CS 3423 Operating Systems
Fall Semester 2017
Prof. Pai H. Chou

Assignment 8

Due Date: Sunday, November 12, 2017, 11:59pm
Up to one-day late submission without penalty
Up to one-week late submission with 20% penalty
Submit electronically on iLMS

What to submit: One zip file named <studentID>-hw8.zip (replace <studentID> with your own student ID). It should contain four files:

- one PDF file named **hw8.pdf** for Section 1 and Section 2.3 (bonus). Write your answers in **English**. Check your spelling and grammar. Include your name and student ID!
- Section 2: Python source files. Include your name and student ID in the program comments on top.
    - Section 2.1, 2.2: **memalloc.py**, **typescript1**
    - Section 2.3 (bonus): show your code and output in your malloc.py and typescript 1, plus written explanation in the PDF.

1. [40 points] Problem Set

1. [10 points] 8.15 Consider a logical address space of 256 pages, with a 4KB page size, mapped onto a physical memory of 64 frames.
    a. How many bits are required in the logical address?
    b. How many bits are required in the physical address?
2. [20 points] 8.16 Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512-MB of physical memory. How many entries are there in each of the following?
    a. A conventional single-level page table
    b. An inverted page table
3. [10 points] 8.20 Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0, 430 (segment#, logical address)
b. 1, 10
c. 2, 500
d. 3, 4000
e. 4, 112

2. [60 points] Programming Exercise

In this programming exercise, you are to implement algorithms for contiguous memory allocation, similar to `malloc()` and `free()` in the standard library (`stdlib`).

`malloc()`, for memory-allocate, is a stdlib function for dynamically allocating a contiguous block of memory. The parameter is the number of bytes to allocate. The return value is the pointer (here an `int` in Python) to the allocated memory block, or `None` if it cannot be allocated, possibly due to memory fragmentation.

`free()` will free a previously allocated memory (as returned by a previous call to `malloc()`). The textbook talks about three policies: First-Fit, Best-Fit, and Worst-Fit. You are to implement all three policies in Python. Use the following API:

```
class MemAlloc:
   _POLICIES = ('FirstFit', 'BestFit', 'WorstFit')
   def __init__(self, totalMemSize, policy = 'BestFit'):
      if not policy in MemAlloc._POLICIES:
         raise ValueError('policy must be in %s' % MemAlloc._POLICIES)
      self.allocation = { } # use this dictionary to map allocated
                            # pointer to the allocated size
      # keep a list of holes, which are tuples with (pointer, size)
      self.holes = [(0, totalMemSize)] # sorting by pointer
      # your own code here …

   def malloc(self, reqSize):
      '''return the starting address of the block of memory, or None'''
      # your code here

   def free(self, pointer):
      '''free the previously allocated memory starting at pointer'''
      # your code here
```

You will find some test cases in the template file. Rename it `memalloc.py`

2.1 [30 points] malloc()

malloc() and free() use of two data structures:

- list of holes (`self.holes`), which consists of tuples (address, size)
- mapping from allocated addresses to sizes (`self.allocation`)

malloc() will iterate over the list of holes, kept in sorted order by address.

- if the policy is First-Fit, then it uses the first hole that is big enough to serve the requested size

- if the policy is Best-Fit, then it continues looking for the smallest hole that is big enough to serve the requested size.
- if the policy is Worst-Fit, then it looks for the biggest hole that can serve the requested size.

If no holes are big enough, then `malloc()` returns `None`.
But if there is one hole that can work, then

- if the chosen hole is used up completely by this malloc request, then it should be deleted from the list of holes.
- Otherwise, if there is still some remaining unused space in this hole, then update the hole's address and size.

In any case, the new allocation should be recorded in the `self.allocation` dictionary. Use the address as the key and size as the value. Finally, return the address for the newly allocated memory chunk.

You should test this part thoroughly, possibly with your own test cases, before proceeding to the next part.

2.2 [30 points] free(void* p)

`free()` is the inverse operation of `malloc()`. It takes a previously allocated address as parameter, looks up the size from the allocation, and

- update the holes list
- delete the freed entry from the allocation dictionary

Updating the holes list is potentially the tricky part, because there are several possible cases. Let (p, s) denote the pointer to the freed block and the size of the block to be freed.

- if empty holes list: just add (p, s) to the holes list.
- if (p, s) goes before the first hole on the list:
  - if (p, s) and first hole are disjoint, just prepend (p, s) to holes list
  - if contiguous, then merge (p, s) into the first hole by updating the first hole's starting address and size.
- if (p, s) goes after the last hole on the list:
  - mirror image to the "before the first hole"
- if (p, s) goes between hole [i] and hole [i+1]:
  - if all three are contiguous, merge all three (and delete hole [i+1])
  - if (p, s) contiguous with [i], merge them
  - if (p, s) contiguous with [i+1], merge them
  - if all three are disjoint, insert (p, s) between [i] and [i+1] on the list

Here is sample output:

```
a=malloc(10):
 FirstFit symbols={'a': 0} holes=[(10, 10)] allocation={0: 10}
 BestFit symbols={'a': 0} holes=[(10, 10)] allocation={0: 10}
 WorstFit symbols={'a': 0} holes=[(10, 10)] allocation={0: 10}
b=malloc(1):
 FirstFit symbols={'a': 0, 'b': 10} holes=[(11, 9)] allocation={0: 10, 10: 1}
 BestFit symbols={'a': 0, 'b': 10} holes=[(11, 9)] allocation={0: 10, 10: 1}
```

```
 WorstFit symbols={'a': 0, 'b': 10} holes=[(11, 9)] allocation={0: 10, 10: 1}
c=malloc(4):
 FirstFit symbols={'a': 0, 'b': 10, 'c': 11} holes=[(15, 5)] allocation={0:
10, 10: 1, 11: 4}
 BestFit symbols={'a': 0, 'b': 10, 'c': 11} holes=[(15, 5)] allocation={0:
10, 10: 1, 11: 4}
 WorstFit symbols={'a': 0, 'b': 10, 'c': 11} holes=[(15, 5)] allocation={0:
10, 10: 1, 11: 4}
free(c)
 FirstFit symbols={'a': 0, 'b': 10} holes=[(11, 9)] allocation={0: 10, 10: 1}
 BestFit symbols={'a': 0, 'b': 10} holes=[(11, 9)] allocation={0: 10, 10: 1}
 WorstFit symbols={'a': 0, 'b': 10} holes=[(11, 9)] allocation={0: 10, 10: 1}
free(a)
 FirstFit symbols={'b': 10} holes=[(0, 10), (11, 9)] allocation={10: 1}
 BestFit symbols={'b': 10} holes=[(0, 10), (11, 9)] allocation={10: 1}
 WorstFit symbols={'b': 10} holes=[(0, 10), (11, 9)] allocation={10: 1}
d=malloc(9):
 FirstFit symbols={'b': 10, 'd': 0} holes=[(9, 1), (11, 9)] allocation={10:
1, 0: 9}
 BestFit symbols={'b': 10, 'd': 11} holes=[(0, 10)] allocation={10: 1, 11: 9}
 WorstFit symbols={'b': 10, 'd': 0} holes=[(9, 1), (11, 9)] allocation={10:
1, 0: 9}
e=malloc(10):
 FirstFit symbols={'b': 10, 'd': 0, 'e': None} holes=[(9, 1), (11, 9)]
allocation={10: 1, 0: 9}
 BestFit symbols={'b': 10, 'd': 11, 'e': 0} holes=[] allocation={10: 1, 11:
9, 0: 10}
 WorstFit symbols={'b': 10, 'd': 0, 'e': None} holes=[(9, 1), (11, 9)]
allocation={10: 1, 0: 9}
------------------------
a=malloc(3):
 FirstFit symbols={'a': 0} holes=[(3, 17)] allocation={0: 3}
 BestFit symbols={'a': 0} holes=[(3, 17)] allocation={0: 3}
 WorstFit symbols={'a': 0} holes=[(3, 17)] allocation={0: 3}
b=malloc(6):
 FirstFit symbols={'a': 0, 'b': 3} holes=[(9, 11)] allocation={0: 3, 3: 6}
 BestFit symbols={'a': 0, 'b': 3} holes=[(9, 11)] allocation={0: 3, 3: 6}
 WorstFit symbols={'a': 0, 'b': 3} holes=[(9, 11)] allocation={0: 3, 3: 6}
c=malloc(2):
 FirstFit symbols={'a': 0, 'b': 3, 'c': 9} holes=[(11, 9)] allocation={0: 3,
3: 6, 9: 2}
 BestFit symbols={'a': 0, 'b': 3, 'c': 9} holes=[(11, 9)] allocation={0: 3,
3: 6, 9: 2}
 WorstFit symbols={'a': 0, 'b': 3, 'c': 9} holes=[(11, 9)] allocation={0: 3,
3: 6, 9: 2}
d=malloc(5):
 FirstFit symbols={'a': 0, 'b': 3, 'c': 9, 'd': 11} holes=[(16, 4)]
allocation={0: 3, 3: 6, 9: 2, 11: 5}
 BestFit symbols={'a': 0, 'b': 3, 'c': 9, 'd': 11} holes=[(16, 4)]
allocation={0: 3, 3: 6, 9: 2, 11: 5}
```

```
 WorstFit symbols={'a': 0, 'b': 3, 'c': 9, 'd': 11} holes=[(16, 4)]
allocation={0: 3, 3: 6, 9: 2, 11: 5}
free(a)
 FirstFit symbols={'b': 3, 'c': 9, 'd': 11} holes=[(0, 3), (16, 4)]
allocation={3: 6, 9: 2, 11: 5}
 BestFit symbols={'b': 3, 'c': 9, 'd': 11} holes=[(0, 3), (16, 4)]
allocation={3: 6, 9: 2, 11: 5}
 WorstFit symbols={'b': 3, 'c': 9, 'd': 11} holes=[(0, 3), (16, 4)]
allocation={3: 6, 9: 2, 11: 5}
free(c)
 FirstFit symbols={'b': 3, 'd': 11} holes=[(0, 3), (9, 2), (16, 4)]
allocation={3: 6, 11: 5}
 BestFit symbols={'b': 3, 'd': 11} holes=[(0, 3), (9, 2), (16, 4)]
allocation={3: 6, 11: 5}
 WorstFit symbols={'b': 3, 'd': 11} holes=[(0, 3), (9, 2), (16, 4)]
allocation={3: 6, 11: 5}
e=malloc(2):
 FirstFit symbols={'b': 3, 'd': 11, 'e': 0} holes=[(2, 1), (9, 2), (16, 4)]
allocation={3: 6, 11: 5, 0: 2}
 BestFit symbols={'b': 3, 'd': 11, 'e': 9} holes=[(0, 3), (16, 4)]
allocation={3: 6, 11: 5, 9: 2}
 WorstFit symbols={'b': 3, 'd': 11, 'e': 16} holes=[(0, 3), (9, 2), (18, 2)]
allocation={3: 6, 11: 5, 16: 2}
free(b)
 FirstFit symbols={'d': 11, 'e': 0} holes=[(2, 9), (16, 4)] allocation={11:
5, 0: 2}
 BestFit symbols={'d': 11, 'e': 9} holes=[(0, 9), (16, 4)] allocation={11: 5,
9: 2}
 WorstFit symbols={'d': 11, 'e': 16} holes=[(0, 11), (18, 2)] allocation={11:
5, 16: 2}
f=malloc(11):
 FirstFit symbols={'d': 11, 'e': 0, 'f': None} holes=[(2, 9), (16, 4)]
allocation={11: 5, 0: 2}
 BestFit symbols={'d': 11, 'e': 9, 'f': None} holes=[(0, 9), (16, 4)]
allocation={11: 5, 9: 2}
 WorstFit symbols={'d': 11, 'e': 16, 'f': 0} holes=[(18, 2)] allocation={11:
5, 16: 2, 0: 11}
```

2.3 [up to 20 points] Bonus: test case showing advantage of First-Fit

In the provided test cases, we included one example that shows Best-Fit succeeding while the other two fail, and another example showing Worst-Fit succeeding. For this bonus problem, you are to generate a test case that shows First-Fit succeeding and Best-Fit and Worst-Fit fail. You must provide the test case in the same format as in the template. You must provide an explanation in the PDF file and a typescript. If multiple students submit identical test cases, then the bonus points will be divided evenly among them.