# HOST DISPATCHER DOCUMENTATION

**Author:** Ashwin Ramesh
**Student ID:** 311254012

# 1. Introduction

The seconds assignment in the Operating System Internals course focuses on the implementation of a hypothetical host dispatcher system. The main inspiration is to try and mimic the actual process dispatching mechanism used by the various operating systems in today's computers.

This design document will, in detail, explain the reasoning behind the memory allocation algorithm used, the various data structures created to make the dispatching algorithm work and possible issues stemming from this implementation and potential improvements in the next iteration of the program.

# 2. Memory Allocation Algorithms

The specification for the Host Dispatcher Assignment was to ensure that the memory allocation algorithm used was implemented using contiguous allocation. This means that memory is consecutively allocated to each process. The size of the process is compared to the amount of contiguous main memory remaining and if there is a block of memory, the process is placed in memory otherwise it is placed in queue awaiting memory. [1] This constraint resulted in a subset of memory allocation algorithms that could be implemented for this assignment.

## 2.a. First Fit

The First Fit Algorithm allocates the first large enough contiguous block of memory found to the process. The algorithm uses a linked list type data structure, where each block of memory is connect to its neighbours. It starts at the first block, iterating through each of the next blocks until it finds a large enough block of memory. If that block of memory is too large, it will be split into two separate blocks such that the extra memory portion is left free for other processes. [2]

The obvious advantages of the First Fit algorithm is that it allocates memory very quickly and manages to merge smaller, neighbouring block to become a larger block. However, as the algorithm always starts at the front of the linked list, smaller holes tend to accumulate at the beginning of memory, increasingly fragmenting the whole memory block. [3] This is also known as internal fragmentation.

## 2.b. Next Fit

As mentioned above, the First Fit algorithm has the downfall of heavy fragmentation at the beginning of the memory block. An improvement would be to leave the pointer to the last allocated location rather than moving back to the start of the list. This would better distribute the allocation process. This method of memory allocation is known as the Next Fit algorithm. However, the Next Fit algorithm does create the issue of external fragmentation, such that the memory allocation causes many small blocks of free memory to be scattered throughout the total memory space. This results in larger processes not being able to run due to a lack of contiguous memory. [4]

## 2.c. Best Fit

The Best Fit algorithm is much slower than both the First Fit and Next Fit algorithms. Instead of allocating to the first largest block of free contiguous memory, it will search through the whole memory space to find the smallest block which will fit the process memory required. [5] This means that the memory is efficiently allocated. The performance of this algorithm is the worst among the various algorithms described in this report.

## 2.d. Worst Fit

The Worst Fit algorithm will allocate the largest remaining block to the next process, which will allow future processes to use the remaining memory from that block. The idea is to maximise the split to increase the probability of allocating memory to other processes. [6] This reduces the chances of external fragmentation but is also very slow to run.

## 2.e. Buddy System

Unlike the various “Fit” systems described above, which use a linear linked list data structure to store contiguous memory blocks, the buddy system uses a binary tree type data structure to allocate memory between processes. The allocator will only assign blocks of specific sizes, and has many free lists one for each permitted size. Usually these sizes are powers of two.

If the requested memory size is not exactly available, then the algorithm will split a larger block in two again and again until a block of the correct size is created. The main advantage of this system is that merging (coalescence) of adjacent blocks is cheap since the neighbour of any free block can be calculated from its address.

This type of system can either react well to incoming processes or can cause severe internal fragmentation due to the rounding of block sizes. Typically, the process of freeing and merging of memory is extremely fast, maxing out at $\log(\frac{\text{max. block size}}{\text{min. block size}})$ . [7]

## 2.f. Personal Choice

For this assignment, my personal choice was to use the First Fit Algorithm. Although the Buddy System does offer a better memory management scheme in the long run, it is far too complex for this type of assignment. On the other hand, the other "Fit" algorithms were not as efficient as the First Fit or resulted in a far more fragmented memory allocation. Finally, the First Fit algorithm is extremely easy to implement, allowing me to focus on the actual dispatcher and queuing system in more detail.

# 3. Data Structures used in the Host Dispatcher

This section of the report will outline the various data structures and algorithms used to create this hypothetical host dispatcher.

## 3.a. Process Structure

The process data structure is fairly simplistic. It stores the process id created by the system, various I/O devices, memory size required as well as its arrival time and processing time. The processes also work in a linked list nature, allowing easy en-queuing and de-queuing from the various queues. All memory allocated to this process will be retained for the life of the process, once the processing time is up, the memory and the process will be freed.

## 3.a. Queue Structure

The Host Dispatcher is implemented using six different queues, each utilising an underlying linked list. As illustrated on the diagram below, there is a multi-level feedback queue such that any process having completed one iteration of the clock will have it's priority reduced (unless it is already on the lowest priority level). For example, a process beginning on priority 1 will move to priority 2 then priority 3 over the course of it's processing.

For this assignment, the maximum amount of processes is 1000, which will be read from an input file, into the Job Dispatch List. From here, items are allocated to either the Real Time or User Job queues when the arrival time of the process matches the current system time.

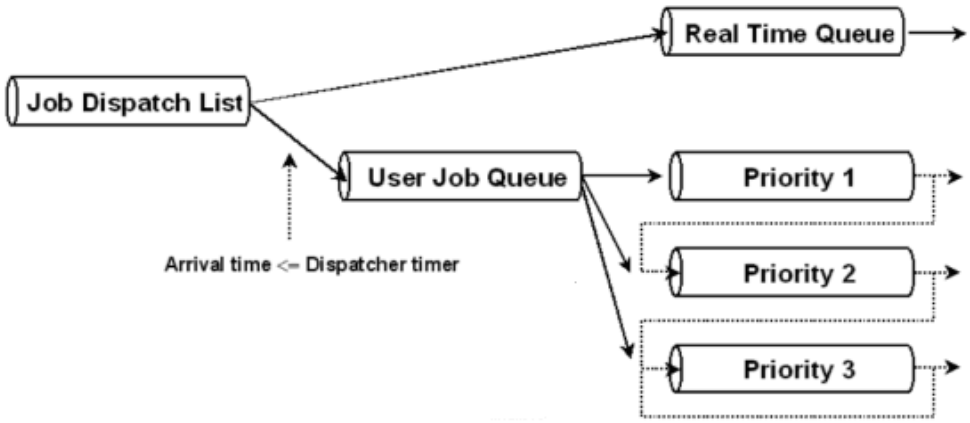


*Figure 1*

The feedback queues only run for “user job” processes, or those processes that require I/O resources. They run on priority levels one through to three. On the other hand, “real-time” processes are automatically priority zero, allowing them to continually run before any user jobs.

## 3.b. Memory Structure

The memory allocation block (MAB) is also implemented using a linked list data structure. From the assignment specification, all real time processes will automatically be allocated 64 MB of memory, while user jobs will be allocated 960 MB altogether. This adds up to 1024 MB of memory.

As described in section 2 of this report, the First Fit algorithm will be used, meaning that after finding a suitable MAB, any extra MB will be split into a separate MAB. Similarly, when the memory is freed, the MAB will try to merge with it's neighbours to create a larger block. All user jobs will only be queued into the feedback queues when an appropriate MAB is located.

## 3.c. Resource Structure

I/O resource allocation uses a simple data structure storing the maximum I/O devices available. In this system, there are 2 printers, 1 scanner, 1 modem and 2 CDs. This can easily be changed by re declaring values in the source code. Similar to memory, user jobs cannot enter the feedback queues until resources are freed. Each process carries its resources until the process is terminated.

## 3.d. Dispatcher Structure

With the use of the aforementioned structures, the dispatcher runs while there are still any processes remaining to run or while a process is currently running. The dispatcher's main role is to place processes onto correct queues and simultaneously allocate memory and resources using the systems mentioned previously. As a hypothetical system, the dispatcher does not actually process using I/O devices or actual processes, but rather uses a predefined program provided by the University.

# 4. Program Structure and Modules

## 4.a. File System

The implementation of the host dispatcher provided with this report was carefully structured into source files (*.c) and include files (*.h). This allowed appropriate abstraction while allowing for a neat folder structure. This is listed below:

**[host_dispatcher]**
├── makefile ← Compiles the dispatcher
├── readme.md
├── report.pdf
├── **[inc]**
│ ├── dispatcher.h
│ ├── mab.h
│ ├── pcb.h
│ ├── readinput.h
│ └── resources.h
└── **[src]**
├── dispatcher.c
├── hostd.c
├── mab.c
├── pcb.c
├── readinput.c
├── resources.c
└── sigtrap.c

The makefile is used to compile the dispatcher. To compile with GCC, type “make”. To compile with CLANG type “make clang”. The compile will result in “hostd” and “sigtrap” object files, where “hostd” is the dispatcher and “sigtrap” is the fake process.

## 4.b. Interface Functions

The host dispatcher contains six major source files. The interface functions of each will briefly be explained below: (NOTE: smaller functions are not defined here)

**4.b.i. hostd.c** – is the main file, calls all other functions

- *main():* Checks that input file is provided, reads the input into a queue then runs the dispatcher with this queue.

**4.b.ii. readinput.c** – functions to read and validate input

- *count_file_lines():* counts the number of lines in the input file
- *validate_input_line():* validates that the input provided follows the rules of the dispatcher
- *read_file():* iterates through each line in the file and adds it to the input queue if it is correctly validated

**4.b.iii. dispatcher.c** – provides the interface to pass processes between queues and start/suspend/stop them while acting as the dispatcher itself.

- *running_processes():* checks if a process is currently running. Stops it if time is up, suspends and requeues if not. Force runs real time processes.
- *start_process():* starts or resumes a process from the highest level queue possible
- *enqueue_roundrobin():* enqueues processes from the user queue to the appropriate feedback queue when process has “arrived”
- *enqueue_user_real_queues():* enqueues processes from the input queue to either the real time or user queue
- *dispatcher():* runs the above processes and acts as the dispatcher. Controls the clock

**4.b.iv. pcb.c** – represents the functions and struct for the process

- *pcb_dequeue():*dequeue head process from a queue and return it
- *pcb_enqueue():*enqueue a process to the tail of a queue
- *pcb_create():*create a process with default values and return it
- *pcb_free():*free a given process from memory and queue
- *pcb_start():*start the process (send SIGCONT if already suspended)
- *pcb_terminate():*stop the process (send SIGINT)
- *pcb_suspend():* suspend the process (send SIGTSTP)

**4.b.v. mab.c** – the memory struct and functions relating to manipulating the memory using the First Fit Algorithm

- *mabCreate():*create a MAB struct with initial values and return it
- *memChk():*check if a given size block is free in the MAB linked list and return it
- *memAlloc():*allocate a block of memory of a given size if a free block exists
- *memFree():*free the MAB and merge the size with neighbouring blocks
- *memMerge():*merge a given MAB with it's right neighbour
- *memSplit():* split a MAB into a given size and the remaining size

**4.b.vi. resources.c** – the resource struct and functions relating to manipulating the resources used

- *create_resource():*create a resource struct with predefined values
- *allocate_resource():* allocate the resources required for a process
- *check_resource():* check if resources for a process are available
- *free_resource():* free the resources used by a completed process

## 4.c. Justification of Structure

The most important part of developing software is to ensure future readability and maintainability. The manner in which this version of the host dispatcher is structured allows future developers to easily search and modify specific source files, while the consistent manner of function definitions allows for faster (future) development. Finally, the makefile is defined to not only provide a normal compile action, but also a CLANG compile, which provides far more verbose debugging statements. These reasons are the basis for this modular structuring of the host dispatcher program files.

# 5. An Examination into the Host Dispatcher and Real World Systems

The Host Dispatcher system implemented for this assignment is a simplistic version of what real world operating systems use. The key focus of the assignment was not to implement a full fledged dispatcher, but rather to allow students to learn the workings of such a dispatcher from a high level perspective. Obviously there were key problems with this developed version of the host dispatcher and improvements can definitely be made.

## 5.a. A Comparison of Shortcomings in the Host Dispatcher and Operating Systems

Multi-level dispatchers have been implemented and used in many real world applications, having been created back in 1962 by Corbato et al. [8] In actual operating systems (OS) this type of queuing will give preference to short and I/O bound processes, while longer running processes will be given a lower priority. [9]

Unlike the dispatcher created in this course, which only uses four levels of priority, actual OS have a far greater number of priority rankings. For example, Windows 7 has priority levels from zero to 31 [10], while Linux 2.4 had priority levels from zero to 140. [11] In both these cases, the OS would decide where to place each process depending on how small or how important the task was. One of the key problems for the OS is to decide how to weight the processes and how to prioritise them accordingly .For example, video playback should be given a far higher priority level compared to image processing.

The memory allocation scheme used by the host dispatcher also does not conform with the implementation in real world OS. Although memory is allocated, the host dispatcher does not actually store any program information in that memory block. Additionally, the memory space for the hypothetical dispatcher is limited to 1024 MB (64MB for real time), while in reality, this size is far greater while allowing real time processes to accumulate a greater percentage of memory. This is achieved by utilising virtual memory. Instead of storing a process in memory (RAM), the MAB will store an address to memory located on the disk (this is known as the virtual memory). This allows OS to over step its limited memory space. [12]

The time quantum in scheduling processes represents the time slice for each process to run before it is suspended and en-queued back into one of the priority queues. In the case of the host dispatcher, this quantum is defined to be 1000ms (1 second). In OS this quantum is ever changing, allowing some high priority tasks to have a greater slice of time. For example, the Linux OS has a 200ms quantum for real time tasks, while “nice” (user jobs) processes were only allocated 10ms per iteration.[13] This also goes to show that 1000ms is far too long for any one process, but this fact can be forgiven as the host dispatcher's output would only be readable at this speed.

The resource allocation process in the dispatcher created is extremely naive. It assumes that all resources required will be allocated at the start of the process and does not take into account possible additional resources during run time. OS have memory areas that point to these resources and during allocation, use semaphores to stop potential deadlock and starvation issues. [14] Furthermore, OS have the capability to force free resources that are being overused, a feature not present in the dispatcher.

Finally, most OS nowadays have multiple cores (processors) which allow for parallel processes. The dispatcher written for this assignment only assumes one core, as only one process ever runs per time quantum iteration. Obviously with the implementation of parallel processes, locks and other sort of checks must be made to ensure there are no race conditions or deadlocks. The algorithms used by OS are far more complex, whereas the algorithm used for the host dispatcher is far more linear.

## 5.b. Possible Improvements to the Host Dispatcher

Operating Systems have continually been improved upon for the past fifty years. The methods used to allocate memory, resources and time to processes are far too complex to simply re implement for an university assignment, however some improvements can be made to make the host dispatcher more realistic. These improvements are listed in dot points below:

- Reduce the time quantum and provide various quantum sized for the different queues.
- Increase the number of real time queues and user queues to better mimic real OS.
- Allow for a more dynamic allocation of memory through paging and virtual memory.
- The hypothetical process should allow for dynamic resource allocation.
- Allow for parallel processing with the use of threading

# 6. References


1. Janssen, C. (1956) *What is Contiguous Memory Allocation? - Definition from Techopedia*. [online] Available at: http://www.techopedia.com/definition/3769/contiguous-memory-allocation [Accessed: 2 Jun 2013].
2. Memorymanagement.org (n.d.) *The Memory Management Reference: Beginner's Guide: Allocation*. [online] Available at: http://www.memorymanagement.org/articles/alloc.html [Accessed: 2 Jun 2013].
3. Www2.cs.uregina.ca (n.d.) *Operating Systems*. [online] Available at: http://www2.cs.uregina.ca/~hamilton/courses/330/notes/memory/segmentation.html [Accessed: 2 Jun 2013].
4. Answers.com (n.d.) *What is the difference between external and internal fragmentation?*. [online] Available at: http://wiki.answers.com/Q/What_is_the_difference_between_external_and_internal_fragmentation [Accessed: 2 Jun 2013].
5. Courses.cs.vt.edu (n.d.) *Online CS Modules: Memory Allocation*. [online] Available at: http://courses.cs.vt.edu/csonline/OS/Lessons/MemoryAllocation/ [Accessed: 2 Jun 2013].
6. Research.cs.vt.edu (2011) *Section 2.1.4 - Worst Fit*. [online] Available at: http://research.cs.vt.edu/AVresearch/MMtutorial/WorstFit.php [Accessed: 2 Jun 2013].
7. Memorymanagement.org (n.d.) *The Memory Management Reference: Beginner's Guide: Allocation*. [online] Available at: http://www.memorymanagement.org/articles/alloc.html [Accessed: 2 Jun 2013].
8. Unknown. (n.d.) *Scheduling: The Multi-Level Feedback Queue*. [e-book] http://pages.cs.wisc.edu/~remzi/OSFEP/cpu-sched-mlfq.pdf.
9. Answers.com (n.d.) *What is Multilevel feedback queue?*. [online] Available at: http://wiki.answers.com/Q/What_is_Multilevel_feedback_queue [Accessed: 2 Jun 2013].
10. Msdn.microsoft.com (2012) *Scheduling Priorities (Windows)*. [online] Available at: http://msdn.microsoft.com/en-us/library/windows/desktop/ms685100(v=vs.85).aspx [Accessed: 2 Jun 2013].
11. En.wikipedia.org (2008) *Scheduling (computing) - Wikipedia, the free encyclopedia*. [online] Available at: http://en.wikipedia.org/wiki/Scheduling_(computing)#Linux [Accessed: 2 Jun 2013].
12. En.wikipedia.org (2012) *Virtual memory - Wikipedia, the free encyclopedia*. [online] Available at: http://en.wikipedia.org/wiki/Virtual_memory [Accessed: 2 Jun 2013].
13. Oreilly.com (2000) *Understanding the Linux Kernel: Chapter 10: Process Scheduling*. [online] Available at: http://oreilly.com/catalog/linuxkernel/chapter/ch10.html [Accessed: 2 Jun 2013].
14. En.wikipedia.org (2006) *Resource (computer science) - Wikipedia, the free encyclopedia*. [online] Available at: http://en.wikipedia.org/wiki/Resource_(computer_science) [Accessed: 2 Jun 2013].