COSC519 Operating Systems

Memory Management Simulation

December 8, 2010

Introduction

Memory management is a vital process for an operating system to run properly. Memory can be considered a scarce resource in that the memory requirements of all the programs being used in an operating system can possibly be larger than the amount of physical memory readily available. As a result, the data and instructions loaded into memory for processes must be continually managed through allocation and de-allocation in memory. The goal of this project was to simulate the operating system's use of memory by creating dummy processes and loading them into a conceptual memory modeled after typical memory found in modern computers. This established a better understanding how data is read, written, and removed from memory and the underlying components and processes that drive these actions.

Problem

In order to simulate memory management from an operating system's perspective, we need to appropriate all of the components that would be typically involved throughout a process lifespan. These components lie in the operating system, the memory hardware, and in the software that is being executed by the operating system. We must:

1. synthesize a process
2. prepare it for execution by placing it into a process queue
    - load the process into memory with the appropriate constraints
        - remains in its address space
        - does not violate logical or physical size constraints
        - is valid
        - follows the allocation rules set in place
3. ensure that the process in memorydoes not interfere with other processes in memory or the queue during its execution
4. run the process to completion
5. clear out the process from memory upon termination to create room for more processes

All memory management is performed logically from an operating system's perspective. However to create the whole picture of what is going occurring within memory, it was necessary to also simulate the behaviors of software and hardware to fully realize the entire process of memory management.

Approach

The group's approach to constructing a memory management simulator for the operating system consisted of the following:

Programming Language

The memory management simulation was written in the Java programming language using the Eclipse IDE. Specifically, the version of Eclipse used to code this application was Helios Service Release 1. The Java programming language was chosen primarily because of its familiarity to the project group members. In addition, once it was revealed that this was an entirely conceptual simulation requiring no interaction with any low-level hardware entities, the Java programming language seemed appropriate as the portability of the language could be easily written, compiled, run, and tested on all platforms.

Major Program Components

Our program will require the basic implementation of an operating system and the hardware/software entities for which it acts as an intermediary. As a result, our object model for our program relies on three main classes to model these three entities and a collection of helper classes that perform additional responsibilities to model memory management. The three main classes modeling the behavior of the operating system, memory hardware, and the computer software as well as the helper classes will be outlined later in this report and how they facilitate memory management.

Conceptual Overview

We intend to demonstrate an understanding of memory from a conceptual standpoint via the simulator. We planned to address the following concepts specifically:

Process Queue

The operating system portion of our program utilizes a simple process queuing algorithm to ensure that processes are continuously loaded into memory as other processes finish their run and terminate. This process queue continuously loads new processes until the total amount of processes to run designated at the beginning of the program is reached.

Paging

We will demonstrate the process of paging by dividing our physical memory into fixed-sized blocks (frames) and dividing our logical memory into blocks of the same size called pages. The memory management simulator should be able to keep track of the free frames. Then when it is time to run a program that is of n pages in size, we will be able to find n free frames in the physical memory and load the program into these. The physical memory frame sizes are configurable prior to initialization of the program as is total physical memory size.

Paging Table

The program uses a paging table to effectively translate logical addresses to physical addresses and vice-versa. Our paging table is resized on-the-fly and not only translates addresses but also assigns new addresses to physical addresses with empty logical addresses and logical addresses to empty physical addresses.

Base and Limit Registers for Running Processes

The simulation implements proper establishment of process boundaries when a process is first loaded into memory. These two boundaries were the base register from which the remaining logical addresses for a process would be derived and a limit register, which would indicate the highest logical address a process should be able to gain access to. In essence, we will simulate the length of a logical address space for a process by specifying these registers.

Address Binding

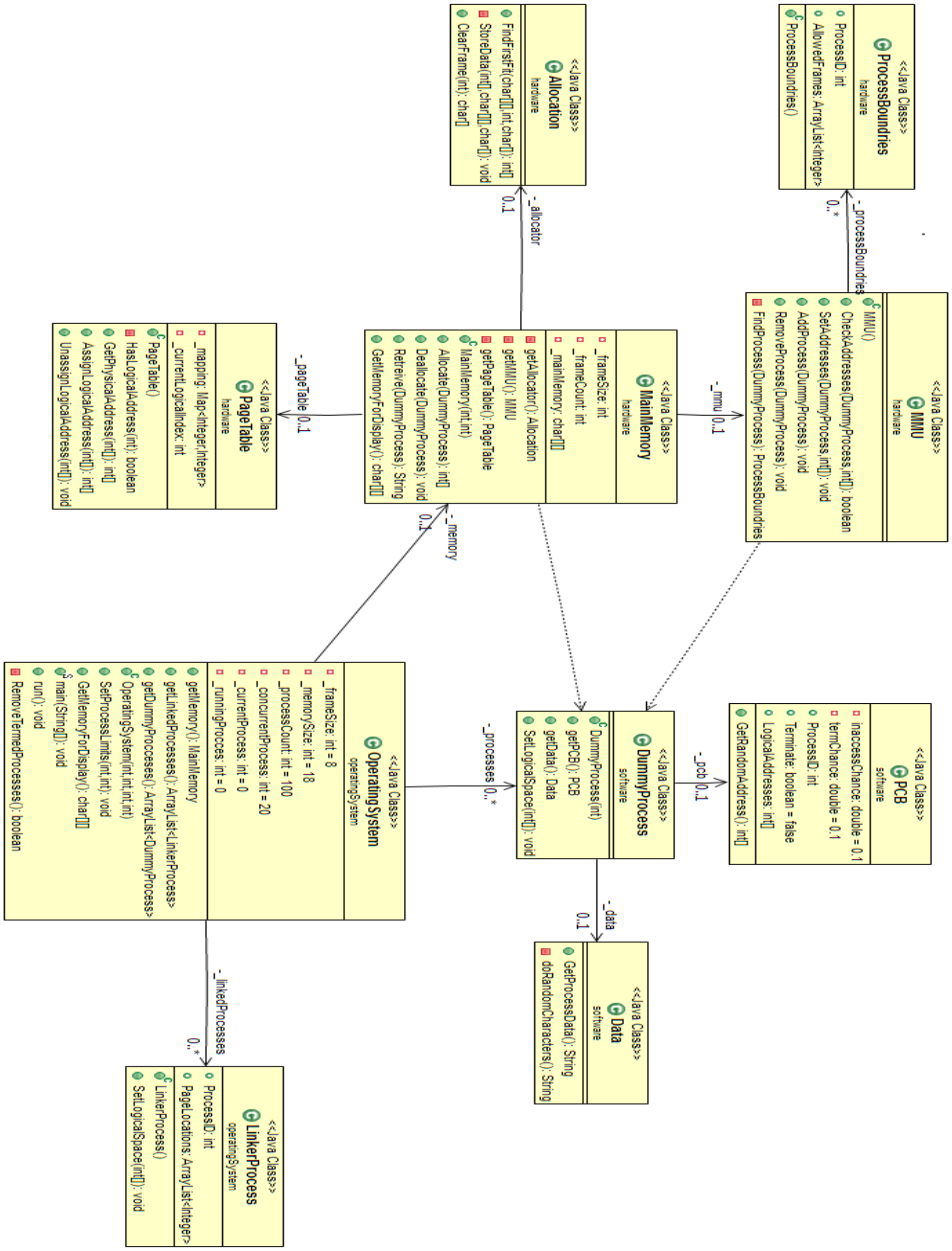
We achieved the simulation of address binding at load time when the process is executed. Compile time address binding would be possible but we do not initialize our processes until the program begins running as the program itself is an extremely basic version of three entities: an operating system, hardware, and software.

Fragmentation

Our program demonstrates the process of fragmentation that can occur in noncontiguous memory allocation. Thus if a block of 1000 bytes is requested, although the total available memory may be over 1000 bytes, this space might not be usable because the space is noncontiguous (i.e., we have four 300 byte blocks available for use but they are interspersed throughout the address space and not continuous). We will attempt to relocate memory pages should this happen in our simulation to allow for situations like this to be overcome.

Object Model

The object model displayed below shows the overall structure of the classes that comprise the memory management simulation. These objects are entities created to mimic real-life computing components. The object model includes the fields and methods that comprise each object as well as indicators of objects interacting with other objects. These objects were designed to generally adhere to the following process:



<<Java Class>>
ProcessBoundries
hardware
ProcessID: int
AllowedFrames: ArrayList<Integer>
ProcessBoundries()
<<Java Class>>
Allocation
hardware
FindFirstFit(char[][],int,char[]): int[]
StoreData(int[],char[][],char[]): void
ClearFrame(int): char[]
-_allocator
0..1
-_processBoundries
0..*
<<Java Class>>
PageTable
hardware
_mapping: Map<Integer,Integer>
_currentLogicalIndex: int
PageTable()
HasLogicalAddress(int): boolean
GetPhysicalAddress(int[]): int[]
AssignLogicalAddress(int[]): int[]
UnassignLogicalAddress(int[]): void
-_pageTable
0..1
<<Java Class>>
MainMemory
hardware
_frameSize: int
_frameCount: int
_mainMemory: char[][]
getAllocator(): Allocation
getMMU(): MMU
getPageTable(): PageTable
MainMemory(int,int)
Allocate(DummyProcess): int[]
Deallocate(DummyProcess): void
Retrieve(DummyProcess): String
GetMemoryForDisplay(): char[][]
-_mmu
0..1
<<Java Class>>
MMU
hardware
MMU()
CheckAddresses(DummyProcess,int[]): boolean
SetAddresses(DummyProcess,int[]): void
AddProcess(DummyProcess): void
RemoveProcess(DummyProcess): void
FindProcess(DummyProcess): ProcessBoundries
-_memory
0..1
<<Java Class>>
OperatingSystem
operatingSystem
_frameSize: int = 8
_memorySize: int = 18
_processCount: int = 100
_concurrentProcess: int = 20
_currentProcess: int = 0
_runningProcces: int = 0
getMemory(): MainMemory
getLinkedProcesses(): ArrayList<LinkerProcess>
getDummyProcceses(): ArrayList<DummyProcess>
OperatingSystem(int,int,int,int)
SetProcessLimits(int,int): void
GetMemoryForDisplay(): char[][]
main(String[]): void
run(): void
RemoveTermedProcesses(): boolean
-_processes
0..*
<<Java Class>>
DummyProcess
software
DummyProcess(int)
getPCB(): PCB
getData(): Data
SetLogicalSpace(int[]): void
-_pcb
0..1
<<Java Class>>
PCB
software
inaccessChance: double = 0.1
termChance: double = 0.1
ProcessID: int
Terminate: boolean = false
LogicalAddresses: int[]
GetRandomAddress(): int[]
-_data
0..1
<<Java Class>>
Data
software
GetProcessData(): String
doRandomCharacters(): String
-_linkedProcesses
0..*
<<Java Class>>
LinkerProcess
operatingSystem
ProcessID: int
PageLocations: ArrayList<Integer>
LinkerProcess()
SetLogicalSpace(int[]): void

Detailing the Object Model

As was previously mentioned, to appropriately simulate memory management, we created classes that model the behaviors of the three interfaces over which memory management occurs: hardware memory, the operating system, and computer software.

Operating System Component

OperatingSystem

The OperatingSystem class is the **core component of operating system** and it interfaces directly with the MainMemory (hardware) and DummyProcess (software) classes. This class inherits from the java Thread class such that it is run as a thread itself. By including a run() method in the class in cooperation with its inherited Thread properties, it is possible to use this class in the same manner as a main() method would to initialize the entire memory management simulation.

This class initializes the array lists that represent our collections of dummy and linked processes and imposes limits on memory size, frame size, process concurrency count, and total processes allowed.

It is in this class that the basic process queue is implemented. Processes are queued using a loop where if the number of processes active is below concurrency limit, a process is added and subsequently run. T

Finally, the OperatingSystem class has been given the ability to return statistical information regarding used space, free space, and total space.

Linker and LinkerProcess

The Linker and LinkerProcess classes were helper classes that actually track what is loaded into memory although they were not implemented entirely.

In theory, these classes would have been expanded to allow for dynamic linked libraries (DLLs) such that frequently used libraries could be reused without having to be reloaded to memory each time they are needed by separate processes to conserve space and resources.

Software Component

ProgramControlBlock

The program control block was designed to ensure that processes do not try to access memory outside of their address space. To receive confirmation of this implementation, we are throwing an error when a process tries to access a logical address outside of the address space.

This was done by using a built-in ten percent failure rate of hitting an attempt to access an inaccessible address. This ten percent failure rate was implemented with the help of the java.util.random class which performs many functions on randomly generated numbers.

DummyProcess

This class is the **core class of software** component and it interfaces directly with the OperatingSystem class (operating system) and the MMU and Main Memory classes (hardware).

The DummyProcess class performs the following functions:

1. **Conceptually emulates the behavior of operating system processes –**dummy processes are created by this class that contain strings of data. The process encapsulates the string data where it is eventually stored into memory.

2. **Initializes Data (random process data generator) and PCB**

3. **Defines logical address space for the process via the PCB**

Data

The Data helper class performs the following actions:

1. **Creates random strings of characters which serve as the “data” for our dummy processes**

2. **Returns these strings when called from other classes / methods**

Memory Hardware Component

MainMemory

The MainMemory class is the **core component of the classes modeled after memory hardware** and it interfaces directly with the Operating System class (operating system) and the DummyProcess class (software).

The MainMemory class also performs the following actions:

1. **initializes the PageTable, Allocator, and MMU classes**

2. **allocates and deallocates the process data from memory**

3. **returns the data displaying frame contents and the currently running processes for the GUI for display**

Allocator

The Allocator class is a helper class that is called from within the MainMemory class to perform three tasks:

1. **Performs first fit allocation algorithm** – the allocator checks out the entire length of memory and finds the first empty frame. It continues to do this until it finds enough frames to satisfy the size of the process that needs to be stored in memory. The frame locations are saved away.

2. **Stores data in frames designated –** the designated frame locations are used when the data is actually stored in these frames.

3. **Clears frames –** Frames are cleared by being reinitialized within the program.

PageTable

The PageTable class is essentially a translation table that keeps track of what physical addresse ranges are mapped to which logical address ranges and vice-versa. The PageTable class utilizes heavily the java.util.map class which is basically a hash table. The PageTable mapping object is resizable and is resized continuously as:

1. **Physical addresses are designated for logical addresses without a physical address pair**

2. **Logical addresses are designated for physical addresses without a logical address pair**

3. **Page table entries are cleared out once processes complete**

MMU

The MMU class is a helper class and performs the following actions:

1. **Calls FindProcess instance of ProcessBoundaries class to find the boundaries of the Dummy Process**

2. **Sets the logical address of a process based on looping through an array stored of allowed addresses. This loop ensures that the set address is not illegal, nor has it already been used**

ProcessBoundaries

The ProcessBoundaries class is a helper class that works to track what frames a process can physically access given its defined logical address space. This ensures that processes do not access frames outside of their allocated physical address space.

Program Flow

The program flow for a typical process being added to memory, executed, terminated, and removed typically occurs as follows:

Until the final process has run:

1. Add a new process if number of current running processes is less than concurrency limit

2. Get process IDs from PCB for linker and dummy processes

3. Allocate space for new dummy process

4. Process is filled with dummy data

5. Calculate the number of frames required to store the process containing data

6. Process is added to the MMU after finding process boundaries and frames that it is allowed to access

7. Page table updated with process' physical address

8. Logical address of the process returned to the operating system

9. Allocator runs the find first fit algorithm on entire memory object

10. Data is stored for this particular process when enough free frames to hold data are found

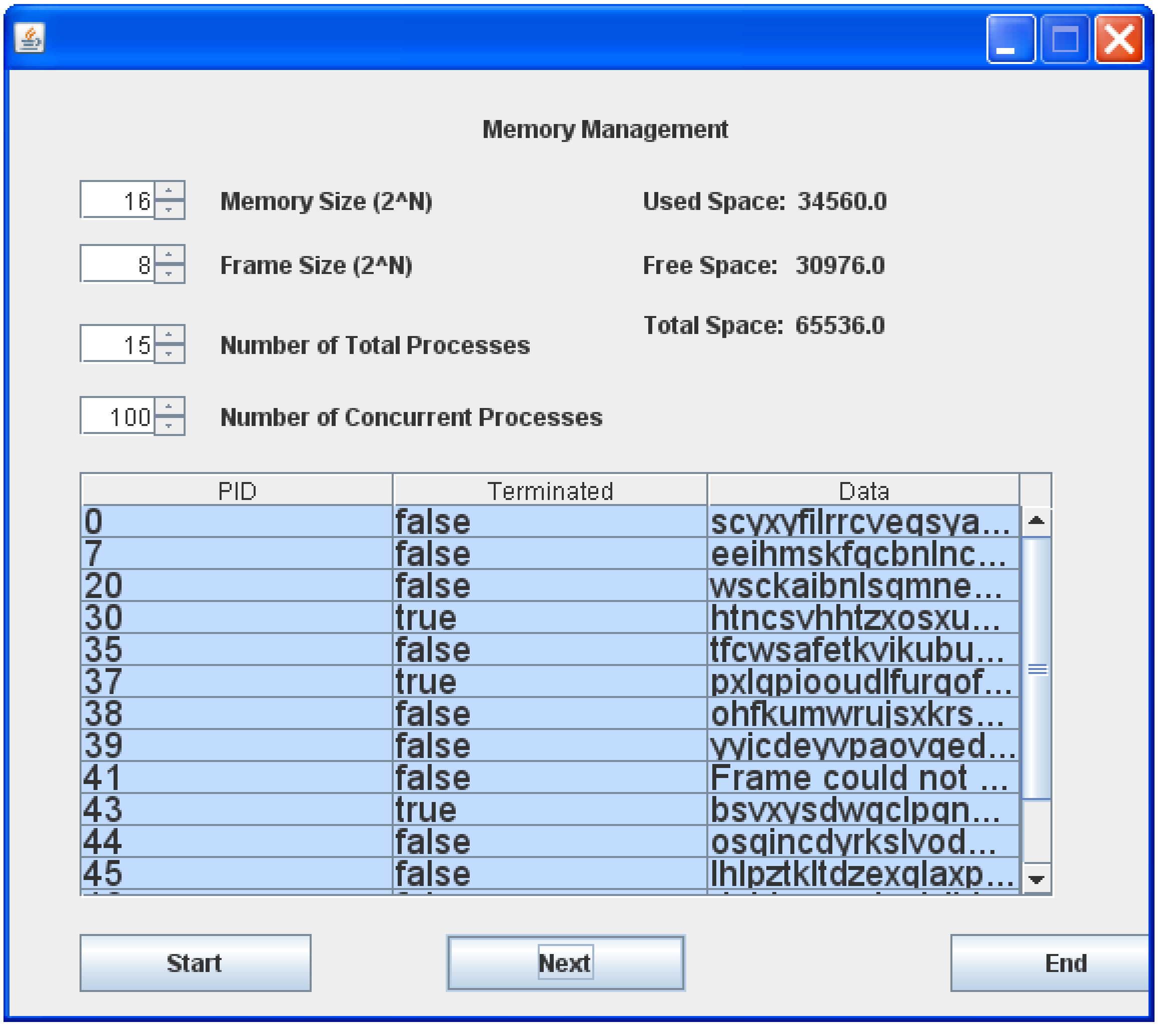
11. Check for processes that have terminated and continue for steps 12 - 15

When process has terminated:

12. Deallocate process and its data from the actual memory
13. Clear the page table entry corresponding to the process
14. Clear frames by reinitializing
15. Return to Step 1 if not the final process

Graphical User Interface

Our graphical interface serves to visually display the processes being loaded into memory and the data they contain. The graphical user interface was constructed using the Swing API for Java applications.



Memory Management

16 Memory Size (2^N)

8 Frame Size (2^N)

15 Number of Total Processes

100 Number of Concurrent Processes

Used Space: 34560.0

Free Space: 30976.0

Total Space: 65536.0

PID	Terminated	Data
0	false	scyxyfilrrcveqsya...
7	false	eeihmskfqcbnlnc...
20	false	wsckaibnlsqmne...
30	true	htncsvhhtzxosxu...
35	false	tfcwsafetkvikubu...
37	true	pxlqpiooudlfurqof...
38	false	ohfkumwrujsxkrs...
39	false	yyjcdeyvpaovqed...
41	false	Frame could not ...
43	true	bsvxysdwqclpqn...
44	false	osqincdyrkslvod...
45	false	lhlpztkltdzexqlaxp...

As shown above, the interface is essentially a table that displays snapshots of memory overtime. These snapshots detail the current Process IDs of the processes currently loaded into memory, whether or not they are terminated, and the string data that the processes are holding.

The program is initially populated with the first amount of processes determined by the spinner. After the user clicks next, output information is relayed regarding the state of the memory in terms of space.

Inputs

Our inputs for these processes are spinners with defaults set within the range what we know to function properly given our tests on the application once it was completed. There is a set range of maximums and minimums for each spinner.

Our inputs are as follows:

Memory Size (default of 2^16 bytes) – This input specifies our memory size. The minimum size of our memory is the same as the default at 2^16 bytes. Our maximum size that we have tested for and confirm as working in our program is 2^24 bytes.

Frame Size (default of 2^8 bytes) – This input specifies the paging that is performed on our memory. This can be altered to range from 2^2 bytes to 2^12 bytes.

Number of Total Processes (Defaults at 100) – This specifies the number of processes that will be generated over the lifespan of the program. This can range from 50 to 200 processes. These processes will be continually created until the total processes limit is reached given that the concurrency ceiling is not maxed out.

Number of Concurrent Processes (Defaults at 15) – This is the number of processes that can be running simultaneously. This can range from 25 to 5 processes running in parallel. When the concurrency cap is not reached, a new processes is created given that the number of total processes to be run throughout the life of the program has not been reached.

Output

Process ID (PID) – the assigned ID of the process that has been generated

Terminated flag – this flag denotes whether or not the process has finished running. If it has, when the user clicks next, the process will no longer populate the table and will be replaced by a new process if the concurrency limit has not been reached and there are still processes remaining to run in the process queue.

Data – this merely displays the string data that populates the process. There is a built-in ten percent change that the frame cannot be accessed in our Program Control Block object. When this occurs, the Data field is populated with "Frame could not be accessed." In addition, when there is fragmentation, this is displayed in the table by the data block. While frames that are completely full will have all the characters expected in their frame, fragmented frames (where the entire frame wasn't needed to capture the remaining process data) will not have completely populated data fields.

The output for our program, aside from the table of PIDs, terminated flags, and the actual program data in the strings is basic. We display the free space information, the used space information, and total space information in bytes of the memory.

Conclusion

The memory management simulation accurately simulates the majority of concepts the group aimed to implement.

If we were to go further, we would have worked on the Linker and Linker Process more to make them more robust. In addition, we would likely have a more customizable GUI so that additional testing and experimentation could be performed on the memory management simulation.

Due to the language barrier between the software architect and project manager and the two developers, we felt that the simulation was adequate. In addition, the loss of two group members over the course of the semester was detrimental to the overall progress of our simulation and as group feel that the simulation was sufficient given the overall language barrier and loss of contributors.