OS’25 Project

PART IV: **OVERALL TESTING & BONUSES**

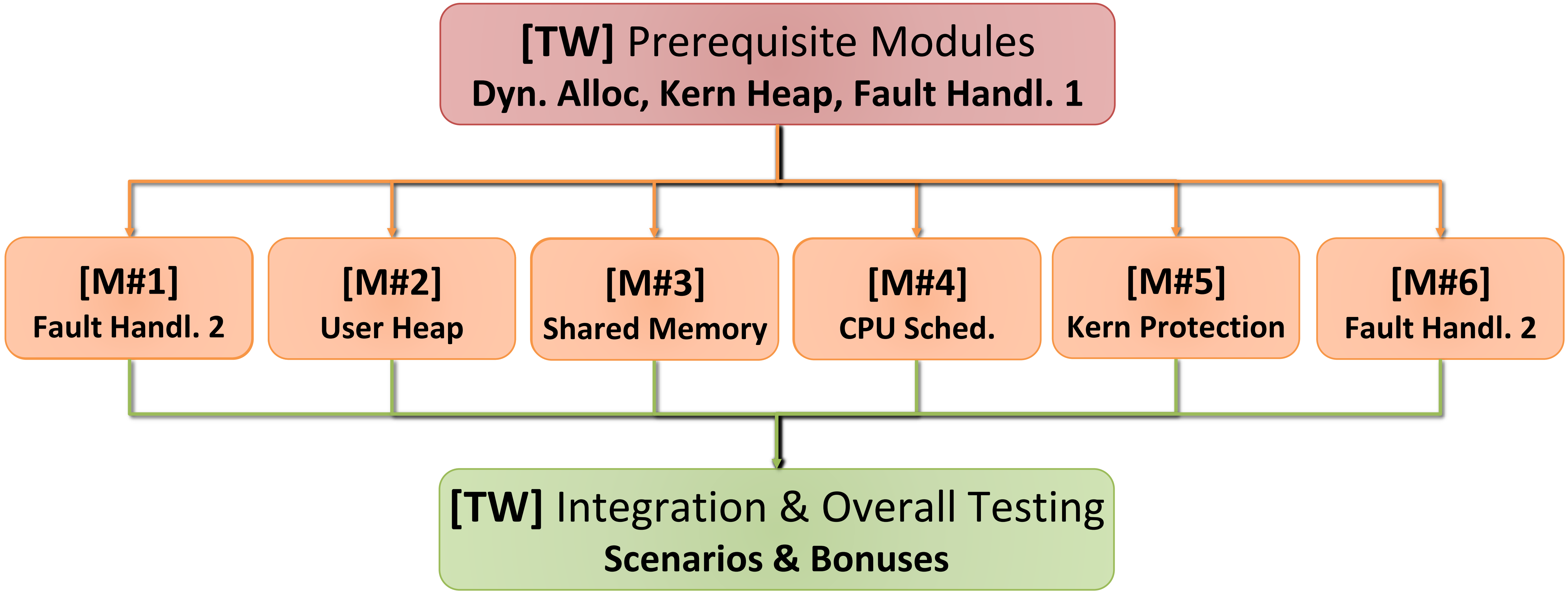
Agenda

Agenda

- **PART IV: OVERALL TESTING & BONUSES**

1. Overall Testing
2. Bonuses

Project Overview



Overall Testing

PART IV: OVERALL TESTING & BONUSES

Objective

Testing the **entire project** and the **interaction**

between different modules

Project ENTIRE Tests: **Intro**

GIVEN

- Set of **ready-made C programs** to test the entire project in different scenarios

REQUIRED

- Use these programs to **test & validate** that the entire project will run successfully

EVALUATION

- **FIVE UNSEEN Scenarios** (1 mark/each). The time limit of each one: **max of 1 min / each**

Project ENTIRE Tests: **Programs**

To run each program: `FOS> run <prog name> <WS Size> [<priority>]`

To load multiple programs: `FOS> load <prog name> <WS Size> [<priority>]`

`…`

`FOS> runall`

I	Program	Params to Play With!
1	***fos_factorial.c (fact):*** calculate the factorial of the given integer (recursive code)	1. Input integer 2. Working set size 3. Priority
2	***fos_fibonacci.c (fib):*** calculate the Fibonacci value of the given index (recursive code)	1. Fibonacci index 2. Working set size 3. Priority
3	***arrayOperations_Master.c (arrop):*** test the shared memory & semaphore modules by creating & initializing a shared array, then run 3 programs that apply different operations on this array (quicksort, mergesort and statistics). The four processes use semaphores for sync.	1. Array size 2. Initializ. (Asc, Ident., Random) 3. Working set size 4. Priority

Project ENTIRE Tests: **Programs**

To run each program: `FOS> run <prog name> <WS Size> [<priority>]`

To load multiple programs: `FOS> load <prog name> <WS Size> [<priority>]`

```
...
FOS> runall
```

I	Program	Params to Play With!
4	***quicksort_noleakage.c (qs1):*** apply the quick-sort recursive algorithm to sort a pre-initialized array of the given size (using malloc()). The created array will be deleted every time (using free())	1. Array size 2. Initializ. (Asc, Desc, Random) 3. Working set size 4. Priority
5	***quicksort_leakage.c (qs2):*** apply the quick-sort recursive algorithm to sort a pre-initialized array of the given size (using malloc()). The created array will NOT be deleted, leading to memory leakage.	1. Array size 2. Initializ. (Asc, Desc, Random) 3. Working set size 4. Priority

Project ENTIRE Tests: **Programs**

To run each program: `FOS> run <prog name> <WS Size> [<priority>]`

To load multiple programs: `FOS> load <prog name> <WS Size> [<priority>]`

…

`FOS> runall`

I	Program	Params to Play With!
6	***mergesort_noleakage.c (ms1):*** apply the merge-sort recursive algorithm to sort a pre-initialized array of the given size (using malloc()). The auxiliary arrays in the "Merge" function are deleted every time (using free())	1. Array size 2. Initializ. (Asc, Desc, Random) 3. Working set size 4. Priority
7	***mergesort_leakage.c (ms2):*** apply the merge-sort recursive algorithm to sort a pre-initialized array of the given size (using malloc()). The auxiliary arrays in the "Merge" function are **NOT** deleted (i.e. causing memory leakage)	1. Array size 2. Initializ. (Asc, Desc, Random) 3. Working set size 4. Priority

Project ENTIRE Tests: **Scenarios**

Possible Scenarios:

1. Run Single Program with Different Params
   1. Can be used to test **ALL** Modules **except** CPU scheduling
   2. To Run any program:

      ```
      FOS> run <prog name> <WS Size> [<priority>]
      ```

   3. Examples for Some Scenarios
      1. Test the same program with **different WS sizes** (very small, medium, large and very large sizes)
      2. Compare **different replacement** algorithms on same WS sizes
      3. Compare **mem-leakage** program **vs. non-leaky** ones

Project ENTIRE Tests: **Scenarios**

Possible Scenarios:

2. Run Multi-Programs at the Same Time with Different Params
    1. Can be used to test **ALL** Modules **including** CPU scheduling
    2. To load multiple programs & run them at once:

       ```
       FOS> load <prog name> <WS Size> <priority>
       FOS> load <prog name> <WS Size> <priority>
       …
       FOS> runall
       ```

    3. Examples for Some Scenarios
        1. Run **instances of the same program** with different priorities. Play with the starvation threshold.
        2. Run **set of programs** with different priorities. Play with the starvation threshold.

Project ENTIRE Tests: **Scenarios**

Possible Scenarios:

3. Internal Kernel Functionalities

SleepLock, Semaphores & Channel sleep/wakeup are currently used in the FOS to:

1. **Lock the Console**:
   - Allow the user program to **print message(s) and receive input** in a **critical section**
   - So, **no other processes** can receive inputs or print messages while another process do
2. **Allow Interrupt-based Keyboard Input**:
   - While a process waiting for a **keyboard input**, it's slept (i.e. **blocked**) until the input is received
   - Upon receiving a char, an **interrupt** is issued to allow the process to get this char
3. **Allow Interrupt-based Disk Access**:
   - While a process waiting for a **faulted page**, it's slept (i.e. **blocked**) until the fault is handled
   - Upon loading from disk, an **interrupt** is issued to allow the kernel thread to get the content

Project ENTIRE Tests: **Scenarios**

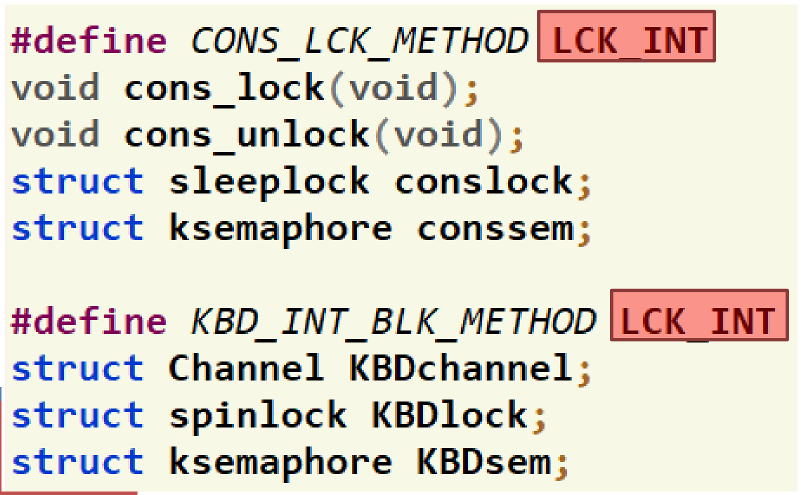
Possible Scenarios:

3. Internal Kernel Functionalities

SleepLock, Semaphores & Channel sleep/wakeup are currently used in the FOS to:

To test the correct functionality of SleepLock, Semaphores & channel sleep/wakeup for the first 2 pts:

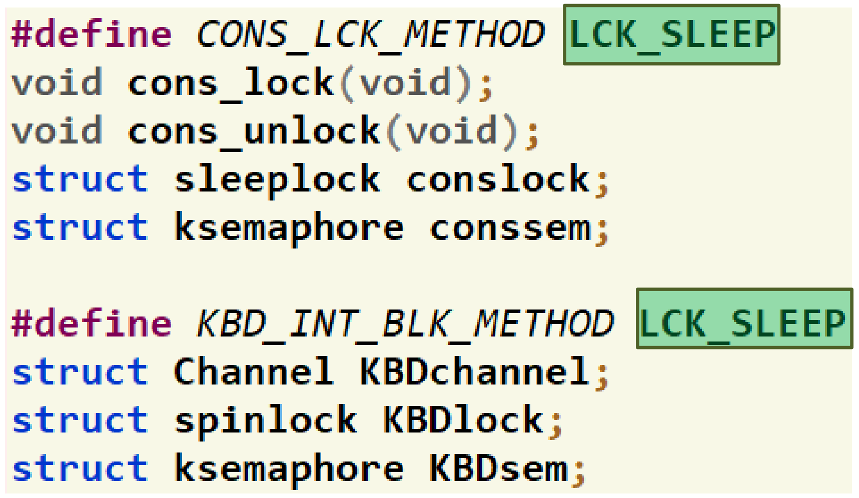
1. Change the method of handling both points from INTERRUPT-based into SLEEP-based, as follows



```
#define CONS_LCK_METHOD LCK_INT
void cons_lock(void);
void cons_unlock(void);
struct sleeplock conslock;
struct ksemaphore conssem;

#define KBD_INT_BLK_METHOD LCK_INT
struct Channel KBDchannel;
struct spinlock KBDlock;
struct ksemaphore KBDsem;
```

(kern/cons/console.h)



```
#define CONS_LCK_METHOD LCK_SLEEP
void cons_lock(void);
void cons_unlock(void);
struct sleeplock conslock;
struct ksemaphore conssem;

#define KBD_INT_BLK_METHOD LCK_SLEEP
struct Channel KBDchannel;
struct spinlock KBDlock;
struct ksemaphore KBDsem;
```

Project ENTIRE Tests: **Scenarios**

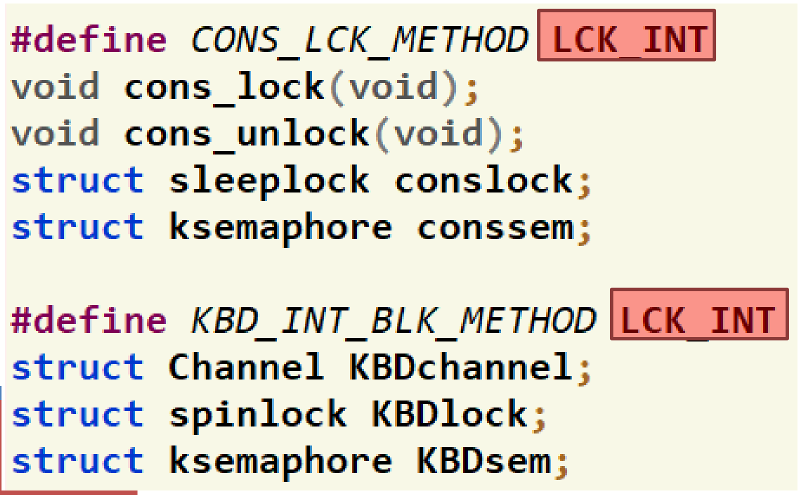
Possible Scenarios:

3. Internal Kernel Functionalities

SleepLock, Semaphores & Channel sleep/wakeup are currently used in the FOS to:

To test the correct functionality of SleepLock, Semaphores & channel sleep/wakeup for the **KB I/O**:

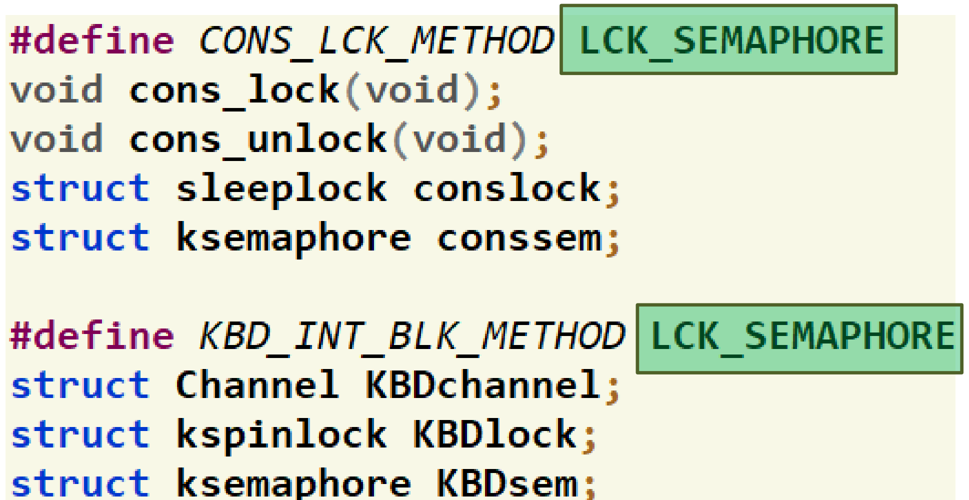
1. Change the method of handling both points from INTERRUPT-based into SLEEP-based, as follows



```
#define CONS_LCK_METHOD LCK_INT
void cons_lock(void);
void cons_unlock(void);
struct sleeplock conslock;
struct ksemaphore conssem;

#define KBD_INT_BLK_METHOD LCK_INT
struct Channel KBDchannel;
struct spinlock KBDlock;
struct ksemaphore KBDsem;
```

(kern/cons/console.h)



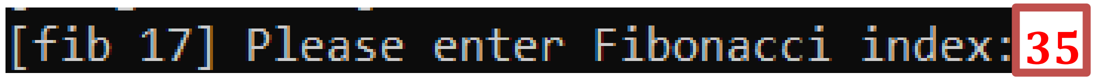
```
#define CONS_LCK_METHOD LCK_SEMAPHORE
void cons_lock(void);
void cons_unlock(void);
struct sleeplock conslock;
struct ksemaphore conssem;

#define KBD_INT_BLK_METHOD LCK_SEMAPHORE
struct Channel KBDchannel;
struct kspinlock KBDlock;
struct ksemaphore KBDsem;
```

Project ENTIRE Tests: **Scenarios**

2. Run the following THREE programs
   - **`FOS> load mergesort`** //apply mergesort on array of size 800,000
   - **`FOS> load fib`** //calculate the Fibonacci at the given index
   - **`FOS> load fib`** //calculate the Fibonacci at the given index
   - **`FOS> runall`** //run all processes

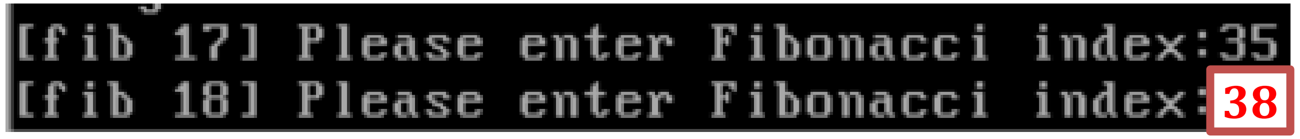
3. During the run, do the following:
   a) When prompting for the Fibonacci index of the **FIRST `fib`** program: write **35** and **press Enter**



```
[fib 17] Please enter Fibonacci index:35
```

Press Enter

   a) When prompting for the Fibonacci index of the **SECOND `fib`** program: write **38, WAIT for ~15sec**



```
[fib 17] Please enter Fibonacci index:35
[fib 18] Please enter Fibonacci index:38
```

WAIT for ~15sec

   a) Upon pressing Enter, BOTH **`mergesort`** and first **`fib`** are **finished <u>IMMEDIATELY</u> without any delay** since both are actually running during the previous **30sec waiting period**
   b) After ~20sec, the second **`fib`** will be finished

➢ Repeat steps 2 & 3 but without the changes in step1 (i.e. using **`LCK_INT`** instead of **`LCK_SLEEP`**): at step 3.c), note all programs will be **running for a noticeable period** of time before any of them is finished

Project ENTIRE Tests: **Scenarios**

Possible Scenarios:

3. Internal Kernel Functionalities

SleepLock, Semaphores & Channel sleep/wakeup are currently used in the FOS to:

To test the correct functionality of SleepLock, Semaphores & channel sleep/wakeup for the **disk I/O**:

1. Change the method of handling the disk from PROGRAMMED-I/O into INTERRUPT-based:

```
#define DISK_IO_METHOD PROGRAMMED_IO
```

(inc/disk.h)

```
#define DISK_IO_METHOD INT_SLEEP
```

```
#define DISK_IO_METHOD INT_SEMAPHORE
```

1. Run any set of programs with **small working set** size and observe the difference in **response time** between the two modes (PROGRAMMED-I/O & INTERRUPT-based)

Bonuses

Bonuses

1. Dynamic Allocator: Block If No Free Block

- If there’s no free block, instead of panic, **BLOCK** the process until a block becomes available

Bonuses

2. Kernel Realloc

1. Attempts to resize the allocated space at given virtual address to "**new size**" bytes, possibly moving it in the heap.
2. If **successful**, returns the **new virtual address**.
3. On **failure**, returns a **NULL** pointer, and the old virtual address remains valid.
4. A call with virtual_address = null is equivalent to kmalloc()
5. A call with new_size = zero is equivalent to kfree()
6. It should work in **[BLOCK ALLOCATOR]** , **[PAGE ALLOCATOR]** and **CROSS-ALLOCATORS**

Bonuses ⭐

3. Efficient Implementation of Kernel Page Allocator

- **EFFICIENT** implementation of the **Page Allocator** using **suitable data structures**
- Compare the performance of at least two different implementations to show the speedup

Bonuses

4. Free the Entire Process (env_free) V.1 (without shared variables)

1. Block allocator of User Heap (if any)
2. All pages in the page working set
3. Working set itself
4. All page tables in the entire user virtual memory
5. Directory table
6. User kernel stack
7. All pages from page file, this code ***is already*** written for you ☺

Bonuses

5. Free the Entire Process (env_free) V.2 (including shared variables)

1. Block allocator of User Heap (if any)
2. All pages in the page working set
3. Working set itself
4. **ALL shared objects (if any)**
5. **ALL semaphores (if any)**
6. All page tables in the entire user virtual memory
7. Directory table
8. User kernel stack
9. All pages from page file, this code ***is already*** written for you ☺

Bonuses

6. FOS Enhancement

- If you **discover** any **issue** in the FOS:
  1. Performance issue
  2. Security/Protection issue
  3. Any other design/technical issue to act as a real OS
- Try to **get a solution** for it. (no implementation, just the solution idea)
- Fill-up this document with TWO main sections:
  1. The **issue** explained in a detailed and clear way.
  2. The **solution** explained in a detailed and clear way.

DELIVERY: Discuss it with the Lecturer in online session isA

THE END

☺ Enjoy **developing** your **own OS** ☺