CS-4348 Project 3

You will extend your old code from Project 2 so that each process has an assigned memory space. Processes are only allowed to perform read/write operations within their own memory space, and memory spaces are not allowed to overlap.

Remember, ***your program must run on the UTD Giant Server!*** Any programs which cannot compile or run on the Giant Server will suffer from an automatic -50 point penalty. Instructions for logging into the Giant Server are given on the eLearning page.

You will be given a set of functions you are required to implement for each computer component. You are allowed to define additional helper functions if you wish, and you are allowed to modify what arguments are passed into each function. However, you should do your best to stick with the general architecture that has been presented in this document.

disk.c

You will modify your disk.c functions that load instruction set programs into memory. Just like in Project 2, you will read a list of program file names to determine which programs to load. Each row in the list of programs will have two arguments.

The first argument is an integer that gives how large of a memory space will be allocated to the process, while the second argument gives the file name that contains the instructions that will be loaded. A process can request more memory than it needs to hold the instructions. Each argument will be separated by a space. An example list is given below:

```
64 program_add.txt
80 program_if.txt
40 program_add.txt
```

When a program wishes to be loaded, an allocation request will be made to the Simple Memory Manager (SMM). If the SMM rejects the request by returning a value of 0 from the allocate() function, then the program is not loaded. If the request is accepted and a value of 1 is returned, then the program is loaded at the base address located within the allocation table. The SMM, the allocate() function, and the allocation table will be described in the next section.

smm.c

You will create a new file called smm.c which will implement your Simple Memory Manager (SMM). The SMM will perform memory management for your operating system by serving as an interface between user programs and the memory array. The SMM will use contiguous memory allocation with dynamic partitioning, as is described in the lecture slides.

With dynamic partitioning, you will need to create a linked list to keep track of the holes in memory. Every time a new hole is created, this linked list will be modified to reflect the current state of the system. Each entry in the linked list must keep track of the beginning address and the size of the hole.

You must use a global variable to count the number of times a new hole is created. When the OS terminates, this number should be printed to the command line.

Whenever a new process is created, a request is made to the SMM to allocate memory for the process. This request will include the amount of memory that the process wishes to reserve as well as the PID of the process. The SMM will then use the First-Fit algorithm to select a hole. To implement First-Fit, simply iterate through all the holes in the linked list, and select the first hole that is found which is large enough to meet the process' request. If no hole is large enough, then the process will be rejected, and an error message will be printed in the console. When a rejection occurs, the OS should continue operating as if the rejected process never existed, for this would be a poor operating system if it crashed every time a user process has an error. Remember, **an error message should be printed** when the SMM rejects a process.

When you choose a hole to allocate the process in, you can select the process base address to be the base address of the hole, and then update the base address and size of the original hole to reflect the memory that has been lost. If the entire hole is lost, then it should be removed from the linked list.

The SMM will also maintain an int array to represent an allocation table. The array will be of size (256, 3), where each row of the allocation table stores the PID, the base address, and the size of the process. Whenever a process successfully has memory reserved, this information about the process is added to the allocation table. To determine where a process should be placed in the table, iterate through the table and use the first row found which has a size of 0.

When a process is terminated, the memory it occupied must be deallocated. To do so, you must update the holes linked list to reflect that the memory space formerly occupied by the process is now considered unused and available for new processes. A simple way to do this is to add a new hole to the linked list in the appropriate position (i.e. the new hole's base address is greater than the previous hole's base address but lower than the next hole's base address). Then, iterate through the linked list. If you find two holes that are adjacent to each other in the linked list, you can remove and replace them with a new hole that merges the two of their spaces together. Also, you must set the size of this process' row in the allocation table to 0 to signify the position is available.

Whenever the memory is about to be modified with the mem_read and mem_write functions, a request is made to the SMM to see if the operation is allowed. The SMM will allow the read/write operation if it is being made within the memory space of the process. To determine if the operation is occurring within the memory space of the process, search for the row of the allocation table which contains the PID of the process in question. Check if the address that the operation is requesting is greater than the base address of the process and is less than the sum of the base address plus the size of the process. If the operation address is within these boundaries, then the operation is allowed to continue. If not, then the operation should be rejected, an error message should be printed, and the process that ordered the illegal operation should be terminated. The OS should then continue running normally. Remember, **an error message should be printed** when the SMM rejects an operation.

With all this considered, the following functions should be defined:

Function	**Description**
int allocate(int pid, int size)	Finds a base address by using the find_hole() function. Creates an entry in the allocation table for a given PID and size with the calculated base address. If no hole could be found, then a 0 is returned to signal that the process should be rejected. If a hole is found, a 1 is returned.
void deallocate(int pid)	Deallocates the memory held by a process with the given PID. Adds a new hole with the same base address and size as the process, and sets the size of corresponding row in the allocation table to 0.
void add_hole(int base, int size)	Creates a new hole in the holes linked list with the given base address and size. This hole should be placed such that its base address is greater than the base address of the previous hole but less than the base address of the next hole. Ties should not occur. After a hole is added, adjacent holes should be merged.
void remove_hole(int base)	Removes the hole at the given base address.
void merge_holes()	Iterates through the linked list, and merges any adjacent holes that are found.
int find_hole(int size)	Finds and returns a base address with the First-Fit hole selection algorithm, and updates the holes linked list. If no viable hole can be found, then return a -1 to signal that the process should be rejected.
int get_base_address(int pid)	Search the allocation table for the row that matches the given pid. Return the base address of this row.
int find_empty_row()	Search the allocation for any row which has a size of 0. Returns the index of this row, or a -1 if no rows are found.
int is_allowed_address(int pid, int addr)	Checks if the address given by addr is allowed for the process with the given PID. Returns 1 if valid, and 0 if not. Only addresses within the address space of the process are allowed.

scheduler.c

You will add a new function which removes a process from the ready queue. This function takes the PID of the process as input, and will search the ready queue for the process. Once found, the process will be removed. You will also add a function that returns the PID of the currently running process.

memory.c

You will modify the mem_read() and mem_write() functions of memory.c. These functions will now check with the SMM to see if their desired read/write operations are allowed for a given process. This can be done by calling the is_allowed_address() function and providing the address that the operation wishes to access as well as the PID of the currently running process.

If the is_allowed_addr() function returns a 1, then proceed with the operation as normal. If it returns a 0, then you must deallocate the process with the deallocate() function, and remove the process from the scheduler's ready queue.

main.c

The system should continue executing until there are no more processes left in the ready queue. Once the system is finished executing, print out the variable which counted how many holes were created during the runtime of the OS. You will also print out the following locations from memory:

- { 30, 150, 230 }

You can hard-code file locations and system configurations in main.c

Example Files

The following example files were uploaded with this PDF file:

- loop50.txt
    - Loops 50 times, and writes the value 1 to absolute address 30
- loop100.txt
    - Loops 100 times, and writes the value 2 to absolute address 150
- loop200_valid.txt
    - Loops 200 times, and writes the value 3 to absolute address 230
- loop200_invalid.txt
    - Loops 200 times, and writes the value 3 to absolute address 30. This will cause a memory access error, as absolute address 30 is outside the memory space allocated in program_list_invalid_access.txt.
- program_list_valid.txt
    - Loads 3 valid programs and gives them all reasonable sizes.
- program_list_invalid_allocation.txt
    - Loads 3 valid programs, but gives 2 of them unreasonable sizes.
- program_list_invalid_access.txt
    - Loads 2 valid programs, and 1 invalid program that will write outside its memory space.

Submission Requirements

Your submission must meet the following requirements:

- Your code will need to run the given instruction set programs, and then exit gracefully while printing the required memory locations.
- Your code will need to successfully compile and run on the UTD Giant Server.
- The following items must submitted in a zip file on eLearning
    - cpu.c
    - memory.c
    - disk.c
    - main.c
    - scheduler.c
    - smm.c
    - A readme.txt file that describes your code and how to run it.
    - Your makefile (if you used one)
    - Any necessary .h header files.