# OS 2025: MS2 Project Testing Cases

## A- Instructions

Please consider the following IMPORTANT notes regarding the project

1. Test each part from the project independently.
2. After completing all parts, test the whole project using the testing scenarios.
3. The individual tests and scenarios MUST meet the following time limits:

    1. *Scenarios:* ***max of 4 min / each***
    2. *All other individual tests:* ***max of 1 min / each***

4. During your solution, don't change any file EXCEPT those who contain "TODO".
5. In bonuses & challenges, if you change any other file during your solution, **kindly MAKE SURE to tell us when you deliver the code.**

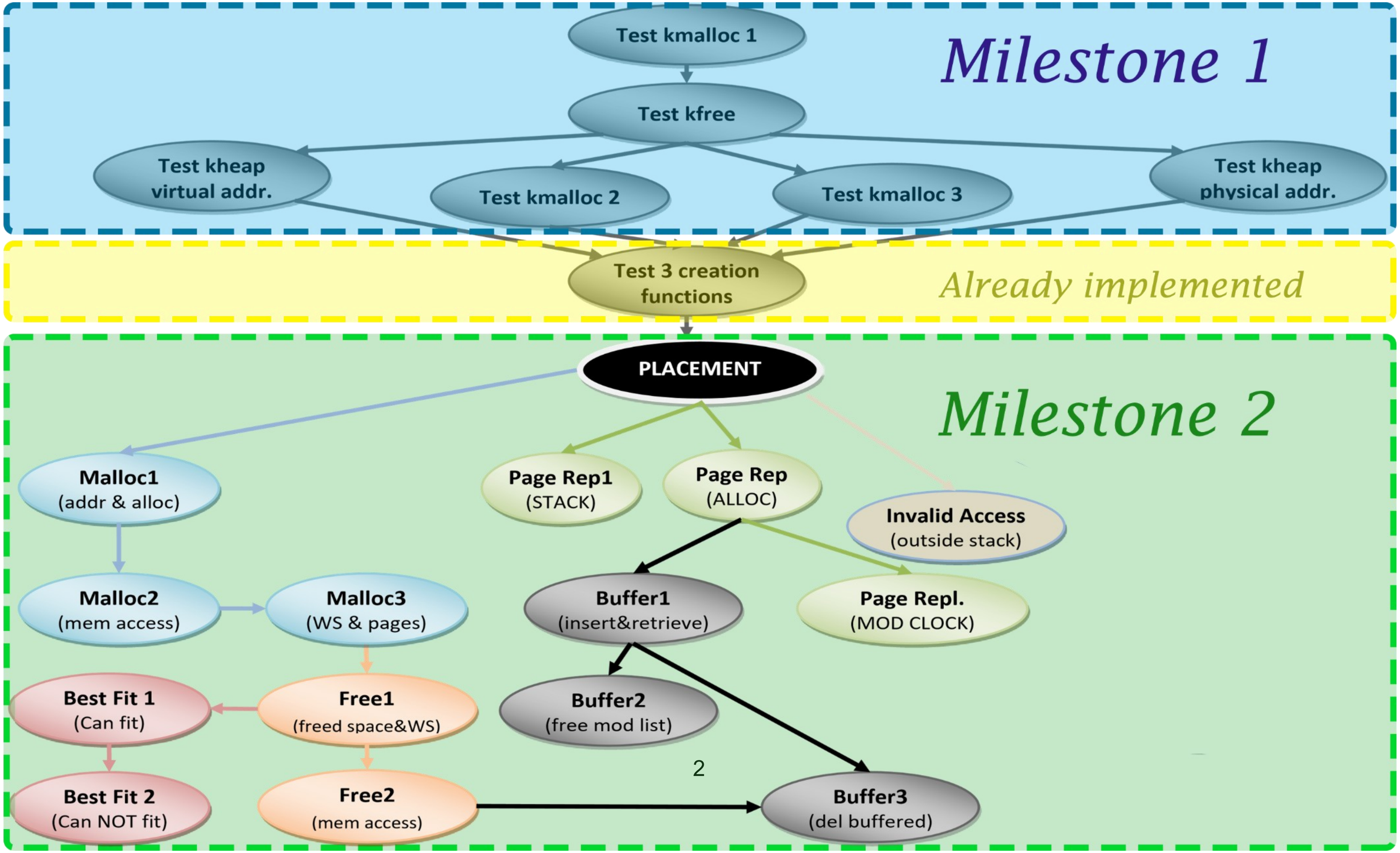
## B- Dependency Graph of Ready-Made Tests

The following graph shows the dependencies between the ready-made tests.

For example:

- To test **Placement**, you first need to successfully test the following: kmalloc, kfree, kheap_virtual_address, kheap_physical_address from **Milestone 1**.

All tests are based on the page placement, which in turn is based on **KERNEL HEAP** tests. So, you need to first implement the KERNEL HEAP functions.



## C- Responsibility of Each Ready-Made Test

The following tables show the main points that each of the test programs will check for!!

| Placement | Invalid Access | Page Replace#1 (Alloc) | Page Replace#2 (Stack) | Page Replace#3 (ModClk) |
|---|---|---|---|---|
| 1. Updating WS & last index<br>2. Mem. Allocation (increased)<br>3. Adding new stack pages to Page File | Illegal memory access to page that's not exist in Page File and not STACK | 1. Mem. Allocation (no change)<br>2. Page File allocation (no change) | 1. Add new stack pages to Page File for 1$^{st}$ time ONLY, then update<br>2. Mem. Allocation<br>3. Victimize and restore stack page | 1. Working set after removing ModClk pages.<br>2. WS last index. (No empty locations in the WS) |

| Buffer 1 | Buffer 2 (free modified list) | Buffer 3 |
|---|---|---|
| 1. Buffering/restoring modif. & not modif. pages from both lists<br>2. Adding to/ Removing from two buffers (list size)<br>3. Page File | 1. Page File allocation (no change)<br>2. Modified list size (decreased)<br>3. Free list size (increased)<br>4. Modified bit (=0)<br>5. Remove pages that belong to ANY env | 1. Freeing modified & not modified from both lists<br>2. Modified list size (decr.)<br>3. Free list size (increased)<br>4. Can't access its pages again |

| Malloc1 | Malloc2 | Malloc3 | Free1 (with placement) | Free2 (with placement) |
|---|---|---|---|---|
| 1. Return addresses (4KB boundary)<br>2. Page File allocation<br>3. Memory allocation (nothing) | Memory access (read & write) of the allocated spaces | After accessing: check num of pages and WS entries | 1. Deleting from page file<br>2. Deleting WS pages<br>3. Deleting empty tables<br>4. Updating WS | 1. Clear entry of dir. & table<br>2. Can't access any page again (i.e. fault on it lead to invalid access) |

| Best Fit 1 | Best Fit 2 |
|---|---|
| Requesting allocations that always fit in one of the free segments. (All requests should be granted) | Requesting allocations that can't fit in any of the free segments. (All requests should NOT be granted) |

# D- Testing Procedures

## FIRST: Testing Each Part

Run every test of the following. If a test succeeds, it will print and success message on the screen, otherwise the test will panic at the error line and display it on the screen.

**IMPROTANT NOTES:**

1. Run each test in **NEW SEPARATE RUN**
2. If the test of certain part is failed, then there's a problem in your code
3. Else, this NOT ensures 100% that this part is totally correct. So, make sure that your logic matches the specified steps exactly

---

## Testing Page Fault Handler:

***tst_placement.c (tpp):*** tests page faults on stack + page placement

```
FOS> run tpp 20
```

***tst_invalid_access.c (tia):*** tests handling illegal memory access (request to access page that's not exist in page file and not belong to the stack)

```
FOS> run tia 15
```

***tst_page_replacement_alloc.c (tpr1):*** tests allocation in memory and page file after page replacement.

```
FOS> run tpr1 11
```

***tst_page_replacement_stack.c (tpr2):*** tests page replacement of stack (creating, modifying and reading them)

```
FOS> run tpr2 6
```

***tst_page_replacement_mod_clock.c (tmodclk):*** tests page replacement by MODIFIED CLOCK algorithm

```
FOS> run tmodclk 11
```

***tst_buffer_1.c (tpb1):*** tests page buffering and un-buffering during replacement

```
FOS> run tpb1 11
```

***tst_buffer_2.c (tpb2):*** tests freeing the modified frame list when it reaches max size

```
FOS> modbufflength 10

FOS> run tpb2 11
```

***tst_buffer_3.c (tpb3):*** tests removing the buffered pages inside freeMem

```
FOS> run tpb3 11
```

## Testing User Heap:

### 1. Testing User Heap Dynamic ALLOCATION for LARGE Sizes:

*tst_malloc_1.c (tm1):* tests the implementation **malloc()** & **allocateMem()**. It validates both the return addresses from the malloc() and the number of allocated frames by allocateMem().

```
FOS> run tm1 2000
```

*tst_malloc_2.c (tm2):* tests the implementation **malloc()** & **allocateMem()**. It checks the memory access (read & write) of the allocated spaces.

```
FOS> run tm2 2000
```

*tst_malloc_3.c (tm3):* tests the implementation **malloc()** & **allocateMem()**. After accessing the memory, it checks the number of allocated frames and the WS entries.

```
FOS> run tm3 2000
```

### 2 . Testing User Heap Dynamic DEALLOCATION for LARGE Sizes:

*tst_free_1.c (tf1):* tests the implementation **free()** & **freeMem()**. It validates the number of freed frames by freeMem().

```
FOS> run tf1 2000
```

*tst_free_2.c (tf2):* tests the implementation **free()** & **freeMem()**. It checks the memory access (read & write) of the removed spaces.

```
FOS> run tf2 2000
```

### 3. Testing User Heap Dynamic ALLOCATION Using FIRST FIT: (If your strategy is first fit)

*tst_first_fit_1.c (tff1):* tests the **first fit strategy** by requesting allocations that always fit in one of the free segments. All requests should be granted.

```
FOS> run tff1 2000
```

*tst_first_fit_2.c (tff2):* tests the **first fit strategy** by requesting allocations that can't fit in any of the free segments. All requests should NOT be granted.

```
FOS> run tff2 2000
```

### 4. Testing User Heap Dynamic ALLOCATION Using NEXT FIT: (If your strategy is next fit)

*tst_nextfit.c (tnf):* tests the Next fit strategy by requesting allocations that either fit of not fit in one of the free segments. Some requests should be granted while others should not.

```
FOS> run tnf 2000
```

### 5. Testing User Heap Dynamic ALLOCATION Using BEST FIT: (If your strategy is best fit)

*tst_best_fit_1.c (tbf1):* tests the **Best fit strategy** by requesting allocations that always fit in one of the free segments. All requests should be granted.

```
▯ FOS> run tbf1 1000
```

*tst_best_fit_2.c (tbf2):* tests the **Best fit strategy** by requesting allocations that can't fit in any of the free segments. All requests should NOT be granted.

```
▯ FOS> run tbf2 1000
```

6. **Testing User Heap Dynamic ALLOCATION Using WORST FIT:** (If your strategy is worst fit)

*tst_worstfit.c (tnf):* tests the **Worst fit strategy** by requesting allocations that either fit of not fit in one of the free segments. Some requests should be granted while others should not.

```
▯ FOS> run twf 2000
```

## Testing env_free function to free all the memory allocated for an environment

1. **test env free without using dynamic allocation/de-allocation**

```
▯ FOS> run tef1 10
```

a success message should be displayed

2. **test env free without using dynamic allocation/de-allocation**

```
▯ FOS> run tef2 20
```

a success message should be displayed

## SECOND: Testing Whole Project

You should run each of the following scenarios successfully

***WRITE ONE OF THE FOLLOWING 4 LINES BEFORE RUNNING ANY OF THE FOLLOWING SCENARIOS BASED ON YOUR KERNEL HEAP STARTEGY:***

```
▯ FOS> khfirstfit //if your strategy is first fit

▯ FOS> khbestfit //if your strategy is best fit

▯ FOS> khnextfit //if your strategy is next fit

▯ FOS> khworstfit //if your strategy is worst fit
```

***AND WRITE ONE OF THE FOLLOWING 4 LINES BEFORE RUNNING ANY OF THE FOLLOWING SCENARIOS BASED ON YOUR USER HEAP STARTEGY:***

```
▯ FOS> uhfirstfit //if your strategy is first fit

▯ FOS> uhbestfit //if your strategy is best fit

▯ FOS> uhnextfit //if your strategy is next fit

▯ FOS> uhworstfit //if your strategy is worst fit
```

## Scenario 1: Running single program to Test ALL MODULES TOGETHER

**REQUIRED MODULES:**

1. KERNEL Heap
2. USER Heap (malloc & free)
3. Page Fault Handler (placement + replacement)

```
FOS> run tqsfh 7  //run tst_quicksort_freeHeap test it
```

according to the following steps:

- Number of Elements = **1,000**

  Initialization method : **Ascending**

  Do you want to repeat (y/n) : **y**

- Number of Elements = **5,000**

  Initialization method : **Descending**

  Do you want to repeat (y/n) : **y**

- Number of Elements = **300,000**

  Initialization method : **Semi random**

  Do you want to repeat (y/n) : **n**

  **"At each step, the program should sort the array successfully"**

## Scenario 2: Running multiple programs with PAGES suffocation

**REQUIRED MODULES:**

1. KERNEL Heap
2. USER Heap (malloc only)
3. Page Fault Handler (replacement)

```
1. FOS> load fib 7          //load Fibonacci program
2. FOS> load tqs 7          //load Quick sort program [with leakage]
3. FOS> load ms2 7          //load Merge sort program [with leakage]
4. FOS> runall              //run all of them together
```

Test them according to the following steps:

**[Fibonacci]**

- Fibonacci index = **30** **"Result should = 1346269"**

**[QuickSort]**

- Number of Elements = **1,000**

  Initialization method : **Ascending**

  Do you want to repeat (y/n) : **y**

- Number of Elements = **1,000**

  Initialization method : **Semi random**

  Do you want to repeat (y/n) : **n**

**"At each step, the program should sort the array successfully"**

**[MergeSort]**

- Number of Elements = **32**

  Initialization method : **Ascending**

  Do you want to repeat (y/n) : **y**

- Number of Elements = **32**

  Initialization method : **Semi random**

  Do you want to repeat (y/n) : **n**

**"At each step, the program should sort the array successfully"**

## Scenario 3: Running multiple programs WITHOUT MODIFIED LIST

**REQUIRED MODULES:**

1. KERNEL Heap
2. USER Heap (malloc only)
3. Page Fault Handler (replacement)

**Run this scenario two times to compare with MAX_MODIFIED_LIST_COUNT = 1 vs. 1000**

***Compare the time between both cases and note the effect of writing each modified victim into H.D.D when MAX SIZE = 1 (also observe the led of H.D.D in the Bochs)***

```
1. FOS> modbufflength 1  // set modified buffer length to be 1
2. FOS> load qs 7            // load Quick sort program [with leakage]
3. FOS> load qs 7            //load Quick sort program [with leakage]
4. FOS> runall          //run both of them together
```

Test them according to the following steps:

**[QuickSort]**

- Number of Elements = **200,000**

  Initialization method : **Semi random**

  Do you want to repeat (y/n) : **n**

**“At each step, the program should sort the array successfully”**

**[QuickSort]**

- Number of Elements = **300,000**

  Initialization method : **Semi random**

  Do you want to repeat (y/n) : **n**

**“At each step, the program should sort the array successfully”**

---

# Enjoy writing your own OS

# □ GOOD LUCK □