COMP3230 Principles of Operating System

Assignm

Tesla Factory Production Line

Details of as se refer to the
source code t 2 document

Objectives

- Use Pthread library to write multithreaded program
- Use semaphores to han
- Use semaphores to limit resource usage

- Learn parallel programming
- Solve deadlock problem

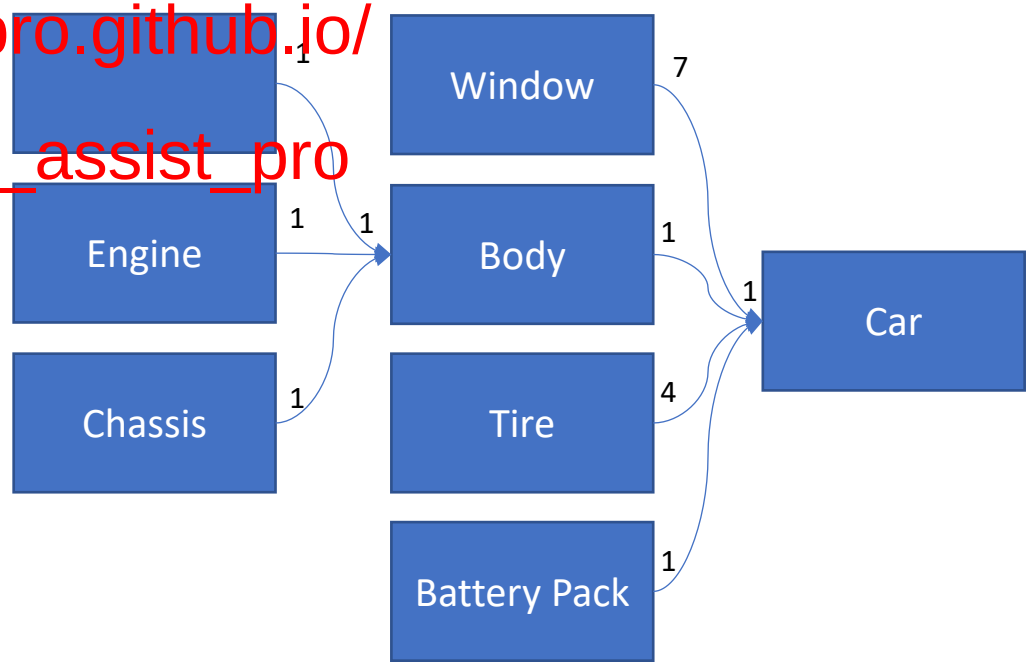
Prerequisites

- Program in C (prerequisite of this course)
  - Review Tutorial 1
  - Self-learning materials on Moodle
  - C Library: https://youtu.be/JbHmin2Wtmc
- Tutorial 3 & 4
  - Multithread programming with Pthread
  - Thread synchronization with Semaphore

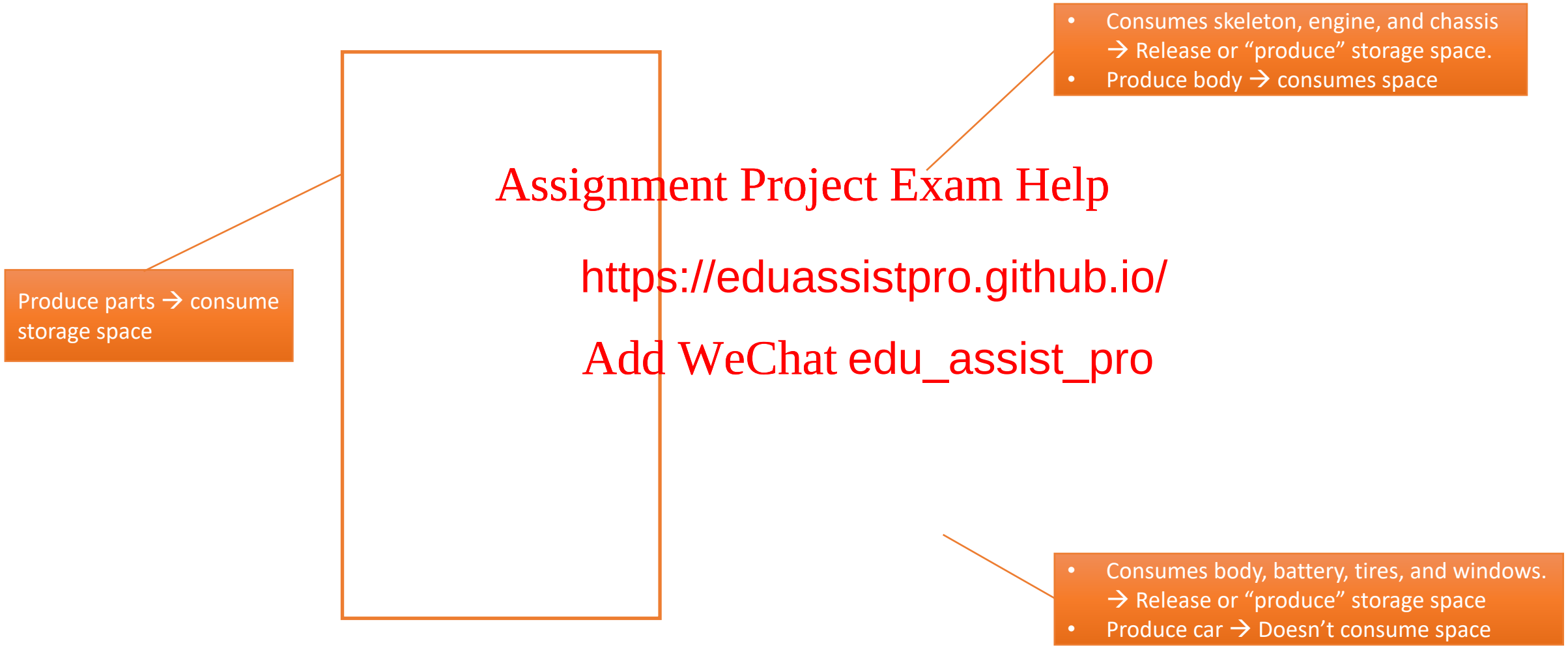
System Overview

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

- 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



Dependency Relationship



Inside libTeslaFactory.a

These semaphores are not directly accessible to you. However, the value of semaphores will be changed as you call functions defined in production.h.

- Resource tracking


```
int getNumFreeSpace();
int getNumProducedSkeleton();
int getNumProducedEngine();
int getNumProducedChassis();
int getNumProducedBody();
int getNumProducedWindow();
int getNumProducedTire();
int getNumProducedBattery();
int getNumProducedCar();
```

```
void makeSkeleton(Robot robot);
void makeEngine(Robot robot);
void makeChassis(Robot robot);
void makeBody(Robot robot);
void makeWindow(Robot robot);
void makeTire(Robot robot);
void makeBattery(Robot robot);
void makeCar(Robot robot); // ma
```

```
int tryMakeSkeleton(Robot robot);
int tryMakeEngine(Robot robot);
int tryMakeChassis(Robot robot);
int tryMakeBody(Robot robot);
int tryMakeWindow(Robot robot);
int tryMakeTire(Robot robot);
int tryMakeBattery(Robot robot);
```

```
int timedTryMakeSkeleton(int waitTime, Robot robot);
int timedTryMakeEngine(int waitTime, Robot robot);
int timedTryMakeChassis(int waitTime, Robot robot);
int timedTryMakeBody(int waitTime, Robot robot);
int timedTryMakeWindow(int waitTime, Robot robot);
int timedTryMakeTire(int waitTime, Robot robot);
int timedTryMakeBattery(int waitTime, Robot robot);
```

Inside libTeslaFactory.a

- makeXXX():

```
static int _makeItemWithSpace(int makeTime, sem_t *item) {
  I_DEBUG_HEAD
  int ret;
  if ((ret = _requestSpace()) == 0)
    _makeItemOnly(makeTime, item);
  return ret;
}
```


sem_wait() is used for makeXXX()
sem_trywait() is used for tryMakeXXX()

```
static void _makeItemOnly(int makeTime, sem_t *item) {
  I_DEBUG_HEAD
  sleep(makeTime);
  sem_post(item);
}
```

Inside libTeslaFactory.a

```
static int _timedTryMakeItemWithSpace(int waitTime, int makeTime, sem_t *item) {
  I_DEBUG_HEAD
  int ret = -1;
  if ((ret = _timedTryrequestSpace(waitTime)) == 0)
    _makeItemOnly(makeTime, item);
  return ret;
}
```

Inside libTeslaFactory.a

```
static int _tryGetItem(sem_t *item) {
  I_DEBUG_HEAD
  int ret;
  if ((ret = sem_trywait(item)) == 0)
    _releaseSpace();
  return ret;
}
```

makeBody() will call _requestSpace() here.

Once the product process starts, it will run until the end. If some parts are missing, tryMakeBody() will keep trying until all parts are acquired.

Inside libTeslaFactory.a

makeCar() won't request space

Q1. Complete the simple multithreaded version

Job IDs are continuous numbers. Here we can use a for loop to enqueue Job IDs to the queue.
You can also manually type “SKELETON”, “ENGINE”, … if you find it is clearer.

Q1. Complete the simple multithreaded version

Create num_typeB robots for type B robots and num_typeC robots for type C robots.
One pthread represents one rotbot.
Don’t forget to join all robot threads.

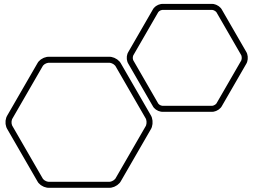
Q2 Implement a deadlock free multithreaded program

- Strategy 1: Deadlock prevention
  - The production process is executed in a way that we can be sure there won't be any deadlock.
  - E.g.: The hungry philos[illegible] use philosophers produce an agreement to ensure there is no deadl[illegible]
  - Hint for this strategy: You can either have a clever scheduler to arrange the order of jobs so that there's no deadlock or your robots are smart enough to pick jobs to ensure the whole production process is deadlock free.

Q2 Implement a deadlock free multithreaded program

- Strategy 2: Deadlock detection
  - We don't know if there will be a deadlock situation, but we can detect one.
  - E.g.: If one robot has been waiting for a space for unreasonable amount of time, we can say that there is a deadlock. The he jobs so that the deadlock can be broken.
  - Hint for this strategy: If you only want your rob ain seconds, you can use those timedTryMakeXXX() functions. Be noted, you s e a reasonable amount of waiting time. If you wait too long, deadlock detection will degrade the performance of production.


```
int timedTryMakeSkeleton(int waitTime, Robot robot);
int timedTryMakeEngine(int waitTime, Robot robot);
int timedTryMakeChassis(int waitTime, Robot robot);
int timedTryMakeBody(int waitTime, Robot robot);
int timedTryMakeWindow(int waitTime, Robot robot);
int timedTryMakeTire(int waitTime, Robot robot);
int timedTryMakeBattery(int waitTime, Robot robot);
```



Hint

- Code provided in Q1 is simply
- You can use more queues and help you solve deadlock problem and improve th performance.

Common Parallel Programming Technique

- **Master-slave model**

Master thread (usually the thread which executes main() function) arranges, coordinate the jobs. It is keep tracking the progress. It may cally adjust the job arrangement in e something goes wrong.

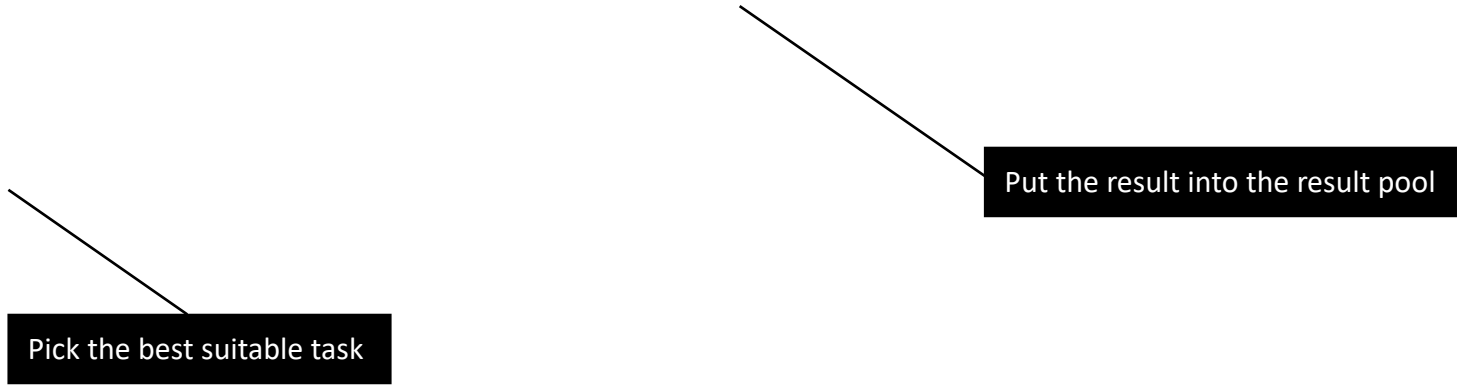
Working threads don't need to "think" or consider the whole picture of the process. Just do the job it is asked to do.

E.g.: T3 Exercise 1. Each working thread doesn't care about the total range. They just calculate the numbers they are assigned to.

Common Parallel Programming Technique

- **Autonomous thread model**

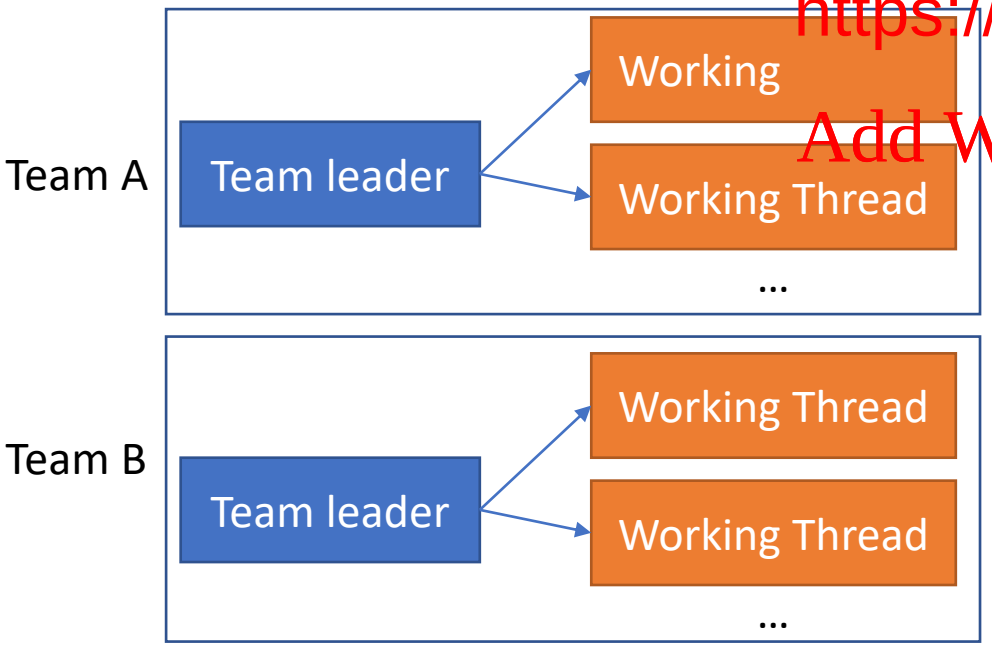
- No centralized command system
- Each working thread can work by their own
- All working threads share the same goal
- Each working thread must be wise
- Each working thread knows what to do to contribute the most to the goal.



Common Parallel Programming Technique

- **Hybrid model**

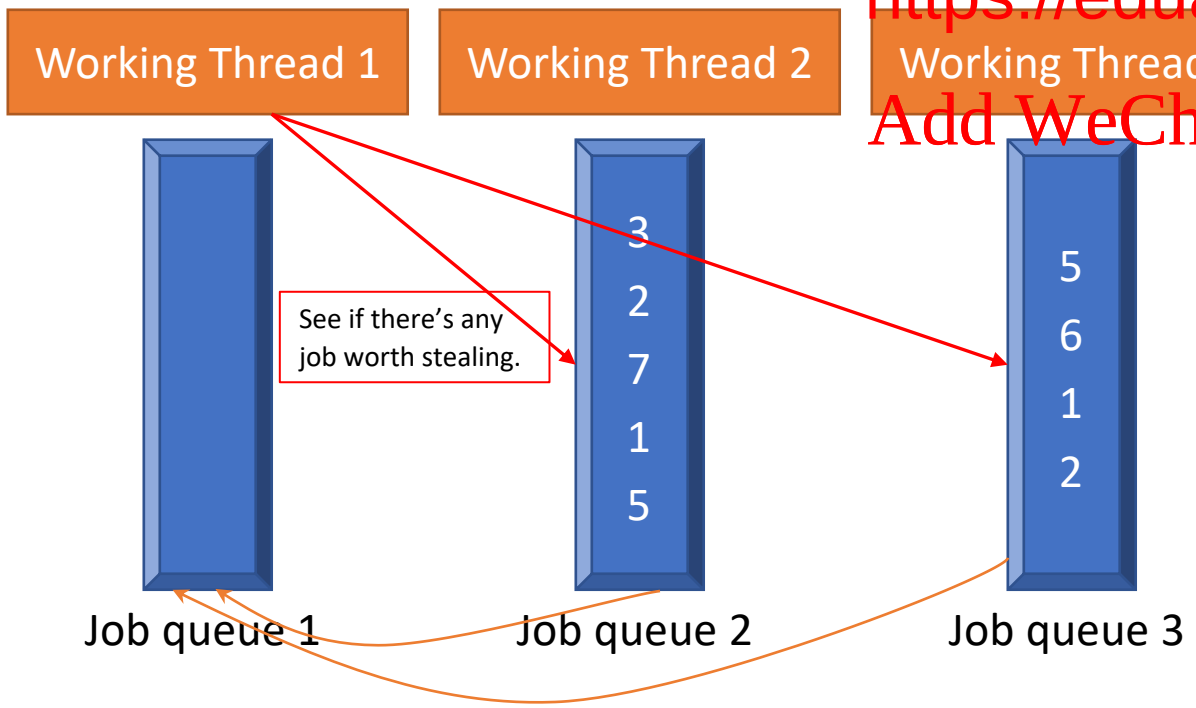
You may combine Master-slave model and autonomous thread model together.
For example, some autonomous working threads can serve as a "team leader" while other working threads just do what the leader tell



- If th ng process is time consuming or needs to compete to access certain resources, hybrid model may reduce the decision-making overhead while keeping certain level of autonomy.

Common Parallel Programming Technique

- Work stealing
  - Assume each working threads has a job queue.
  - If one working thread has completed all jobs in its job queue, this thread can "steal" jobs from other threads.
  - Purpose: keep all working t rmance.



1 has finished all its jobs in job queue 1.

- Thread 1 checks job queues of other threads
- Thread 1 steals jobs from other threads by dequeuing other threads' job queue and enqueue jobs to its own job queue.
  - What kind of jobs should thread 1 steal so that the overall performance can be improved?
  - How many jobs should thread 1 steal before it starts to work again?