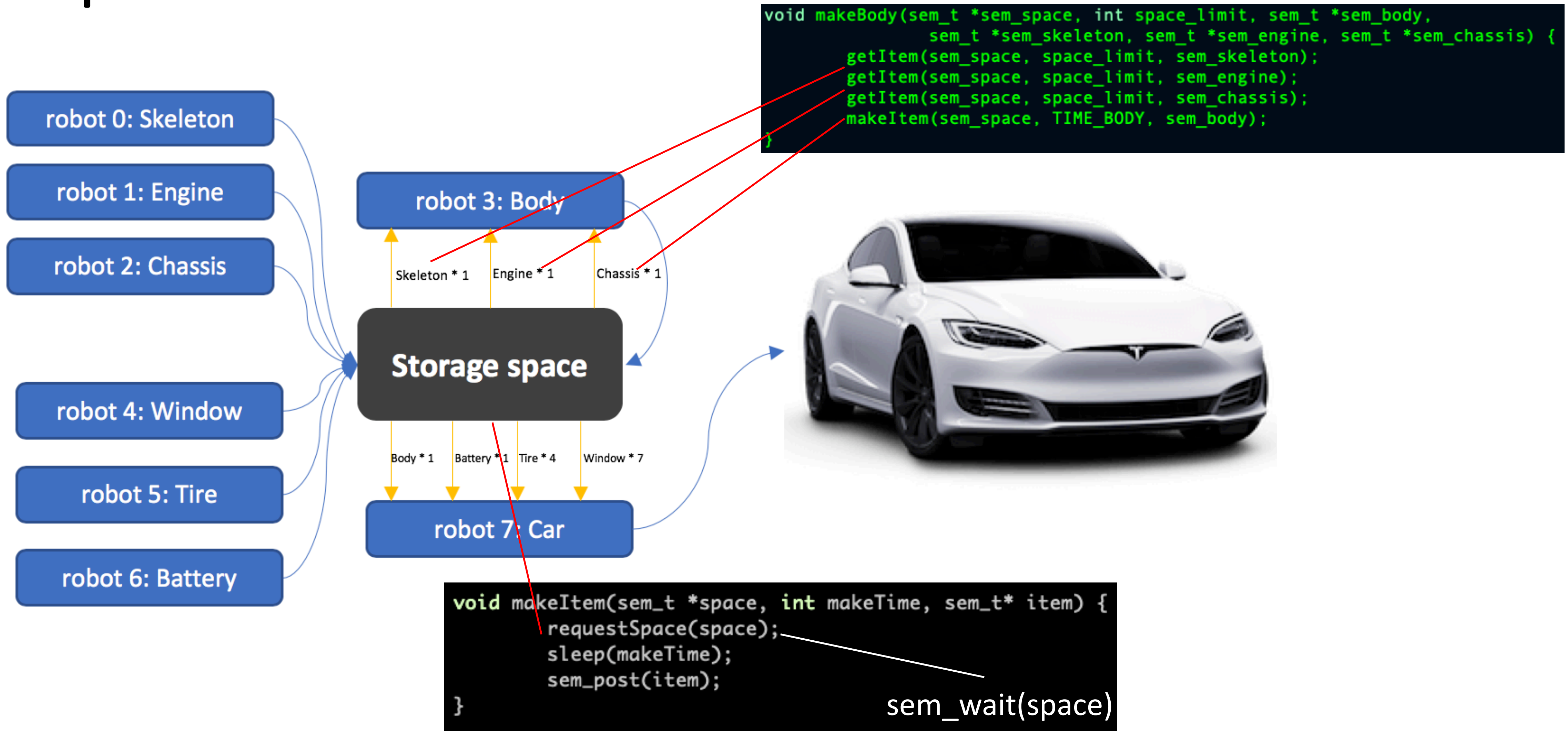


```
void makeItem(sem_t *space, int makeTime, sem_t* item) {
        requestSpace(space);
        sleep(makeTime);
        sem_post(item);
}

void getItem(sem_t *space, int space_limit, sem_t *item) {
        sem_wait(item);
        releaseSpace(space, space_limit);
}
```



```
#define SKELETON   0
#define ENGINE     1
#define CHASSIS    2
#define BODY       3
#define WINDOW     4
#define TIRE       5
#define BATTERY    6
#define CAR        7
                    definitions.h
```

Implementation Details



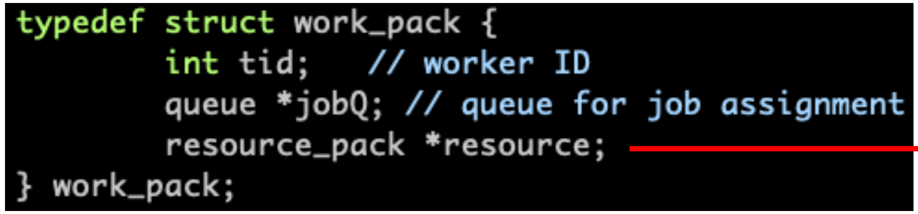
```
void makeBody(sem_t *sem_space, int space_limit, sem_t *sem_body,
                sem_t *sem_skeleton, sem_t *sem_engine, sem_t *sem_chassis) {
        getItem(sem_space, space_limit, sem_skeleton);
        getItem(sem_space, space_limit, sem_engine);
        getItem(sem_space, space_limit, sem_chassis);
        makeItem(sem_space, TIME_BODY, sem_body);
}
```

```
void makeItem(sem_t *space, int makeTime, sem_t* item) {
        requestSpace(space);
        sleep(makeTime);
        sem_post(item);
}
```

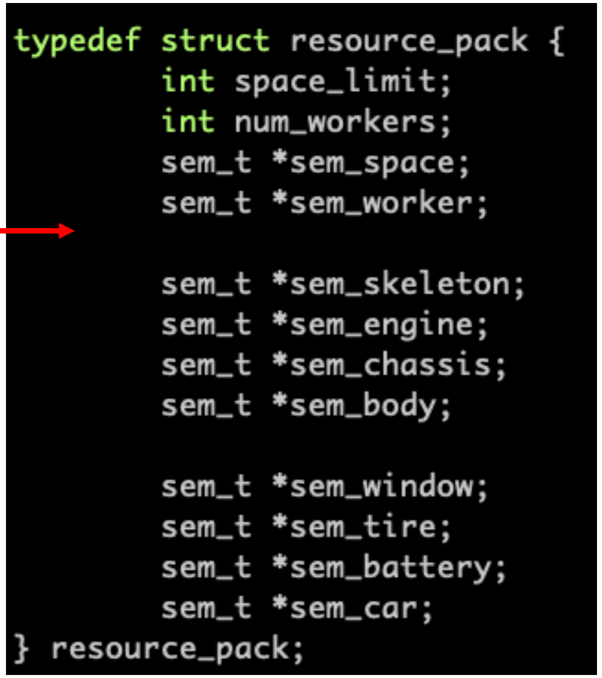
sem_wait(space)

Implementation Details

- Worker thread creation
  - work_pack: pass to worker threads when calling pthread_create()
  - resource_pack: a package of resource semaphores



```
typedef struct work_pack {
        int tid;    // worker ID
        queue *jobQ; // queue for job assignment
        resource_pack *resource;
} work_pack;
```



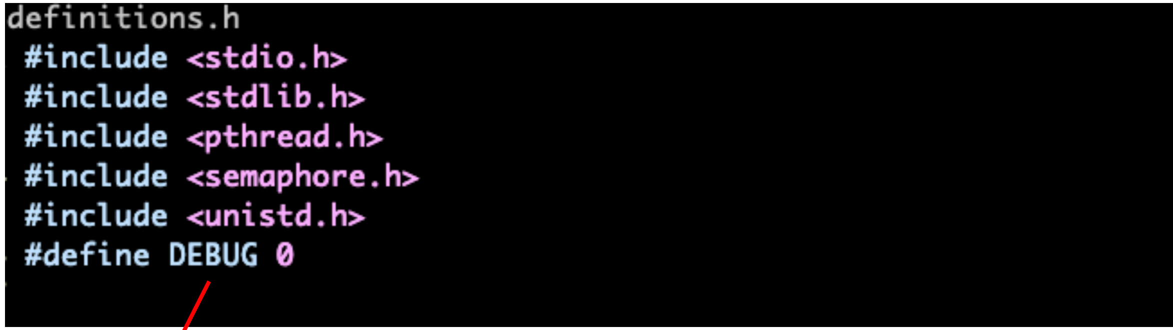
```
typedef struct resource_pack {
        int space_limit;
        int num_workers;
        sem_t *sem_space;
        sem_t *sem_worker;

        sem_t *sem_skeleton;
        sem_t *sem_engine;
        sem_t *sem_chassis;
        sem_t *sem_body;

        sem_t *sem_window;
        sem_t *sem_tire;
        sem_t *sem_battery;
        sem_t *sem_car;
} resource_pack;
```

Assignment 2 Questions

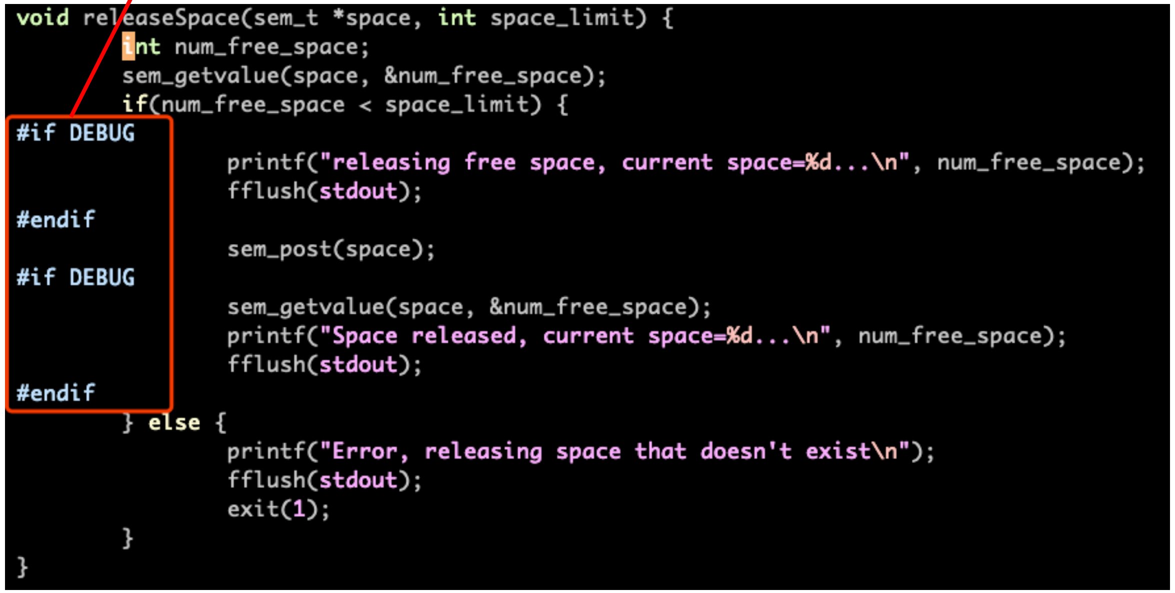
- Q1 Complete the single threaded version
- Q2 Implement a naïve multithreaded program
  - Q2.1 Implement Thread-safe queue
  - Q2.2 Multithreaded production
- Q3 Make it stable, make it run fast

Debug

- gdb debug
  - https://sourceware.org/gdb/onlinedocs/gdb/Threads.html
  - Google “gdb multiple threads”
- printf debug
  - Change DEBUG back to 0 before submit
  - Add more printf() if you need, remove them before submit. Or you can put them into #if DEBUG … #endif so that it won’t print out when debug mode is disabled



```
definitions.h
 #include <stdio.h>
 #include <stdlib.h>
 #include <pthread.h>
 #include <semaphore.h>
 #include <unistd.h>
 #define DEBUG 0
```

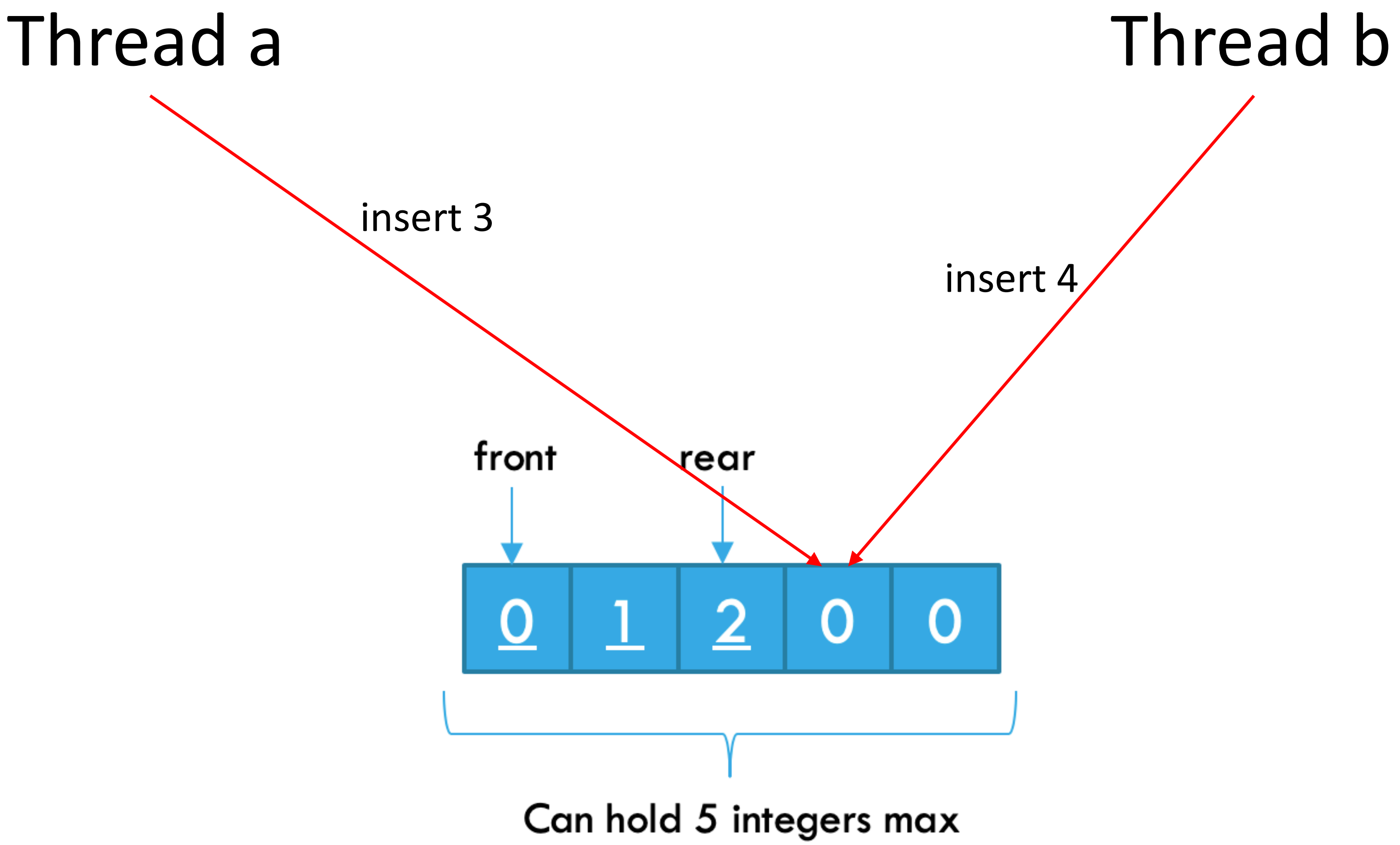


```
void releaseSpace(sem_t *space, int space_limit) {
        int num_free_space;
        sem_getvalue(space, &num_free_space);
        if(num_free_space < space_limit) {
#if DEBUG
                printf("releasing free space, current space=%d...\n", num_free_space);
                fflush(stdout);

#endif
                sem_post(space);

#if DEBUG
                sem_getvalue(space, &num_free_space);
                printf("Space released, current space=%d...\n", num_free_space);
                fflush(stdout);

#endif
        } else {
                printf("Error, releasing space that doesn't exist\n");
                fflush(stdout);
                exit(1);
        }
}
```

Q1. Complete Single Thread Version

- Get Familiar with the program
- Tasks
  1. Copy the **queue.c** and **queue.h** from your first tutorial exercise to directory q1.
  2. In file ***job.c***, you should complete 2 functions: *makeBattery* and *makeCar*.
  3. Then you need to add lines to ***main.c*** to complete the rest of the program so that all parts will be made sequentially.
     - There are 2 phases: Task scheduling and production. You need to finish both parts. To make it simple for this question, only one car will be made. (15 marks for coding)
  4. After you finish your code, you can compile your code by typing in command <u>*make*</u> in your Linux console (**makefile** has been provided). If there's no error, you can run your program by executing **./tesla_factory.out**.
  5. Include a screenshot of your program. **Please add a line in *main.c* to print out your own name and your university ID at the beginning of your program**. (5 marks for screenshot)

Queue from Tutorial 1



Thread a

Thread b

insert 3

insert 4

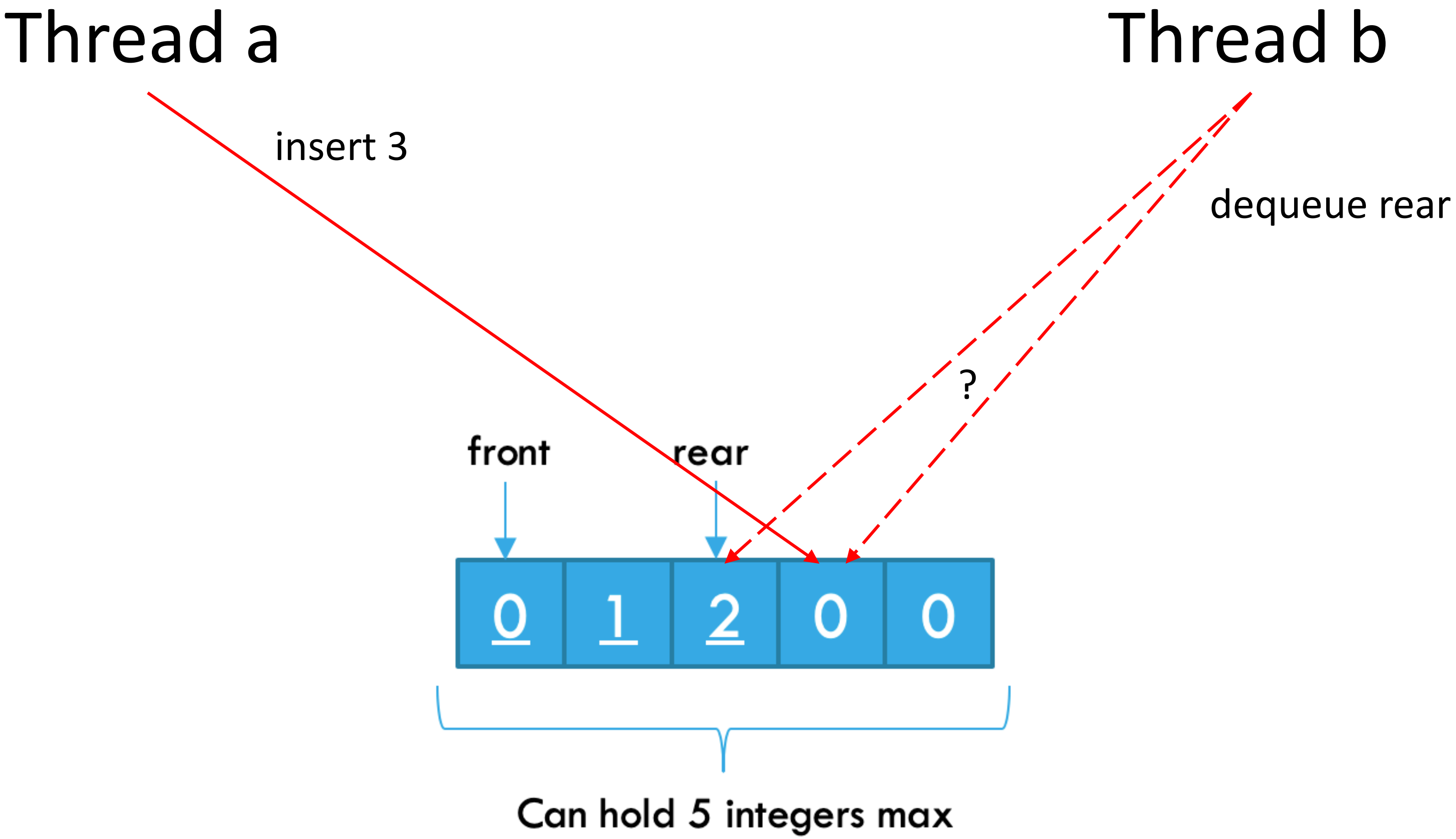
front

rear

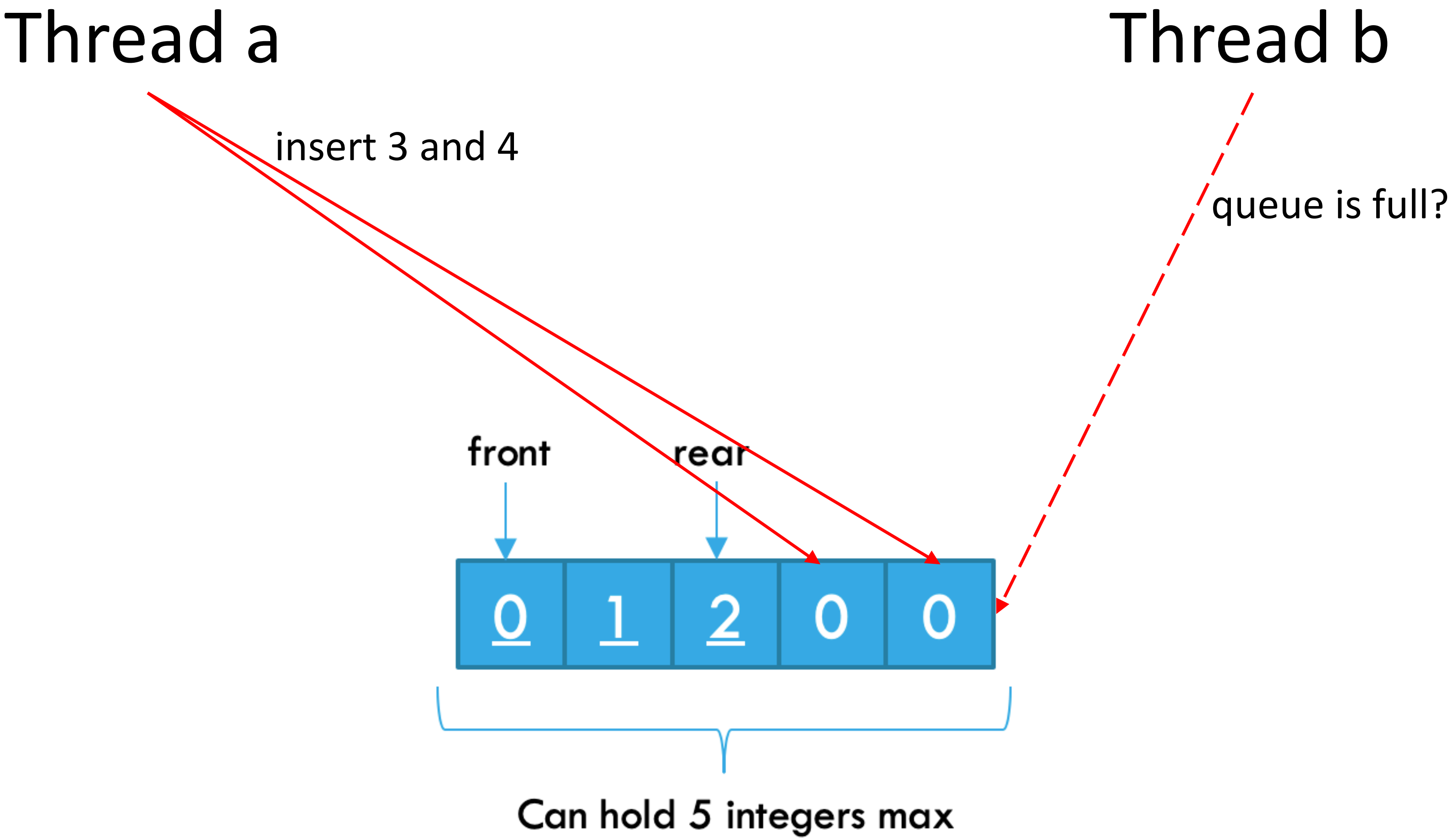
0 1 2 0 0

Can hold 5 integers max

Queue from Tutorial 1



Queue from Tutorial 1



Q2.1 Implement Thread-safe queue

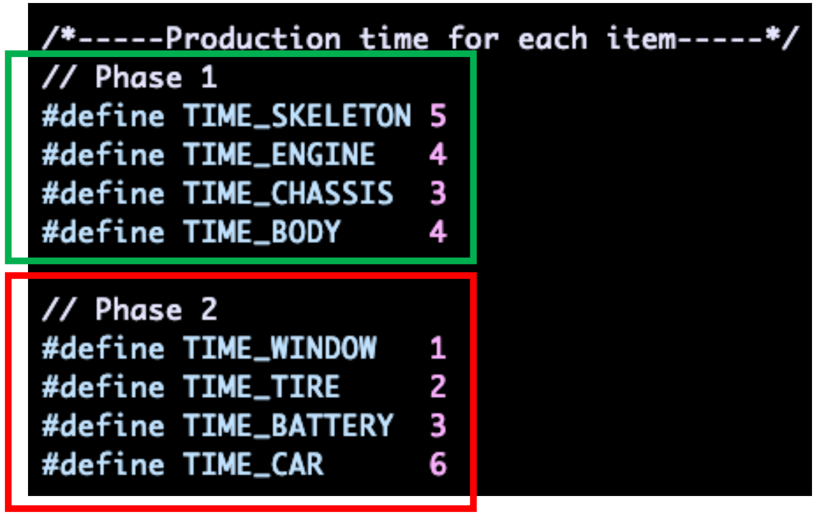
- Make your queue thread safe with **semaphore**
- **Test enqueue**
  - N threads will be created, and each thread will enqueue number ‘1’ to the queue. When all threads are done with enqueuing, sum up all the elements in the queue. If it’s thread safe, all threads can successfully enqueue and the value of sum should be equal to the number of thread N.
- **Test dequeue (front/rear)**
  - First enqueue N elements in the queue. Then launch N threads and each will dequeue once from the queue. If the queue is empty after being dequeued N times, then the dequeue function is thread safe.

Q2.2 Implement a naïve multithreaded

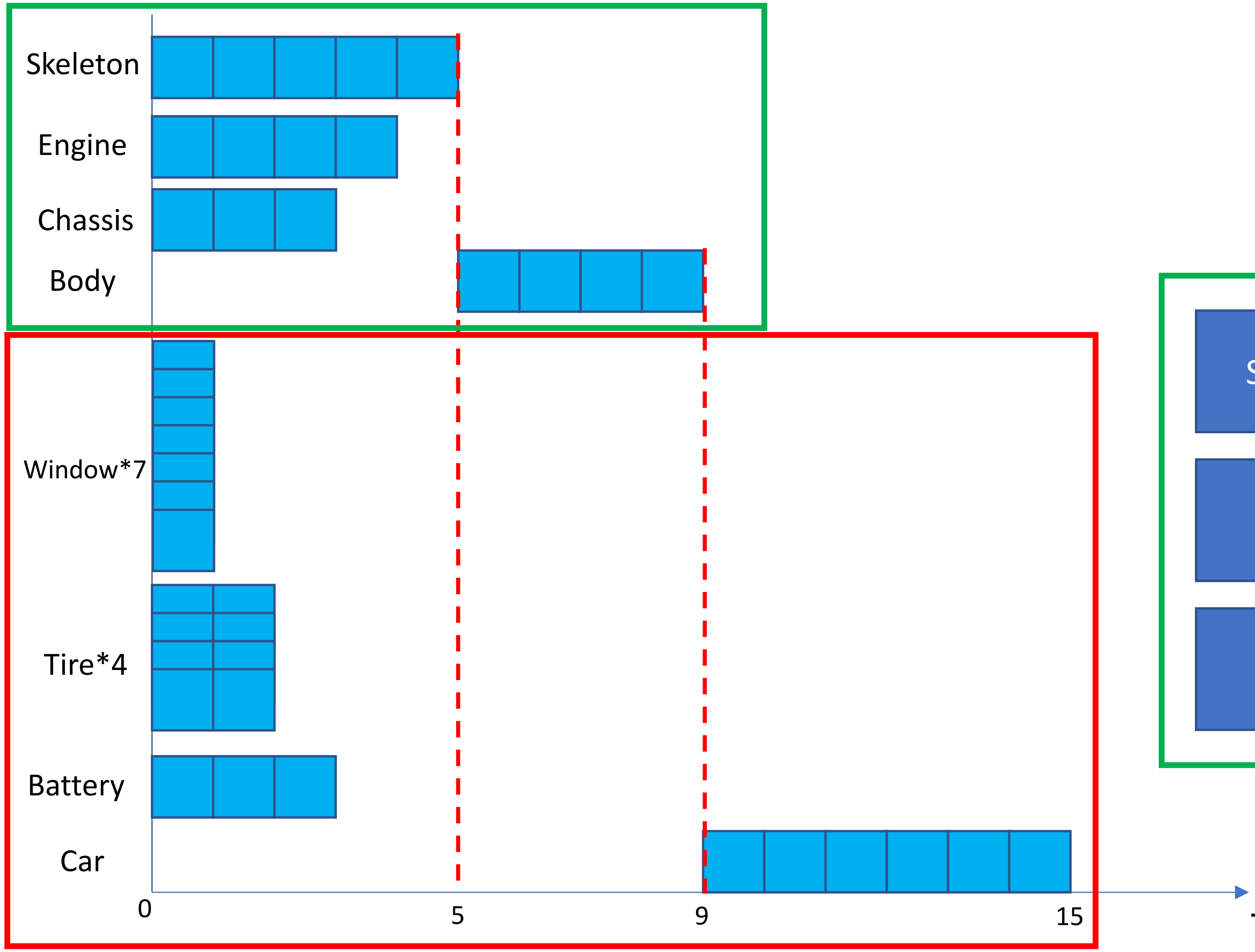
- Assume we have infinite space storage
- Copy the completed code from q1 to q2 and your thread-safe queue from q2_queue to q2
- Multiple workers will work simultaneously to speed up the production process

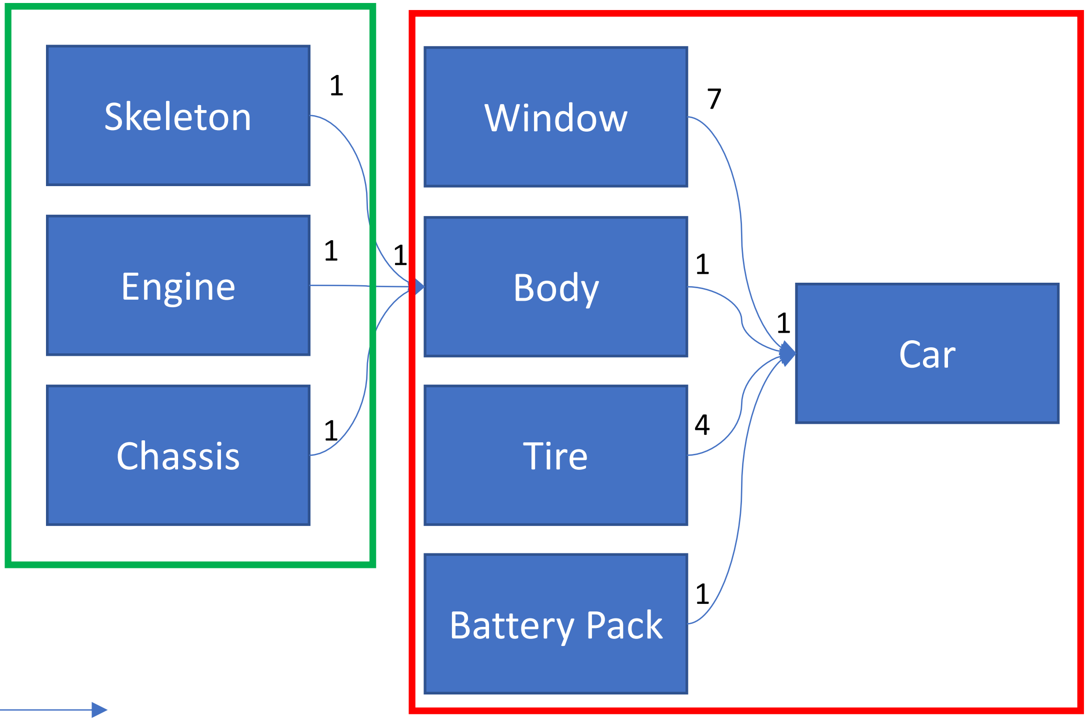
Theoretical Upper Limit (1 car)

No limit on number of workers and storage space



```
/*-----Production time for each item-----*/
// Phase 1
#define TIME_SKELETON 5
#define TIME_ENGINE   4
#define TIME_CHASSIS  3
#define TIME_BODY     4

// Phase 2
#define TIME_WINDOW   1
#define TIME_TIRE     2
#define TIME_BATTERY  3
#define TIME_CAR      6
```





Q3 Make it stable, make it run fast

- In Q2, there’s no limitation on the storage space.
- What if we take storage space into consideration…
- Example: 2 workers, 1 unit of storage space
    1. Worker 1 gets jobID=0, and requests 1 unit of storage space and make a skeleton.
    2. Worker 2 gets jobID=1 and wants to build an engine. Worker 2 requests 1 unit of storage space but failed, because it’s been taken to store the skeleton. Worker 2 stops and waits for space…
    3. Worker 1 gets jobID=2 building a chassis. Worker 1 requests 1 unit of storage space but failed either. Worker 1 stops production and waits for space…
    4. Both worker 1 and worker2 are waiting for a free space infinitely…

Deadlock!!!

Q3 Make it stable, make it run fast

- Tasks
  - Tackle deadlock problem
  - Optimize for speed

- Requirements
  1. You **must** use your own implementation of thread-safe queue in Q3.
  2. Your program should accept any numbers of workers to produce different number of cars.
  3. Performance **scalability** analysis.
  4. Clearly introduce your deadlock handling algorithm in your report

Q3 Make it stable, make it run fast

- Marking scheme
  - Deadlock free implementation: 30 marks (20 bonus marks included);

    Your total mark = $30 \times \sum_{i=1}^{N} \frac{T_{i\ min}}{Ti}$,

    where T_i is your runtime of the ith test case, $T_{i\ min}$ is the minimum time among all your classmates.
  - Deadlock free program performance competition among the whole class
  - If any deadlock case is found with your program, you get 0 mark for both report explanation and implementation. Your performance won't be recorded either.
    - Deadlock judgement: sequentially producing car parts one by one to make a car costs 40s. If your program fails to finish producing N cars within N*60 seconds, it'll be considered as a deadlock situation.

General requirements

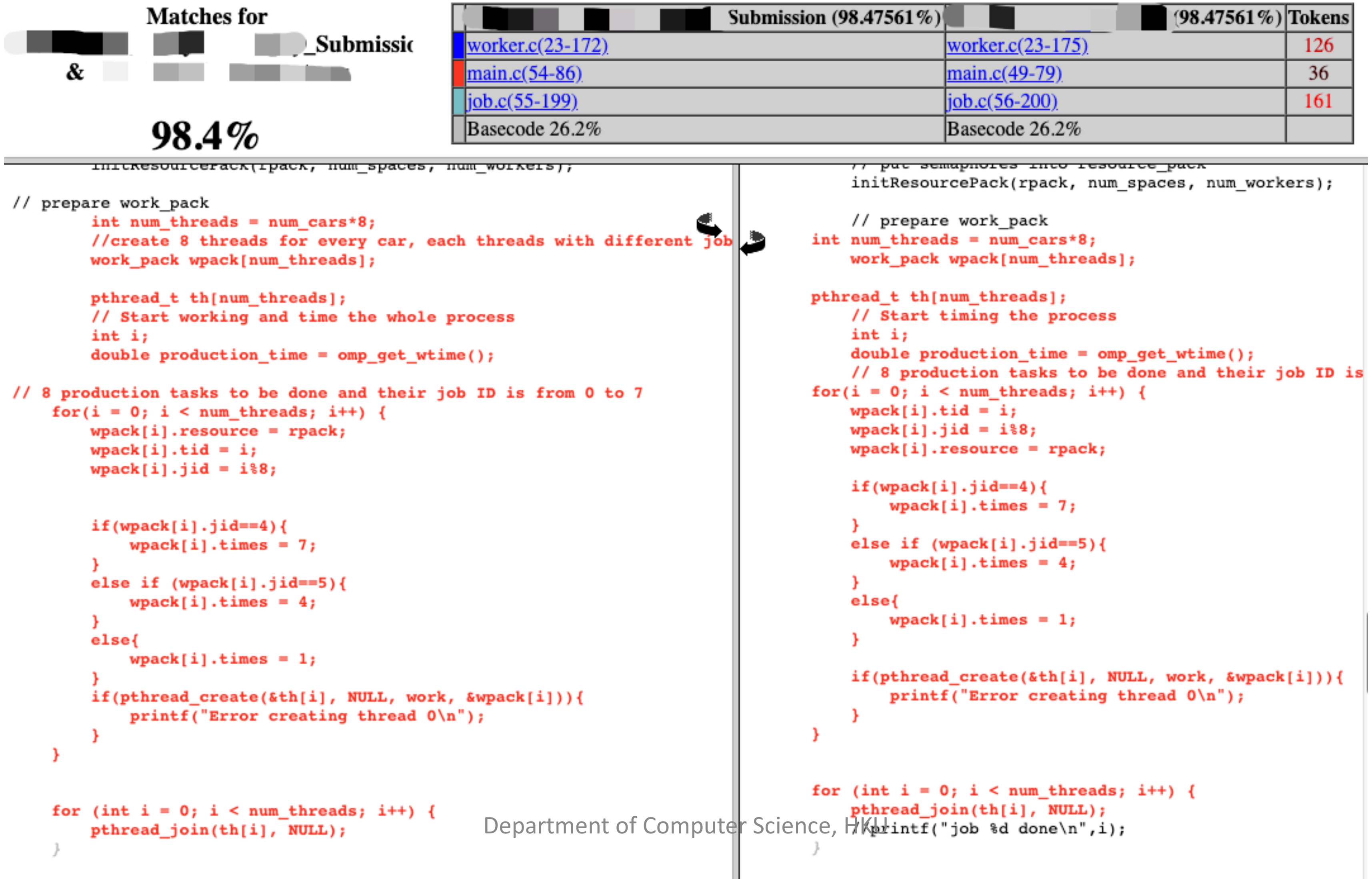
- Make sure that your code for each question can be compiled and run without problem on workbench, or you will get **0** mark
- If you create more worker threads than num_workers, **0** mark will be given for that entire question
- Questions not allowed to asked:
  - Ask for answers to compare with your own code or check your answer with TA before you submit it
  - Ask if your idea/algorithm work or not.
    - You have an idea, you should find ways to proof/disproof it.
  - Questions related to programme in C.
  - Debug your program. Try to debug your program with ***printf*** or ***gdb*** by yourself.

NO PLAGIARISM

NO PLAGIARISM

- Source code from previous year course won’t work this year
- Source code will be compared with all submissions this year and previous years
- Basecode (code provided) won’t be compared
- Once caught, minimum penalty is getting 0 mark for this assignment.

Direct copying



Matches for [redacted] _Submissic & [redacted]

98.4%

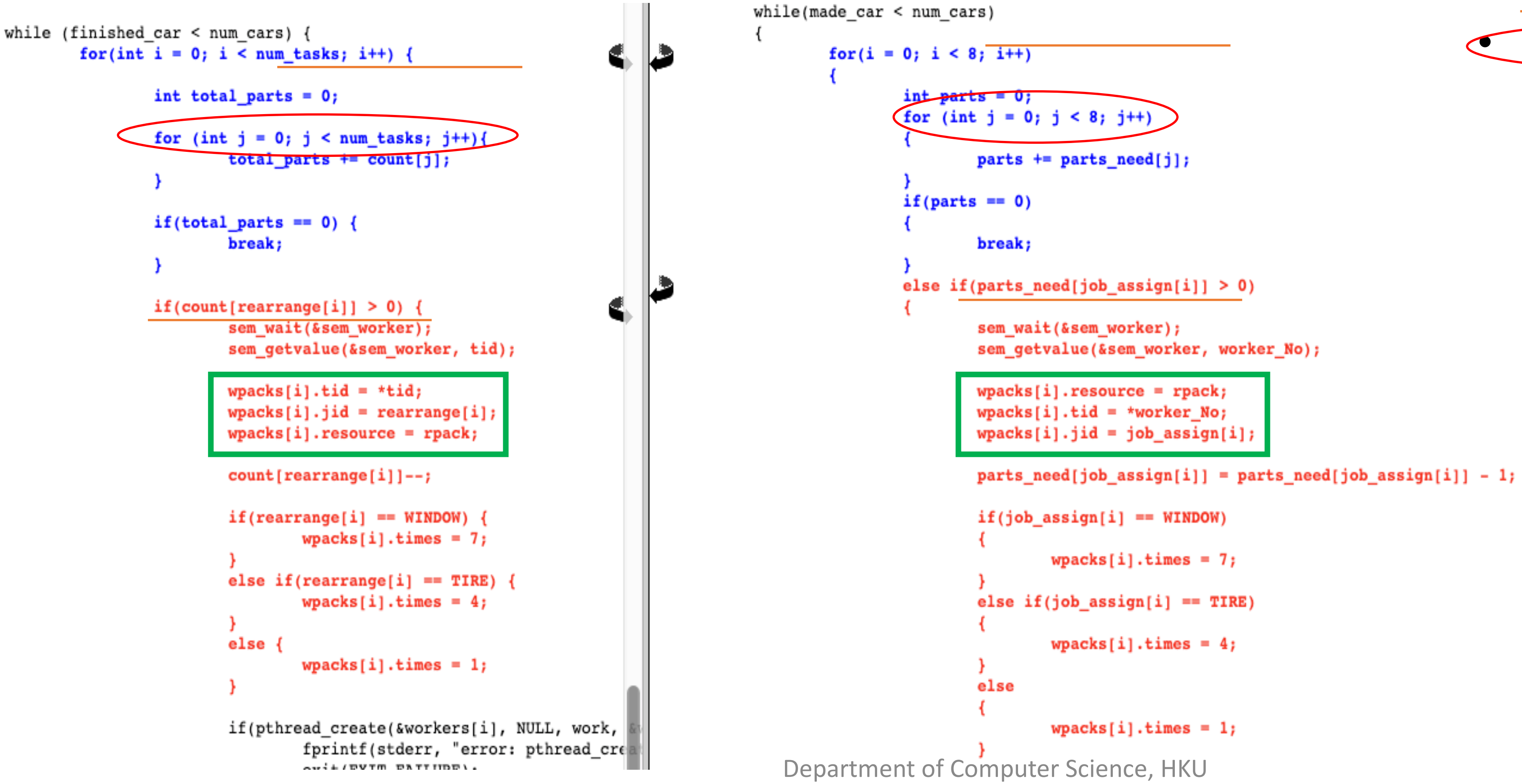
[redacted] Submission (98.47561%)	[redacted] (98.47561%)	Tokens
worker.c(23-172)	worker.c(23-175)	126
main.c(54-86)	main.c(49-79)	36
job.c(55-199)	job.c(56-200)	161
Basecode 26.2%	Basecode 26.2%	

```
        initResourcePack(rpack, num_spaces, num_workers);

// prepare work_pack
        int num_threads = num_cars*8;
        //create 8 threads for every car, each threads with different job
        work_pack wpack[num_threads];

        pthread_t th[num_threads];
        // Start working and time the whole process
        int i;
        double production_time = omp_get_wtime();

// 8 production tasks to be done and their job ID is from 0 to 7
    for(i = 0; i < num_threads; i++) {
        wpack[i].resource = rpack;
        wpack[i].tid = i;
        wpack[i].jid = i%8;


        if(wpack[i].jid==4){
            wpack[i].times = 7;
        }
        else if (wpack[i].jid==5){
            wpack[i].times = 4;
        }
        else{
            wpack[i].times = 1;
        }
        if(pthread_create(&th[i], NULL, work, &wpack[i])){
            printf("Error creating thread 0\n");
        }
    }


    for (int i = 0; i < num_threads; i++) {
        pthread_join(th[i], NULL);
    }
```

```
    // put semaphores into resource_pack
    initResourcePack(rpack, num_spaces, num_workers);

    // prepare work_pack
int num_threads = num_cars*8;
    work_pack wpack[num_threads];

pthread_t th[num_threads];
    // Start timing the process
    int i;
    double production_time = omp_get_wtime();
    // 8 production tasks to be done and their job ID is
for(i = 0; i < num_threads; i++) {
    wpack[i].tid = i;
    wpack[i].jid = i%8;
    wpack[i].resource = rpack;

    if(wpack[i].jid==4){
        wpack[i].times = 7;
    }
    else if (wpack[i].jid==5){
        wpack[i].times = 4;
    }
    else{
        wpack[i].times = 1;
    }

    if(pthread_create(&th[i], NULL, work, &wpack[i])){
        printf("Error creating thread 0\n");
    }
}


for (int i = 0; i < num_threads; i++) {
    pthread_join(th[i], NULL);
    printf("job %d done\n",i);
}
```

Copying with minor change

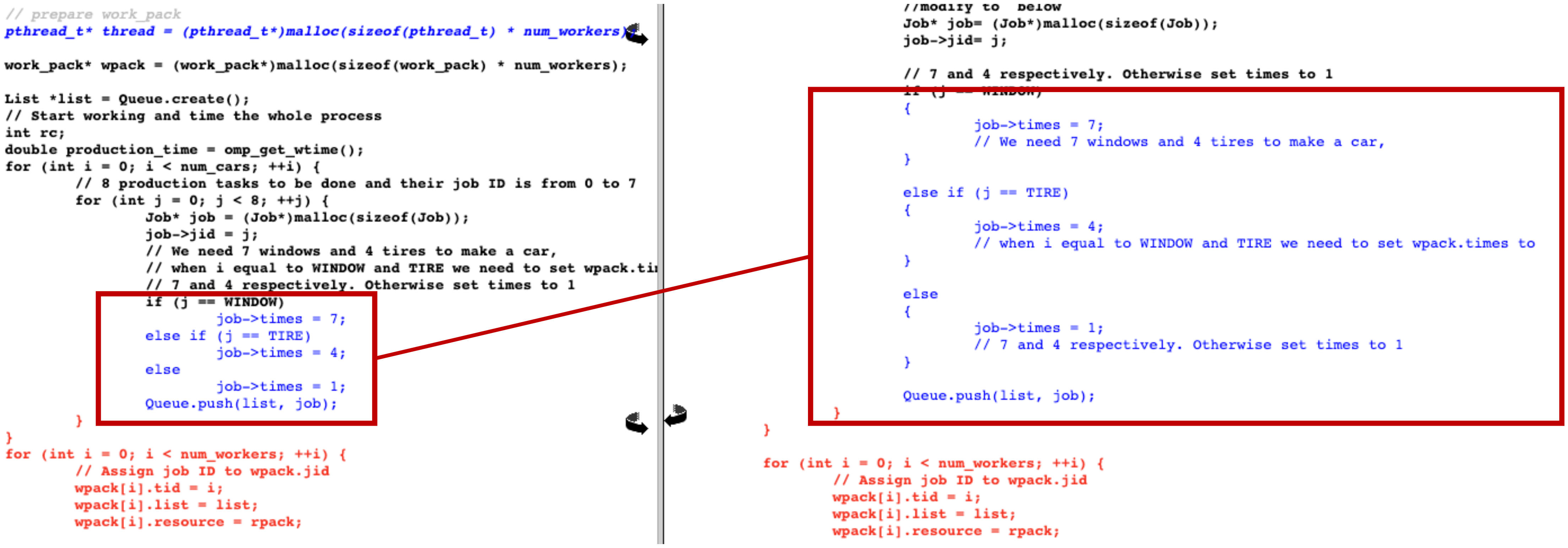
- Change code order
- Change variable name
- Hard-coded value



```
while (finished_car < num_cars) {
        for(int i = 0; i < num_tasks; i++) {

                int total_parts = 0;

                for (int j = 0; j < num_tasks; j++){
                        total_parts += count[j];
                }

                if(total_parts == 0) {
                        break;
                }

                if(count[rearrange[i]] > 0) {
                        sem_wait(&sem_worker);
                        sem_getvalue(&sem_worker, tid);

                        wpacks[i].tid = *tid;
                        wpacks[i].jid = rearrange[i];
                        wpacks[i].resource = rpack;

                        count[rearrange[i]]--;

                        if(rearrange[i] == WINDOW) {
                                wpacks[i].times = 7;
                        }
                        else if(rearrange[i] == TIRE) {
                                wpacks[i].times = 4;
                        }
                        else {
                                wpacks[i].times = 1;
                        }

                        if(pthread_create(&workers[i], NULL, work, &
                                fprintf(stderr, "error: pthread_cre
                                exit(EXIT_FAILURE);
```

```
while(made_car < num_cars)
{
        for(i = 0; i < 8; i++)
        {
                int parts = 0;
                for (int j = 0; j < 8; j++)
                {
                        parts += parts_need[j];
                }
                if(parts == 0)
                {
                        break;
                }
                else if(parts_need[job_assign[i]] > 0)
                {
                        sem_wait(&sem_worker);
                        sem_getvalue(&sem_worker, worker_No);

                        wpacks[i].resource = rpack;
                        wpacks[i].tid = *worker_No;
                        wpacks[i].jid = job_assign[i];

                        parts_need[job_assign[i]] = parts_need[job_assign[i]] - 1;

                        if(job_assign[i] == WINDOW)
                        {
                                wpacks[i].times = 7;
                        }
                        else if(job_assign[i] == TIRE)
                        {
                                wpacks[i].times = 4;
                        }
                        else
                        {
                                wpacks[i].times = 1;
                        }
```

Logical copying



```
// prepare work_pack
pthread_t* thread = (pthread_t*)malloc(sizeof(pthread_t) * num_workers);

work_pack* wpack = (work_pack*)malloc(sizeof(work_pack) * num_workers);

List *list = Queue.create();
// Start working and time the whole process
int rc;
double production_time = omp_get_wtime();
for (int i = 0; i < num_cars; ++i) {
        // 8 production tasks to be done and their job ID is from 0 to 7
        for (int j = 0; j < 8; ++j) {
                Job* job = (Job*)malloc(sizeof(Job));
                job->jid = j;
                // We need 7 windows and 4 tires to make a car,
                // when i equal to WINDOW and TIRE we need to set wpack.ti
                // 7 and 4 respectively. Otherwise set times to 1
                if (j == WINDOW)
                        job->times = 7;
                else if (j == TIRE)
                        job->times = 4;
                else
                        job->times = 1;
                Queue.push(list, job);
        }
}
for (int i = 0; i < num_workers; ++i) {
        // Assign job ID to wpack.jid
        wpack[i].tid = i;
        wpack[i].list = list;
        wpack[i].resource = rpack;
```

```
        //modify to  below
        Job* job= (Job*)malloc(sizeof(Job));
        job->jid= j;

        // 7 and 4 respectively. Otherwise set times to 1
        if (j == WINDOW)
        {
                job->times = 7;
                // We need 7 windows and 4 tires to make a car,
        }

        else if (j == TIRE)
        {
                job->times = 4;
                // when i equal to WINDOW and TIRE we need to set wpack.times to
        }

        else
        {
                job->times = 1;
                // 7 and 4 respectively. Otherwise set times to 1
        }

        Queue.push(list, job);
    }
}

for (int i = 0; i < num_workers; ++i) {
        // Assign job ID to wpack.jid
        wpack[i].tid = i;
        wpack[i].list = list;
        wpack[i].resource = rpack;
```

Plagiarism

- Direct copying
- Adding/deleting/reordering comments
- Reordering code lines
- Renaming variables
- Adding meaningless code lines
- Copying online code without reference
- …