*1. Memory Management Simultation*

**Memory Managment Overview**

When processes are started and enter into a system, they must be allocated resources. One critical resource is its’ memory (i.e. RAM) which contains its data, instructors, and other meta data for a process to exist.

Initially, memory is considered one big partition (i.e. a list of one node). When a request for memory is initiated, the memory manager must look to the list of partitions to find one that can satisfy the request for the process. If a partition is large enough, then the requested amount of space is allocated, and the remaining space is added back to the list of partitions. If no space is available, then an error is given.

Space can be allocated among the available list of free partitions based on 3 policies (i.e. algorithms):

1. First Fit (represented by a unsorted list)
2. Best Fit (represented by a sorted list in ascending order)
3. Worst Fit (represented by a sorted list in descending order)

In order to allocate memory, your solution must find the first available partition big enough to satisfy the request. Any remaining memory from the chosen partition is called a fragment, and is placed back into the list according to the allocation policy above.

When processes complete their work, resources must be deallocated (i.e. given back to the system), so that these resources can be shared with other processes. A deallocated memory block is placed back into the list according to the allocation policy above.

Over time, the list of available partitions can be merged to create larger partitions. This can be done by merging physically adjacent partitions (i.e. called coalescing), or shuffling free partitions to one end of memory, and used memory the other end of memory (i.e. called compaction). Compaction, can be very expensive.

**Simulation**

You are to modify the simulation code to implement the three memory allocation policies (i.e. First Fit, Best fit, and Worst Fit). You are to also implement the coalescing method to merge physically adjacent memory blocks into one larger block. Feel free to run the reference code *./mmu\_ref* on the sample input file *input0.txt* and compare your output.

**Input**

Your Input File will contain a single integer representing an initial memory partition size, followed by one or more lines containing two integers:

1. two integers, representing a process id and its requested block size to allocate,
2. one negative interger, representing the process id to deallocate, followed by a 0.
3. another negative number -99999, followed by a 0. The negative number represents a call for the simulation to defragment the free list by coalescing physically adjacent blocks.

<INITIAL PARTITION SIZE>

<PID> <REQUEST BLOCKSIZE>

:

-99999 0

<PID> <REQUEST BLOCKSIZE>

-<PID> 0

-<PID> 0

:

**Sample Input**

100000  
2 50  
3 1000  
5 10000  
4 8900  
6 20000  
-99999 0  
-5 0  
7 500000  
8 1234  
9 4000  
-2 0  
-8 0  
-99999 0

**Process:**

1. Read the first line and create a linked list (i.e. FREE\_LIST) with the Partition (i.e. block) size given in the input file.
2. When reading a line containing , call the function *allocate\_memory()*.
3. When reading a line containing - 0, call the function *deallocate\_memory()*.
4. When reading a line containing -99999 0, call the function *coalese\_memory()*.

Fortunately, this has been done for you. If you would like a better understanding of how this was done, feel free to review the code.

**The Code and Data Structures**

Change to the MMU directory. You are to modify the file *mmu.c*. Use the Makefile to compile.

$ cd MMU

$ make mmu

Speciffically, you will implement the following functions:

**allocate\_memory**()

**deallocate\_memory**()

You must also modify the link list code in Lab 1. I have given you most of the functions, but there are some that need to be implemented. Specifically:

**list\_is\_in\_by\_pid**()

**list\_add\_ascending\_by\_blocksize**()

**list\_add\_ascending\_by\_address**()

**list\_coalese\_nodes**()

The struct node in Lab 1 has been modified to store nodes which contain details about a memory partition (i.e. block).

**typedef** **struct** **block** {

**int** pid; *// process id*

**int** start; *// start of the memory partition*

**int** end; *// end of the memory partition*

}**block\_t**;

**typedef** **struct** **node** {

**block\_t** \*blk; *// pointer to the block*

**struct** **node** \***next**; *// pointer to next node*

}**node\_t**;

Block size can be determined by the following formula *blocksize = end - start*. You will maintain two data structures (i.e. linked list): 1. Free Block List, and 2. Allocated Block List.

**First Fit**

**Implementation**

-ALLOCATE: To allocate a block of memory, search through the Free Block list, starting from the head, for the first block large enough to satisfy the request. If found, remove the block from the Free Block list, and insert it into the Allocated Block list using the function *list\_add\_ascending\_by\_address()*. This function will keep all allocated memory in order based on the physical address. Keep in mind that the block you select may exceed the requested size. Therefore, you will have fragmentation. Make sure to add the fragment back to the Free Block list based on the allocation policy (i.e. FIFO)

* Use the *list\_add\_to\_back()* to insert memory block in FIFO order for FIRST FIT.

If no space is found, print the message "Error: Not Enough Memory".

-DEALLOCATE: To deallocated a block assigned to a process. Remove the block from the Allocated Block list, and insert the block back into the Free Block List based on the allocation policy.

**Best Fit**

**Implementation**

-ALLOCATE: To allocate a block of memory, search through the Free Block list, starting from the head, for the first block large enough to satisfy the request. If found, remove the block from the Free Block list, and insert it into the Allocated Block listusing the function *list\_add\_ascending\_by\_address()*. This function will keep all allocated memory in order based on the physical address. If not found, Error: Not Enough Memory. Remember insert any fragments back into the Free Block List based on the allocation policy.

-DEALLOCATE: To deallocated a block assigned to a process. Remove the block from the Allocated Block list, and insert the block back into the Free Block List based on the allocation policy.

* Use the *list\_add\_ascending\_by\_blocksize()* to insert memory block in FIFO order for BEST FIT.

**Worst Fit**

**Implementation**

-ALLOCATE: To allocate a block of memory, search through the Free Block list, starting from the head, for the first block large enough to satisfy the request. If found, remove the block from the Free Block list, and insert it into the Allocated Block listusing the function *list\_add\_ascending\_by\_address()*. This function will keep all allocated memory in order based on the physical address. If not found, Error: Not Enough Memory. Remember insert any fragments back into the Free Block List based on the allocation policy.

-DEALLOCATE: To deallocated a block assigned to a process. Remove the block from the Allocated Block list, and insert the block back into the Free Block List in descending order.

* Use the *list\_add\_descending\_by\_blocksize()* to insert memory block in FIFO order for WORST FIT.

**Coalescing**

When a call to coalesce is read, the simulation must iterate through the Free Block list and identify adjacent nodes who are physically adjacent.

**Two Physically Adjacent Free Blocks**

**NodeA (start = 1000, end = 20000, pid = 8, Next = NodeB)**  
**NodeB (start = 20001, end = 40000, pid = 1, Next = NodeW)**

NodeA and NodeB are removed from the Free Block List. NodeA and NodeB are merged into one NEWNode, and NEWNode is placed back into the Free Blocl List.

**NEWNode (start = 1000, end = 40000, pid = 0, Next = ?)**

NOTE: It may be easier to sort the free\_list by addresses first, then iterate through the list and merge adjacent blocks.