**Design Document for CS 350 Term Project (Fall 2012)**

**Jonathan Roberts and Jimmy Wilson**

**Preamble:**

The project was to develop a program that would simulate four processors that would allow a computer to run four processes simultaneously. This simulation was meant to create twenty jobs, each with a randomly generated data size and time length. The jobs would be inserted into a memory for the processors. The process was to run in a first fit scenario, a best fit scenario and a shortest job first scenario.

At the beginning of the project development, both team members began writing rough draft codes for the project. Jimmy began writing draft code for setting up memory space, generating the random sizes and time lengths of the jobs, and the first fit and best fit scenarios while Jonathan also began writing codes for the program. Due to busy schedules of both team members, setting a time to meet was difficult to accomplish. Jimmy sent what he had worked on to Jonathan through email.

After some time of individual work a suitable time for a team meeting was found. By the time of the team meeting, Jonathan had typed up a full program for the project. During a team meeting, the coding written by Jonathan Roberts was found to be the more efficient method. Both team members reviewed the codes and the test results of the program.

After reviewing the codes and result Jimmy made a few comments regarding corrections to ensure that the program performed all the need function, accomplished its purpose, and presented the output in the correct format. Jimmy’s review of the codes was conducted in comparing the program method and output to the example fragment provided in the instructions and the written description of the requested methods and manner of displaying data in output.

A few corrections were made to ensure the output was correct and to remove any unnecessary codes within the program. The program had been made to produce two output results in an uncertainty of how the final product was supposed to display the function. The analysis pointed to one output over the other.

Jonathan later obtained additional feedback from the instructor, Dr. Krishnaprasad, to ensure that the program was meeting the requested needs using the requested methods. After this, Jonathan went through the codes making some adjustments according to the minor issues identified in the team meeting and the input from Dr. Krishnaprasad. Jonathan then made the final edits to clean up the project into the final product.

**Assumptions**

For the purposes of this simulation, it was assumed:

-all processors execute concurrently in true parallel

-all simulated jobs arrive at the same time, and never in the middle of the simulation

-as a consequence of the above, the physical memory analyzed begins empty and available

-all job information relevant to priority (such as time needed to execute) is known in advance

-there would never be a job with a memory request outside the range of 16 to 64 (inclusive)

-there would never be a job with execution time shorter than 5 time units or longer than 15 time units

-no jobs depend on each other for execution (i.e. a process will not wait for information from another)

-all resources needed by a job will be immediately available upon request, thus no wasted execution time

**Data Structure Choices**

This simulation places an emphasis on linked lists (acting as queues) and arrays (typically for sorting purposes).

For each of the following, an array is used to:

-keep track of the individual memory segments

-keep track of the predetermined memory sizes (32, 48, 24, 16, 64, 32, 24)

-sort the processes to allocate to memory at the simulation start (when using Shortest Job First scheduling)

-sort the processes to allocate to the CPU after each CPU cycle

Linked lists are used by the schedulers to:

-keep track of the original queue of processes (naturally occurs in the order of process ID)

-enqueue processes to allocate to memory

-keep track of the processes currently assigned to memory (no longer functions as a queue)

-keep track of the processes that have finished executing, with the first to finish being the head

**Algorithm Strategies**

Each hypothetical process is created in the ProcessScheduler class within its addToBase method using the passed parameters representing the process ID and its randomly generated memory request and execution time.

Once all processes have been generated, the updateReadyQueue method is called, which will initially move processes from the base linked list to the waitQueue linked list. From there, it will attempt to allocate as many processes in the waitQueue linked list as possible to available memory segments. This allocation is handled by the allocateProcess method of the MemoryList class, which will search for an available memory slot (according to the memory selection scheme) by checking the size of the process to allocate and the size of each memory segment that is not already taken. Each memory segment stores information on which process is currently allocated to it, along with its maximum capacity and how much space is currently unused. Once a process enters the ready queue, its entry time is set, which will allow processes to gain a priority for the First-Come First-Served job schedule.

After the readyQueue is updated, the assignToCPU method can be called, which will execute on an amount of processors equal to the passed integer. The HypProcess class's executeProcess() method is called for every process that has been determined to go next in the ready queue. By calling this method, the HypProcess object that gets CPU time will decrement its time remaining value by an amount equal to the quantum length, and if its time remaining reaches 0, it will set its current state to FINISHED. If there is still time left, the process's state changes to RUNNING.

After this method finishes, the main program calls the readyQueueString and waitQueueString methods, which will give data relevant to the aftermath of the current time step, and output this data to a text file.

For the next time step, the updateReadyQueue method will be called, and then the assignToCPU method again. This process continues for each time step thereafter, with two main differences relevant to the updateReadyQueue and assignToCPU methods: The updateReadyQueue method will first check for any processes in the FINISHED state within the ready queue, then remove them and add them to the finished linked list. Furthermore, the assignToCPU method will check for any processes that may run again (i.e., there are more cores than processes in READY state), and the processes with highest priority (either shortest time remaining or entered the ready queue first) will be assigned to the CPU again.

**Conclusion**

From these simulations, it becomes apparent that the Shortest Job First scheduling strategy has an extremely strong effect on how many processes can complete during a simulation. However, due in part to the lack of memory segments that can satisfy the average randomly-generated process (under this simulation's assumptions and conditions), there is a high chance of starvation for processes requiring more than 32 KB of memory. Thus, the beneficial effects of best-fit allocation are not as apparent as would be expected. Furthermore, in every scheme, near the end of a 30 time-unit simulation, there is a significant amount of wasted memory. Even using the best fit allocation scheme does not alleviate this in an obvious manner; sometimes, the fragmentation and/or count of total processes that complete is even *worse* near the end of a 30 time unit simulation than the first fit allocation scheme simulations. However, the effects are not always obvious with a 30 time unit simulation.

**MemorySegment**

**Global Fields:**

int size

Represents the maximum space for this memory segment

int memoryNum

Represents the id of this segment

boolean filled

Value stating whether this segment is occupied by a process

HypProcess process

Object of type HypProcess that this segment "holds" currently

**Methods:**

MemorySegment(int mem, int s)

Constructor taking the memory segment id and size as parameters; initializes filled to false and process to null

int getMemoryNum()

Returns this memory segment's id

int getSize()

Returns this memory segment's maximum size

boolean isEmpty()

Returns false if the "held" process is null, true otherwise

HypProcess getProcess()

Returns (but does not remove) the current HypProcess object held

int wastedSpace()

Returns either the size of this segment (if empty) or the size of the internal fragmentation of this segment (if filled)

boolean fill(HypProcess h)

Adds a hypothetical process to this segment, and sets filled to true

HypProcess remove()

Removes the currently held process and returns it, then sets process to null and filled to false

**MemoryList**

**Global Fields:**

final int[] MEM\_SIZES = {32, 48, 24, 16, 64, 32, 24}

Array representing the sizes of memory segments, in order of id

enum MemorySelectScheme {FIRSTFIT, BESTFIT}

Enumeration representing schemes for allocation of a process to memory

MemorySegment[] memory

An array of MemorySegment objects

MemorySelectScheme scheme

Scheme for memory allocation used for this instance

int processesHeld

Represents how many processes are "held" in this list of memory

**Methods:**

MemoryList(MemorySelectScheme fit)

Constructor that takes the desired scheme of memory allocation as a parameter

boolean allocateProcess(HypProcess hp)

Method that will take an object representing a process, then attempt to allocate it to an available memory segment. If successful, this will return true. If there is no segment available or an error occurs, this will return false.

private int firstAvailable(HypProcess hp, int start)

A private method used by this class to find the first available memory segment, within the memory array, which will fit a process. The start parameter represents the index to begin the search

HypProcess deallocateProcess(HypProcess hp)

Method that removes a process from its respective memory segment and returns it

boolean isFull()

Returns true if and only if every MemorySegment object in the memory array is full

int getWastedMemory()

Returns the total wasted memory, found by calling each MemorySegment object's wastedSpace() method

**HypProcess**

**Global Fields:**

enum ProcessState {WAITING, READY, RUNNING, FINISHED}

An enumeration representing the 4 states of processes that are relevant to this simulation

int memoryRequest

Represents the amount of memory needed by this process

int executionTime

Represents the time this process needs to complete its instructions (at the start)

int timeRemaining

Represents the current time necessary for this process to complete its instructions

int idNo

Represents this process's ID, for differentiation in output

int inTime

Represents the time this process enters memory (default -1)

int memSegment

Represents the ID of the memory segment in which this process is located (default -1)

ProcessState currentState

Represents the state of this process

ProcessScheduler.ScheduleOrder schedule

Used by the job scheduler, this is the order by which this process should be sorted

**Methods:**

HypProcess(int id, int mem, int exTime, ProcessScheduler.ScheduleOrder sched)

Constructor that sets this process's id, memory request, and time request, in addition to the order of scheduling that should take place in compareTo() method override. Default state is WAITING and timeRemaining is initialized to the same value as the time request of this process. The in-time is initialized to -1 and memory segment id is initialized to -1 as well, to indicate that this process has neither actually started nor is in memory yet.

int getId()

Returns this process's ID number

int getSize()

Returns this process's memory request

void setInTime(int t)

Sets this process's in time based on the total theoretical time units that have elapsed since the simulation's start

int getInTime()

Returns the simulation time at which this process was allocated to an available memory segment

int getTimeRemaining()

Returns the amount of time needed before this process's instructions are complete

int getExecutionTime()

Returns the initial time request of this process

int compareTo(Object otherProcess)

compareTo method override used by the ProcessScheduler class for sorting (Arrays.sort() method)

void setState(ProcessState state)

Sets the current state of this process, if needed

ProcessState getState()

Returns the current state of this process

int getMemorySegment()

Returns the memory segment id this process currently occupies

void setMemorySegment(int address)

Sets the memory segment id this process occupies upon allocation

**Private Class:**

InvalidOrNullSchedulingException extends RunTimeException

Exception class used for debugging

**ProcessScheduler**

**Global Fields:**

enum ScheduleOrder {FCFS, SJF}

Enumeration of values representing whether to prioritize a process's entry time or its time request

LinkedList<HypProcess> baseQueue

A linked list of hypothetical processes in the order of creation (i.e., in the order of ID)

LinkedList<HypProcess> waitQueue

A linked list of hypothetical processes ordered by priority of job scheduling (e.g., if using Shortest-Job First scheduling, the lowest time request processes will occur first in this list); this represents all processes waiting to be assigned to memory

LinkedList<HypProcess> readyQueue

A linked list of hypothetical processes that are currently in memory and able to execute

LinkedList<HypProcess> finished

A linked list of processes that have finished executing and are no longer enqueued

int timeElapsed

The amount of time that has taken place in this simulation, incremented with each CPU cycle

int quantum

The time slice allotted for each "turn" of round robin scheduling to processes that will be executed by the CPU

ScheduleOrder schedule

The job schedule priority scheme used by this particular instance

MemoryList ram

The MemoryList object that is used for this simulation

**Methods:**

ProcessScheduler(int quant, MemoryList mem, ScheduleOrder sched)

Constructor that initializes the length of each round robin timeslice, sets the scheduling order of processes, and receives the memory object which is used throughout scheduling; timeElapsed is initialized to 0 and all linked lists are instantiated

void addToBase(int id, int spaceRequest, int execTime)

Takes in the id, memory request, and time request of a process and creates a new HypProcess object with appropriate parameters, then finally adds this process to the end of the baseQueue linked list

int totalWaiting()

Returns the size of the waitQueue linked list

int jobsCompleted()

Returns the size of the finished linked list

boolean exhaustedQueues()

Returns true if and only if waitQueue and readyQueue are empty (no processes to execute or allocate)

void updateReadyQueue()

At the first time step, will either set the waitQueue linked list equal to baseQueue, or sort baseQueue and set waitQueue equal to this (depending on scheme). This method then updates data in the following order:

any process in readyQueue in the FINISHED state will be removed from memory and added to the finished linked list; then, any process in waitQueue that can be allocated to a memory segment will be allocated using MemoryList's appropriate methods

void assignToCPU(int numThreads)

Takes as a parameter the maximum number of threads the CPU can handle (i.e., the number of cores), then determines how many threads will receive CPU time for the current time step. Afterward, an array of this size is constructed which will represent the next processes to receive CPU time. It is then filled with as many processes in READY state (i.e., in memory and did not go last turn) as possible, and if there are still cores available (i.e., the array has not been filled), the remainder is filled with processes that had their turn during the last time step. These final processes are added in such a way that only processes with the highest priority (based on the initialized schedule enumeration value) will go twice in a row.

String[] readyQueueString()

Returns an array of strings with the following information for each process that is currently allocated memory (each element in the array representing a process's information) in given order:

current time step, process ID, memory segment the process has been allocated to, memory request of the process, time remaining for the process before execution is finished, and current state of the process (either READY or RUNNING while in ready queue)

String[] waitQueueString()

Returns an array of strings with exactly the same information as the readyQueueString() method, but for the processes currently waiting to be assigned to memory