OS 2023: 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.**

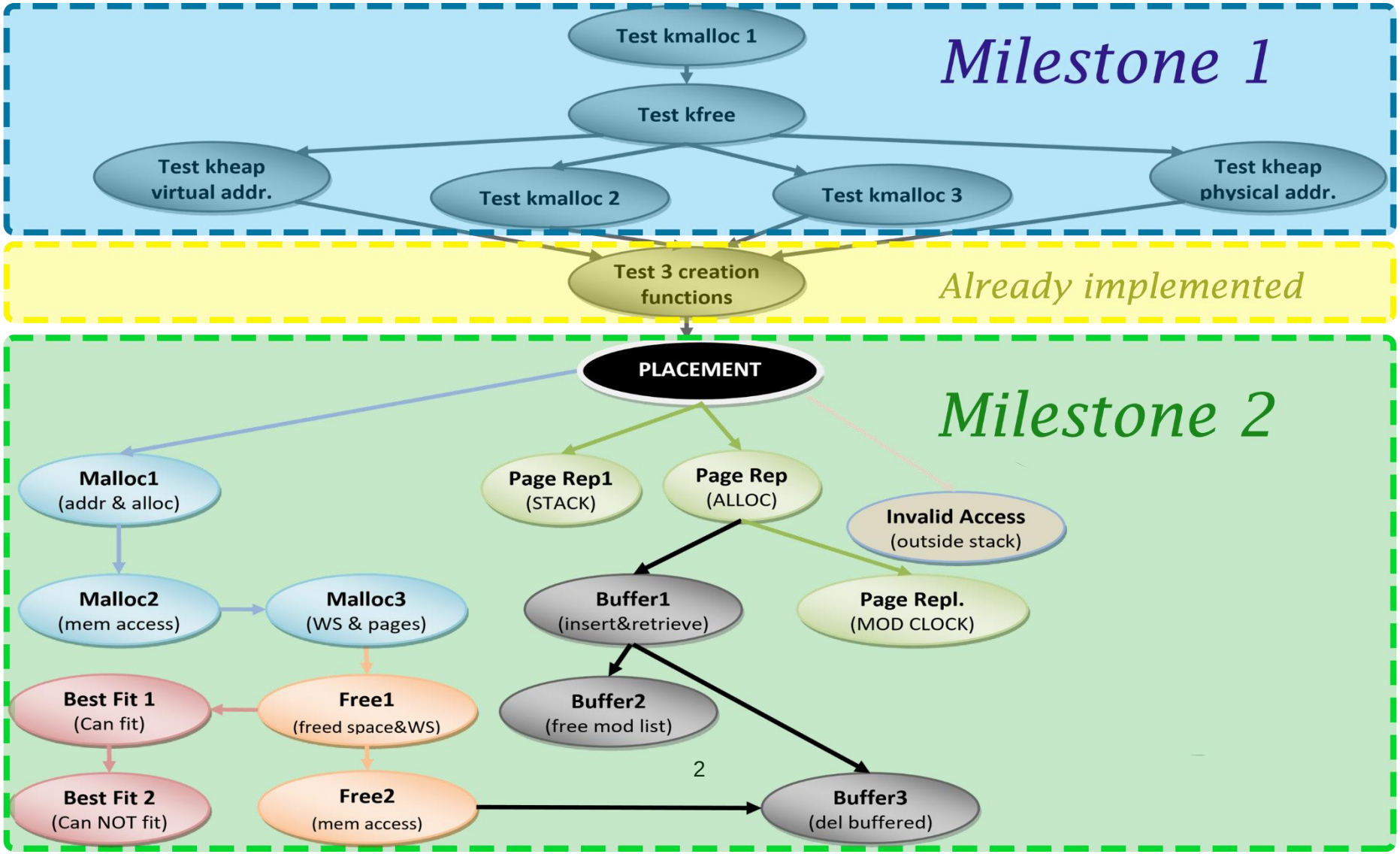
B- Dependency Graph of Ready-Made Tests

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 2. Mem. Allocation (increased) 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) 2. Page File allocation (no change)	1. Add new stack pages to Page File for 1st time ONLY, then update 2. Mem. Allocation 3. Victimize and restore stack page	1. Working set after removing ModClk pages. 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 2. Adding to/ Removing from two buffers (list size) 3. Page File	1. Page File allocation (no change) 2. Modified list size (decreased) 3. Free list size (increased) 4. Modified bit (=0) 5. Remove pages that belong to ANY env	1. Freeing modified & not modified from both lists 2. Modified list size (decr.) 3. Free list size (increased) 4. Can't access its pages again

Malloc1	Malloc2	Malloc3	Free1 (with placement)	Free2 (with placement)
1. Return addresses (4KB boundary) 2. Page File allocation 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 2. Deleting WS pages 3. Deleting empty tables 4. Updating WS	1. Clear entry of dir. & table 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

---

1. 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
```

2. Testing User Heap:

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`

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`

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`

BONUS:

- **Start in bonus after finishing USER HEAP functions**.

1. User Realloc

tst_realloc_1.c (tr1): tests the real location both FIT and NOT FIT in the same location.

- **FOS>** `run tr1 3000`

tst_realloc_2.c (tr2): tests the **SPECIAL CASES** of reallocation, ex: Reallocate (with size = zero, no given address, can't fit , in existing internal fragment).

- **FOS>** `run tr2 3000`

tst_realloc_3.c (tr3): tests the data after reallocation.

- **FOS>** `run tr3 3000`

2. env_free

Scenario 1: without using dynamic allocation/de-allocation.

1. **FOS>** `run tef1 10`

Scenario 2: using dynamic allocation and free.

1. **FOS>** `run tef2 20`

Scenario 3: buffering pages then removing them by calling env_free without using dynamic allocation/deallocation.

1. FOS> ***meminfo*** — ***denote free frames as "X"***
2. FOS> ***load fib 5*** — ***//load Fibonacci program***
   - *Fibonacci index = 30* — *"Result should = 1346269"*
3. FOS> ***load fact 5*** — ***//load Factorial program***
   - *Fact number = 30* — *"Result should = 1409286144"*
4. FOS> ***load fos_add 5*** — ***//load fos_add program***
5. FOS> ***runall*** — ***//run all of them***
6. FOS> ***killall*** — ***//kill all of them***
7. FOS> ***meminfo*** — ***free frames SHOULD equal to "X"***

SECOND: Testing Whole Project

You should run each of the following scenarios successfully

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