COMP3520 Assignment 2
HOST Dispatcher Design Document

Tianyu Pu
310182212

Semester 1, 2012

1 Introduction

In this report we shall examine the Hypothetical Operating System Testbed (HOST) Dispatcher, starting wih a discussion of memory allocation algorithms as well as data structures for the memory and resources, the overall structure of the dispatcher, and concluding with a discussion and comparison of the dispatching schemes of HOST and real operating systems.

2 Memory allocation

For the purposes of the assignment, the memory used by the processes is a contiguous block of memory assigned to that process for the entire lifetime of that process. The system only has 1024 megabytes (MB) of memory available for processes, and 64 MB must always be available for real-time processes – this leaves 960 MB available for all active user jobs.

In addition, there is no virtual memory support, nor is the system paged.

Within these constraints, a number of variable memory partitioning algorithms are possible. The most commonly used are described below.

2.1 Overview of memory allocation algorithms

The need for memory management becomes apparent in a multiprogramming system such as the HOST dispatcher. This is because in addition to a "system" part of memory, the "user" part must also be divided to allow for multiple processes. In order to implement this, we examine some of the ways in which memory can be dynamically partitioned. Dynamic partitioning (i.e. allocating to a process exactly how much memory it needs) was considered over fixed partitioning (i.e. allocating a fixed block of memory to a process which may be more than it needs, effectively wasting the extra memory) because it more efficiently utilised the limited memory resources available in the HOST dispatcher.

As stated earlier, dynamic partitioning involves allocating to a process exactly how much memory it requires. Over time, as more and more processes are created and more memory is allocated and recycled, we end up with many small holes in memory, a phenomenon known as *external fragmentation.* It then follows that if we carefully choose which section of memory to allocate to a process we can reduce the number of these so-called fragments. With this context we will consider three different ways of placing these partitions, best-fit, first-fit, and next-fit – these are called placement algorithms. In addition, the buddy system will also be described.

2.1.1 Placement algorithms – best-fit, first-fit, next-fit

These three algorithms are all limited to selecting from the free blocks of memory that are equal or larger than the memory that the process about to be run needs.

Best-fit selects the free block that is as close to the required size as possible. *First-fit* starts from the beginning of the block and selects the first free block that is large enough (regardless of how large or small the difference in size is), and *next-fit* is like first-fit except that it scans from the place of the last allocation instead of at the beginning every time.

2.1.2 Buddy system

The buddy system is an attempt to compromise between the fixed and dynamic partitioning schemes. Firstly, the size of the smallest possible memory block that can be allocated is determined. (This can depend on actual hardware limitations or on a balance between reducing memory wastage and excessive maintenance overhead). Then, the entire space available for allocation is treated as a single block of size 2^N. If a request of size s is less than 2^N and greater than 2^{N-1} (half of the block), then the entire block is allocated. If not, the block is split into two "buddies" of size 2^{N-1} and the request is allocated to one of these buddies. Now, if this request is less than the size of the block 2^{N-1} but greater than 2^{N-2} (i.e., half of one of the buddy blocks), the process is allocated to one of the buddies. Otherwise, similar to the previous allocation, one of the buddies is split in half. This goes on until the process is allocated.

2.2 Justification of final choice

The final memory allocation system used in the HOST dispatcher is the first-fit placement algorithm. There are a number of reasons why this algorithm was chosen over the others discussed above.

Of course, out of the three placement algorithm choices that are possible, the one that is best will depend on the size and swapping sequence of processes. However, Stallings [2012] gathers and presents some general comments that have been made about these algorithms: first-fit tends to be the best and fastest; next-fit performs slightly worse as it tends to leave many fragments at the end of the block of free space, requiring more frequent merging of adjacent free blocks (*compaction*) that incur a lot of overhead; and best-fit is usually the worst performer, as it leaves behind many blocks too small to satisfy subsequent allocation requests due to the fact that it tries to guarantee that the remaining free space after allocation is as small as possible (and hence requiring a great deal of memory compaction).

The buddy system has been found to be a reasonable compromise to try and overcome the weaknesses of the fixed as well as dynamic schemes of memory alloation. However, for the purposes of this assignment, the first-fit memory allocation scheme has been selected for its performance and simplicity of implementation and maintenance.

3 Data structures

3.1 Queuing and dispatching

The process queues are implemented using a singly-linked list data structure. There are separate queues for the input queue (when jobs are first read into the program), the user job queue, the real-time queue and the round robin feedback queues. In all queues, when a job is queued, it is added to the end of the linked list. When a job is dispatched from a particular queue, it is released from the head of the queue. This makes dispatching jobs very efficient (as only the head is examined), but enqueuing less efficient (as we need to traverse from the head to find the end of the list). The linked list implementation is also an intuitive representation of the underlying queue abstract data type, and is relatively easy to understand and maintain, without losing too much efficiency.

3.2 Allocating resources

The memory of the HOST dispatcher is represented as a doubly-linked list, with each list element a block of memory (that may be allocated or free). Initially, the memory begins as a one-element linked list, and then is split and merged as processes require memory and then release it when they terminate. A doubly- linked list was used because the extra link makes it much easier to perform compaction of free memory – to check if adjacent blocks are free, one only needs to check the previous and next links (if it were singly linked, to find the previous block another traversal through the list would have been required, decreasing the efficiency). Each process gets a block of memory allocated to them before they can run.

Resources are represented using a C structure. Because their allocation is discrete rather than continuous (for example, you can only allocate 1 or 2 printers, not half a printer, whereas you can allocate any amount of memory that fits within the available space of the operating system), a “master” resource monitor keeps the counts of available resources (printers, scanners, modems and CD drives) during the execution of the HOST dispatcher. The resources a process needs is stored when the process is read from input, but marked as unallocated. Just before the process is sent to a non-realtime job queue (since realtime jobs do not use resources), the process’s resource requirements are checked against the available counts in the main resource monitor. This is a simple comparison of the four fields of the structure and is very efficient. All allocations and deallocations simply update the resource counts in the main resource monitor structures. This provides a central way of keeping track of resources counts while also being able to track the resources that each process needs.

4 Program structure

4.1 Overview

The HOST dispatcher consists of several different sub-functions that work together to run the whole program. These can be broken down into the main program, and separate functions for processes, memory allocation and resource tracking.

4.2 Dispatcher

The function of the dispatcher is to retrieve the jobs and run them based on the status of the various queues. It also maintains the memory and the resources. It does this by calling functions contained in the other modules for process creation, termination, suspension, memory allocation and deallocation, as well as resource management. As a result, the dispatcher contains only the high-level control flow of the dispatcher – the details of the management of processes, memory and resources are delegated to the other modules (described below).

4.3 Processes

Process management is implemented in the `pcb.c` file, which includes the following functions:

- `struct pcb *startpcb(struct pcb *process)` takes a process block and attempts to start the process. It prints the various characteristics of the process such as its process ID, priority, remaining CPU time, memory offset location and size, status and resources allocated. If the process failed to start (the call to `execvp()` or `fork()` failed), then NULL is returned. Else, the process is started and returned.

- `struct pcb *terminatepcb(struct pcb *process)` takes a process block and uses `kill()` to send the interrupt signal to the process. It then waits for the process to terminate before returning.

- `struct pcb *suspendpcb(struct pcb *process)` takes a process and sends it the terminal stop signal (the catchable version of SIGSTOP) to suspend it. It then waits for the process to suspend before returning.

- `struct pcb *restartpcb(struct pcb *process)` takes a process and attempts to restart the process by sending it the SIGCONT signal, returning NULL if unsuccessful.

- `struct pcb *createnullpcb(void)` creates a new process control block and returns it.
- `struct pcb *enqpcb(struct pcb **head, struct pcb *process)` takes the pointer to the head of the process queue as well as the desired process, and enqueues the process by traversing through the linked list queue from the head.
- `struct pcb *deqpcb(struct pcb **head)` takes the pointer to the head of the job queue and simply returns the process at the head and updates the new head of the queue.
- `void printq(struct pcb **head)` is a helper function for printing out the contents of a queue, given a pointer to the head of that queue.

4.4 Memory

Memory management is implemented in the `mab.c` file, which includes the following functions:

- `struct mab *memchk(struct mab *m, int size)` takes a memory block and traverses the memory block list to find a block of at least *size* large. If it finds the block, it returns it, otherwise it returns NULL.
- `struct mab *memalloc(struct mab *m, int size)` uses memchk() to find the available block (if any), and then calls memsplit() on that block. The behaviour of memsplit() will be described in more detail below.
- `struct mab *memfree(struct mab *m)` takes a memory block to free as an argument, then calls memerge() on that block.
- `struct mab *memmerge(struct mab *m)` takes a memory block and attempts to merge additional free blocks to its left. It takes the result of the left merge and performs a right merge, and then returns the result.
- `struct mab *mergeleft(struct mab *m)` takes a memory block and traverses to its left to find additional adjacent free blocks, and returns the resulting merged block.
- `struct mab *mergeright(struct mab *m)` takes a memory block and merges to its right. Like mergeleft(), it also returns the resulting merged block.
- `struct mab *memsplit(struct mab *m, int size)` splits the memory block given if its size is greater than the desired size, and creates a new memory block. If the size of the block is just enough for the required process, then there is no splitting and the block is simply marked as being allocated.
- `void printmem(struct mab *head)` takes the head of the memory doubly-linked list and prints out the various blocks existing in memory. A helper function and used for debugging purposes only.

4.5 Resources

Resource management is very similar to the memory management described above. The functions are implemented in `rsrc.c`:

- `struct rsrcb *rsrcchk(struct rsrcb *host, struct rsrcb *r)` takes a resource monitor and a resource, and compares the fields of both. If there are enough of each resource in the monitor for the particular set of resources required, then return the resource (indicating true), else return NULL (indicating false).
- `struct rsrcb *rsrcalloc(struct rsrcb *host, struct rsrcb *r)` takes a resource monitor and a resources, and allocates the resource by substracting the resources counts from the main resource monitor.
- `void rsrcfree(struct rsrcb *host, struct rsrcb *r)` dellocates the resources by adding the counts back to the main resource monitor. Also frees the memory corresponding to the particular resource request.

5 Dispatching schemes

5.1 Dispatching scheme of HOST

The dispatching scheme of the HOST dispatcher consists of two types of dispatchers:

- a first-come-first-served (FCFS) queue for real-time processes, and
- three round robin feedback dispatchers.

In this scheme, real-time processes are run until completion without interruption, even if there are multiple user jobs ready to be run. All real-time jobs in the queue are run until the queue is empty. If there are no real-time processes to run, then the job with the highest priority in the feedback queues is run for the length of a quantum (defined to be 1 in the current implementation), before being suspended. After each quantum, the real-time queue is checked for any additional processes that have been added: if there are new processes, then those are run first, otherwise the highest priority non-realtime process is run for another quantum. Its priority is decreased (if possible) and placed on the appropriate round robin queue.

Such a scheme allows high priority processes to be run before all other processes. This is particularly important for jobs that have a strict deadline and are critical to the system. Real-time processes also do not decrease in priority while they are running, ensuring that they pre-empt all other processes in the HOST dispatcher. If no real-time processes are currently running, then allowing the non-realtime jobs to run for only a small quantum ensures that other processes get executed without too much of a delay in running any real-time processes that arrive while another process is executing.

One of the biggest shortcomings in this scheme is its scheduling of real-time processes. Because they are executed in the order that they arrive, and only one may be run at a time, there is no way for the scheduler to know if some real-time processes are more urgent than others, and some can potentially go over the deadline if they have to wait for a long real-time process before it to finish. Some ways have been come up with in order to solve this problem in current operating systems that are used today, and they will be discussed in the next section.

In addition, as the number of non-realtime processes increases, the scheduler alternates between each one for one second. Once all of these processes have been decreased to the lowest priority, each process will run in order until they complete. This makes sense to ensure that each process gets an equal opportunity to run and to complete, however there is no way of distinguishing between processes that have run for a while which could be a problem if certain non-realtime jobs need to run more than others. The problem with round robin scheduling is its lack of throughput when too-small quantum sizes are used. There is also no opportunity for multiple processors and parallelisation.

5.2 Schemes used by popular operating systems

Here we discuss some of the more commonly used operating systems and how they schedule their real-time and non-realtime processes, with comparisons between them and the HOST dispatcher we have discussed above. The information about these systems have been based off the material in Stallings [2012].

5.2.1 Linux

The scheduler in Linux systems before version 2.4 consisted of a real-time and non-realtime scheduler. The real-time scheduler differed from the HOST dispatcher: it was made up two classes, a FCFS queue and a round-robin system. This is in contrast to the FCFS queue solely used in the HOST dispatcher. This allows the absolutely most urgent processes to be placed in the FCFS queue to complete (generally) without interruptions, while also handling other less pressing processes in the round-robin queues. This ensures that all real-time processes can be executed at a frequency suitable to the process.

The non-realtime scheduling is also different from the HOST dispatcher – the priority of each process is dynamically computed during its execution as a function of its static priority (the priority it starts with, and is specified by the user) and its execution behaviour. Generally, tasks that have spent a lot of time sleeping are given a higher priority, and the scheduler tends to favour I/O-bound tasks over processor-bound tasks. Different processes are also assigned different size timeslices (between 10-200ms). Higher-priority tasks are generally allocated longer timeslices.

For memory allocation, Linux has two main aspects: virtual memory and kernel memory. Virtual memory in Linux is addressed using a three-level page table consisting of a page directory, page middle directory, and a page table. Pages are allocated using the buddy system, which enhances reading in and writing out pages to and from main memory. Pages in memory are replaced using the least frequently used policy, which relies on the assumption that the pages that are the oldest and least used are less likely to be used again (and hence are marked for replacement). The HOST dispatcher does not have any sort of memory replacement functionality – once a process acquires a block of memory, it holds onto that memory until it finishes executing, even while it waits for other processes to execute before it can complete. For kernel memory, Linux uses a scheme called *slab allocation* for small short-term chunks, which essentially is a set of linked lists for each chunk of memory. Chunks may be merged and moved in a similar way to the buddy algorithm.

5.2.2 Windows

The scheduler implemented in the Windows operating system is a pre-emptive one with a system of round-robin scheduling for certain levels as well as dynamic priority variation for other levels. Unlike the systems described so far, threads are the unit of scheduling rather than processes.

Similar to the HOST dispatcher, there are two types (or "classes") of priorities – real-time and variable (likeable to user jobs). Each class is further separated into 16 priority levels, with threads in each classes of varying priority levels. As a result of the pre-emptive scheduler, real-time tasks are given preference over other threads, similar to HOST. In contrast, however, the real-time priority class runs threads of a certain priority level using a round-robin queue. This allows different levels of real-time tasks to be run, unlike the single FCFS queue we implemented for the HOST dispatcher. All real-time threads have a fixed priority for their lifetime. This is not the case for variable (or non-realtime) threads – they begin with an initial priority value and this value may receive a temporary boost during its lifetime. It is worth noting, however, that variable priority class threads are never boosted into the priority levels of real-time threads. Similar to the real-time class, there is a queue for every priority level, however in this instance it is a FCFS queue. As the priority level of a thread in this class can change, it will also move to a different queue based on its most current priority level. Like the Linux scheduling system, this scheme favours interactive I/O-bound threads over processor-bound ones.

Windows uses virtual memory paging and a manager controls how the memory is allocated and the way paging is carried out. On 32-bit systems, the available user address space is almost 2GB, which is divided into pages of a fixed size. When a process is created, it can make use of all of this space. When it is first activated, data structures are assigned to it which managed its set of working memory. The set consists of regions that are available (not currently used by the process), reserved (addresses set aside for a process which cannot be allocated for another purpose), or committed (addresses initialised for the process's use). As sets of active processes grow, Windows recovers memory for the system by removing pages that are less recently used out of the working memory sets.

References

William Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall, Upper Saddle River, New Jersey, USA, 7th edition, 2012.