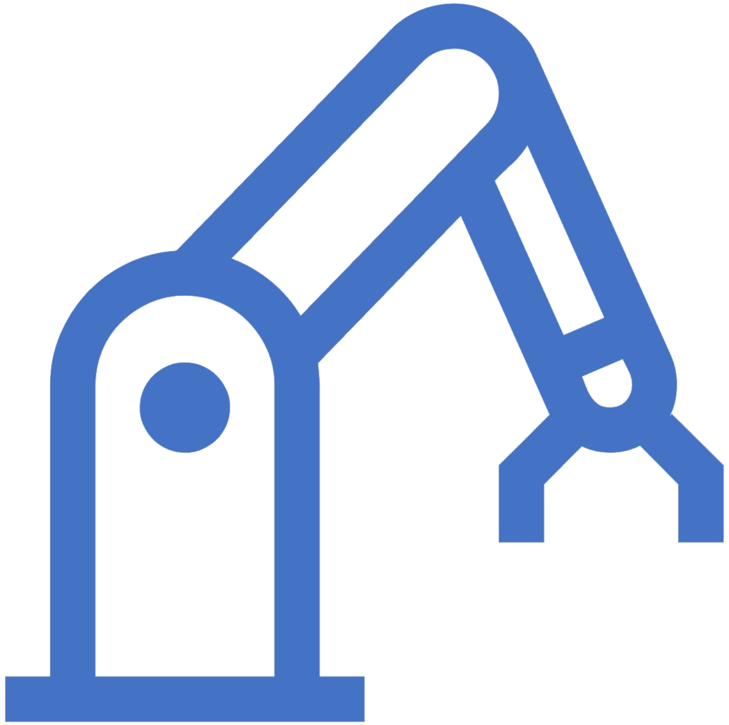
COMP3230 Principles of Operating System

Assignment 2

Tesla Factory Production Line



Details of assignment 2 please refer to the source code and assignment 2 document

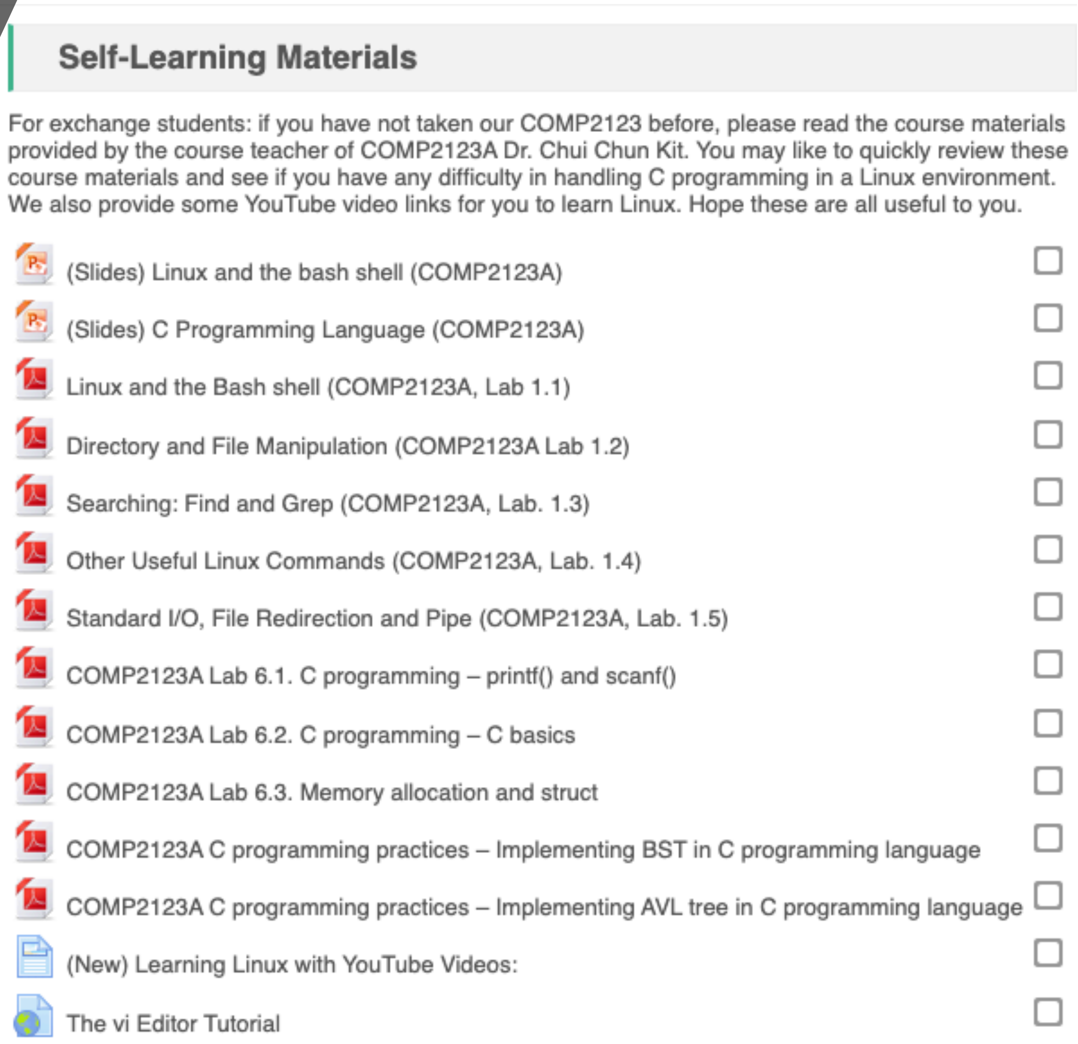
Objectives

- Use Pthread library to write multithreaded program
- Use semaphores to handle thread synchronization
- 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



Self-Learning Materials

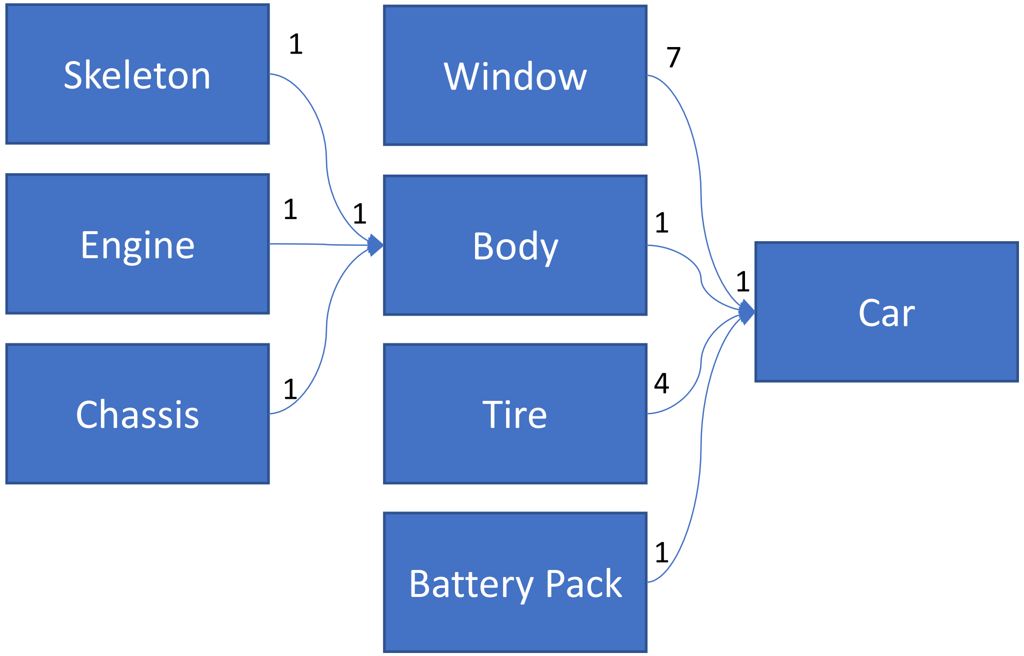
For exchange students: if you have not taken our COMP2123 before, please read the course materials provided by the course teacher of COMP2123A Dr. Chui Chun Kit. You may like to quickly review these course materials and see if you have any difficulty in handling C programming in a Linux environment. 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

System Overview

- 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

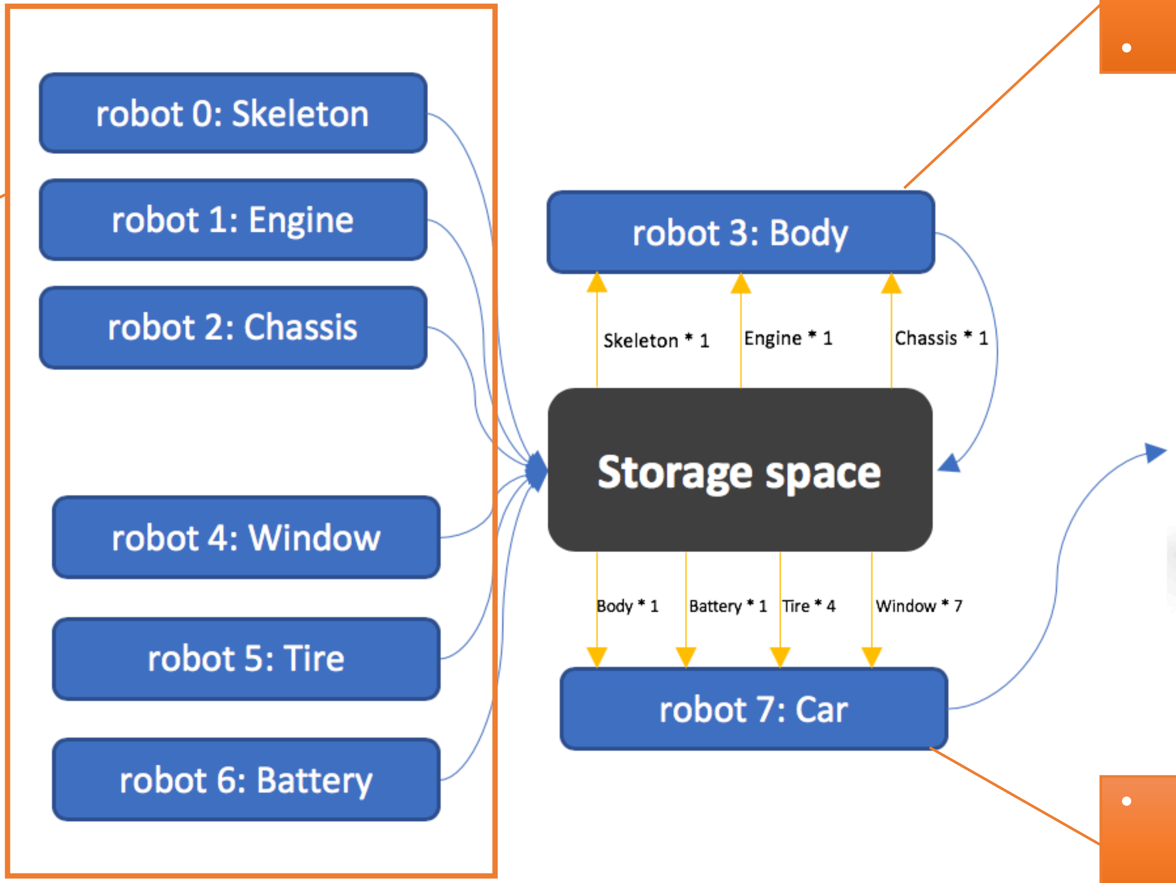
```
//Job ID
#define SKELETON  0
#define ENGINE    1
#define CHASSIS   2
#define BATTERY   3
#define WINDOW    4
#define TIRE      5
#define BODY      6
#define CAR       7
```



Dependency Relationship

- Consumes skeleton, engine, and chassis → Release or "produce" storage space.
- Produce body → consumes space

Produce parts → consume storage space



- Consumes body, battery, tires, and windows. → Release or "produce" storage space
- Produce car → Doesn't consume space

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

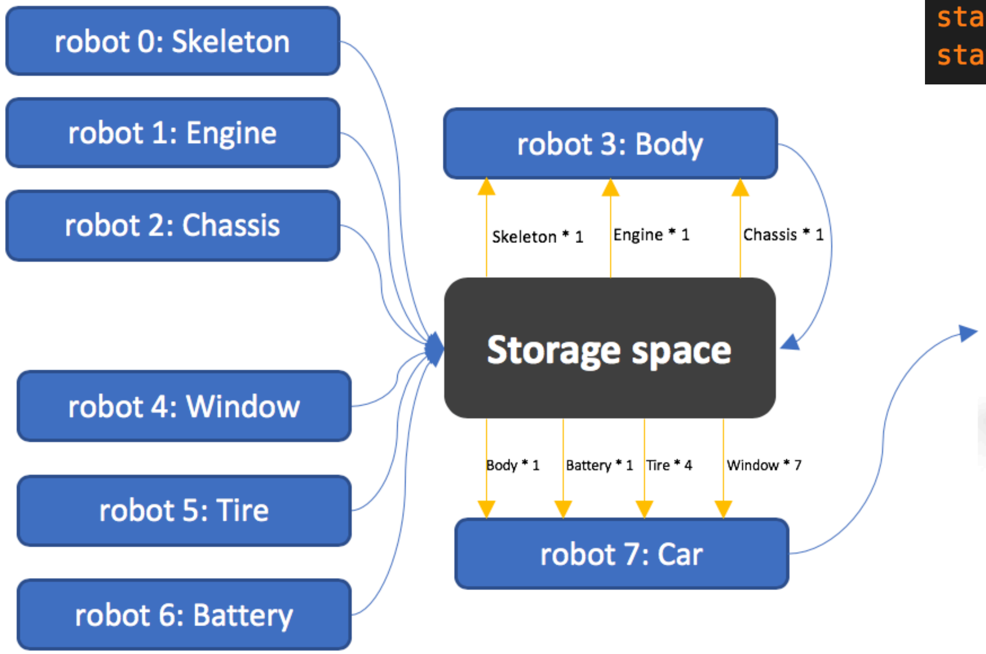
```
static sem_t _space;
static int _space_limit;

static sem_t _producedSkeleton;
static sem_t _producedEngine;
static sem_t _producedChassis;
static sem_t _producedBody;
static sem_t _producedWindow;
static sem_t _producedTire;
static sem_t _producedBattery;
static sem_t _producedCar;
```

```
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():

```c
#define MAKE(WHAT, What)                                                  \
  void make##What(Robot robot) {                                          \
  I_DEBUG_HEAD\
    switch (robot->robotType) {                                           \
    case TypeA:                                                           \
      _makeItemWithSpace(TYPE_A_TIME_##WHAT, &_produced##What);           \
      break;                                                              \
    case TypeB:
      _makeItemWithSpace(TYPE_B_TIME_##WHAT, &_produced##What);
      break;
    case TypeC:
      _makeItemWithSpace(TYPE_C_TIME_##WHAT, &_produced##What);
      break;
    default:
      break;
    }
  }

MAKE(SKELETON, Skeleton)
MAKE(ENGINE, Engine)
MAKE(CHASSIS, Chassis)
MAKE(WINDOW, Window)
MAKE(TIRE, Tire)
MAKE(BATTERY, Battery)
```

```c
#define TYPE_A_TIME_SKELETON 5
#define TYPE_A_TIME_ENGINE   4
#define TYPE_A_TIME_CHASSIS  3
#define TYPE_A_TIME_BODY     4
#define TYPE_A_TIME_WINDOW   1
#define TYPE_A_TIME_TIRE     2
#define TYPE_A_TIME_BATTERY  3
#define TYPE_A_TIME_CAR      6

#define TYPE_B_TIME_SKELETON 4
#define TYPE_B_TIME_ENGINE   5
#define TYPE_B_TIME_CHASSIS  4
#define TYPE_B_TIME_BODY     3
#define TYPE_B_TIME_WINDOW   2
#define TYPE_B_TIME_TIRE     2
#define TYPE_B_TIME_BATTERY  4
#define TYPE_B_TIME_CAR      5

#define TYPE_C_TIME_SKELETON 3
#define TYPE_C_TIME_ENGINE   3
#define TYPE_C_TIME_CHASSIS  4
#define TYPE_C_TIME_BODY     6
#define TYPE_C_TIME_WINDOW   3
#define TYPE_C_TIME_TIRE     1
#define TYPE_C_TIME_BATTERY  4
#define TYPE_C_TIME_CAR      4
```

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

```c
static int _requestSpace() {
  I_DEBUG_HEAD
#ifdef DEBUG
  int num_free_space;
  sem_getvalue(&_space, &num_free_space);
  debug_printf(__func__, "Requesting space, current space=%d...\n",
               num_free_space);
#endif
  int ret = sem_wait(&_space);
#ifdef DEBUG
  if (ret == 0) {
    sem_getvalue(&_space, &num_free_space);
    debug_printf(__func__, "Space requested, current space=%d...\n",
                 num_free_space);
  } else {
    debug_printf(__func__, "Space request failed, no space available\n");
  }
#endif
  return ret;
}
```

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

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

Inside libTeslaFactory.a

```c
#define TIMED_TRY_MAKE(WHAT, What)                                        \
  int timedTryMake##What(int waitTime, Robot robot) {                     \
  I_DEBUG_HEAD\
    int ret;                                                              \
    switch (robot->robotType) {                                           \
    case TypeA:                                                           \
      ret = _timedTryMakeItemWithSpace(waitTime, TYPE_A_TIME_##WHAT,      \
                                        &_produced##What);                \
      break;                                                              \
    case TypeB:                                                           \
      ret = _timedTryMakeItemWithSpace(waitTime, TYPE_B_TIME_##WHAT,      \
                                        &_produced##What);                \
      break;                                                              \
    case TypeC:                                                           \
      ret = _timedTryMakeItemWithSpace(waitTime, TYPE_C_TIME_##WHAT,      \
                                        &_produced##What);                \
      break;                                                              \
    default:                                                              \
      break;                                                              \
    }                                                                     \
    return ret;                                                           \
  }

TIMED_TRY_MAKE(SKELETON, Skeleton)
TIMED_TRY_MAKE(ENGINE, Engine)
TIMED_TRY_MAKE(CHASSIS, Chassis)
TIMED_TRY_MAKE(WINDOW, Window)
TIMED_TRY_MAKE(TIRE, Tire)
TIMED_TRY_MAKE(BATTERY, Battery)
```

```c
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;
}
```

```c
static struct timespec _ts;
static int _timedTryrequestSpace(int sec) {
  I_DEBUG_HEAD
  if (clock_gettime(CLOCK_REALTIME, &_ts) == -1) {
    err_printf("clock_gettime", __LINE__, "Failed to get time\n");
    return -1;
  }
  _ts.tv_sec += sec;

#ifdef DEBUG
  int num_free_space;
  sem_getvalue(&_space, &num_free_space);
  debug_printf(__func__, "Requesting space, current space=%d...\n", num_free_space);
#endif
  int s = sem_timedwait(&_space, &_ts);
#ifdef DEBUG
  if (s == 0) {
    sem_getvalue(&_space, &num_free_space);
    debug_printf(__func__, "Space requested, current space=%d...\n", num_free_space);
  } else {
    debug_printf(__func__, "Space request failed, no space available\n");
  }
#endif
  return s;
}
```

Inside libTeslaFactory.a

```c
int tryMakeBody(Robot robot) {
  I_DEBUG_HEAD
  int ret;

  if ((ret = _tryRequestSpace()) != 0)
    return ret;

  int req_skeleton = 1;
  int req_engine = 1;
  int req_chassis = 1;
  while (req_skeleton > 0 || req_engine > 0 || req_chassis > 0) {
    if (req_skeleton > 0 && _tryGetItem(&_producedSkeleton) == 0)
      req_skeleton--;
    if (req_engine > 0 && _tryGetItem(&_producedEngine) == 0)
      req_engine--;
    if (req_chassis > 0 && _tryGetItem(&_producedChassis) == 0)
      req_chassis--;
  }

  switch (robot->robotType) {
  case TypeA:
    _makeItemOnly(TYPE_A_TIME_BODY, &_producedBody);
    break;
  case TypeB:
    _makeItemOnly(TYPE_B_TIME_BODY, &_producedBody);
    break;
  case TypeC:
    _makeItemOnly(TYPE_C_TIME_BODY, &_producedBody);
    break;
  default:
    break;
  }

  return 0;
}
```

makeBody() will call _requestSpace() here.

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

```c
static void _releaseSpace() {
  I_DEBUG_HEAD
  int num_free_space;
  sem_getvalue(&_space, &num_free_space);
  if (num_free_space < _space_limit) {
#ifdef DEBUG
    debug_printf(__func__, "Releasing space, current space=%d...\n",
                 num_free_space);
#endif
    sem_post(&_space);
#ifdef DEBUG
    sem_getvalue(&_space, &num_free_space);
    debug_printf(__func__, "Space released, current space=%d...\n",
                 num_free_space);
#endif
  } else {
    err_printf(__func__, __LINE__,
            "Fatial Error, releasing space that doesn't exist\n")
    exit(-1);
  }
}
```

Once the production 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

```
void makeCar(Robot robot) {
  I_DEBUG_HEAD
  int req_window = 7;
  int req_tire = 4;
  int req_battery = 1;
  int req_body = 1;

  while (req_window > 0 || req_tire > 0 || req_battery > 0 || req_body > 0) {
    if (req_window > 0 && _tryGetItem(&_producedWindow) == 0)
      req_window--;
    if (req_tire > 0 && _tryGetItem(&_producedTire) == 0)
      req_tire--;
    if (req_battery > 0 && _tryGetItem(&_producedBattery) == 0)
      req_battery--;
    if (req_body > 0 && _tryGetItem(&_producedBody) == 0)
      req_body--;
  }

  switch (robot->robotType) {
  case TypeA:
    _makeItemOnly(TYPE_A_TIME_CAR, &_producedCar);
    break;
  case TypeB:
    _makeItemOnly(TYPE_B_TIME_CAR, &_producedCar);
    break;
  case TypeC:
    _makeItemOnly(TYPE_C_TIME_CAR, &_producedCar);
    break;
  default:
    break;
  }
}
```

Q1. Complete the simple multithreaded version

```
/* Prepare task */
Task task = calloc(1, sizeof(Task_t));
task->jobQ = queueCreate(num_cars * 17);
for (int k = 0; k < num_cars; k++){
  for(int i = 0; i < 8; i++) {
    if(i == WINDOW) {
      for(int j = 0; j < 7; j++) queueEnqueue(task->jobQ, &i);
    } else if(i == TIRE) {
      for(int j = 0; j < 4; j++) queueEnqueue(task->jobQ, &i);
    } else queueEnqueue(task->jobQ, &i);
  }
}
/* Prepare task end*/
```

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

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

```
/* Production start */
// Create robot, assign task, and start to work
for (int i = 0; i < num_typeA; ++i) {
  robotsA[i] = createRobot(TypeA);
  robotsA[i]->task = task;
  pthread_create(&robotsA[i]->pthread, NULL, simpleRobotRoutine, robotsA[i]);
}
// TODO: create typeB and typeC robots

// wait until work done
for (int i = 0; i < num_typeA; ++i) {
  pthread_join(robotsA[i]->pthread, NULL);
}
// TODO: join typeB and typeC robot threads

/* Production end */
```

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 philosopher example in T4 → those philosophers produce an agreement to ensure there is no deadlock.
  - 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. Then you may need to rearrange the jobs so that the deadlock can be broken.
  - Hint for this strategy: If you only want your robot wait for certain seconds, you can use those timedTryMakeXXX() functions. Be noted, you should determine a reasonable amount of waiting time. If you wait too long, deadlock detection will degrade the performance of production.

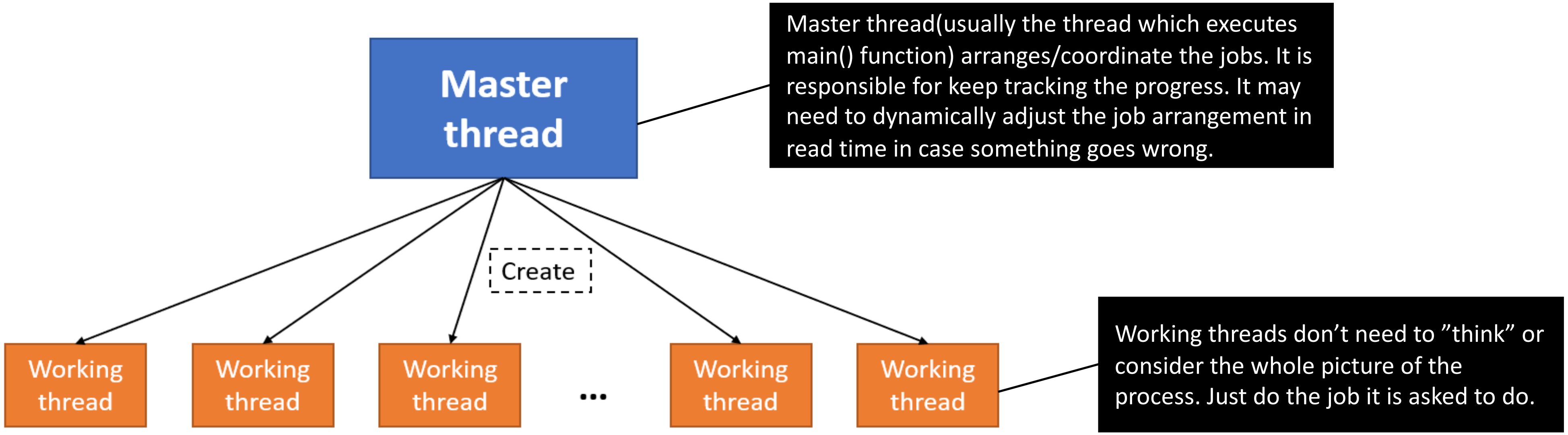
```
int timedTryMakeSkeleton(int waitTime, Robot robot);
int timedTryMakeEngine(int waitTime, Robot robot);
int timedTryMakeChassis(int waitTime, Robot robot);
int timedTryMakeBody(int waitTime, Robot robot);
int timedTryMakeWindow(int waitTime, Robot robot);
int timedTryMakeTire(int waitTime, Robot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

Hint

- Code provided in Q1 is simply an example.
- You can use more queues and semaphores to help you solve deadlock problem and improve the performance.

Common Parallel Programming Technique

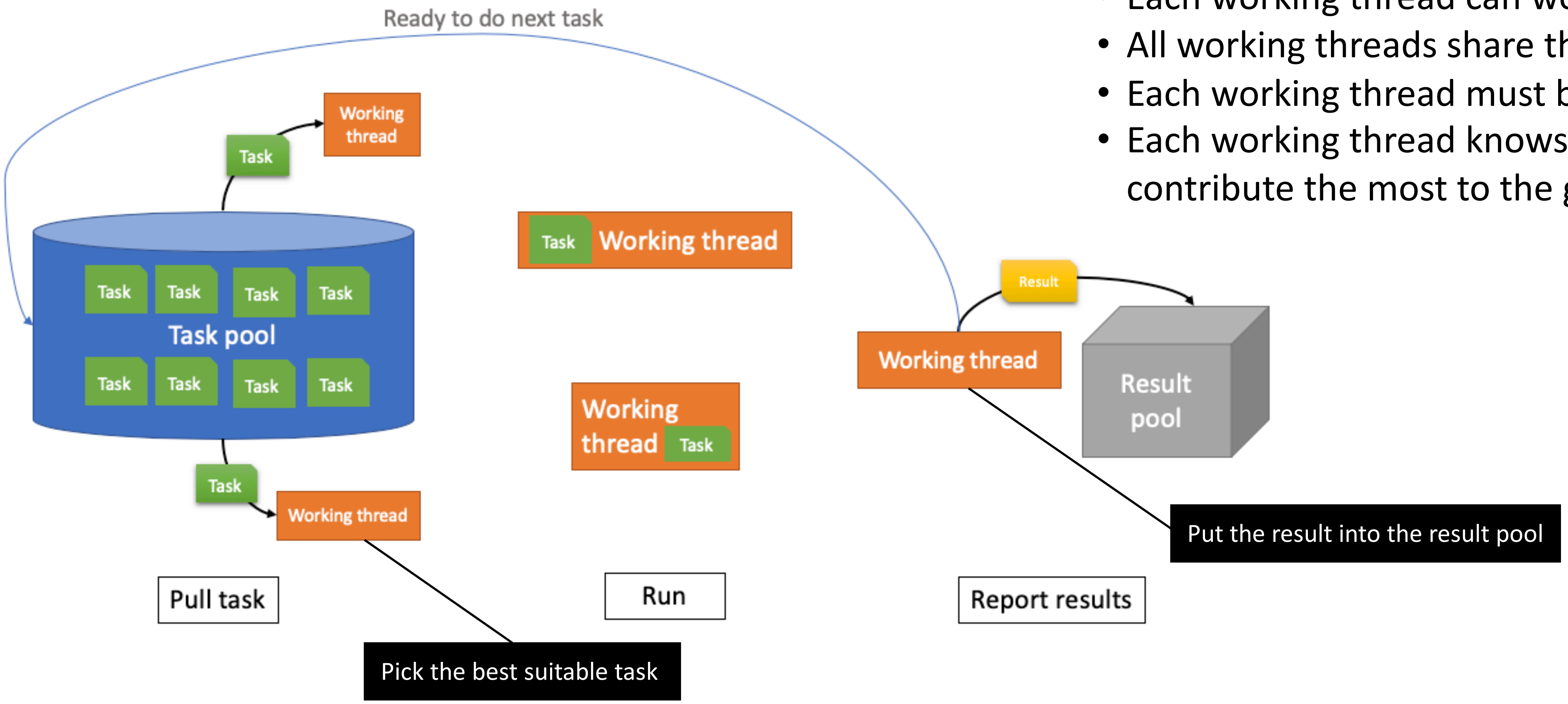
- **Master-slave model**



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**

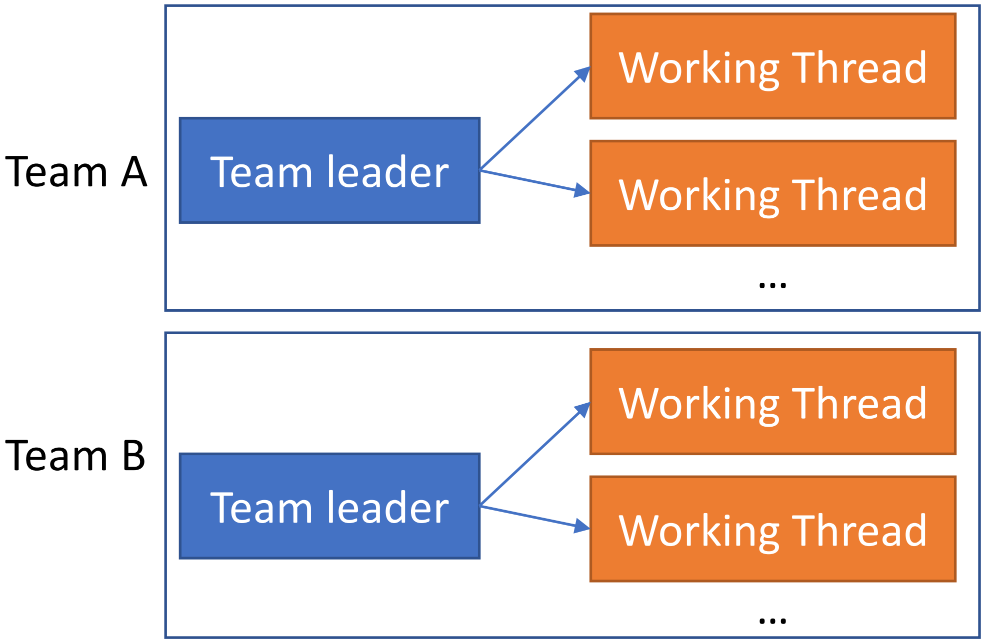


- No centralized command system
- Each working thread can work by their own
- All working threads share the same goal
- Each working thread must be wise
- Each working thread knows what to do to contribute the most to the goal.

Common Parallel Programming Technique

- **Hybrid model**

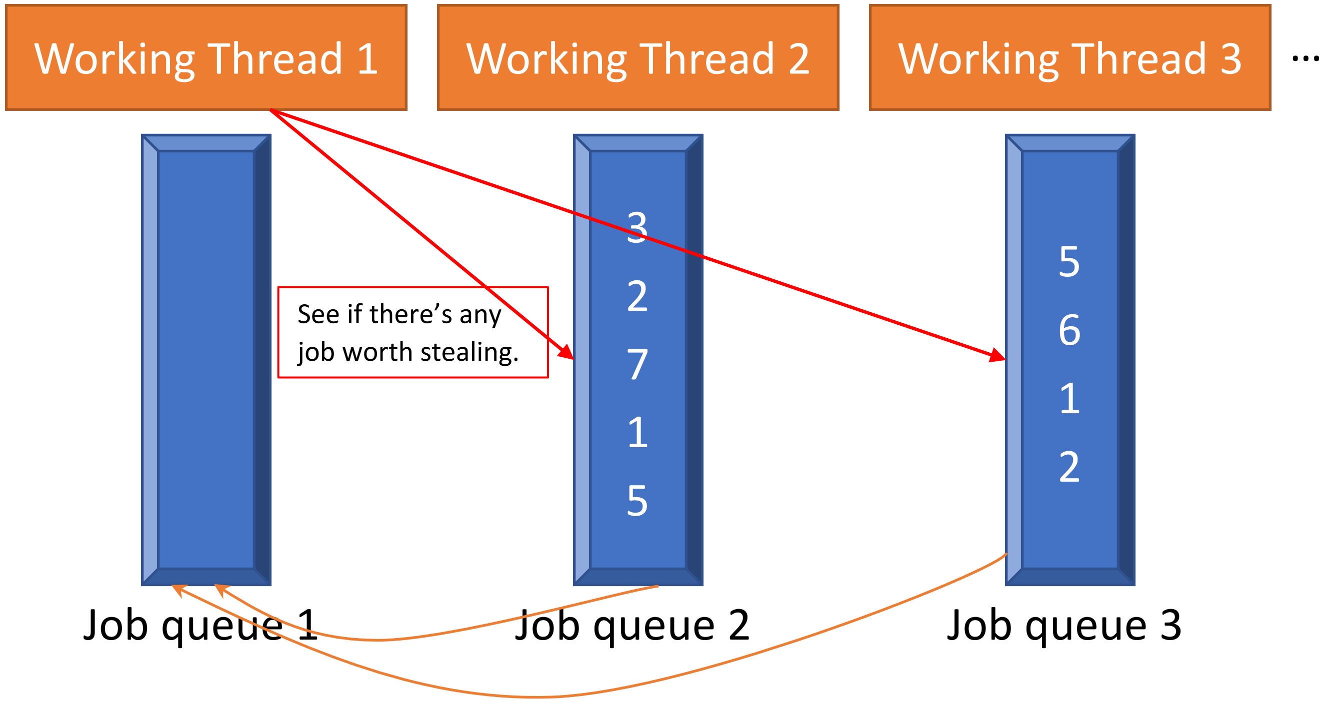
You may combine Master-slave model and autonomous thread model together.
For example, some autonomous working threads can serve as a "team leader" while other working threads just do what the leader tell them to do.



- If the decision-making 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 threads busy to maximize performance.



Example

- Thread 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?