OS 2021 Project Testing Cases

A- Instructions

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
6. **Before testing the whole project (scenarios), make sure to add the 2 files in the released folder (UPDATED FILES) and replace the existing ones.**

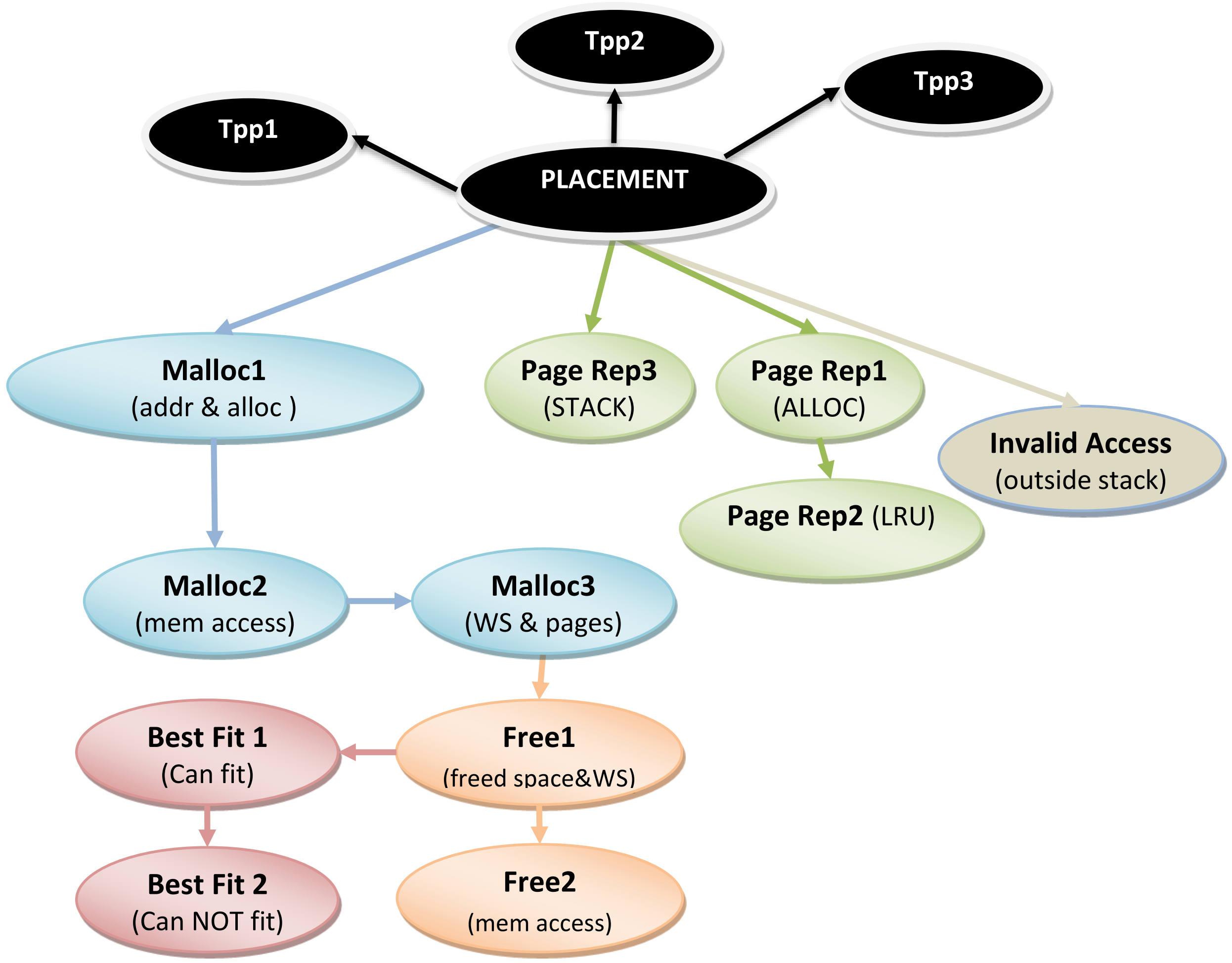
B- Dependency Graph of Ready-Made Tests

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

For example:

- To test **malloc**, you first need to successfully test the following: page fault handler functions.
- On the other hand, testing **Placement** doesn't depend on any test.

All tests are based on the page placement. So you need to first implement the page placement functions and test them with the **given test program**.


Tpp2
Tpp1
Tpp3
PLACEMENT
Malloc1 (addr & alloc)
Page Rep3 (STACK)
Page Rep1 (ALLOC)
Invalid Access (outside stack)
Page Rep2 (LRU)
Malloc2 (mem access)
Malloc3 (WS & pages)
Best Fit 1 (Can fit)
Free1 (freed space&WS)
Best Fit 2 (Can NOT fit)
Free2 (mem access)

C- Responsibility of Each Ready-Made Test

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

Page Placement #1	Page Placement #2	Page Placement #3	Invalid Access
Active list is not full	Active list is full and second list is not full	Fault VA in second List and second list is not full	Illegal memory access to page that's not exist in Page File and not STACK
a. active and second lists content b. Mem. Allocation(increased)	c. Add new stack pages to Page File d. Accessing fault Pages Content		

Page Replace#1 (Alloc)	Page Replace#2 (LRU on 2nd List)	Page Replace#3 (Stack)
1. Mem. Allocation (no change) 2. Page File allocation (no change)	1. LRU lists after handling faults for faulted pages in the second chance list. 2. LRU lists after new page faults to check LRU algorithm on the 2nd chance list.	1. Add new stack pages to Page File for 1st time ONLY, then update 2. Mem. Allocation 3. Victimize and restore stack page

Malloc1	Malloc2	Malloc3	Free1 (with placement)	Free2 (with placement)
1. Return addresses (4KB boundary) 2. Page File allocation 3. Memory allocation (nothing)	1. Memory access (read & write) of the allocated spaces	1. 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 5. Validates the number of freed frames.	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
1. Make number of allocations **2.** Free some of these allocations to make holes. **3.** Make other allocations, that should be allocated in the best fit holes. **4.** Make another continuous hole that should be collapsed. **5.** Make other allocations, that should be allocated into the best fit holes.	1. Make an allocation that is greater than the heap size, so it should not be granted. 2. Make some allocations and holes. 3. Make an allocation with no suitable segment to fit on, this request 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_1.c (tpp1): tests page faults on stack + page placement in case the active list is not FULL.

```
FOS> run tpp1 20 2
```

tst_placement_2.c (tpp2): tests page faults on stack + page placement in case the active list is FULL, and the second active list is NOT FULL.

```
FOS> run tpp2 20 7
```

tst_placement_3.c (tpp3): tests page faults on pages already exist in the second active list (ACCESS).

```
FOS> run tpp3 20 7
```

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 2
```

tst_page_replacement_LRU_Lists_1.c (tpr1): tests allocation in memory and page file after page replacement.

```
FOS> run tpr1 10 5
```

tst_page_replacement_LRU_Lists_2.c (tpr2): tests the LRU algorithm by emitting a page from the second chance for page replacement.

```
FOS> run tpr2 11 5
```

tst_page_replacement_stack_LRU_Lists.c (tpr3): tests page replacement of stack (creating, modifying and reading them)

```
FOS> run tpr3 6 3
```

2. **Testing User Heap Dynamic ALLOCATION:**

tst_malloc_1.c (tm1): tests the implementation **malloc()** & **allocateMem()** for **Sizes > 2KB**. It validates both the return addresses from the malloc() and the number of allocated frames by allocateMem().

- `FOS> run tm1 2000 0`

tst_malloc_2.c (tm2): tests the implementation **malloc()** & **allocateMem()** for **Sizes > 2KB**. It checks the memory access (read & write) of the allocated spaces.

- `FOS> run tm2 2000 0`

tst_malloc_3.c (tm3): tests the implementation **malloc()** & **allocateMem()** for **Sizes > 2KB**. After accessing the memory, it checks the number of allocated frames and the WS entries.

- `FOS> run tm3 2000 0`

3. **Testing User Heap Dynamic DEALLOCATION:**

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

- `FOS> run tf1 1000 200`

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

- `FOS> run tf2 1000 200`

4. **Testing User Heap Dynamic ALLOCATION Using 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 2000 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 2000 1000`

SECOND: Testing Whole Project

You should run each of the following scenarios successfully

IMPORTANT NOTE: **Before testing the whole project (scenarios), make sure to add the 2 files in the released folder (UPDATED FILES) and replace the existing ones.**

Scenario 1: Running single program to Test ALL MODULES TOGETHER

REQUIRED MODULES:

1. USER Heap (malloc & free)
2. Page Fault Handler (replacement – w/o SecondList (i.e. FIFO))

```
FOS> run tqsfh 7 0        //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. USER Heap (malloc only)
2. Page Fault Handler (replacement)

```
1. FOS> load fib 7 0        //load Fibonacci program
2. FOS> load tqs 7 0        //load Quick sort program [with leakage]
3. FOS> load ms2 7 0        //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"

You should run each of the following scenarios successfully

Scenario3: Approx-LRU vs FIFO

REQUIRED MODULES:

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

Approx-LRU

```
FOS> run qs 500 250       //run QuickSort with NO memory leakage
```

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

  Initialization method : **Semi-Random**

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

  "the program should sort the array successfully"

FIFO

```
FOS> run qs 500 0         //run QuickSort with NO memory leakage
```

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

  Initialization method : **Semi-Random**

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

  "the program should sort the array successfully"

Which is faster? Which cause fewer # disk reads & writes?

[BONUS] Scenario4: Fast Implementation of the Best Fit

REQUIRED MODULES:

1. USER Heap
2. Page Fault Handler

If BEST FIT runs in less than 10 sec, take your BONUS ☺

```
1. FOS> load ms1 20 0
2. FOS> load ms1 20 0
3. FOS> runall              //run both of them together
```

Test them according to the following steps:

[MergeSort]

- Number of Elements = **64**

  Initialization method : **Ascending**

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

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

[MergeSort]

- Number of Elements = **64**

  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 ☺