CS 342302 Operating Systems

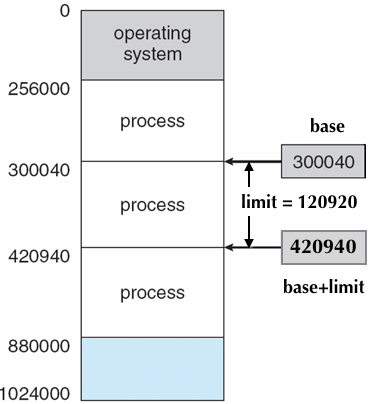
Fall Semester 2021

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?

A: Process 1: base: 256000 base + limit = 300039 inclusive

Process 2: base: 300040 base + limit = 420939 inclusive

Process 3: base: 420940 base + limit = 879999 inclusive

* 1. Who loads the values of the base and limit registers?

A: The OS loads the values of the base and limit registers which uses a special privileged instruction. This scheme allows the OS to change the value of the registers but prevents user programs from changing the registers contents.

* 1. What happens when a process attempts to address memory location outside its address space?

A: The OS must ensure that every memory access from a process is made withing its base and limit registers. If an attempt is made to access a memory location outside its address space, then the hardware would catch it and generate a trap to the OS for the OS to handle it. Usually in the form of a segmentation fault.

1. Does a compiler determine the absolute address of the memory references? If not, what kind of address does it generate?

A: No, a compiler does not generate absolute addresses. A compiler typically binds symbolic addresses to relocatable addresses.

1. Which tool (compiler, linker, loader) determines the actual address to access memory for **compile-time binding**? What must be known to perform binding?

A: The linker or loader determines the actual address to access memory for. compile time binding. Memory location must be known a priori or before such that the linker can generate absolute code.

1. For **load-time binding**, what does the loader do? And what kind of code must the compiler-linker generate?

A: The compiler or linker will generate relocatable code if the memory location is not known at compile time. The loader then fills the address and adds the offset. Example: During loading .BS = 2000 and offset = 0x18. Then, program is loaded at address 2018.

1. 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?

A: The hardware oversees the mapping of virtual addresses to physical addresses and thus it is assumed only the hardware knows the base address for each process. The user program then only deals with logical addresses.

1. What is the meaning of a **logical address**?

A: A logical address is generated by the CPU (also referred as a virtual address). The logical address space is the set of all logical addresses generated by a program.

What is the meaning of a **physical address**?

A: A physical address is the address seen by the memory unit. The physical. address space is the set of all physical addresses generated by a program.

How is a logical address different or same as the physical address for?

* 1. compile-time binding – Same as physical
  2. load-time binding – Same as physical
  3. execution-time binding – Remapped (Physical vs Logical addresses)

1. What does **dynamic loading** mean? Why is it a good idea?

A: Dynamic loading means a routine is not loaded until is called. It is a good idea because it ensures better memory-space utilization and does not require the entire program image to be loaded into the OS for it to be loaded in memory.

1. What is a problem with the combination of **dynamic loading with static linking**?

A: Because each process has its own logical address space, dynamic loading with static linking may load multiple copies of the same routine in the physical address space.

1. What is the mechanism for **dynamic linking**? How does it address the problem of dynamic loading with static linking?

A: Dynamic linking uses a stub mechanism. The stub is used to locate the appropriate memory-resident library routine. The stub then replaces itself. with the address of the routine and executes it. Thus, it is not necessary to load the routine multiple times in physical memory.

1. What are two partitioning schemes of doing contiguous (physical memory) allocation?

A: Fixed and variable partition are the schemes of doing contiguous allocation.

1. What is a limitation imposed by **fixed partition** scheme of contiguous allocation?

A: Advantage: Each process is loaded into one partition of a fixed size.

Problem: The main limitation is that your degree of multiprogramming or the number of processes that can be effectively scheduled is bounded by the # of partitions.

1. What is a problem caused by **variable partition** scheme of contiguous allocation?

A: Advantage: Each process gets its own partition size. Its more efficient. since partitions are sized to a given process’s needs.

Problem: Holes or blocks of contiguous free memory of different sizes are scattered in memory as processes spawn and terminate.

1. 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?

A: First-fit: Allocate the first hole that is big enough. Generally faster than other schemes. Better storage utilization.

Best fit: Allocates the smallest hole that is big enough. It produces the smallest leftover whole.

Worst fit: Allocate the largest hole. Produces the largest leftover hole.

\* Best and worst fit must search the entire list unless they are ordered by size.

1. What is the rationale for using **worst-fit**?

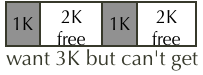
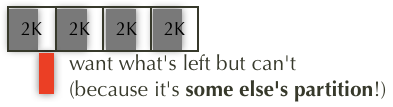
A: If fragmentation happens we don’t end with many smaller holes (best fit) but fewer bigger holes.

1. What is **external fragmentation**?

A: In external fragmentation, total memory space exists to satisfy a request but not contiguous. Occurs in variable-sized allocation.

1. What is **internal fragmentation**?

A: In internal fragmentation, free memory is internal to each partition, but it is too small to be used. In addition, in fixed partition once you give a partition to a process, only that process is allowed to use the memory from that partition. Degree of multiprogramming is limited to # of partitions.

1. In the following two scenarios, identify what kind of fragmentation it is. Gray color means allocated; white color means free.
   1. External fragmentation   
      
   2. Internal fragmentation.  
      
2. **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?

A: Execution time binding is required for compaction to work. There are several situations where compaction may not work correctly:

1) Relocation is not dynamic

2) I/O buffer may be in use – I/O is happening, and I/O space does not see the relocation of the process.

3) Backing store may have same fragmentation problems.

1. Paging is a way of organizing noncontiguous allocation. What does **noncontiguous** mean in this case?

A: Noncontiguous (physical) memory allocation means that a processes logical space may be allocated to physical memory wherever such memory is available. This is the strategy used by paging.

1. Does paging use variable partition or fixed partition?

A: Fixed partitions.

1. Does paging have internal fragmentation? external fragmentation?

A: External fragmentation is avoided. Could still have internal fragmentation.

1. Why does paging tend to have less external fragmentation than contiguous allocation?

A: Because processes are no longer required to be allocated in contiguous chunks of physical memory. They can be allocated wherever memory is available and thus avoiding the creation of holes.

1. What is a **page** and what is a **frame**?

A: A ***page*** is a logical memory block while a ***frame*** is a physical memory block. Both should be of the same size and size should be of the power of 2. 512 bytes to 16MB.

1. 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?

A: A logical address mut be divided in two parts: A page number *p* and a page offset *d*. The page number is mapped to a page table which contains the base address of each frame while the page offset stays the same and is. combined with the base address to define the physical memory address.

1. What are the advantages and disadvantages of a
   1. smaller page size

A: Advantages: Internal fragmentation is minimized.

Disadvantages: Needs more page table entries to track in a page table.

* 1. larger page size

A: Disadvantages: Internal fragmentation increases.

Advantages: Fewer page table entries to track in a page table.

? 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**?

A: Page table maps entire logical memory space into frames. One page table is needed per process. On the other hand, a frame table determines which frames are available and maps frame to the tuple (page, process). One frame table is required for the entire system.

1. What are two **registers** that identify a page table?

A: PTBR (Page-table base register)– Pointer to page table in memory.

PTLR (Page-table length register) – Size of page table.

1. What does TLB stand for? Is it hardware or software, and what does it do?

A: TLB – Translation look-aside buffer (A cache) for fast lookup of frame number. It is provided by hardware support.

1. What happens on a TLB miss? What happens if all TLB entries are occupied?

A: OS loads page-table entry into the TLB for faster access next time. If the TLB is full, some entries must be replaced. Replacement policies must be considered.

1. What is ASID in a TLB entry? Is it mandatory? What are its benefits?

A: An ASID is an address space identifier. It uniquely identifies each process. It ensures protection for processes data in physical memory by ensuring ASID matches current process.

1. What is the purpose of bits for indicating **access rights** of a page?

A: They enforce access rights. Indicate if read-only, read-write or execute-only. Any violations? – Trap to Kernel

1. Why would some pages be marked **invalid** in a page table entry?

A: Invalid bit -> Page is NOT part of the processes logical address space.

1. Why is the reason for using a PTLR (page table length register)? Isn't the size of a page table fixed?

A: It doesn’t have to be of fixed size since many entries may be not allocated (sparse table). It is useful because to keep track of the page table length because it is likely to be smaller than its worst-case size. Thus, is saves memory when most of the page table entries are unused.

1. Can **shared memory** between processes be supported? Do different processes need to use the same virtual address? Do the virtual addresses of the different processes map to the same physical address?

A: Yes, shared memory between processes can be supported. No, they may have different virtual address spaces. The same physical copy in the physical address is used by all processes, so every virtual address maps to the same physical address.

1. 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?

A:

1. How does a **hashed page table** store its entries?

A:

1. Is a **clustered page table** a form of a hashed page table? How is it more economical?

A: Yes, it is a form of a hashed page table,

## 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.