CS2106 Introduction to Operating Systems

Semester 1 2021/2022

Tutorial 7

Contiguous Memory Allocation

1. (Fixed partitioning) Consider six memory partitions of size 200 KB, 400 KB, 600 KB, 500 KB, 300 KB and 250 KB. These partitions need to be allocated to four processes of sizes 357 KB, 210 KB, 468 KB and 491 KB in that order.

   Perform the allocation of processes using:

   a. First Fit Algorithm
   b. Best Fit Algorithm
   c. Worst Fit Algorithm

   Which algorithm makes the most efficient use of memory in this particular case? Which algorithm has the best average runtime? Which algorithm is the best to use overall?

   ANS:

   According to question, the main memory has been divided into fixed size partitions as:

200KB	400KB	600KB	500KB	300KB	250KB

   Let us say the given processes are:

   - Process P1 = 357 KB
   - Process P2 = 210 KB
   - Process P3 = 468 KB
   - Process P4 = 491 KB

   Allocation Using First Fit Algorithm-

200KB	400KB(P1)	600KB(P2)	500KB(P3)	300KB	250KB

   Process P4 can not be allocated the memory. This is because no partition of size greater than or equal to the size of process P4 is available.

   Allocation Using Best Fit Algorithm:

200KB	400KB(P1)	600KB(P4)	500KB(P3)	300KB	250KB(P2)

   Allocation Using Worst Fit Algorithm:

200KB	400KB	600KB(P1)	500KB(P2)	300KB	250KB

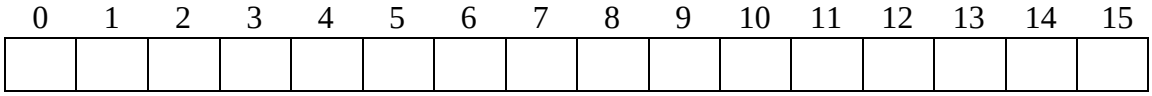
Process P3 and Process P4 can not be allocated the memory. This is because no partition of size greater than or equal to the size of process P3 and process P4 is available.

In this particular example, Best Fit Algorithm turns out to be the best in terms of memory efficiency. However, that does not need to hold true in general. Regarding the runtime, Best Fit and Worst Fit must go through the entire list (O(N)) to find the best (worst) candidate. First Fit has the best runtime as the search stops as soon as the first free hole that accommodates the request is available.

2. (Dynamic partitioning) In the lecture, we used linked list to store partition information under the dynamic allocation scheme. One common alternative is to use bitmap (array of bits) instead.
Basic idea: A single bit represents the smallest allocatable memory space, 0 = `free`, 1 = `occupied`. Use a collection of bits to represent the allocation status of the whole memory space.
As a tiny example, suppose the memory size is 16KB and the smallest allocatable unit is 1KB. We need 16 bits (2 bytes) to keep track of the allocation status:

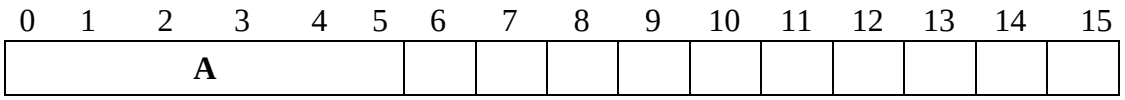
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Below is the corresponding physical memory layout at this point. Note that the allocatable unit != partition. At this point the whole memory is a single free partition. The boxes are drawn to illustrate the allocation unit clearly.



After placing process **A** (6KB), the bitmap and the corresponding physical memory layout become:

1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0



Give brief pseudo code to:

a. Allocate X KB using the bitmap using **first-fit**.
b. Deallocate (free) X KB with start location Y.
c. Merge adjacent free space.

ANS:

The key observation is that the bits cannot be accessed directly as if it is an array. Bitwise operations (logical AND / OR and shifting) are needed in order to retrieve a bit. Bit manipulation is the main overhead of using bitmap.
For ease of discussion, the pseudo codes below are presented as if array indexing is provided for bitmap.

a. 1. Start ← 0.
   2. Start ← Location of the first '0' in the bitmap after **Start**
      3. If there are X consecutive zeroes, mark [Start…Start+X-1] as 1, success!
      4. Else Start ← The first '1' in the bitmap after Start, repeat step 2.
      5. Stop when bitmap is exhausted.

b. Straightforward, mark [Y… Y+X-1] as 0. Done!

c. No need to do anything ☺. This is the benefit of using bitmap as adjacent free spaces are "merged" automatically.

3. (Buddy System) Consider a buddy memory allocation system that has byte addressing. Memory space starts at `00000` and ends at `11111` = 2^5 = 32 bytes. Smallest allocatable block size is 1 byte. Assume that all possible merges are already done on buddy blocks.

a. Find the largest possible size of each of the free blocks.
b. Find the **minimum** sum of all allocated memory in the system.

The following are memory addresses (in binary) of free blocks:

```
[a] = 00000
[b] = 00110
[c] = 01000
[d] = 01100
[e] = 01110
[f] = 10000
[g] = 11100
```

Hint: Represent the memory as an array of 32 bytes. Split the blocks such that the free blocks will be the above addresses.

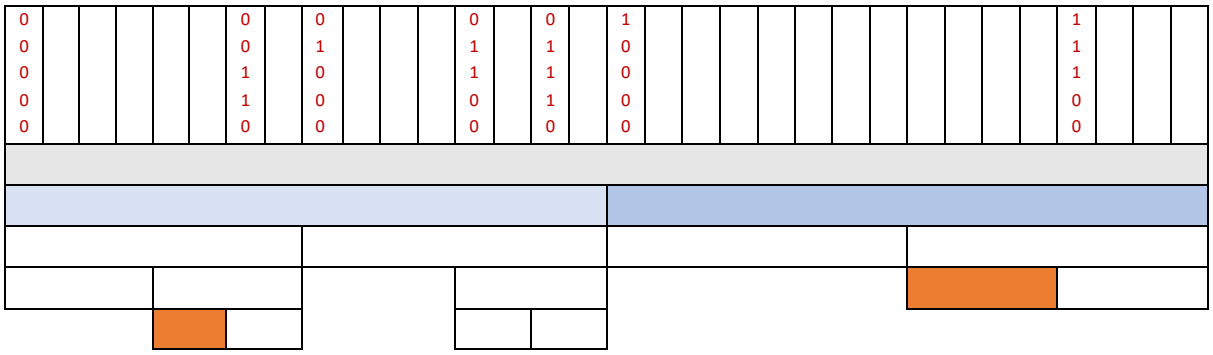
ANS:

Allocated memory regions are 00100 (size = 2 bytes), 01101 (size = 1 byte), 11000 (size = 4 bytes)
Total memory allocated = 7 bytes.
Any other answer would not be the **minimum** possible memory allocation.

Steps to derive the answer

- By maximising the free block sizes, we can find the minimum sum of allocated memory.
- Represent the free addresses on an array as shown below.
- Consider how large the first free block [a] could be. This would depend on the address of the next free block [b].
- To get free block [b], notice that the original block had to be split 4 times.
- As we assumed that free buddy blocks were already merged, the buddy block on the left of free block [b] must be occupied.
- We can conclude from the above discussion that free block [a] can have a maximum size of 4 bytes, free block [b] can have a maximum size of 2 bytes and there is a occupied block in between them of size 2 bytes.

Repeat the same argument to find the remaining occupied blocks



Below is a tree-based representation of the allocation.

- 00000 (32 bytes)
  - 00000 (16 bytes)
    - 00000 (8 bytes)
      - 00000 (4 bytes) (free)
      - 00100 (4 bytes)
        - **00100 (2 bytes) (allocated)**
        - 00110 (2 bytes) (free)
    - 01000 (8 bytes)
      - 01000 (4 bytes) (free)
      - 01100 (4 bytes)
        - 01100 (2 bytes)
          - 01100 (1 byte) (free)
          - **01101 (1 byte) (allocated)**
        - 01110 (2 bytes) (free)
  - 10000 (16 bytes)
    - 10000 (8 bytes) (free)
    - 11000 (8 bytes)
      - **11000 (4 bytes) (allocated)**
      - 11100 (4 bytes) (free)

4. (Overhead of Bookkeeping Information) Regardless of the partitioning schemes, the kernel needs to maintain the partition information in some way (e.g. linked lists, arrays bitmaps etc). These kernel data, which is the overhead of the partitioning scheme, can consume considerable memory space.

   Given an initially free memory space of 16MB (2^{24} Bytes), briefly calculate the overhead for each of the scheme below. You should try to find a representation that reduces the overhead if possible.

   For simplicity, you can assume the following size during calculation:

   - Starting address, Size of partition or Pointer = 4 bytes each
   - Status of partition (occupied or not) = 1 byte

   a. Fixed-Size Partition: Each partition is 4KB size (2^{12}). What is minimum and maximum overhead?

   b. Dynamic-Size Partition (Linked List): The smallest request size is 1KB (2^{10}), the largest request size is 4KB. What is the minimum and maximum overhead using linked list? Allocations happen in multiples of 1 KB.

   c. Dynamic-Size Partition (Bitmap, see Q1): The smallest request size is 1KB (2^{10}), the largest request size is 4KB. What is the minimum and maximum overhead using bitmap?

ANS:

a. As we know the total number of partitions (16MB/4KB = 4K (2^{12}) partitions) in this scheme. The simplest representation is to have an array of 4K entries, each entry represent the status of a partition (occupied or free).
Total overhead = 4K entries * 1byte each = 4096 bytes

As the partition number is fixed, maximum overhead = minimum overhead.

b. A common linked list node structure contains
{ Start Address, Partition Size, Status, Next Node Pointer}
Size of one node = 3*4 + 1 = 13 bytes

Minimum overhead:
When the whole partition is free, only one node is needed, overhead = 13 bytes

Maximum overhead:
If every request is of the smallest size, we have the maximum number of partitions:
(16MB / 1KB = 16K (2^{14}) Partitions).

Overhead = 16K partitions * 13 bytes per partition information node
= 2^{14} * 13 = 212,992 bytes

Alternative representation:
Another possibility is to store **two separate linked lists** for free and occupied partitions, respectively. In that case, the **status** field can be removed, reducing the node size to 12 bytes instead of 13 bytes, giving the min and max overhead as 12bytes, and 2^{14} * 12 = 196,608 bytes

Another possibility is to drop the size information of partition in the node of the linked list and only keep the status Boolean and starting address. One could tell the size of the partition by seeing the next node in the linked list or referring to the total space allocated to the process if the next node is null. Minimum is still 4 + 9 = 13 bytes. We need to keep track of the total size the process is allocated = 4 bytes. Maximum space taken will be 2^{14} * 9 + 4 = 147460 bytes.

Common misconception:
Note that we do not need to have additional space to store pointers to the linked lists' heads. We can store the linked lists' heads at a fixed address. If linked list is empty, that address will be null. Else, that address will have the first node. For the two separate linked list representation, we can have the heads at address x and x + sizeof (node) respectively.

c. (Assuming we can store the status with 1 bit): Each bit in the bitmap represents the smallest allocatable unit, 1KB in this case. The size of the bitmap is not affected by the number of partitions, hence minimum == maximum overhead.

Overhead = 2^{14} bits / 8 = 2^{11} = 2,048 bytes.

Questions for own exploration

5. (Buddy System) Given a 1024KB memory with smallest allocatable partition of 1KB, use buddy system to handle the following memory requests.
    a. Allocate: Process A (240 KB)
    b. Allocate: Process B (60 KB)
    c. Allocate: Process C (100 KB)
    d. Allocate: Process D (128 KB)
    e. Free: Process A
    f. Free: Process C
    g. Free: Process B

Show the physical memory layout as a way to track the results, i.e. below is the physical memory layout after request (a).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A [256KB (240)]				Free[256KB]				Free [512KB]							

Note: For a more condensed representation, the numbers on the first row represent the starting address as multiples of 64KB. e.g. Process A starts at **0** x 64KB = 0, the free 256KB partition starts at **4** x 64KB = 256KB, the last partition is at **8** x 64KB = 512KB.

You should try to maintain the actual array of linked lists (as shown in lecture) throughout to gain a better understanding of the algorithm. For consistency, we assume the partitions of the same size are arranged in ascending order by their starting address.

ANS:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A [256KB (240)]				Free[256KB]				Free[512KB]							
A				B	Free		Free[128]	Free[512KB]							
A				B	Free		C	Free[512KB]							
A				B	Free		C	D		Free		Free			
Free				B	Free		C	D		Free		Free			
Free				B	Free		Free	D		Free		Free			
Free								D		Free		Free			

6. (First fit) Suppose we use the following allocation algorithm: for the first allocation traverses the list of free partitions from the beginning of the list until the first partition which can accommodate the request. The following allocations, however, do not will start from the beginning, but instead start from the partition where the previous allocation was performed. Compare this algorithm with First Fit in terms of runtime and efficiency of memory use.

ANSWER*:*

First Fit always starts the search from the beginning of the free list. Over time, the holes close to the beginning will become too small, so the algorithm will have to look further down the list, increasing the search time.

The described algorithm is known as *Next Fit* and avoids the above problem by changing the starting point of the search. This in turn leads to a more uniform distribution of hole sizes across the free list, and will therefore lead to **faster allocation**.