CS2106: Operating Systems
Lab 4 – Memory Management Simulation

Important:

- The deadline of submission through E-Submit system is **3rd April 5pm**
- The total weightage is 4%:
  - Exercise 1: 2%
  - Exercise 2: 2%
- System/OS restriction: Use **any Unix based** OS for this lab. You can either use **Sunfire** or any **Linux** installation.

Section 1. Overview

- There are **two exercises** in this lab. In lecture 8, we have learned about basic memory management. In this lab, we are going to **simulate** the process of managing physical memory. As memory management is performed by the Operating System internally, it is not easy to understand the actual process from outside. Hence, we are trying to write a **simulation** of memory allocation to have a deeper understanding of this process.

General outline of exercises:
- Exercise 1: Simulating the dynamic partitioning using various searching algorithms.
- Exercise 2: Simulating the buddy system partitioning.

Section 2. Exercises in Lab 4

2.1 Exercise 1

General Algorithm Description

In dynamic partitioning, the partition information is kept in a linked list. Each list node holds the following information:
- ***Starting Memory address***
- ***Size of partition***
- ***Status of partition (occupied or free)***
- ***Pointer to the next node.***

Initially, the whole memory range is a single free partition. There are two types of memory requests:

1. **Memory Allocate:** A memory size `M` is specified. The partition information list is search for a free partition that can fit `M`. The three searching algorithms that we are going to use are:
    a. **First Fit-** The list is searched from the beginning, the first partition that is large enough to fit `M` is chosen
    b. **Best Fit** – The list is searched from the beginning to end, the **smallest partition** that is large enough to fit `M` is chosen. When there are multiple suitable partitions, the **first** of such partitions is used
    c. **Worst Fit** – The list is searched from the beginning to end, the **largest partition** that can fit `M` is chosen. When there is multiple suitable partitions, the **first** of such partitions is used

    Suppose a free partition with size `N` is found, it is then split into two partitions:
    - Partition `P` with the first `M` bytes, is allocated to meet the request.
        a. **The starting address of P is returned as the result**
    - Partition `Q` with the remaining `N-M` bytes, is a new free partition.

    It is possible to fail the allocate operation when no suitable partition can be found.

2. **Memory Free:** The starting address `S` of a partition is specified. The partition info list is searched to find the occupied partition with starting address `S`. If found, that partition `P` is indicated as free.
    a. Check the preceding block of `P`, if the block is free, then the two free blocks are merged as a new free block
    b. Check the succeeding block of `P`, if the block is free, then the two free blocks are merged as a new free block

    In the best case, both the preceding and succeeding blocks of `P` are free, then the three free blocks can be merged as a new free block.

    Note that it is possible to fail the free operation when the specified address **S** is not an occupied block or not a valid starting address.

Program Specification

Write a program that perform the allocate and free operations as described above. The program takes the following **command line arguments**
- **Initial Free Memory Size**
- **Search Algorithm to use:**
    - **1 == First-Fit**
    - **2 == Best-Fit**
    - **3 == Worst-Fit**

Example:

```
a.out 1024 2                //1024 free memory, use Best-Fit Search Algorithm
```

Input Specification:

```
N       // Number of operations
```

N lines of input should follow, with the following format

```
Request Argument     //Request = 1 for allocate, 2 for free
                     //Argument = argument for the request
                     //For allocate, argument specify requested memory size
                     //For free, argument specify the starting address
```

Sample Input:

```
6               //6 requests
1 100           //request for 100 bytes
1 50            //request for 50 bytes
1 1000          //request for 1000 bytes
2 0             //free the occupied block at address 0
2 0             //free the occupied block at address 0
2 100           //free the occupied block at address 100
```

Output Specification:

Each request will generate one line of output.

- For allocate request, output the following:
  starting address of allocated block OR
  -1 if request cannot be satisfied

- For free request, output the following:
  `ok` for successful deallocation OR
  `failed` if request cannot be satisfied

At the end of program, compute the following statistic:

- Total occupied region in bytes
- Total free region in bytes
- **`Free region / Total Memory Size * 100%`**
  - an indication of ***external fragmentation***

Sample output (assuming the initial size of 1024, using Best-Fit algorithm):

```
0          //Successful allocation for "1 100" request
100        //Successful allocation for "1 50" request
-1         //Failed allocation for "1 1000" request
ok         //Successful deallocation for "2 0" request
failed     /Failed deallocation for "2 0" request
ok         //Successful deallocation for "2 100" request

0          //Occupied region
1024       //Free region
100.00     //External Fragmentation percentage, total waste in this example ☺
```

Skeleton File **`ex1.c`**:

To simplify the exercise, the skeleton file **`ex1.c`** has several helper functions coded for you. A simple demo that shows the written functions as well as basic way to approach the question is also included in the **`main()`**. You can simply compile the program by "**`gcc ex1.c`**" and run it to get some basic idea.

Note that you are **free to write the program from scratch if you want ☺**.

Usage of ex1:

You can see that ex1 can serve as an experiment tool to compare/contrasts the various searching algorithms. Using the same set of requests, you can try using different fitting algorithms to see the final fragmentation ratio. Also, the percentage of successful allocation is also an interesting statistic.

2.2 Exercise 2

The description of buddy system is slightly changed from the lecture note version to make your implementation easier.

In buddy system, the partition information is kept in an array of linked lists. The array **`A`** has **`(K+1)`** entries, where each entry at index **`I`** represents the free block(s) of size 2^I. Each partition is represented as a node that contains:

- ***Starting Memory address***
- ***Actual Occupied Size*** //when request size is < partition size
- ***Status*** //Occupied or Free
- ***Pointer to the next node***

The two types of memory operations can be handled as follows:

```
1. Memory Allocate: A memory size M is specified.
   i.  Find the smallest S, such that 2^S >= M
   ii. Access A[S] for a free block
          a. If free block exists:
                ❑ Allocate the block by changing the partition status
                ❑ Indicate the actual occupied size M
          b. Else
                ❑ Find the smallest R from S+1 to K, such that A[R] has a
                  free block B
                ❑ For (R-1 to S)
                      ❑ Repeatedly split B ➔ A[S...R-1] has a new free
                        block
                ❑ Goto Step ii
```

Note that allocate can fail if no suitable free block can be found.

2. **Memory Free:** The starting address S of an occupied partition P is specified.
   i. Check in **A[S]**, where **2^S == size of P**
   ii. If the buddy **Q** of **P** exists (also free)
      i. Remove **P** and **Q** from list
      ii. Merge **P** and **Q** to get a larger block **P'**
      iii. Goto step **i**, where **P 🡐 P'**
   iii. Else (buddy of **P** is not free yet)
      iv. Indicate **P** as free in **A[S]**

<u>Program Specification</u>

Write a program that perform the allocate and free operations using Buddy System. The program takes the following **command line arguments**

- **Initial Free Memory Size**

Example:

```
a.out 1024          //1024 free memory
```

You can assume the initial free memory size is some power-of-two.

<u>Input Specification:</u>

```
N       // Number of operations
```

N lines of input should follow, with the following format

```
Request Argument    //Request = 1 for allocate, 2 for free
                    //Argument = argument for the request
                    //For allocate, argument specify requested memory size
                    //For free, argument specify the starting address
```

Sample Input:

```
6                   //6 requests
1 300               //request for 300 bytes
1 512               //request for 512 bytes
1 1000              //request for 1000 bytes
2 0                 //free the occupied block at address 0
2 300               //free the occupied block at address 300
2 512               //free the occupied block at address 512
```

Output Specification:

Each request will generate one line of output.

- For allocate request, output the following:
  starting address of allocated block and block size OR
  -1 if request cannot be satisfied

- For free request, output the following:
  `ok` for successful deallocation OR
  `failed` if request cannot be satisfied

At the end of program, compute the following statistic:

- Total occupied partitions in bytes
- Total free partitions in bytes
- **`Wasted memory / Total Memory Size * 100%`**
  - as an indication of **internal fragmentation**
  - **`Wasted memory = Block size – Actual occupied size`**

Sample output (assuming the initial size of 1024):

```
0 512        //Successful allocation for "1 300" request with block size 512
512 512      //Successful allocation for "1 512" request with block size 512
-1           //Failed allocation for "1 1000" request
ok           //Successful deallocation for "2 0" request
failed       //Failed deallocation for "2 300" request
ok           //Successful deallocation for "2 512" request

0            //Occupied region
1024         //Free region
0.00         //Internal Fragmentation percentage, no waste in this example ☺
```

Skeleton File **`ex2.c`** and general advice:

The given skeleton file **`ex2.c`** comes with the following:

- A sample declaration of the node used by the array **`A[]`** in buddy system
- Sample code to declare and initialize **`A[]`**
- A few helper functions to get you started

You can try out the demo program by "**`gcc ex2.c`**", then "**`a.out XXXX`**", where **`XXXX`** is the initial memory size. For the demo purpose, use a small initial size that is > 32.

Note that the demo code is given as a guide only. You are free to rewrite the code from scratch.

General Advice:

- The size of an occupied block can be found by exhaustively going through all the blocks in `A[]`.
    - ▪ You are free to find a more efficient way (not a requirement).
- Although the general idea of block splitting is shown in the skeleton code, it may not be the best/easiest way to do it. You are free to redesign the whole code.

Section 3. Submission

Submit **only the following files:**

a. **`ex1.c`**
b. **`ex2.c`**

Head over to the following website:

https://mysoc.nus.edu.sg/~esubmit/assignment/prog/submit/

Use your NUSNET id and password to login. The submission procedure should be quite intuitive. However, if you have problem, don't hesitate to post it in the IVLE forum or email the lecturer at (sooyj@comp.nus.edu.sg).