CS 342302 Operating Systems

Fall Semester 2021

Prof. Pai H. Chou

Weekly Review 10

(Scope: Ch. 9)

## 1. Definitions and Short Answers

1. In the base-and-limit registers scheme such as pictured below,


	1. What are the **address spaces** of the three processes?
	2. Who loads the values of the base and limit registers?
	3. What happens when a process attempts to address memory location outside its address space?
2. Does a compiler determine the absolute address of the memory references? If not, what kind of address does it generate?
3. Which tool (compiler, linker, loader) determines the actual address to access memory for **compile-time binding**? What must be known in order to perform binding?
4. For **load-time binding**, what does the loader actually do? And what kind of code must the compiler-linker generate?
5. If **execution-time binding** is used, what assumption can be made about the base address of the process? Who calculates the base address + offset to form the actual memory address, and at what time?
6. What is the meaning of a **logical address**? What is the meaning of a **physical address**? How is logical address different or same as the physical address for
	1. compile-time binding
	2. load-time binding
	3. execution-time binding
7. What does **dynamic loading** mean? Why is it a good idea?
8. What is a problem with the combination of **dynamic loading with static linking**?
9. What is the mechanism for **dynamic linking**? How does it address the problem of dynamic loading with static linking?
10. What are two partitioning schemes of doing contiguous allocation?
11. What is a limitation imposed by **fixed partition** scheme of contiguous allocation?
12. What is a problem caused by **variable partition** scheme of contiguous allocation?
13. What are the three common schemes for variable-partition contiguous memory allocation? Which one is generally faster? Which ones may need to search the entire list of holes?
14. What is the rationale for using **worst-fit**?
15. What is **external fragmentation**?
16. What is **internal fragmentation**?
17. In the following two scenarios, identify what kind of fragmentation it is. Gray color means allocated, white color means free.
	1. 
	2. 
18. **Compaction** may be a way to reduce fragmentation, but it is not always possible. What kind of **address binding** (compile time, load time, execution time) is required for compaction to work? Even if it is supported, under what situation may it still not work correctly?
19. Paging is a way of organizing noncontiguous allocation. What does **noncontiguous** mean in this case?
20. Does paging use variable partition or fixed partition?
21. Does paging have internal fragmentation? external fragmentation?
22. Why does paging tend to have less external fragmentation than contiguous allocation?
23. What is a **page** and what is a **frame**?
24. To support paging, how should a logical address be divided so that it can be mapped to a physical address? Which part of the address is mapped and which part is the same?
25. What are the advantages and disadvantages of a
	1. smaller page size
	2. larger page size

? Express in terms of page table size (i.e., number of entries) and fragmentation (say which kind fragmentation).

1. What does a **page table** map from and to? What does a **frame table** map from and to? Do you need **one per process** or **one for the entire system**?
2. What are two **registers** that identify a page table?
3. What does TLB stand for? Is it hardware or software, and what doe it do?
4. What happens on a TLB miss? What happens if all TLB entries are occupied?
5. What is ASID in a TLB entry? Is it mandatory? What are its benefits?
6. What is the purpose of bits for indicating **access rights** of a page?
7. Why would some pages be marked **invalid** in a page table entry?
8. Why is the reason for using a PTLR (page table length register)? Isn't the size of a page table fixed?
9. Can **shared memory** between processes be supported? Do different process need to use the same virtual address? Do the virtual addresses of the different processes map to the same physical address?
10. How does a 2-level **hierarchical page table** scheme divide the logical address into different fields, and what are the steps in looking up the frame number?
11. How does a **hashed page table** store its entries?
12. Is a **clustered page table** a form of a hashed page table? How is it more economical?

## 2. 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](https://drive.google.com/file/d/1CF86oT4I6TDim0eqqaIiEUjjZXrFB2CT/view?usp=sharing) file. Rename it memalloc.py

### 2.1 malloc(size\_t size)

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 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 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.