Operating Systems Exam Solutions and Grading Rubric

May 27, 2025

Q1. Short Questions [15 marks]

Note: Questions were given separately - not available for solution or rubric creation.

Q2. Compilation & IPC [5 marks]

Part (a): Compilation, Linking, and Execution [2.5 marks]

i) Compilation Diagram [1.25 marks]

Solution:

Source Files (.c) → [Compiler] → Object Files (.o) → [Linker] → Executable → [Loader] → Memory/Execution

```
module1.c      module1.o
module2.c  --> module2.o --> a.out --> Process in RAM
module3.c      module3.o
```

Rubric:

- **Requirements:**
  - Clear visual representation of compilation → linking → execution flow
  - Must show multiple source files → object files → executable
  - Include hints/labels for each stage
- **Grading Breakdown:**
  - Excellent (1.0-1.25): Complete diagram showing: source files (.c) → compiler → object files (.o) → linker → executable → loader → execution
  - Good (0.75-1.0): Diagram present with most stages but missing 1-2 components
  - Fair (0.5-0.75): Basic diagram with some stages but unclear flow
  - Poor (0-0.5): Incomplete or incorrect diagram
- **Common Student Faults:**
  - Forgetting to show multiple source files
  - Missing the linker stage entirely
  - Confusing compiler and linker roles
  - Not showing object files as intermediate step

ii) Missing Symbols and Unresolved References [1.25 marks]

Solution:

- **Missing Symbols:** Functions/variables declared but not defined in any object file during linking.
- **Unresolved References:** Symbols referenced in code but linker cannot find their definitions across all object files.

Rubric:

- **Requirements:**
  - Define ”missing symbols” in context of linking
  - Define ”unresolved references” in context of linking
  - Relate to Bilal’s multi-module scenario
- **Grading Breakdown:**
  - Excellent (1.0-1.25): Clear explanation that missing symbols occur when linker cannot find definitions for declared functions/variables, and unresolved references are symbols used but not defined
  - Good (0.75-1.0): Generally correct but missing some technical precision
  - Fair (0.5-0.75): Basic understanding but incomplete explanation
  - Poor (0-0.5): Incorrect or no explanation
- **Common Student Faults:**
  - Confusing compilation errors with linking errors
  - Not distinguishing between the two terms
  - Providing examples without conceptual explanation

Part (b): IPC Comparison [2.5 marks]

i) Bidirectional Sensor ↔ Alert Communication [1.25 marks]

Solution:
Shared Memory - Most efficient due to direct memory access, minimal latency, supports bidirectional communication without system call overhead.

Rubric:

No marks w/o shared memory. Minimal latency mentioned in the question

- **Requirements:**
  - Compare pipes, messages, shared memory for bidirectional communication
  - Consider minimal latency requirement
  - Identify most efficient option
- **Expected Answer:** Shared memory (fastest) or named pipes/message queues (structured)
- **Grading Breakdown:**
  - Excellent (1.0-1.25): Correct identification with proper justification considering latency
  - Good (0.75-1.0): Correct choice with basic justification
  - Fair (0.5-0.75): Reasonable choice but weak justification
  - Poor (0-0.5): Incorrect choice or no justification

ii) Safe Concurrent Writes to Log File [1.25 marks]

No marks if synchronization not mentioned explicitly

Solution:
Message Queues - Provides serialization of writes, built-in synchronization, prevents race conditions in file access.

Rubric:

Accept shared memory with mutex/semaphores as correct as well

- **Requirements:**
  - Consider thread safety for concurrent writes
  - Identify best IPC mechanism for file safety
- **Expected Answer:** Message queues or pipes with single writer process
- **Grading Breakdown:**
  - Excellent (1.0-1.25): Correct identification with synchronization considerations
  - Good (0.75-1.0): Correct choice with basic reasoning
  - Fair (0.5-0.75): Partially correct understanding
  - Poor (0-0.5): Incorrect or no answer
- **Common Student Faults:**
  - Not considering concurrency issues
  - Mixing up efficiency with safety requirements
  - Generic answers without scenario-specific reasoning

Q3. Performance and Scheduling [10 marks]

Part (a): Amdahl's Law Analysis [5 marks]

i) Speedup Calculations [1.5 marks]

Solution:

Formula: $S = \frac{1}{(1-P)+\frac{P}{N}}$

- Strategy A: $P = 0.6, N = 4$

$$S = \frac{1}{0.4 + \frac{0.6}{4}} = \frac{1}{0.4 + 0.15} = \frac{1}{0.55} = 1.82$$

- Strategy B: $P = 0.8, N = 2$

$$S = \frac{1}{0.2 + \frac{0.8}{2}} = \frac{1}{0.2 + 0.4} = \frac{1}{0.6} = 1.67$$

Rubric:

- **Requirements:**
  - Apply Amdahl's Law formula: $S = \frac{1}{(1-P)+\frac{P}{N}}$
  - Strategy A: $P = 0.6, N = 4$; Strategy B: $P = 0.8, N = 2$
  - Show complete stepwise calculations
- **Expected Calculations:**
  - Strategy A: $S = \frac{1}{0.4+\frac{0.6}{4}} = \frac{1}{0.55} = 1.82$
  - Strategy B: $S = \frac{1}{0.2+\frac{0.8}{2}} = \frac{1}{0.6} = 1.67$
- **Grading Breakdown:**
  - Excellent (1.25-1.5): Correct formula, correct substitution, correct calculations for both strategies
  - Good (1.0-1.25): Correct approach with minor calculation errors
  - Fair (0.5-1.0): Formula used but significant errors in application
  - Poor (0-0.5): Wrong formula or no systematic approach

ii) Performance Comparison [3.5 marks]

Solution:

Strategy A gives better performance (1.82 > 1.67). Adding more cores provides greater benefit than reducing sequential portion when parallelizable work is substantial. Hardware scaling outperforms software optimization in this scenario.

Rubric:

- **Requirements:**
  - Compare calculated speedups
  - Identify better strategy
  - Provide reasoning (maximum 3 lines as specified)
- **Expected Answer:** Strategy A (1.82 > 1.67) gives better performance
- **Grading Breakdown:**
  - Excellent (3.0-3.5): Correct identification, proper numerical comparison, concise reasoning within 3 lines
  - Good (2.5-3.0): Correct choice with adequate reasoning
  - Fair (1.5-2.5): Correct choice but weak or lengthy explanation
  - Poor (0-1.5): Incorrect choice or no reasoning
- **Common Student Faults:**
  - Mathematical errors in Amdahl's Law application
  - Not showing step-by-step calculations
  - Exceeding the 3-line limit for explanation
  - Misunderstanding parallel vs sequential portions

Part (b): Scheduling Algorithm Comparison Table [2.5 marks]

Solution:

	FCFS	Round Robin	SJF
How is Scheduling done?	Processes executed in arrival order using FIFO queue	Each process gets fixed time quantum, preempted after quantum expires	Process with shortest CPU burst time selected first
Response Time	Poor - long processes delay others	Good - all processes get CPU quickly	Excellent for short jobs, poor for long jobs
How is Context-Switching done?	Only when process completes or blocks	Forced after time quantum expires	Only when process completes or shorter job arrives
Key Advantages	Simple implementation, no starvation	Fair time distribution, good response time	Minimizes average waiting time
One Drawback	Convoy effect with long processes	High context-switch overhead	Starvation of long processes

Rubric:

- **Requirements:**
  - Create table in landscape format
  - Compare FCFS, Round Robin, and SJF
  - Use specified headings from Figure #1 (not provided in document)
- **Grading Breakdown:**
  - Excellent (2.0-2.5): Well-organized table with accurate comparisons across all criteria
  - Good (1.5-2.0): Good table with minor inaccuracies
  - Fair (1.0-1.5): Basic table but missing key comparisons
  - Poor (0-1.0): Poorly organized or largely incorrect
- **Common Student Faults:**
  - Not using landscape format
  - Missing key comparison criteria
  - Inaccurate algorithm characteristics
  - Poor table organization

Part (c): Synchronization with Mutex and Semaphore [2.5 marks]

i) Why Both Mutex and Semaphore? [1.25 marks]

Solution:

- **Mutex M:** Ensures mutual exclusion for critical section entry
- **Semaphore S:** Controls resource availability/counting mechanism

Rubric:

- **Requirements:**
  - Explain the need for both synchronization primitives
  - Relate to the specific scenario described
- **Expected Answer:** Mutex provides mutual exclusion for critical section entry, semaphore controls resource availability/counting
- **Grading Breakdown:**
  - Excellent (1.0-1.25): Clear distinction between mutex (mutual exclusion) and semaphore (resource counting) roles
  - Good (0.75-1.0): Generally correct but less precise
  - Fair (0.5-0.75): Basic understanding but incomplete
  - Poor (0-0.5): Incorrect or no explanation

ii) Effect of Removing Semaphore [1.25 marks]

Solution:

Both processes could enter critical section simultaneously, violating mutual exclusion and causing race conditions.

Rubric:

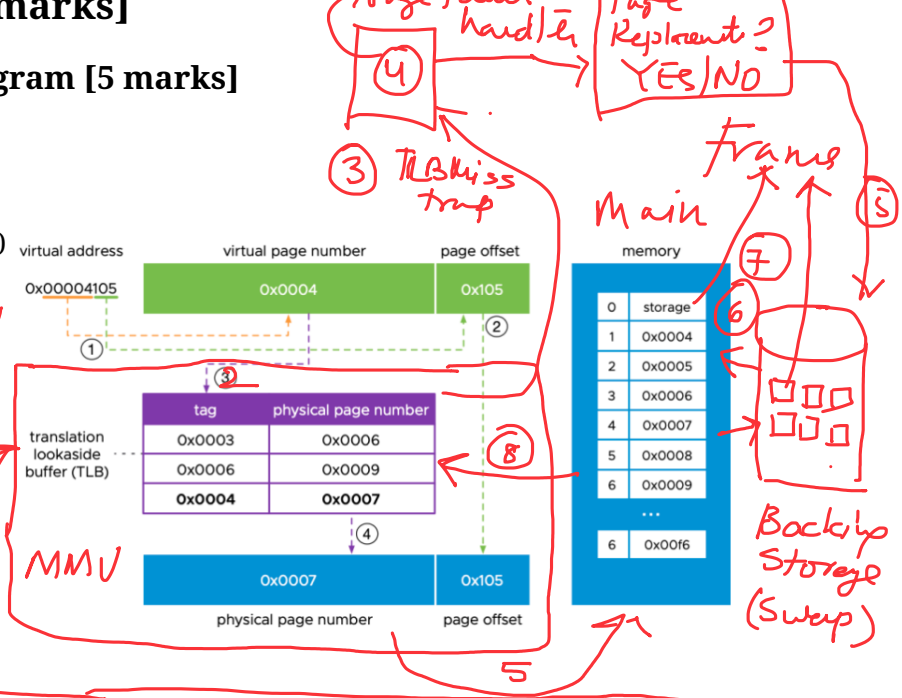
- **Requirements:**
  - Analyze what happens without the semaphore
  - Consider the specific scenario with two processes
- **Expected Answer:** Without semaphore, both processes could enter critical section simultaneously, defeating the purpose of synchronization
- **Grading Breakdown:**
  - Excellent (1.0-1.25): Correct analysis of synchronization failure
  - Good (0.75-1.0): Generally correct understanding
  - Fair (0.5-0.75): Partial understanding
  - Poor (0-0.5): Incorrect or no analysis
- **Common Student Faults:**
  - Confusing mutex and semaphore purposes
  - Not analyzing the specific two-process scenario
  - Generic answers without context

Q4. Memory Management [10 marks]

Part (a): Virtual Memory Access Diagram [5 marks]

Solution:

1. CPU generates Virtual Address
2. Check TLB (Translation Lookaside Buffer)
3. (TLB Miss) Access Page Table in Memory
4. (Page Fault) Page Fault Handler activated
5. Access Swap File on Disk
6. Load Page into Physical Memory
7. Update Page Table Entry
8. Update TLB Entry
9. Restart Memory Access
10. Return Data to CPU



TLB Function: Hardware cache storing recent virtual-to-physical address translations for fast lookup.

Rubric:

- **Requirements:**
  - Complete labeled diagram showing CPU → virtual memory → physical memory → swap file
  - Include TLB location and operation
  - Show numbered flow with technical vocabulary
  - Include persistent storage interaction
- **Required Components:**
  - CPU generating virtual address
  - TLB (Translation Lookaside Buffer) check

- Page table lookup (if TLB miss)
- MMU (Memory Management Unit)
- Physical memory access
- Page fault handling
- Swap file on persistent storage
- Numbered flow sequence

- **Grading Breakdown:**
  - Excellent (4.5-5.0): Complete diagram with: CPU, MMU, TLB, page table, physical memory, swap file, all properly labeled with technical terms and numbered flow
  - Good (3.5-4.5): Most components present but missing 1-2 elements or some labeling issues
  - Fair (2.5-3.5): Basic diagram with major components but unclear flow or missing technical vocabulary
  - Poor (0-2.5): Incomplete diagram or major conceptual errors

- **Common Student Faults:**
  - Missing TLB entirely
  - Not showing swap file interaction
  - Unclear flow sequence
  - Lack of technical vocabulary
  - Not numbering the steps

Not showing TLB miss up trap.
Not showing page fault & Page replacement
Not mentioning Instruction Restart.

Part (b): Second Chance Algorithm Simulation [5 marks]

Solution: * Deduct 1 wt. if given data copied

Because no replacement so far

Page Ref #	Action	Frame (R-Bit) Used	Evicted Page #	Remarks
16	Replace	[16(0), 2(1), 3(0), 9(0), 11(0)]	7	Clock hand at frame 0, R-bit = 0, replace page 7
14	Replace	[16(0), 2(0), 14(0), 9(0), 11(0)]	3	Clock hand finds frame 2, R-bit = 0, replace page 3 WHY?
12	Replace	[16(0), 2(0), 14(0), 12(0), 11(0)]	9	Clock hand at frame 3, R-bit = 0, replace page 9
79	Replace	[16(0), 2(0), 14(0), 12(0), 79(0)]	11	Clock hand at frame 4, R-bit = 0, replace page 11

2 not replaced
2nd chance

Algorithm: Clock hand moves circularly. If R-bit = 1, set to 0 and continue. If R-bit = 0, replace page.

Rubric:

* Page #2 given 2nd chance

- **Requirements:**
  - Complete table for pages 16, 14, 12, 79
  - Show frame states with R-bits
  - Indicate actions (Load/Hit/Replace)
  - Show evicted pages when applicable
  - Provide brief remarks explaining algorithm execution
  - Use landscape table format
- **Key Algorithm Points:**
  - Clock hand moves circularly
  - Check R-bit: if 0, replace; if 1, set to 0 and move to next
  - New pages get R-bit = 0
  - Hits set R-bit = 1
- **Grading Breakdown:**
  - Excellent (4.5-5.0): Complete accurate table with correct R-bit management, proper replacements, and clear remarks
  - Good (3.5-4.5): Mostly correct with minor errors in R-bit handling or remarks
  - Fair (2.5-3.5): Basic understanding but some algorithm errors

- Poor (0-2.5): Major errors in algorithm execution or incomplete table

- **Common Student Faults:**
  - Incorrect R-bit manipulation
  - Not understanding circular pointer movement
  - Copying given data instead of computing new entries
  - Missing or incorrect remarks
  - Not showing evicted page numbers
  - Incorrect frame state representations

Q5. Write C code for any **TWO parts below.** All over attempts shall be penalized with **2 Wt.** [10 mark]

Note: Write straightforward code that addresses multithreading (and synchronization if needed) as per the question. Invalid or filler, or fancy logic will not be graded. Each missing/invalid step and syntax error shall be penalized.

Part (a): Multiprocess with Pipes [5 marks]

i) Main Function [3.5 marks]

Solution:

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int pipes[4][2], n1=10, n2=5;
    for(int i=0; i<4; i++) {
        pipe(pipes[i]);
    }
    for(int i=0; i<4; i++) {
        if (fork() == 0) {
            close(pipes[i][0]);
            int result;
            switch(i) {
                case 0: result = n1 + n2; break;
                case 1: result = n1 - n2; break;
                case 2: result = n1 * n2; break;
                case 3: result = n1 / n2; break;
            }
            write(pipes[i][1], &result, sizeof(int));
            close(pipes[i][1]);
            exit(0);
        }
    }
    int results[4];
    for(int i=0; i<4; i++) {
        close(pipes[i][1]);
        read(pipes[i][0], &results[i], sizeof(int));
        close(pipes[i][0]);
        wait(NULL);
    }
    printf("Add: %d, Sub: %d, Mul: %d, Div: %d\n", results[0], results[1], results[2], results
        [3]);
    return 0;
}
```

Rubric:

- **Requirements:**
  - Create 4 child processes (P1, P2, P3, P4)
  - Each child performs different arithmetic operation
  - Use pipe system calls for communication
  - Parent collects all results

- Display results using printf in main()

- **Grading Breakdown:**
  - Process Creation (1.0 mark): Correct use of fork() for 4 children
  - Pipe Implementation (1.0 mark): Proper pipe creation and usage
  - Arithmetic Operations (1.0 mark): Each child performs different operation
  - Result Collection (0.5 mark): Parent reads from all pipes
  - Code Quality/Syntax (0.5 mark): Proper C syntax and logic flow

- **Common Student Faults:**
  - Not handling pipe creation properly
  - Forgetting to close unused pipe ends
  - Incorrect fork() usage
  - Not waiting for child processes
  - ~~Missing error handling~~ No asked in the question

ii) Gantt Chart [1.5 marks]

Solution:

```
P0: [Create P1,P2,P3,P4] [Wait] [Collect] [Print]
P1: [Execute+] [Exit]
P2: [Execute-] [Exit]
P3: [Execute*] [Exit]
P4: [Execute/] [Exit]
```

Rubric:

- **Requirements:**
  - Show P0 creating P1, P2, P3, P4
  - Show execution timeline
  - Show control returning to P0
  - Show final output phase

- **Grading Breakdown:**
  - Excellent (1.25-1.5): Clear timeline with all process states
  - Good (1.0-1.25): Good representation with minor issues
  - Fair (0.5-1.0): Basic chart but unclear transitions
  - Poor (0-0.5): Incorrect or incomplete chart

Part (b): Pthread Word Counting [5 marks]

i) Main Function [1.5 marks]

Solution:

```c
#include <pthread.h>
#include <stdio.h>
int total_words = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int main() {
    pthread_t threads[100];
    int thread_args[100];
    for (int i = 0; i < 100; i++) {
        thread_args[i] = i;
        pthread_create(&threads[i], NULL, work, &thread_args[i]);
    }
    for (int i = 0; i < 100; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Total: %d\n", total_words);
    return 0;
}
```

Rubric:

- **Requirements:**
  - Create multiple concurrent threads
  - Each thread processes 10,000 lines
  - 1 million total lines (100 threads needed)
  - Wait for all threads to finish
  - Print total word count
- **Grading Breakdown:**
  - Thread Creation (0.5 mark): Correct pthread$_{c}reateusageWorkDistribution(0.5mark)$: $Properdivisionof1Mlines$
  - Synchronization (0.5 mark): Using pthread$_{j}oincorrectly$

In case shared global variable use for word count

ii) Work Function [3.5 marks]

Solution:

```
void* work(void* arg) {
    int thread_id = *(int*) arg;
    int start_line = thread_id * 10000;
    int local_count = 0;
    for(int line = start_line; line < start_line + 10000; line++) {
        local_count += readline_wordcnt("verylarge.txt", line);
    }
    pthread_mutex_lock(&mutex);
    total_words += local_count;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

wrong or complex calculation will result in deduction

Update on global count is a choice and need mutex

Rubric:

- **Requirements:**
  - Use provided readline$_{w}ordcnt()functionProcessassignedlinesforeachthread$
  - Handle concurrent file access
  - Accumulate word counts
- **Grading Breakdown:**
  - Function Signature (0.5 mark): Correct thread function format
  - Line Processing (1.5 marks): Proper use of readline$_{w}ordcnt()ThreadSafety(1.0mark)$: $Handlingconcurrentacces$
  - Logic Flow (0.5 mark): Correct algorithm implementation
- **Common Student Faults:**
  - Incorrect thread function signature
  - Not handling thread arguments properly
  - Race conditions in shared data access
  - Incorrect calculation of line ranges per thread
  - ~~Missing pthread library includes~~

X Not applicable. We have asked two function only main() - work() -

Part (c): Dining Philosophers Variation [5 marks]

i) Problem Modeling [1.5 marks]

Solution: i) Model this situation as a Dining Philosophers Problem. (diagram + justification) **[1.5]**

- Dept1-Charge1-Dept2-Charge2-Dept3-Charge3-Dept4-Charge4-Dept5-Charge5-Dept1
- 5 departments = 5 philosophers, 5 stations = 5 forks, 2 adjacent ports = 2 forks, limit 4 = prevent deadlock.

Rubric:

Mapping using diagram

Deadlock Prevention: By restricting the system to allow only four departments to charge simultaneously, the facilities team ensures that at least one department is not competing for resources, breaking the potential for a circular wait (a necessary condition for deadlock). This is a standard technique in the Dining Philosophers Problem, where a semaphore limits concurrent access to resources.

Justification

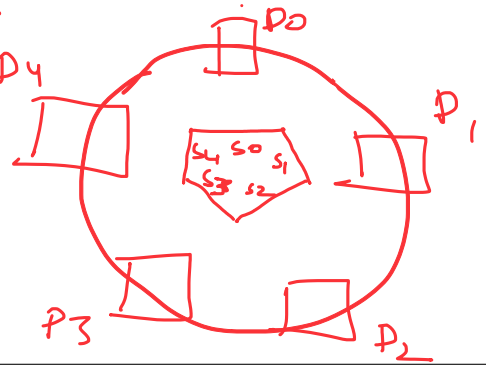
- **Requirements:**
  - Diagram showing 5 departments as philosophers
  - 5 charging stations as forks
  - Justify the mapping
  - Show constraint of only 4 departments charging simultaneously
- **Grading Breakdown:**
  - Diagram (0.75 mark): Clear visual representation
  - Justification (0.75 mark): Proper mapping explanation

ii) Pthread Implementation [3.5 marks]

Solution:

ii) Write a Pthread C code snippet for philosophers. Dry run line by line to show that deadlock is avoided. **[3.5]**
Note: Reproducing generic Dining philosopher code from memory will not be graded.

```c
#include <pthread.h>
#include <semaphore.h>
sem_t room;
pthread_mutex_t stations[5];
void* department(void* arg) {
    int id = *(int*) arg;
    sem_wait(&room);
    pthread_mutex_lock(&stations[id]);
    pthread_mutex_lock(&stations[(id+1) % 5]);
    printf("Dept %d charging \n", id);sleep(10);
    pthread_mutex_unlock(&stations[(id+1) % 5]);
    pthread_mutex_unlock(&stations[id]);
    sem_post(&room);
    return NULL;
}
int main() {
    sem_init(&room, 0, 4);
    return 0;
}
```



Semaphore limits to 4 philosophers, preventing circular wait deadlock.

Rubric:

- **Requirements:**
  - Implement solution preventing deadlock
  - Use semaphore to limit to 4 concurrent philosophers
  - Show line-by-line dry run
  - Demonstrate deadlock avoidance
- **Grading Breakdown:**
  - Semaphore Usage (1.0 mark): Limiting to 4 philosophers
  - Mutex/Lock Implementation (1.0 mark): Proper resource locking
  - Deadlock Prevention (1.0 mark): Correct algorithm logic
  - Dry Run (0.5 mark): Clear execution trace
- **Common Student Faults:**
  - Using generic dining philosophers code without adaptation
  - Not implementing the "only 4 departments" constraint
  - Incorrect semaphore initialization
  - Missing deadlock prevention mechanism
  - Inadequate dry run explanation

Seq#	C Statement	ID	Room	Operation
1	sem_init(&room, 0, 4)	-	4	Initialize room semaphore with capacity 4
2	pthread_mutex_init(&stations[0], NULL)	0	-	Initialize mutex for charging station 0
3	pthread_mutex_init(&stations[1], NULL)	1	-	Initialize mutex for charging station 1
4	pthread_mutex_init(&stations[2], NULL)	2	-	Initialize mutex for charging station 2
5	pthread_mutex_init(&stations[3], NULL)	3	-	Initialize mutex for charging station 3
6	pthread_mutex_init(&stations[4], NULL)	4	-	Initialize mutex for charging station 4
7	pthread_create(&threads[0], NULL, department, &ids[0])	0	-	Create thread for department 0
8	pthread_create(&threads[1], NULL, department, &ids[1])	1	-	Create thread for department 1
9	pthread_create(&threads[2], NULL, department, &ids[2])	2	-	Create thread for department 2
10	pthread_create(&threads[3], NULL, department, &ids[3])	3	-	Create thread for department 3
11	pthread_create(&threads[4], NULL, department, &ids[4])	4	-	Create thread for department 4
12	sem_wait(&room) (Dept 0)	0	3	Department 0 enters room (room count decrements)
13	pthread_mutex_lock(&stations[0]) (Dept 0)	0	-	Department 0 acquires left station (station 0)
14	pthread_mutex_lock(&stations[1]) (Dept 0)	0	-	Department 0 acquires right station (station 1)
15	printf("Dept 0 charging")	0	-	Department 0 begins charging
16	sem_wait(&room) (Dept 1)	1	2	Department 1 enters room (room count decrements)
17	pthread_mutex_lock(&stations[1]) (Dept 1)	1	-	Department 1 blocks waiting for station 1
18	sem_wait(&room) (Dept 2)	2	1	Department 2 enters room (room count decrements)
19	pthread_mutex_lock(&stations[2]) (Dept 2)	2	-	Department 2 acquires left station (station 2)
20	pthread_mutex_lock(&stations[3]) (Dept 2)	2	-	Department 2 acquires right station (station 3)