

# COMP3520 Host Dispatcher Design Document

Justin Ting, jtin2945, 430203826

May 2015

# 1 Memory Allocation Algorithm

A number of memory allocation algorithms were available to choose from for this assignment, and each one will be detailed below - with a description of what they do, their advantages and disadvantages, and a worked example that will be consistent across each algorithm, using the specific version of the algorithm detailed in each subsection.

The algorithms that were allowed to be chosen from were First Fit, Worst Fit, Next Fit, Best Fit, and the Buddy System, all of which at least partially fall under the category of ‘Dynamic Partitioning’ (the latter being in a sense a ‘combination’ of fixed and dynamic partitioning), in contrast to ‘Fixed Partitioning’. The former allows a variable number of partitions as well as their sizes, while the latter has both a fixed number of partitions as well as their sizes (although fixed sizes does not necessarily indicate fixed *equal* sizes - unequal fixed sizes also still fit the category as well).

At this point, part of the narrowing down of algorithm choice has already been done for us due to the limitations of the specifications of the task - the exclusion of Fixed Partitioning algorithms means that we automatically avoid some of their main disadvantages, including having a fixed (and hence) inflexible number of processes that can be running at any one time, inefficient use of partition space by smaller jobs, and internal fragmentation within partitions. Not only this, but virtually no real world systems still employ the use of fixed partitioning, a clear indicator of its inefficiencies and pitfalls.

For our available class of algorithms, they share a number of properties which for the sake of brevity will be outlined in the introduction here rather than repeated in each subsection. Memory blocks managed through dynamic algorithms do not suffer from *internal fragmentation*, but rather, *external fragmentation*. Whereas internal fragmentation is a more inherent problem that cannot be solved without major modifications to an algorithm itself, external fragmentation is an issue that can be solved more simply (although not without a cost) - through the use of compaction. While internal fragmentation alludes to unused space within fragments, external fragmentation is the space between allocated partitions which are unused (and potentially can’t have any pending jobs slot into it). Compaction solves this by periodically (based on a factor determined by the algorithm) by packing all the memory together back into a contiguous block, external fragmentation is at that moment in time eliminated - though this has issues of its own, as it requires both time and processing power, but more importantly, the now-modified locations of memory blocks can’t invalidate references that previously existed which programs relied upon.

The following algorithms are named based on the way in which they allocate memory to incoming programs’ requests.

## 1.1 Best Fit

The **Best Fit** memory allocation algorithm, as the name suggests, will find the smallest block available that is able to fit the request which it is given, starting from the beginning. This means that each search for an appropriate block is of  $O(n)$  complexity, as an entire memory block must be traversed to determine the best-fitting block. This algorithm performs dismally, as always finding the best block means that the resulting *size* of unused space (external fragmentation) is minimised - the wording of which can lead one to incorrectly believe is a positive thing, but in fact, rapidly creates a large number of miniscule, unusable blocks of memory, requiring compaction, which, as described above, is an expensive process which we want to minimise - while the Best Fit algorithm essentially maximises it instead. As a result, the best fit algorithm is in fact the worst performer of the pure dynamic allocation algorithms.

**However**, it is worth noting that despite its theoretical worst performance, this worst-case behaviour is often not observed, and when these cases do not surface simply due to the nature of the programs making requests

to a block of memory, it doesn't necessarily perform worse than some of the other dynamic memory allocation algorithms. While some sources will find this point contentious, some (and possibly a majority) will actually claim that research has shown that Best Fit and First Fit aren't actually superior to each other in terms of wasted memory (though the latter holds)

## 1.2 Next Fit

The **Next Fit** memory allocation algorithm will remember the last spot in which memory was last allocated (naturally, this will be the beginning if any memory is yet to be allocated), and serve requests by finding the next available block that is sufficiently large - taking  $O(n)$  time. Under normal operation, this means that what usually happens is the largest block at the 'end' of memory will be quickly fragmented as processes are finishing and their allocated memory freed, and new processes with different requirements take their place. This points to a relatively frequent need to compact memory, though less so than the theoretical worst case that the Best Fit algorithm requires.

## 1.3 First Fit

The **First Fit** memory allocation algorithm search from the beginning of a memory block to find the first block that is sufficiently large to satisfy the request being made. This traversal has a complexity of  $O(n)$ . Due to searching from the start every time, it takes longer than Next Fit to require breaking up blocks in the later areas of memory, which also means that theoretically, fragmentation will occur slower and hence compaction doesn't have to happen as frequently, reducing the overhead of the algorithm in this regard. However, though it holds the same complexity as Next Fit for each search, the *actual* time to search is slightly higher as it has to check from the start each time, rather than simply the remaining memory left from the last allocated location. This means that during operation when not in the process of compacting, the First Fit algorithm is slower than the Next Fit algorithm.

## 1.4 Worst Fit

The **Worst Fit** memory allocation algorithm will search from the beginning of a full memory block and find the block that is largest in difference to the request actually being made - hence it being named 'worst fit', and takes  $O(n)$  time per search. Due to the requirement from searching the entire memory space each time a request is made, its speed during operation is the same speed as that of Best Fit.<sup>1</sup> However, its major difference to the Best Fit algorithm is that rather than trying to minimise 'wasted' space, i.e. maximise small, fragmented blocks, it does the reverse - in place of the theoretical 'small, unusable' blocks created by Best Fit, Worst Fit will instead have the largest block remaining possible, leaving them available for later processes that make larger requests for available memory.

## 1.5 Buddy System

The **Buddy System** memory allocation algorithm differs from the above in that it neither purely uses fixed or dynamic partitioning schemes, but rather, a sort of combination of both. This is due to its aim to circumvent the inherent negative aspects of both - to avoid limitation of the number of processes and potential inefficient use of memory of fixed schemes, and the constant need for compaction and its computational cost of dynamic allocation schemes.

The buddy system assumes that the total size of available memory is  $2^U$ , where  $U$  is a positive integer. To obtain a block of memory to serve a program's request, the smallest block that is greater than or equal to the request

---

<sup>1</sup>D. Samanta. 'Classic Data Structures' 2004. p. 76

size that is also  $2^k$  where  $k$  is a positive integer is allocated. This block is retrieved by taking the full  $2^U$  block, and spawning two children of size  $2^{U-1}$ , each of which spawn children of  $2^{U-2}$ , etc...until the correctly size block is found, at which point it is given to the program requesting it. It is apparent that this represents a binary tree - which is one of the data structures in which the Buddy System can be represented by. Hence, the complexity of allocation and freeing of memory is  $\log(n)$ , a considerable increase in speed over the previous dynamic allocation methods detailed.

In its unadulterated form, the reader would be correct in observing one critical flaw with the buddy system - that it further exaggerates the problem of internal fragmentation, arguably beyond that of any reasonably designed fixed partitioning scheme - if we were to start with a 4096MB block of memory, and a program requests 2049MB, that would render the entire block in use, leaving 2047MB empty, yet unusable. This can be remedied through the use of slab allocation - a concept beyond the scope of the course.

## 1.6 Algorithm Selected

The algorithm that was selected for implementation in the assignment was the **Buddy System**. The reasons for this were both its superior speed when allocating, in comparison to the previous dynamic partitioning schemes, and eliminating the issues of external fragmentation, although it instead has issues of internal fragmentation - which can be fixed by other means. In the context of the assignment, however, this fact can be overlooked (or, more accurately, slab allocation does not need to be implemented) - and it is enough to identify that a solution exists for it. The other major advantage of choosing Buddy is that it is computationally cheaper to free memory - and seeing as the assignment specification says up to 1000 jobs can be in a particular instance, the constant freeing of jobs would impact performance (although not in a way that the user would notice given the nature of the application and the way jobs are 'assigned').

Upon further investigation, it can also be seen that many real-world systems even today use the Buddy System at least in part, in their memory allocation schemes. For example, the open source library *jemalloc* uses the buddy system<sup>2</sup>, and is used in very well known projects today, such as the Firefox browser for managing fragmentation during its use on Windows platforms<sup>3</sup>, as well as the FreeBSD OS.

---

<sup>2</sup>[people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf](http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf)

<sup>3</sup><https://github.com/jemalloc/jemalloc/wiki/History>

## 2 Queuing, Dispatching, Memory Allocation, and Resource Allocation Structures

All the structures that were used in the implementation of the host dispatcher, including both the main program and the modules which it relied on, comprise of *structs*, *singly linked lists* which were used as queues in all instances where they appeared, *arrays*, and *binary trees*.

### 2.1 Queueing and dispatching processes

*C structs* were used for a Process Control Block(*Pcb*) which held the data for processes, containing their process ID, arguments (for the default ‘process’ process that simply prints incremental ‘ticks’ at a rate of 1/sec), arrival time, priority, remainingcputime, (psuedo) memory required in MB, a C struct containing resource requirements, status, a pointer to a C struct containing memory block information, and a pointer to the next *Pcb*.

To actually queue and dispatch these processes (represented using *Pcb*’s holding process info), singly linked lists were used, as we only ever needed to take *Pcb*’s from the front of queues, never in the middle of one or from the end. A Realtime Queue linked list was used to hold real time processes, an array of Feedback Queues were used to hold the feedback queues, with its indices offset by one compared to each job’s actual priority (priority 1 jobs were in the feedback queue at index 0 in the array, priority 2 at index 1, priority 3 at index 2).

### 2.2 Allocating memory

As mentioned in *Queueing and dispatching processes*, C structs were used to hold information pertaining to memory blocks (Memory Allocation Blocks, or Mabs) - i.e., their offset in main memory, the block’s size, amount allocated (though this does not include memory allocated in any children blocks). Because the Buddy System was used for managing memory, a binary tree was the data structure of choice to maintain the available memory - and hence, each Mab also contains a pointer to its pointer Mab, as well as its left and right child Mabs, where they existed. Two binary trees were maintained - one for user jobs, and one for realtime jobs.

### 2.3 Allocating resources

Once again, as mentioned in *Queueing and dispatching processes*, C structs were used to hold information regarding required resources for a particular *Pcb* - i.e., the printers, scanners, modems, and cds it needed. While each *Pcb* contains a resource struct to keep track of the number of resources required, one more Resource struct existed in the main program to keep track of the available resources in the overall system at any one time.

### 2.4 Appropriateness

Due to C being a non-object oriented language, these were the best structures to use given the problem at hand. Queues are explicitly stated in the assignment spec, and hence queues were needed - implemented as singly linked lists. The only way to hold blocks of information together in C is using structs, hence their use for multiple parts of the program. The very nature of the Buddy System points towards it using a binary tree, which was why that was also used - and many sources verify this too, not the least of which includes the Art of Programming Volume 1.<sup>4</sup>

Some alternatives do exist to the structures used. On a more detailed level, while the implementation that this Design Document described used nested structs, all the information from each struct could have been instead all

---

<sup>4</sup>D. Knuth. ‘The Art Of Programming: Volume 1 Fundamental Algorithms, Third Edition’ 2013. p. 650 (ebook Edition)

compressed into the one struct - in this way, memory would have been allocated marginally more contiguously, but this difference would have been so minor that it would be more advantageous for the sake of code clarity to separate information into appropriate C structs.

On a more fundamental level, the problem at hand would have lended itself nicely to the use of classes in C++ as well - something which could have been emulated in a bastardised form of C, by containing the functions relating to the purpose of each struct with the use of function pointers. However, this goes against common usage of C, as well as arguably violating using 'straight' C, as the assignment required.

## 3 Program Structure

### 3.1 Overall Program Structure

The overall program structure of the Host Dispatcher utilises three modules - that of the Process Control Block, the Memory Allocation Block, and Resource block modules. This results in the code base in its entirety representing a modular, logical structure, accurately breaking down the problem into its constituents in terms of each individual module. By structuring the code around the inherent structure of the problem itself, it makes it easier to understand at first glance, as well as to modify when parts of the host dispatcher (or program in general) needs to be modified or extended. This may not be as apparent when looking at this assignment in isolation, but for argument's sake, if there were subsequent assignments where we received other students' code and were required to perform major extensions on it, well-implemented modularity would make it a much less daunting task. As a better (and more real-world relevant) example, perhaps a much larger program needs to incorporate this relatively miniscule project into it - and if it needed to use individual elements of it, the programmer could look directly to the appropriate module, rather than digging through a single, large file, which would likely be a monolithic, unorganised mess.

The main program contains several key elements:

- An **inputqueue**, which takes in all jobs from an input text file
- A **userjobqueue**, which holds all user jobs from the inputqueue
- An *array of pointers* to **feedback queues**, containing jobs from the userjobqueue that have been supplied the sufficient amount of memory and resources they require
- A **realtimebuffer**, which contains all real time jobs from the inputqueue
- A **realtimequeue**, which contains jobs from the realtimebuffer that have been given enough memory to run

And without getting into any code or even too deep into specifics, the general flow of the program is as follows (note that the following list is not intended to act as instructions to follow to be able to implement the problem at hand to completion and perfection - but rather, to get a concise idea of how the problem was approached - hence, it is missing implementation specific details such as reading in files, specific error handling, actual flow logic, etc.). Steps 2 and 3 represent the Real Time Dispatcher module, while steps 4 and 5 represent the Feedback Dispatcher module.

1. Place all jobs from the inputqueue into the userjobqueue and realtimebuffers respectively
2. Allocate memory to jobs at the front of the realtimebuffer as long as it is available
3. Run all jobs in the realtimequeue from beginning to end until completion (hence skipping the following steps until such time is reached)
4. Allocate memory and resources to jobs in the userjobqueue as they become available, and enter it in the appropriate feedback queue (array) based on the specified priority
5. Run the process at the front of the highest priority feedback queue for one quanta, before placing it at the end of the next lower priority feedback queue

The following subsections will detail each individual module and its major functions, and by doing so the way each one ties into the functionality of the program as a whole will become apparent. Note that trivial helper functions have been excluded, as they provide no useful insight into how the program functions, even to someone looking to utilise the program as a whole needing to understand roughly how everything operates and weaves together. Unlike the previous paragraph, however, considerable detail will be given with regards to each modules' functions, as they are the foundation of the main program.

## 3.2 Process Control Block (pcb) Module

### **startPcb**

This function takes a Pcb pointer and checks the process' status - it will then either start an unstarted process using C's *fork* and *exec* functions, or continue a paused one with a *SIGCONT* signal- and this is done by checking the process ID (which is stored in a Pcb's struct) - a zero ID indicates a new process, while a non-zero one indicates it already exists but has simply been paused by a *SIGTSTP* signal. The function's return value indicates whether it carried out its operation successfully - a valid pointer indicates success, while a *NULL* pointer means that the start/restart operation it attempted failed. This function gets called whenever a process is first being executed from a feedback queue, or being resume from a paused state. As such, there is no 'resumePcb' function, as the reader may have observed.

### **suspendPcb**

This function takes a Pcb pointer and sends the process it stores a *SIGTSTP* signal, causing it to pause. The function will similarly return a *NULL* pointer if the signal fails. This function is called only on jobs in feedback queues, each time it has executed for one quanta.

### **terminatePcb**

This function takes a Pcb pointer and sends the process it stores a *SIGINT* signal, causing it to die. It is only called when a program's remainingcputime has reached zero.

### **enqPcb**

This function takes a pointer to the head of a queue (specifically, a Pcb), and a Pcb pointer to the process block that is to be queued. The head of the existing queue will be stored in a temporary variable, and then modified to point at the back of the queue, allowing the last item in the queue to then be pointed at the target Pcb to be enqueued. The *head* of this new queue is then returned, or in the case where a queue is yet to be created, simply the *process* to be enqueued is returned instead - as it is the head of this newly created queue. This function is called whenever Pcb's need to be enqueued - whether it is into the userjobqueue, from this to the realtimebuffer/userjobqueue, to the realtimequeue/feedbackqueues respectively, or from one feedbackqueue to another.

### **deqPcb**

This function takes a double pointer to the head of a queue (specifically, a Pcb), and takes the frontmost item off the queue by assigning a temporary pointer to the current head, and then pointing the head to the next item (in this case, the next of the temporary Pcb pointer). Hence, the new head after dequeuing will be returned if successful, or *NULL* returned if unsuccessful, when a queue is already empty. With the exception of when the inputqueue is being loaded with data from an external text file, deqPcb will always be called in pairs with enqPcb - a Pcb is only enqueued onto one queue after being dequeued from another.

### **checkRsrcs**

This function takes a pointer to a Pcb and checks whether it requests valid resources. The role of this function is twofold, as it checks for both realtime and user jobs. It returns true for the former only if no resources are requested, whereas it returns true for the latter so long as the requested amount doesn't exceed the system's total available resources (not simply the amount available at the time - that allocation is handled in the main program). This function is called whenever a process is trying to obtain resources in order to be queued to run.

### **checkMem**

This function takes a pointer to a Pcb and checks whether it requests valid memory. The role of this function is similarly twofold, as it checks for both realtime and user jobs. It returns true for the former when memory requested doesn't exceed 64mb, whereas it returns true for the latter so long as it doesn't return more than half the available size. This seemingly abnormal restriction is because of the naive implementation of the Buddy System,



and reserved memory for realtime jobs - the largest available block for a system with 1024MB memory using this implementation of the Buddy System is only 512MB. This function is called whenever processes are trying to reserve a block of memory in order to be queued to run.

At this point, multiple justifications are in order regarding some of the design choices made above.

- The check functions do not simultaneously allocate memory/resources if they are available. This is necessary, as both resources and memory must be available at any point in time before they are allocated - if each pre-allocates them within their isolated functions, we will end up with cases where one allocates memory/resources and the other doesn't because it wasn't able to - resulting in either a bug, or extra code to do otherwise unnecessary freeing if both conditions are satisfied at once
- This point must may have occurred to the reader as soon as they happened upon the check functions - why are resource and memory checks not in the resource and memory modules? This is justified and even preferred, because first of all - they should take pointers to Pcb's and not Rsrc/Mab blocks specifically, as the programmer using these interfaces should not have to worry about such specifics. Secondly, given the previous assertion, we can't place the functions in the memory/resource modules as circular dependencies would arise as a result

### **3.3 Memory Allocation Block (mab) Module**

### **3.4 Resource Block (rsrc) Module**

## 4 Discussion - Use of Multilevel Dispatching Scheme