Project 2 Report
Name: Ciel Student id: 7

Contents

1 Introduction

1.1 Project Background and Objective

In modern computing systems, interprocess communication and synchronisation are fundamental to ensuring the efficient and correct execution of concurrent processes or threads. This project investigates two classic synchronisation problems: the **Stooge Farmers Problem** and the **Bicycle Factory Problem**. These problems model real-world scenarios in which multiple entities must collaborate under specific constraints, necessitating precise coordination to prevent race conditions, deadlocks, and starvation. The objectives of this project are the following.

- Design and implement synchronisation mechanisms using POSIX Pthreads and semaphores to solve the Stooge Farmers Problem and the Bicycle Factory Problem.
- Ensure correct inter-process communication and mutual exclusion while preventing deadlocks and starvation.
- Incorporate real-world delays and stochastic behaviour into the simulations to enhance the robustness of solutions.
- Evaluate the performance and behaviour of the solutions implemented under the specified constraints.

1.2 Key Technologies Involved

Our key technical details include: POSIX Pthreads, synchronisation and programme timing.

1.3 Report Structure Overview

1. **Problem Description:** We first provide a concise overview of the two synchronisation problems, the *Stooge Farmers Problem* and the *Bicycle Factory Problem*, along with their respective requirements.
2. **Design and Implementation:** Next, we present the design rationale, detailing the synchronisation mechanisms used. We then discuss the specific implementation strategies and optimisation techniques applied to each problem.
3. **Performance Evaluation:** Finally, we perform a comprehensive series of tests to assess the performance of the programme under varying conditions. The test results are presented and analysed in detail to validate the effectiveness and robustness of our solutions.

2 Problem Analysis

2.1 Overview of Problems

This project addresses two key problems: the *Stooge Farmers Problem* and the *Bicycle Factory Problem*. To effectively solve these tasks, a comprehensive understanding of several technical concepts is required, including POSIX threads, thread sync mechanisms, and timing.

2.2 Detailed Requirements

2.2.1 Stooge Farmers Problem

In this problem, we simulate the collaboration of three tree planting workers-Larry, Moe, and Curly. Larry digs holes, Moe places a seed in each hole dug, and Curly fills the holes after the seeds are planted. There are some restrictions on the order in which they work.

1. Moe can only plant a seed if at least one empty hole exists (dug by Larry but not yet planted).
2. Curly can only fill a hole if at least one hole has a seed (planted by Moe but not yet filled).
3. Larry cannot dig more than MAX holes ahead of Curly (to prevent excessive unfilled holes).
4. Only one shovel is shared between Larry (digging) and Curly (filling), requiring mutual exclusion.

To accurately simulate real-world conditions, random delays must be introduced for the three workers. Additionally, to ensure programme robustness, implementation must prevent deadlock and starvation in all possible execution scenarios.
This problem demonstrates producer-consumer-like coordination, where each farmer depends on the previous farmer' s work while adhering to resource constraints.

2.2.2 Bicycle Factory Problem

In this problem, we simulate three types of workers-frame producer, wheel producer, and assembler-who collaborate in a bicycle factory and the warehouse (box) they use to interact. Their working mechanism is as follows. **Frame producers** create bicycle frames and place them in a shared storage box. **Wheel producers** create bicycle wheels (2 per bicycle) and store them in the same box. **Assemblers** retrieve 1 frame and 2 wheels from the box to assemble a complete bicycle. Frames and wheels can be fetched separately, but wheels must be taken 2 at a time. The assembled bicycles will not be placed back in the box.
To ensure data integrity within the shared storage (box), a synchronisation mechanism must be implemented to regulate orderly access among workers. Furthermore, given the finite capacity of

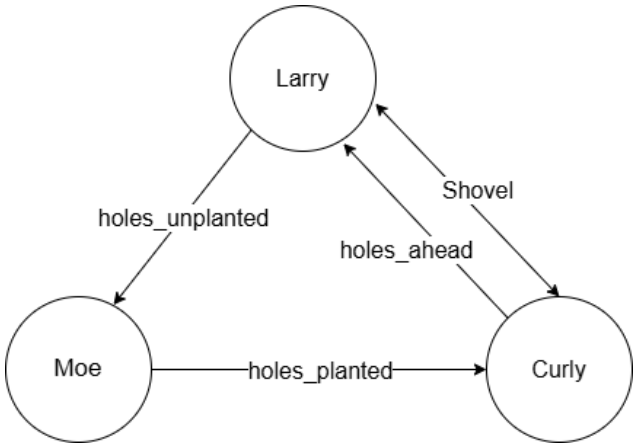

Fig. 1: Implementation of semaphores.

the shared storage (box), the system must enforce strict constraints on the quantities of frame and wheel to prevent deadlock conditions.

This problem models a more complicated multithreaded producer-consumer system with dependencies, which requires careful coordination to ensure efficient and correct bicycle assembly.

3 Design and Implementation of Stooge Farmers Problem

The central challenge in this problem lies in establishing effective synchronisation, which fundamentally requires the correct configuration of semaphores. Having systematically analysed the interdependencies between system components in 2.2.1, we now focus on formally mapping these constraints to appropriate semaphore operations. Fig 1 provides a visual representation of the relationships between the 4 semaphores and their respective interactions with the three farmers within the system. The implementation code is presented in 6.1.

For Constraint 1, we set a semaphore `holes_unplanted`, indicating the number of holes dug but not seeded. Larry releases a semaphore each time he digs a hole; Moe needs to obtain a semaphore before sowing seeds in each hole. If the semaphore is 0, it means that there are no holes that have been dug but not sown, and Moe cannot sow seeds at this time.

Similarly, for Constraint 2, we set a semaphore `holes_planted`, indicating the number of holes sown but not filled. Moe releases a semaphore each time he sows a hole, while Curly has to wait for an available semaphore before filling a hole.

For Constraint 3, we set a semaphore `holes_ahead`. The semaphore is initialised to value MAX (*i.e.*, the maximum number of unfilled holes). Curly performs a semaphore release operation upon filling each hole, while Larry must acquire the semaphore before excavating a new one. This semaphore functions as a credit mechanism, effectively regulating the number of holes Larry is permitted to dig while maintaining system synchronisation.

To enforce Constraint 4, we implement a binary semaphore *shovel*. This synchronisation primitive ensures that digging and filling operations never concurrently access the shovel, thereby preventing race conditions.

4 Design and Implementation of Bicycle Factory Problem

4.1 Baseline Design

4.1.1 Mutual exclusion

The Bicycle Factory Problem is a more complicated scenario of the producer-consumer problem, in which multiple items compete for finite shared storage. Building upon established concurrency control patterns, we employed multiple mutex locks to enforce sequential access to shared resources. Specifically, we implemented:

- A dedicated mutex for synchronising frame producer threads
- A separate mutex for coordinating wheel producer threads
- A box access mutex to mediate storage operations

Design Consideration: The system architecture assumes a single assembler process, thereby eliminating the need for inter-assembler synchronisation mechanisms. This intentional simplification reduces implementation complexity while maintaining the required safety guarantees.

4.1.2 Shared Space Management

The fundamental challenge in the bicycle factory problem is maintaining safe access to a bounded capacity box while preventing deadlock conditions. The solution requires careful coordination between the component producers and the assembler process. Based on intuition, we introduced an `empty_slot`, initialised to the total capacity of the box, which regulates the availability of space by requiring producers to acquire a semaphore before depositing components and being released by the assembler upon removal. In addition to this, separate counting semaphores for `wheel_available` and `frame_available` provide real-time inventory tracking, enabling the assembler to make informed decisions about component retrieval based on current stock levels.
The proposed semaphore system, while effective for basic synchronisation, remains vulnerable to deadlock scenarios arising from component imbalance. A critical failure case occurs when the warehouse becomes saturated with a single component type-for instance, completely filled with frames. In such circumstances, a deadlock emerges through two concurrent constraints: producers are prevented from depositing additional components due to exhausted `empty_slots`, while assemblers cannot proceed with bicycle assembly owing to the absence of complementary components (wheels in

this case). This circular dependency creates a classic deadlock situation in which the system cannot progress without external intervention. The limitation fundamentally stems from the semaphore design's inability to dynamically prioritise component types or enforce balanced production, revealing an inherent constraint in our initial synchronisation approach that requires resolution through additional safeguards or alternative coordination mechanisms.

To resolve the aforementioned deadlock issue, we introduce two additional semaphores: `frame_lim` and `wheel_lim`. These semaphores enforce strict theoretical bounds on component quantities, with `frame_lim` initialised to N-2 (where N represents the total box capacity) to guarantee space for at least two wheels, while `wheel_lim` maintains an N-1 ceiling to preserve single-frame accommodation. This dual-constraint system ensures that the box never enters a single-component saturation state. The mathematical relationship between these limits perpetually maintains the assembly potential, thereby systematically eliminating all deadlock possibilities while preserving full-box use within safe operating parameters.

4.1.3 Assembler Behaviour

The original implementation employs a strictly sequential component acquisition strategy, where the assembler process must first successfully obtain a frame through a blocking wait operation on the frame semaphore before attempting to acquire wheels through a similar blocking mechanism.

This rigid two-phase approach-frame procurement followed by wheel collection-creates an inherent vulnerability in the system's synchronisation design. The blocking nature of both wait operations means the assembler remains indefinitely suspended if either resource becomes unavailable, potentially leading to resource starvation scenarios.

Furthermore, while this design approach requires a minimum box size of 3, the theoretically optimal minimum size is 2. This clearly demonstrates that the current implementation does not achieve optimal efficiency.

4.2 Advancements

4.2.1 Motivation

We observed the following fact: the assembler retrieves components from the warehouse and stores them in its own buffer space.This buffer space can be viewed as a storage capacity of 1 frame and 2 wheels. This observation reveals that by equivalently incorporating the assembler's buffer capacity into the total warehouse capacity, we effectively obtain a larger storage space capable of holding more components.

4.2.2 Implementation Details

Our modification on the baseline method is two-fold.

First, `true_frame_lim` and `true_wheel_lim` are used to indicate the theoretical limits on the quantities of components. Considering the existence of the assembler, the semaphores are initialised to N-1 and N+1, respectively. Since the assembler's buffer capacity has been incorporated into the total box's capacity, we modify the semaphore release mechanism: rather than releasing the two semaphores when components are acquired by the assembler, we now release them only after the assembly process is completed.

Second, we apply a non-blocking implementation for the assembler. The aforementioned design relies on the assembler's ability to freely select frames and wheels. However, the original blocking mechanism `sem_wait` could induce deadlocks and thus is incompatible with our implementation. Consequently, we employ a non-blocking approach `sem_trywait`. In this non-blocking implementation, the assembler adopts a polling mechanism instead of sequential acquisition. If assembly conditions are not met, it iteratively attempts to procure the required components, proceeding only after securing all necessary parts. This strategy effectively circumvents deadlocks.

We present the code of our advanced method in 6.2.

4.2.3 Advantages over Baseline Method

The proposed method introduces an optimised approach to address the box size constraints in the Bicycle Factory Problem, reducing the minimum required box capacity **from 3 to 2**. Compared to the baseline method, this solution demonstrates greater flexibility and efficiency while maintaining a structurally elegant design. The specific differences between the baseline method and the advanced method are listed in Tab 1.

Aspect	Least Slots	Waiting mode	Assembling Scheme	Box Capacity
Baseline	3	blocking	Sequence	N
Advanced	2	non-blocking	polling	N+3 (Equivalently)

Tab. 1: Comparison between the baseline method and the advanced method.

5 Experiments and Testing

5.1 Stooge Farmer Problem

5.1.1 Experiment Settings

We configure the system with N=5 holes and a maximum allowable lead (MAX) of 3 holes for Larry over Curly. To implement randomised delays, we initialise a random seed using `srand()` and generate unique delay values via `rand()`. The delays were restricted by `usleep(rand()%100000)`, with a maximum delay threshold of 100 milliseconds enforced to maintain the system responsiveness.

5.1.2 Experiment Results and Analysis

Here is a running result of our programme:

```
N = 5.
MAX = 3.
Larry gets the shovel.
Larry digs another hole #1.
Larry drops the shovel.
Moe plants a seed in hole #1.
Curly gets the shovel.
Curly fills a planted hole #1.
Curly drops the shovel.
Larry gets the shovel.
Larry digs another hole #2.
Larry drops the shovel.
Moe plants a seed in hole #2.
Curly gets the shovel.
Curly fills a planted hole #2.
Curly drops the shovel.
Larry gets the shovel.
Larry digs another hole #3.
Larry drops the shovel.
Larry gets the shovel.
Larry digs another hole #4.
Larry drops the shovel.
Moe plants a seed in hole #3.
Curly gets the shovel.
Curly fills a planted hole #3.
Curly drops the shovel.
Moe plants a seed in hole #4.
Curly gets the shovel.
Curly fills a planted hole #4.
Curly drops the shovel.
Larry gets the shovel.
Larry digs another hole #5.
Larry drops the shovel.
Moe plants a seed in hole #5.
Curly gets the shovel.
Curly fills a planted hole #5.
Curly drops the shovel.
End.
```

The execution result demonstrates that the planting process was properly implemented according to the defined specifications.

Attention Since both the delay times and thread allocation are non-deterministic, the execution may produce varied outcomes. However, all possible results will consistently adhere to the specified rules.

5.2 Bicycle Factory Problem

5.2.1 Experiment Settings

In order to assess the robustness of our approach under extreme operating conditions, the system was configured with a shared box capacity (N) of 2 and a production target (M) of 5 bicycles. The production process employed 2 frame producers (A), 2 wheel producers (B) and 1 assembler (C). Randomised delays were implemented using the same method described in 5.1.1, maintaining consistency in the experimental setup.

5.2.2 Experiment Results and Analysis

Here is a running result of our programme:

```
N = 2, M = 5
A = 2, B = 2, C = 1
Frame producer 0 produced a frame.
Frame producer 0 placed a frame. { 1 frames, 0 wheels }
Assembler 0 took a frame. { 0 frames, 0 wheels }
Wheel producer 0 produced a wheel.
Wheel producer 0 placed a wheel. { 0 frames, 1 wheels }
Frame producer 0 produced a frame.
Wheel producer 0 produced a wheel.
Wheel producer 0 placed a wheel. { 0 frames, 2 wheels }
Assembler 0 took two wheels. { 0 frames, 0 wheels }
Frame producer 0 placed a frame. { 1 frames, 0 wheels }
Frame producer 1 produced a frame.
Wheel producer 1 produced a wheel.
Wheel producer 1 placed a wheel. { 1 frames, 1 wheels }
Assembler 0 assembled a bicycle.
Assembler 0 took a frame. { 0 frames, 1 wheels }
Wheel producer 1 produced a wheel.
Wheel producer 1 placed a wheel. { 0 frames, 2 wheels }
Assembler 0 took two wheels. { 0 frames, 0 wheels }
Frame producer 1 placed a frame. { 1 frames, 0 wheels }
```

```
Frame producer 0 produced a frame.
Assembler 0 assembled a bicycle.
Assembler 0 took a frame. { 0 frames, 0 wheels }
Wheel producer 0 produced a wheel.
Wheel producer 0 placed a wheel. { 0 frames, 1 wheels }
Frame producer 1 produced a frame.
Wheel producer 0 produced a wheel.
Wheel producer 0 placed a wheel. { 0 frames, 2 wheels }
Assembler 0 took two wheels. { 0 frames, 0 wheels }
Frame producer 0 placed a frame. { 1 frames, 0 wheels }
Wheel producer 1 produced a wheel.
Wheel producer 1 placed a wheel. { 1 frames, 1 wheels }
Assembler 0 assembled a bicycle.
Assembler 0 took a frame. { 0 frames, 1 wheels }
Wheel producer 1 produced a wheel.
Wheel producer 1 placed a wheel. { 0 frames, 2 wheels }
Assembler 0 took two wheels. { 0 frames, 0 wheels }
Frame producer 1 placed a frame. { 1 frames, 0 wheels }
Wheel producer 1 produced a wheel.
Wheel producer 1 placed a wheel. { 1 frames, 1 wheels }
Assembler 0 assembled a bicycle.
Assembler 0 took a frame. { 0 frames, 1 wheels }
Wheel producer 1 produced a wheel.
Wheel producer 1 placed a wheel. { 0 frames, 2 wheels }
Assembler 0 took two wheels. { 0 frames, 0 wheels }
Assembler 0 assembled a bicycle.
End.
```

The experimental results demonstrate that our method successfully avoids deadlock conditions despite the constrained shared box capacity of only 2 slots. This outcome validates the effectiveness of our innovative approach in maintaining system progress under limited resource conditions.

Attention The same as in 5.1.2, the results are uncertain.

6 Reference

6.1 C Code of Stooge Farmer Problem

Here is the code in the source file **lcm.c**. The random delays used for the experiment are all commented out. As **lcm_main.c** and **lcm.h** are given in the template and have not been modified, we do not present them in the report.

```c
#include "lcm.h"

#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

// semaphores
sem_t shovel;
sem_t holes_ahead;  // Larry领先Curly的洞
sem_t holes_unplanted; // 没有放种子的洞
sem_t holes_planted; // 放了种子的洞

void *larry() {
    // some code goes here
    int id = 0;
    while (id < N) {
        sem_wait(&holes_ahead); // 若领先的洞达到了阈值，则等待
        sem_wait(&shovel);

        get_shovel(LARRY);
        if (id >= N) {
            sem_post(&holes_ahead);
            sem_post(&shovel);
            break;
        }
        dig(LARRY, ++id);
        drop_shovel(LARRY);

        sem_post(&holes_unplanted); // 增加一个没有放种子的洞
        sem_post(&shovel);

        // usleep(rand()%100000); // 实验用
    }
    pthread_exit(0);
}

void *moe() {
    // some code goes here
    int id = 0;
    while (id < N) {
        sem_wait(&holes_unplanted); // 等待一个没放种子的洞
        if (id >= N) {
```

```
46              sem_post(&holes_unplanted);
47              break;
48          }
49          plant(MOE, ++id);
50          sem_post(&holes_planted); // 增加一个放了种子的洞
51
52          // usleep(rand()%100000); // 实验用
53      }
54      pthread_exit(0);
55  }
56
57  void *curly() {
58      // some code goes here
59      int id = 0;
60      while (id < N) {
61          sem_wait(&holes_planted); // 等待一个放了种子的洞
62          sem_wait(&shovel);
63
64          get_shovel(CURLY);
65          if (id >= N) {
66              sem_post(&holes_planted);
67              sem_post(&shovel);
68              break;
69          }
70          fill(CURLY, ++id);
71          drop_shovel(CURLY);
72
73          sem_post(&holes_ahead); // 追上一个洞
74          sem_post(&shovel);
75
76          // usleep(rand()%100000); // 实验用
77      }
78      pthread_exit(0);
79  }
80
81  void init() {
82      // some code goes here
83      sem_init(&shovel, 0, 1);
84      sem_init(&holes_ahead, 0, MAX);
85      sem_init(&holes_planted, 0, 0);
86      sem_init(&holes_planted, 0, 0);
87      srand((unsigned)time(NULL));
88  }
89
90  void destroy() {
```

```
    // some code goes here
    sem_destroy(&shovel);
    sem_destroy(&holes_ahead);
    sem_destroy(&holes_planted);
    sem_destroy(&holes_unplanted);
}
```

6.2 C Code of Bicycle Factory Problem

Here is the code in the source file **bicycle.c**. The random delays used for the experiment are all commented out. As **bicycle_main.c** and **bicycle.h** are given in the template and have not been modified, we do not present them in the report.

```
#include "bicycle.h"

#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

box_t box = {0, 0};

sem_t box_mutex; // 仓库的互斥锁
sem_t frame_producer_muxtex; // 车架制造者的互斥锁
sem_t wheel_producer_mutex; // 车轮制造者的互斥锁

sem_t empty_slot; // 仓库剩余容量

sem_t true_frame_lim; // 车架数量的理论上限
sem_t true_wheel_lim; // 车轮数量的理论上限

sem_t frame_available; // 仓库中可用车架
sem_t wheel_available; // 仓库中可用车轮

int bicycles_assembled = 0; // 组装好的自行车数量

int frame_got = 0; // 组装者是否拿了车架
int two_wheels_got = 0; // 组装者是否拿了车轮

int frame_produced = 0; // 生产的车架数量
int wheel_produced = 0; // 生产的车轮数量
const int frame_need = M; // 需要的车架数量
const int wheel_need = 2 * M; // 需要的车轮数量

```

```
int odd_wheel = 0; // 记录是否放置了了奇数个车轮，用于每放2个车轮释放1个信号量

void *frame_producer(void *arg) {
    int id = *(int *)arg;
    while (1) {
        // 制造车架
        sem_wait(&frame_producer_muxtex); // 制造的车架足够就不用制造了
        if (frame_produced >= frame_need) {
            sem_post(&frame_producer_muxtex);
            break;
        }
        else {
            frame_produced++;
            produce_frame(id);
            sem_post(&frame_producer_muxtex);
            // usleep(rand()%100000); // 实验用
        }

        // 存放车架
        sem_wait(&true_frame_lim);
        sem_wait(&empty_slot);
        sem_wait(&box_mutex);

        place_frame(id, &box); // 放入车架，+1操作在函数中执行
        sem_post(&frame_available); // 通知有车架

        sem_post(&box_mutex);
        // usleep(rand()%100000); // 实验用
    }
    pthread_exit(0);
}

void *wheel_producer(void *arg) {
    int id = *(int *)arg;
    while (1) {
        // 制造车轮
        sem_wait(&wheel_producer_mutex);
        if (wheel_produced >= wheel_need) { // 制造的车轮足够就不用制造了
            sem_post(&wheel_producer_mutex);
            break;
        }
        else {
            wheel_produced++;
            produce_wheel(id);
            sem_post(&wheel_producer_mutex);
```

```
            // usleep(rand()%100000); // 实验用
        }

        // 存放车轮
        sem_wait(&true_wheel_lim);
        sem_wait(&empty_slot);
        sem_wait(&box_mutex);

        place_wheel(id, &box); // 放入车轮, +1操作在函数中执行
        // 每制造两个车轮, 通知一次
        if (odd_wheel == 1) {
            sem_post(&wheel_available);
            odd_wheel = 0;
        }
        else
            odd_wheel = 1;

        sem_post(&box_mutex);
        // usleep(rand()%100000); // 实验用
    }
    pthread_exit(0);
}

void *assembler(void *arg) {
    int id = *(int *)arg;
    while (1) {
        // 索要车架
        if(frame_got == 0) {
            if (sem_trywait(&frame_available) == 0) { // 使用非阻塞式等待
                sem_wait(&box_mutex);

                get_frame(id, &box); //拿走一个车架, -1操作在函数中进行
                sem_post(&empty_slot); // 释放一个仓库空间
                frame_got = 1; // 拿到了车架

                sem_post(&box_mutex);
                // usleep(rand()%100000); // 实验用
            }
        }
        // 索要车轮
        if (two_wheels_got == 0) {
            if (sem_trywait(&wheel_available) == 0) { // 使用非阻塞式等待
                sem_wait(&box_mutex);

                get_wheels(id, &box); //拿走一个车轮, -2操作在函数中进行
```

```
123                 sem_post(&empty_slot); // 以下两行释放了两个仓库空间
124                 sem_post(&empty_slot);
125                 two_wheels_got = 1; // 拿到了车轮
126 
127                 sem_post(&box_mutex);
128                 // usleep(rand()%100000); // 实验用
129             }
130         }
131         // 组装自行车
132         if (frame_got && two_wheels_got) {
133             // 我们把组装者的容量等效地纳入仓库的容量，故此时才释放对车架和车轮的理论数
    量限制
134             sem_post(&true_frame_lim);
135             sem_post(&true_wheel_lim);
136             sem_post(&true_wheel_lim);
137             frame_got = 0;
138             two_wheels_got = 0;
139             assemble(id); //组装自行车
140             bicycles_assembled++;
141             // usleep(rand()%100000); // 实验用
142             if (bicycles_assembled == M)
143                 pthread_exit(NULL);
144         }
145     }
146     pthread_exit(0);
147 }
148 
149 void init() {
150     // 组装者可以存放1个车架和2个车轮。我们把组装者的容量等效地纳入仓库的容量。
151     sem_init(&true_frame_lim, 0, N - 1);  // 最多允许N-1个车架
152     sem_init(&true_wheel_lim, 0, N + 1);  // 最多允许N+1个轮子
153 
154     sem_init(&box_mutex, 0, 1);
155     sem_init(&empty_slot, 0, N);
156 
157     sem_init(&frame_available, 0, 0);
158     sem_init(&wheel_available, 0, 0);
159 
160     sem_init(&wheel_producer_mutex, 0, 1);
161     sem_init(&frame_producer_muxtex, 0, 1);
162 
163     // srand((unsigned)time(NULL));
164 }
165 
166 void destroy() {
```

```
167     sem_destroy(&empty_slot);
168     sem_destroy(&frame_available);
169     sem_destroy(&wheel_available);
170     sem_destroy(&true_frame_lim);
171     sem_destroy(&true_wheel_lim);
172     sem_destroy(&box_mutex);
173     sem_destroy(&wheel_producer_mutex);
174     sem_destroy(&frame_producer_muxtex);
175 }
```