**Assignment 5**

**Due, Wednesday, November 10, 2021 for maximum 100%**

**Thursday, November 11, 2021 for maximum 90%**

**Friday, November 12, 2021 for maximum 80%**

**Saturday, November 12, 2021 for maximum 70%**

**Deliverables**

To complete this assignment you must submit your **OSManagement.c** to Webcourses.

**Project description**

This project will require students to simulate the behaviors of an operating system with a series of assignments.

1. Simulate process allocation to memory blocks based on memory management algorithms First Fit, Best Fit, Next Fit, and Worst Fit.
2. Simulate file management of directories and files stored in a directory.
3. Simulate multi-threaded programming with the POSIX (Portable Operating System Interface) threads (a.k.a. pthreads).

**C programming language integrated development environment (IDE)**

1. Code::Blocks ~ NOT Mac compatible
2. Visual Studio Code
3. Atom
4. <https://replit.com/>
5. XCode

**Assignment Scope: Memory Management**

1. **First Fit Implementation**:
   1. Input memory blocks with size and processes with size.
   2. Initialize all memory blocks as free.
   3. Start by picking each process and check if it can be assigned to current block.
   4. If size-of-process <= size-of-block if yes then assign and check for next process.
   5. If not then keep checking the further blocks.
2. **Best Fit Implementation:**
   1. Input memory blocks with size and processes with size.
   2. Initialize all memory blocks as free.
   3. Start by picking each process and find the minimum block size that can be assigned to current process i.e., find min(blockSize[1], blockSize[2],.....blockSize[n]) > processSize[current], if found then assign it to the current process.
   4. If not, then leave that process and keep checking the further processes.
3. **Worst Fit Implementation:**
   1. Input memory blocks with size and processes with size.
   2. Initialize all memory blocks as free.
   3. Start by picking each process and find the maximum block size that can be assigned to current process i.e., find max(blockSize[1], blockSize[2],.....blockSize[n]) > processSize[current], if found then assign it to the current process.
   4. If not, then leave that process and keep checking the further processes.
4. **Next Fit Implementation:**
   1. Input memory blocks with size and processes with size.
   2. Initialize all memory blocks as free.
   3. Start by picking each process and check if it can be assigned to the current block, if yes, allocate it the required memory and check for next process but from the block where we left not from starting.
   4. If the current block size is smaller then keep checking the further blocks.

**Tasks**

|  |  |
| --- | --- |
| Activity | |
| OSManagement.c | 1. Create source code file **OSManagement.c** |
| preprocessor | 1. Include the following C libraries    1. **stdio.h**    2. **stdlib.h**    3. **string.h** 2. Declare the following macros (a.k.a. global constants)    1. **EXIT** as value 0    2. **INVALID** as value -1    3. **MEMORY** as value 1    4. **FILES** as value 2    5. **THREAD** as value 3    6. **FIRST** as value 0    7. **BEST** as value 1    8. **WORST** as value 2    9. **NEXT** as value 3 3. Function prototype for functions    1. **displayMenu**    2. **clearScreen**    3. **memoryManagement**    4. **displayProcess**    5. **firstFit**    6. **worstFit**    7. **bestFit**    8. **nextFit** |
| main | 1. Write the **main** function to do the following 2. Return type **int** 3. Empty parameter list 4. Declare a variable, data type integer, to store the user’s menu selection (i.e. **choice**), initialized to -1 5. Write a loop to continue while variable **choice** is not equal to 0    1. Set variable **choice** equal to function call **displayMenu**    2. Use decision making logic to determine which OS function to run based on the following       1. 1 = **memoryManagement**       2. 2 = fileManagement – TBD; do nothing for this assignment       3. 3 = multiThreads – TBD; do nothing for this assignment       4. 0 = call function **exit()** passing **0** as an argument 6. Exit the program by returning 0 |
| displayMenu | 1. Write function **displayMenu** to do the following    1. Return type **int**    2. Empty parameter list    3. Declare a variable, data type integer, to store the user’s menu selection (i.e. **choice**), initialized to -1    4. Write a loop to continue while variable **choice** is not equal to -1       1. Write a series of **printf** statements to display the OS Management Menu       2. Prompt the user to enter their menu selection, store the input in variable **choice**       3. Use decision making logic to validate the user’s input is between values 0 and 3; if not, set variable **choice** equal to -1    5. Return variable **choice** |
| clearScreen | 1. Write function **clearScreen** to do the following    1. Return type **void**    2. Empty parameter list    3. Outputs to the screen using a call of function **printf** to prompt the user to hit the enter key to move on to the next screen as shown in **Figure 2 Clear screen function output**    4. Declare a variable of data type **char**    5. Call function **scanf** to store input in the **char** variable    6. Call function **system** passing argument explicit text “cls”    7. Call function **system** passing argument explicit text “clear”    8. If your operating system is Windows, comment out the second call to function **system** with argument “cls”    9. If your operating system is Mac or Linux, comment out the first call to function **system** with argument “clear” |
| memoryManagement | 1. Write function **memoryManagement** to do the following    1. Return type **void**    2. Empty parameter list    3. Call function **clearScreen**    4. Write a looping construct to loop for each of the four memory management algorithms       1. Declare a one-dimensional array, data type integer, to store the sizes of the memory blocks (i.e. **blockSize**) initialized with values 15, 10, 20, 35, 80       2. Declare a one-dimensional array, data type integer, to store the sizes of the processes (i.e. **processSize**) initialized with values 10, 20, 5, 30, 65       3. Declare a variable, data type integer, to store the number of blocks (i.e. **blocks**)       4. Declare a variable, data type integer, to store the number of processes (i.e. **processes**)       5. Write decision making logic based on the value of the looping variable (i.e. **algorithm**)          1. When **algorithm** is equal to **FIRST**, call function **firstFit**, passing arguments **blockSize**, **blocks**, **processSize**, and **processes**          2. When **algorithm** is equal to **BEST**, call function **bestFit**, passing arguments **blockSize**, **blocks**, **processSize**, and **processes**          3. When **algorithm** is equal to **WORST**, call function **worstFit**, passing arguments **blockSize**, **blocks**, **processSize**, and **processes**          4. When **algorithm** is equal to **NEXT**, call function **nextFit**, passing arguments **blockSize**, **blocks**, **processSize**, and **processes** |
| nextFit | 1. Write function **nextFit** to do the following    1. Return type **void**    2. Parameter list includes       1. One-dimensional array, data type integer, contains the block sizes (i.e. **blockSize**)       2. Parameter contains the number of blocks, data type integer (i.e. **blocks**)       3. One-dimensional array, data type integer, contains the process sizes (i.e. **processSize**)       4. Parameter contains the number of processes, data type integer (i.e. **processes**)    3. Declare a one-dimensional array, data type integer, to store the block id that a process is allocated to (i.e. **allocation**), size is parameter **processes**    4. Declare a variable, data type integer, to store the block allocation for a process, initialize to 0 (i.e. **id)**    5. Call function **memset**, passing arguments       1. Array **allocation**       2. -1 (i.e. **INVALID**)       3. **sizeof(allocation)**    6. Using a looping construct, loop through the number of **processes**       1. Using a looping construct, loop while **id** is less than the number of **blocks**          1. If the current block size (i.e. index **id**) is greater than or equal to the current process size (i.e. index of outer looping variable)             1. Update the **allocation** array to set the element at index of the outer looping variable equal to variable **id**             2. Reduce available memory of the current block size (i.e. index **id**) by the process size (i.e. index of the outer looping variable)             3. break out of the inner loop       2. Update the value of variable **id** to set the next index in array blockSize by adding **1** to variable **id** then modulus the total by the number of **blocks**    7. Call function **displayProcess** passing arguments **allocation**, **processes**, and **processSize** |
| firstFit | 1. Write function **firstFit** to do the following    1. Return type **void**    2. Parameter list includes       1. One-dimensional array, data type integer, contains the block sizes (i.e. **blockSize**)       2. Parameter contains the number of blocks, data type integer (i.e. **blocks**)       3. One-dimensional array, data type integer, contains the process sizes (i.e. **processSize**)       4. Parameter contains the number of processes, data type integer (i.e. **processes**)    3. Declare a one-dimensional array, data type integer, to store the block id that a process is allocated to (i.e. **allocation**), size is parameter **processes**    4. Call function **memset**, passing arguments       1. Array **allocation**       2. -1 (i.e. **INVALID**)       3. **sizeof(allocation)**    5. Using a looping construct, loop through the number of **processes**       1. Using a looping construct, loop the number of **blocks**          1. If the current block size (i.e. indexof the inner looping variable ) is greater than or equal to the current process size (i.e. index of outer looping variable)             1. Update the **allocation** array to set the element at index of the outer looping variable equal to the inner looping variable             2. Reduce available memory of the current block size (i.e. index of the inner looping variable) by the process size (i.e. index of the outer looping variable)             3. break out of the inner loop    6. Call function **displayProcess** passing arguments **allocation**, **processes**, and **processSize** |
| bestFit | 1. Write function **bestFit** to do the following    1. Return type **void**    2. Parameter list includes       1. One-dimensional array, data type integer, contains the block sizes (i.e. **blockSize**)       2. Parameter contains the number of blocks, data type integer (i.e. **blocks**)       3. One-dimensional array, data type integer, contains the process sizes (i.e. **processSize**)       4. Parameter contains the number of processes, data type integer (i.e. **processes**)    3. Declare a one-dimensional array, data type integer, to store the block id that a process is allocated to (i.e. **allocation**), size is parameter **processes**    4. Call function **memset**, passing arguments       1. Array **allocation**       2. -1 (i.e. **INVALID**)       3. **sizeof(allocation)**    5. Using a looping construct, loop through the number of **processes**       1. Declare a variable, data type integer, to store the current best fit value (i.e. **bestIdx**) initialized to -1 (i.e. **INVALID**)       2. Using a looping construct, loop the number of **blocks**          1. If the current block size (i.e. indexof the inner looping variable ) is greater than or equal to the current process size (i.e. index of outer looping variable)             1. If the value of **bestIdx** is equal to -1 (i.e. **INVALID**)   Set variable **bestIdx** equal to the current block (i.e. the inner looping variable)   * + - * 1. Else if the value of the block size at index **bestIdx** is greater than the value of the block size at index of the inner looping variable   Set variable **bestIdx** equal to the current block (i.e. the inner looping variable)   * + 1. If the value of variable **bestIdx** is not equal to -1 (i.e. **INVALID**)        1. Update the **allocation** array to set the element at index of the outer looping variable equal to variable **bestIdx**        2. Reduce available memory of the current block size (i.e. index **bestIdx**) by the process size (i.e. index of the outer looping variable)   1. Call function **displayProcess** passing arguments **allocation**, **processes**, and **processSize** |
| worstFit | 1. Write function **worstFit** to do the following    1. Return type **void**    2. Parameter list includes       1. One-dimensional array, data type integer, contains the block sizes (i.e. **blockSize**)       2. Parameter contains the number of blocks, data type integer (i.e. **blocks**)       3. One-dimensional array, data type integer, contains the process sizes (i.e. **processSize**)       4. Parameter contains the number of processes, data type integer (i.e. **processes**)    3. Declare a one-dimensional array, data type integer, to store the block id that a process is allocated to (i.e. **allocation**), size is parameter **processes**    4. Call function **memset**, passing arguments       1. Array **allocation**       2. -1 (i.e. **INVALID**)       3. **sizeof(allocation)**    5. Using a looping construct, loop through the number of **processes**       1. Declare a variable, data type integer, to store the current worst fit value (i.e. **wstIdx**) initialized to -1 (i.e. **INVALID**)       2. Using a looping construct, loop the number of **blocks**          1. If the current block size (i.e. indexof the inner looping variable ) is greater than or equal to the current process size (i.e. index of outer looping variable)             1. If the value of **wstIdx** is equal to -1 (i.e. **INVALID**)   Set variable **wstIdx** equal to the current block (i.e. the inner looping variable)   * + - * 1. Else if the value of the block size at index **wstIdx** is less than the value of the block size at index of the inner looping variable   Set variable **wstIdx** equal to the current block (i.e. the inner looping variable)   * + 1. If the value of variable **wstIdx** is not equal to -1 (i.e. **INVALID**)        1. Update the **allocation** array to set the element at index of the outer looping variable equal to variable **wstIdx**        2. Reduce available memory of the current block size (i.e. index **wstIdx**) by the process size (i.e. index of the outer looping variable)   1. Call function **displayProcess** passing arguments **allocation**, **processes**, and **processSize** |
| displayProcess | 1. Write function **displayProcess** to do the following    1. Return type **void**    2. Parameter list includes       1. One-dimensional array, data type integer, that stores the block number allocations (i.e. **allocation**)       2. Parameter that contains the number of processes, data type integer (i.e. **processes**)       3. One-dimensional array, data type integer, that stores the processes (i.e. **processSize**)    3. Write a looping construct to loop through the **processes** (i.e. **processSize**)       1. Display to the console the process number (i.e use the looping variable plus 1)       2. Display to the console the process size (i.e. **processSize** array at the current looping index)       3. Display to the console the memory block assigned based on the following logi          1. If the value stored at the current index of array **processSize** if -1 (i.e. INVALID), output **Not Allocated**          2. Else, output the current allocation (i.e. **allocation**) |
| OSManagement executable |  |
| Test Case 1 | Test Case 1 passes |
| Test Case 2 | Test Case 2 passes |
| Test Case 3 | Test Case 3 passes |
| Test Case 4 | Test Case 4 passes |
| Test Case 5 | Test Case 5 passes |
| Test Case 6 | Test Case 6 passes |
| Test Case 7 | Test Case 7 passes |
| Test Case 8 | Test Case 8 passes |
|  | Source compiles with no errors |
|  | Source compiles with no warnings |
|  | Source runs with no errors |
|  | Source includes comments |

**Perform the following test cases**

|  |  |  |
| --- | --- | --- |
| Test Cases | | |
|  | **Action** | **Expected outcome** |
| Test Case 1 | **Run executable** | The executable runs  The output in the command prompt should be similar to *Figure 1 displayMenu function output* |
| Test Case 2 | **User enters an invalid menu option value, value 2, or value 3** | The output in the command prompt should be similar to *Figure 1 displayMenu function output* |
| Test Case 3 | **User enters menu option value 1; memoryManagement function** | The output in the command prompt should be similar to *Figure 2 memoryManagement function output* |
| Test Case 4 | **User enters menu option value 1; nextFit function** | The output in the command prompt should be similar to *Figure 3 nextFit function output* |
| Test Case 5 | **User enters menu option value 1; firstFit function** | The output in the command prompt should be similar to *Figure 4 firstFit function output* |
| Test Case 6 | **User enters menu option value 1; bestFit function** | The output in the command prompt should be similar to *Figure 5 bestFit function output* |
| Test Case 7 | **User enters menu option value 1; worstFit function** | The output in the command prompt should be similar to *Figure 6 worstFit function output* |
| Test Case 8 | **User enters menu option value 0** | End of functionality; *“Process returned 0 (0x0)”* displayed in output similar to *Figure* ***7*** *Successful end of program* |

Text

Description automatically generated

Figure 1 displayMenu function output

Text

Description automatically generated

Figure 2 memoryManagement function output

A picture containing text

Description automatically generated

Figure 3 nextFit function output

Text

Description automatically generated with low confidence

Figure 4 firstFit function output

Text

Description automatically generated with low confidence

Figure 5 bestFit function output

Text

Description automatically generated with low confidence

Figure 6 worstFit function output

Graphical user interface, text

Description automatically generated

Figure 7 Successful end of program