# CS-323: OS Exercise Week 5 (Fall 2023)

October 18, 2023

Last name, first name: ______________________________

## 1 Fragmentation

Fragmentation can refers to two different notions: internal and external fragmentation.

(a) What is internal fragmentation? Why is it bad?

Internal fragmentation occurs when we allocate resources with not enough granularity. For example, if we store a boolean value in a `int8_t`, we are wasting 7 bits.

In the case of memory allocation, it results from the fact that allocators generally allocate memory in chunks. If we allocate in 1024-byte chunks, a 4-byte request wastes $1,020$ bytes. Larger chunks are easier to track for the allocator (think $1,000$ chunks of 1MB each are easier to track than $1,000,000$ chunks of 1KB each). However, they also waste more space as there is more internal fragmentation. Allocating in terms of bits is the best case to absolutely remove internal fragmentation, however is very inefficient. Its easier for a 64-bit machine to load one word (64 bits from memory) than 1 bit!

This also applies in the case of more generic allocators such as `malloc`, as they may still have a minimum possible chunk size due to needing to store bookkeeping information in free chunks (just as is the case in Lab 2).

(b) What is external fragmentation? Why is it bad?

External fragmentation arises when resources need to be allocated contiguously. It appears when small resource segments are free but interspersed with allocated regions in between, meaning there sufficient free resources in total but only in small usable chunks. A memory allocation may fail because not enough contiguous memory is available, even though there is enough free memory in total. For example, a memory of 5 blocks (xxoxx) with the middle block allocated (o) will not be able to allocate a segment of 3 blocks even if in total 4 blocks (x) are free.

(c) What can be done to counter internal and external fragmentation?

Little can be done to counter internal fragmentation. Allocating resources in a more fine-grained manner is beneficial against fragmentation but comes at a tradeoff with other properties.

External fragmentation can be tackled via compaction. Compaction consists of moving allocated resources to maximize contiguous free space. For the example in the previous answer, moving the allocated block to the beginning frees up 4 contiguous blocks after it. However, this may not always be possible, depending on how the allocation system is designed. Moving memory around in C/C++ would invalidate pointers, but garbage collected languages can usually afford to compact heap blocks by readjusting pointers during the collection phase.

(d) What are real-world examples of internal and external fragmentation?

Internal fragmentation occurs when you leave a small Smart car on a standard parking spot. The parking space was allocated with bigger cars in mind and even if you could hypothetically park two Smarts in the parking space, it is usually not allowed. The tradeoff to counter this type of

Name: ____________________________

fragmentation is easy to see: designing parking spaces with different sizes would reduce fragmentation but make parking cars a nightmare.

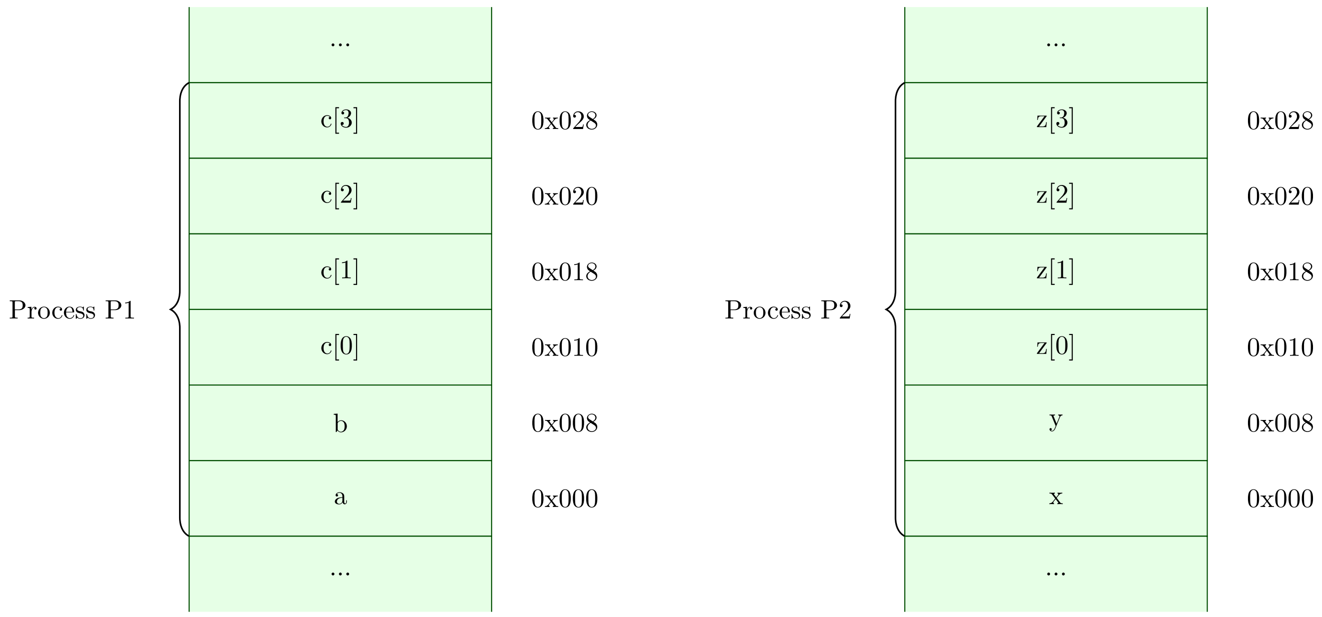
In our parking illustration, external fragmentation occurs when you have trees in the middle of the parking. When drawing the parking spots, we might be left with space for only half a parking sport before each tree. If we were able to move the trees, we might be able to add a few parking spots, but this is a process that is best avoided.

(e) Why is the stack not at risk of fragmentation?

The stack is always freed in reverse order of allocation. The only free space is therefore always on top of the stack and extends up to the heap. External fragmentation is therefore not possible. There can be internal fragmentation within the stack due to structure alignment (how many bytes does `struct { char x; int y; };` occupy?) but it is generally small and also the case for the heap.

## 2 Base and Bounds

Assuming the following process virtual memory structure with the given base registers:
P1 base register: 0x100, P2 base register: 0x200



(a) What is the physical address of `a`?

`0x100`

(b) What is the physical address of `c[0]`?

`0x110`

(c) What is the physical address of `y`?

`0x208`

(d) What is the virtual address of `c[31]`? What is its physical address? Remember that this is perfectly valid C.

This corresponds to the address of `c[0]` $+ 31 \times 8$ bytes $=$ `0x108` virtual, `0x208` physical

(e) What is the result of accessing the address `0x010` from P1?

Acessing physical memory `0x110` a.k.a. `c[0]`

(f) What is the virtual equivalent of physical address `0x108`?

It translates to virtual address `0x008` of P1, which is b

Name: ____________________________

(g) What is the result of accessing the address `0x010` from P2?

Accessing physical memory `0x210` a.k.a. `z[0]`

(h) Assume now that our processes have a bound with value `0x50`. Go through question (a) to (g) again. What changes?

The only difference would be (d) resulting in a segmentation fault as it is over the bounds of P1's address space.

## 3 Base and Bounds - 2

Based on the following code:

```
    char *a = 0x108;
    putchar(*0x100); /* putchar() prints a single character */
    putchar(*(a + 0x10));
    putchar(*0x128)
```

Assuming two processes:

- P1, with base and bound registers of `0x100` and `0x130`
- P2, with base and bound registers of `0x250` and `0x120`

(a) For each line, describe which physical and virtual memory address will be accessed if P1 executes the code above.

Line 1: No memory access strictly necessary, `a` can be stored in a register, or at an unknown stack address otherwise
Line 2: We access P1's virtual memory at `0x100` and physical memory at `0x200`
Line 3: We access P1's virtual memory at `0x118` and physical memory at `0x218`
Line 4: We access P1's virtual memory at `0x128` and physical memory at `0x228`

(b) For each line, describe which physical and virtual memory address will be accessed if P2 executes the code above.

Line 1: Same as (a) Line 1
Line 2: We access P2's virtual memory at `0x100` and physical memory at `0x350`
Line 3: We access P2's virtual memory at `0x118` and physical memory at `0x368`
Line 4: We access P2's virtual memory at `0x128` which is out of bounds, and the MMU issues a segmentation fault as we are trying to access memory outside our segment
Also note how both processes see the same virtual addresses, but infact work on different data at different physical addresses. This element of memory virtualization allows programmers to work without caring about other processes sharing physical memory. Each program sees its own virtual memory space without the presence of any other processes, as if it had the entire memory to itself.

## 4 Allocation Strategies

Assume we have memory of size `0xA0` with an object of size `0x10` at address `0x50`. This is illustrated in Figure 1. For each allocation strategy, what would be the resulting memory layout?

(a) first-fit
(b) worst-fit
(c) best-fit

if we allocate all of the following objects in the given order?

Name: ______________________________

1. An object of size 0x20
2. An object of size 0x5
3. An object of size 0x40

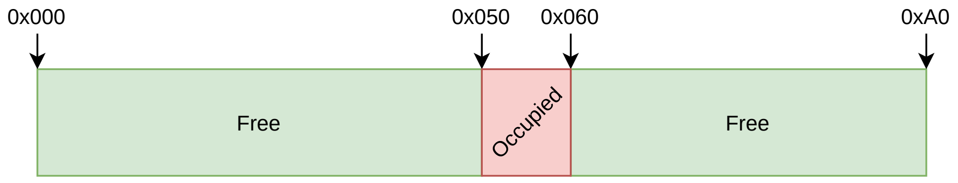


Figure 1: Initial memory allocation

(a) first-fit

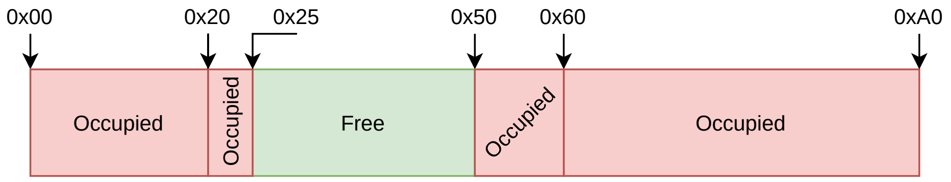


Figure 2: First-fit memory allocation

(b) worst-fit

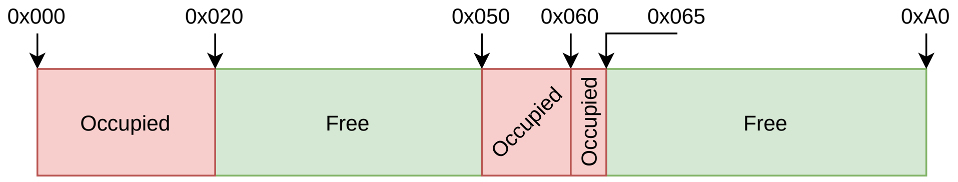


Figure 3: Worst-fit memory allocation, note that it is not possible to allocate the last object.

(c) best-fit

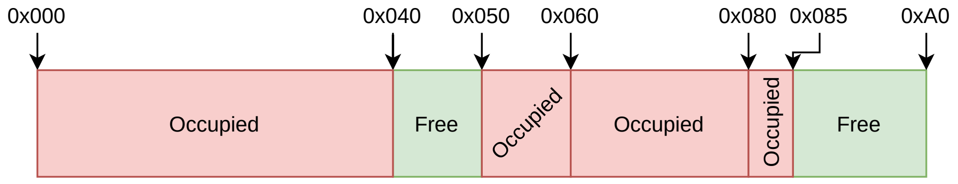


Figure 4: Best-fit memory allocation

Name: ____________________________

## 5 Allocation Strategies - 2

Assuming we are using a list of free blocks as heap management.

(a) What is the advantage and drawback of a first-fit allocation strategy?

The first-fit approach is fast as we only need to iterate through the list until we find a free block big enough to hold our allocation and do not need to check the whole list. It is however prone to fragmentation.

(b) What is the advantage of a best-fit allocation strategy?

The goal of this strategy is to try to minimize the memory fragmentation by allocating a memory region with the size closest to the requested size. It comes however at the cost of the whole list needing to be traversed to find the best region.