COMP3520 Assignment 2

Daniel Collis — 430133523

1 Memory Allocation Algorithms

1.1 First Fit

The First-First memory allocation algorithm is one of the most simplest in existence. It creates and stores a linked list of nodes (representing blocks in memory). When allocating memory with the First-Fit algorithm, the allocation algorithm finds the *first* block which can fit the size of the memory requested. If necessary (when there is a lot of extra memory space), it will then split this block up into two separate blocks, the left half being the size of memory requested, the latter half being free memory that can be allocated in the future.

1.2 Best Fit

The Best Fit algorithm is similar to the First-Fit memory allocation algorithm in many regards. However, unlike the First-Fit algorithm, the Best Fit will scan the entire contents of memory and find the *smallest* possible block a requested size of memory can fit into, unlike First-Fit which will just stop at the first block that can fit the requested memory size. It will then split this block up into two halves, the first being the block of memory requested, with the latter half being the remaining free memory left over.

1.3 Next Fit

With the Next Fit algorithm, rather than searching from the beginning of the memory area like in First-Fit, you store the last block of memory that was last allocated. You then start to search from this point on. Next Fit makes the assumption that the block to the right of the previously allocated block will be free. Like the two aforementioned memory allocation algorithms, when a block of a large enough size has been found, it will split the block in to two parts, the first half being the memory block of the requested size, the second half being the free memory that is left over from the split.

1.4 Worst Fit

This algorithm, rather than trying to find the smallest possible free memory area like in Best Fit, it decides to do the opposite – find the LARGEST free memory area. It searches through the entire memory area finding the largest free block, and if the requested size is smaller than the largest block size, then it will allocate the memory to that block (and

also splitting it up, with the first half being the requested block of the requested size, the rest being free memory space).

1.5 Buddy System

The Buddy System uses a binary tree data structure, rather than a simple linked list like the previous four memory allocation algorithm. The way it works, is that at each node, it splits the memory off into two halves. What this essentially means is that the size of the memory blocks are always powers of two. So, when a block of memory is requested, it starts with the highest leaf node in the tree (the higher the level in the tree, the more free space there is), and recursively divides that leaf node into two more leaf nodes, until a block of an appropriate size has been made.

1.6 Comparison of Memory Allocation Algorithms

1.6.1 First Fit

- **Advantages:**
  - Simple to implement
  - Little overhead/does not need to store pointers to specific sections (just scans through the list sequentially)
- **Disadvantages:**
  - Has a tendency to cut a large portion of free blocks into smaller parts, leading to internal fragmentation.
  - Allocates parts at the beginning of memory, which can lead to more fragments at the beginning of the memory area.

1.6.2 Best Fit

- **Advantages:**
  - By allocating the memory to the smallest available block, it tries to avoid internal fragmentation.
- **Disadvantages:**
  - Can be slower/take more time to find a free memory block than First Fit.

1.6.3 Next Fit

- **Advantages:**
  - Simple to implement.
  - Avoids lot's of internal fragmentation occurring at the beginning of the memory area, unlike First First.

- **Disadvantages:**
  - Requires storing a pointer to the last allocated block of memory.
  - Has a tendency to cut a large portion of free blocks into smaller parts, leading to internal fragmentation.

1.6.4 Worst Fit

- **Advantages:**
  - Minimises internal fragmentation.
- **Disadvantages:**
  - Tends to eliminate all large blocks of memory, and as a result, requests for large blocks of processes cannot be met.

1.6.5 Buddy System

- **Advantages:**
  - Has very little external fragmentation, and can easily merge blocks of memory together with very little overhead.
- **Disadvantages:**
  - Blocks of memory must always be powers of 2. As a result, there could be a lot of left over unused space in the block, thus leading to internal fragmentation.
  - A lot more difficult to implement than the aforementioned memory allocation algorithms.

1.7 Chosen Algorithm

The algorithm I chose was the First-Fit algorithm, because it is very simple to implement, and has minimal overhead when allocating, merging, and freeing memory. It is most appropriate for this assignment, as we are supplying a very little amount of processes to the scheduler, and more advanced systems (e.g. buddy) would be too complex and cause a convoluted assignment.

2 Structures Used By Dispatcher

The HOST Dispatcher we had to implement utilises one memory structure, titled `Pcb` (which stands for Process Control Block). It stores information about a process. It contains:

- the process id (`pid`) of the process,
- the arguments (`args[]`) which contain the program name as the first argument, and any additional program arguments,

- the arrival time of the process, which is after how many quantums since the dispatcher started, should the process be executed,
- the priority of the process, which details whether the process should be executed before any lower priority processes,
- the amount of quantums/CPU cycles there is left until the process is finished (`remainingcputime`),
- the size of the memory block the process has requested (`mbytes`),
- the resources (`req`) that the process has requested to use (e.g. printers, scanners, modems, cds),
- the status of the process, i.e, is it currently running, is it suspended, etc?
- the next process in the linked-list.

The dispatcher also contains Mab structure, which details a specific memory block that has been allocated by the First-Fit algorithm. It contains:

- the offset, that is, where the memory is allocated relative to the start of the memory area,
- the size of the memory block,
- whether this block is free, or has been allocated and in use by a process,
- pointers to previous and next memory blocks (creating a doubly linked list).

Aside from memory management, the dispatches also manages resources, which are represented by the Rsrc structure, which contains the following attributes:

- the number of printers a process has requested to reserve for it,
- the number of scanners a process has requested be reserved to it,
- the number of modems a process has requested to reserve for it, and,
- the number of CDs a process has requested be available for it.

3 Description and Justification of Program Structure

3.1 hostd

The hostd.c file is the primary application file/module. It is where the `main()` function is located. It contains two functions of note, `queues_full()` and `main()`.

- `queues_full` - returns the index of the highest priority feedback queue that has elements in it. -1 otherwise.

- `main` - executes the HOST dispatcher of the assignment. It reads input from the file, creates the input queue, then later assigns those elements to various queues (whether it is a real time process, or a user mode process) throughout execution of the program. It terminates processes if they no longer need to run, otherwise suspends them and lets other higher priority processes take over (if any). It implements the Full HOST Dispatcher algorithm detailed on the assignment specification page word for word.

3.2 pcb

This file contains all of the process management functions/methods. It contains the following functions of note:

- `startPcb` - starts a process for the first time, or resumes a previously suspended process
- `suspendPcb` - pauses a currently running process
- `terminatePcb` - stops a currently running process from executing any further

3.3 mab

This file contains all of the memory management code. It contains the following functions of note:

- `memChk` - checks if there is a free memory block which can fit `size` megabytes into it. This uses the first-fit memory allocation algorithm to determine the next free block.
- `memAlloc` - allocates a new block of memory
- `memMerge` - merges two adjacent blocks of memory, only if they are free (and therefore unallocated)
- `memSplit` - splits a block of unallocated memory into two section, one of `size`, the rest is the remaining free size.

3.4 rsrc

This module manages the resources used by the processes controlled by the dispatches. It contains the following noteworthy functions:

- `rsrcChk` - checks if the requested resources are available
- `rsrcAlloc` - allocates the requested resources, if it can
- `rsrcFree` - frees the supplied resources.

3.5 Justification

This structure is incredibly modularised. As a result, it makes it easy to extend, modify, and maintain the program in the long run. For example, if one wanted to, one could completely swap out the memory management file with another one (provided it adheres to the same interface), and as a result, the HOST dispatcher in `hostd` will then be immediately updated to reflect this new memory management structure. Similarly, should one want to add more resource types, it is quite easy to do so, by simply modifying the structure `Rsrc`.

4 Comparison of Implemented Dispatching Scheme

The HOST dispatcher implemented in this assignment is incredibly simple, compared to what real-world operating systems use today, especially modern ones with multi-tasking capabilities. This dispatcher system only supports four different priorities, whereas on Linux, for example, there are over one hundred different process priorities, and many different realtime priorities too. It was designed to provide a general and introductory overview on how dispatchers may work and how one might implement them. This dispatcher was also limited to a small fixed amount of memory, 1024mb, with only 64mb being allocated for real time processes, and 960mb allocated for userspace processes.

In modern operating systems, the process scheduler will typically pause a process when it has been blocked (for example, on an I/O device such as a hard disk which is significantly slower than a CPU cache), and will resume it when appropriate. Some operating systems also tend to favour foreground processes rather than background, daemon processes (there is an option in Windows that can control this behaviour), which may improve speed for a desktop user. In this regard, modern operating system dispatchers are far more intelligent than this simple little HOST dispatcher that we had to implement.

The memory allocation techniques are also far more advanced that what I had implemented. No real world operating system (aside from maybe a few small niche systems) use the First Fit memory allocation algorithm. Linux, for example, uses the buddy system. However, it uses a much more advanced version with serious mathematics and coding involved that significantly lower the chances of internal fragmentation occurring (a drawback of the buddy system if you read above).

In addition to this, modern system dispatchers tend to account for multiple cores and processes, being able to schedule and run two or more processes/threads in parallel, and manage them successfully and intelligently without any deadlocks or other parallel processing problems ocurring.

Aside from these, the HOST dispatcher has another major drawback: it assumes that all the resources will be available for a process prior to and during its execution. In the real world, operating systems would only supply the resources on demand. And if a resource was not available, it would either return an error code, block the requesting process, or put the request into some sort of queue and execute the command later (e.g. in the case of printing and print spooling).

4.1 Improvements to the HOST Dispatcher

A myriad of improvements could be made to the HOST dispatcher, which could significantly make it more analagous to real world examples. Things such as:

- Support for parallel computing (multiple cores and multiple processes),
- Improve the memory allocation scheme, implementing a buddy system with hacks to reduce the chances of internal fragmentation occurring,
- Provide resources on demand, rather than reserving them prior to a process executing and freeing them at the end,
- Support a much larger memory range, and a custom dynamic memory allocation and expansion regime.