COMP3520: Host Dispatcher

Memory allocation

The memory allocation blocks (MABs) in the host dispatcher regulate and assign available system memory to processes as requested. The memory assignment implemented by the host dispatcher is one of either static partitioning, such that memory partitions are of fixed (but not necessarily equal) sizes; or dynamic partitioning, such that memory partitions are of variable length and number and when a process is brought into main memory it is allocated exactly as much memory as it requires.

Placement algorithms

In an environment that supports dynamic memory allocation, the memory manager must keep a record of the usage of each allocate-able block of memory. This record could be kept by using almost any data structure that implements linked lists. An obvious implementation is to define a free list of block descriptors, with each descriptor containing a pointer to the next descriptor, a pointer to the block, and the length of the block. The memory manager keeps a free list pointer and inserts entries into the list in some order conducive to its allocation strategy. A number of strategies are used to allocate space to the processes that are competing for memory, and these are discussed below.

Best-fit

The best-fit algorithm allocates to the process the memory allocation block that is closest in size to the request. The best-fit algorithm tends to leave small fragments of memory, resulting in the main memory being littered by memory allocation blocks too small to satisfy allocation requests for other processes. This is a property known as external fragmentation. For this reason, memory compaction is required frequently. Allocating memory using the best-fit algorithm is expensive because it involves searching the entire free list to find the best hole. This problem in addition to the regular need for compaction make best-fit a less desirable strategy for memory allocation.

First-fit

The first-fit algorithm scans the available memory allocation blocks from the beginning and chooses the first available block that is large enough. Using this algorithm, small holes tend to accumulate near the beginning of the free list, making the memory allocator search farther and farther each time.

The first-fit algorithm is not only the simplest but usually better and faster than best-fit and next-fit (Stallings). The first-fit algorithm performs reasonably well because it ensures that all allocations are performed quickly. The first-fit allocation algorithm executes with an average time of $\frac{N}{2}$, and a cost of $O(N)$.

Next-fit

Next fit is a variant of the first-fit strategy. The next-fit algorithm begins to scan memory from the location of the last placement, and chooses the next available memory allocation block that is large enough. The problem of small holes accumulating is solved with next fit algorithm, which starts each search where the last one left off, wrapping around to the beginning when the end of the list is reached. The next-fit algorithm is usually worse than the first-fit algorithm and results in more

Joshua Spence
308216350

frequent memory compaction. This algorithm executes with an average time of $\frac{N}{2}$, and a cost of $O(N)$.

Worst-fit

In the worst-fit algorithm, a process is placed in the largest block of unallocated memory available. The idea is that this placement will create the largest hold after the allocations, thus increasing the possibility that (compared to best fit) another process can use the remaining space. Essentially, the worst fit algorithm keeps big holes around. This algorithm executes with an average time of $O\left(\frac{N}{2}\right)$, if the list is sorted ($O(1)$ to find the hole and $O(N)$ to allocate the memory).

Buddy

The buddy system is designed to make merges fast when blocks are freed. In a buddy system, the permitted memory allocation sizes are powers of two, such that if a process requires memory $2^i < memory \leq 2^{i+1}$, then it is allocated a block of size 2^{i+1}. The memory allocator must thus keep many free lists, one for each permitted size. If the free list for the request memory size is empty, then the allocator splits a block of larger memory into two equal blocks, and continues doing so until a block of the required size can be allocated. The main advantage of the buddy system is that memory coalescence is a cheap operation because the "buddy" of any free block can be calculated from its address. The buddy system, however, does suffer from wasted memory known as internal fragmentation. It is also a lot more difficult to implement than the other algorithms.

In comparison to other simpler techniques such as dynamic allocation, the buddy memory system has little external fragmentation, and has little overhead trying to do compaction of memory. Compaction of memory is performed as follows:

- If memory is to be freed
    - Free the block of memory
    - Look at the neighbouring block
    - If the neighbouring block is also free, then combine the two
    - Repeat this process until either the upper limit is reached (all memory is freed), or until a non-free neighbour block is encountered

This method of freeing memory is rather efficient, as compaction is done relatively quickly, with the maximal number of compactions required equal to $\log_2\left(\frac{u}{l}\right)$ where u is the size of the largest possible allocation block and l is the size of the smallest possible allocation block.

Typically the buddy memory allocation system is implemented with the use of a binary tree to represent used or unused split memory blocks.

Final design choice

The memory allocation policy that I chose to implement in the host dispatcher is the first-fit algorithm. I chose this algorithm over other dynamic partition algorithms because of its performance, being faster and more accurate than next-fit and best-fit algorithms. The first-fit algorithm can quickly allocate memory and does not suffer from internal fragmentation. This algorithm does however suffer from external fragmentation, and over time, small holes accumulate

near the start of the memory list. I accepted this disadvantage on the grounds that the algorithm is both simple to implement and performs relatively well.

I chose the first-fit algorithm over the static partition buddy algorithm because of its relative simplicity to implement and the fact that memory allocations can be performed very quickly. In addition, the first-fit algorithm (unlike the buddy algorithm) does not suffer from internal fragmentation. This means that the full capacity of the systems available memory can be better utilised using the first-fit allocation algorithm.

Dispatcher structures

Queuing and dispatching

The host dispatcher uses structures known as Process Control Blocks (PCBs) to store data relating to the status of a process and other process management information. In a fully functional operating system, the PCB forms a small section of the process image, a larger structure which also includes program instructions and program data areas.

PCBs generally store process identification data, processor state data and process control data. Process identification data includes a unique identifier for the process (the process ID) and often other data such as the parent process identifier, the user identifier and the user group identifier. The process ID is particularly relevant because it is used to cross-reference operating system tables, thus identifying which process is using which I/O device or memory area.

The host dispatcher creates queues of PCBs by referencing other PCBs from within the PCB structure. In this way, a queue of PCBs can be maintained and processes can be pushed and popped from the queue as they are executed, suspended or terminated.

The host dispatcher consists of many queues, each storing processes at different stages throughout the processes' life cycle. An input dispatcher queue is used to store PCBs for processes that are to arrive at the dispatcher at a certain time. Once the process' arrival time is reached, the process is then ready to be executed.

Processes are popped from the input dispatcher queue and pushed onto the user job queue when the processes' arrival time has been reached. Processes are popped from the user job queue when the required memory and resources for that process can be allocated. The allocated process is then pushed onto a feedback queue. In this implementation of the host dispatcher, the dispatcher consists of multiple feedback queues to accommodate processes of each possible priority. When a process is popped from the feedback queue and executed/resume, the priority of the process is decremented before it is queued back to a feedback queue. Processes in the feedback queues are executed on a round robin basis.

The only remaining dispatcher queue to be discussed is the real time queue. The real time queue stores processes with a priority higher than any feedback queue and represents processes that arrive at the host dispatcher at any time instant and must be executed immediately, taking priority over all other non-real time processes. Processes queued on the real time queue are executed on a FIFO (first-in first-out) basis.

Allocating memory

To allocate memory resources to processes, a structure named a Memory Allocation Block (MAB) is defined. This structure forms a member of a linked list of memory, collectively representing the total system memory. The MAB structure defines the offset and size of a block of memory, as well as the necessary pointers to form a linked list data structure. The MAB structure has no knowledge of the process to which it has been allocated, but is aware of whether it is free or allocated (through a boolean flag). It is the responsibility of the PCB to remember which memory it has been allocated and to reference this memory accordingly.

Allocating resources

To allocate system resources, a structure named a Resource Allocation Structure (RAS) is defined. This structure is similar to the memory allocation block previously defined, but a key difference is that a Resource Allocation Structure represents an indivisible element of the system, whereas memory blocks can be split into the required size. The RAS structure defines the type of a resource (eg. printer, scanner, modem or CD), as well as the necessary pointers to form a linked list data structure. The RAS structure also contains a pointer to the PCB of the process that it is allocated. In this way a resource can only be allocated to a single process at any given time and is aware of which process it has been allocated to.

Overall structure and dispatching scheme

The overall structure of my implementation of the host dispatcher is logically separated into modules, mostly representing the data structures previously discussed. There is a PCB module, a MAB module, an RAS module, a main host dispatcher module and a utility module for capturing input data from a CSV file.

The PCB module is responsible for creating, starting, suspending, resuming, terminating and freeing processes using a PCB data structure. The PCB module also contains functions for controlling the queue structure, including operators for enqueueing and dequeueing a PCB from the current queue. Freeing a process also involves freeing any memory and resources allocated to that process.

The MAB module is responsible for checking, allocating and freeing memory using a MAB data structure. The RAS module is similar to the MAB module and provides similar functionality for system resources.

The input module is responsible for parsing an input CSV and constructing a PCB input queue based on the input CSV file.

Such a module design is crucial to the design of the host dispatcher and allows for the dispatcher to be easily extended and altered with design requirements. The PCB data structure design is justified because it contains all of the required crucial properties of a process control block, as well as the required properties to form a dispatcher queue. It is a logical decision to allocate a single memory allocation block to a process control block because the assignment specifications specify that processes must use a contiguous region of memory. If this were not the case then a more sophisticated approach would be required.

Joshua Spence
308216350

However, the host dispatcher does not implement any mechanism by which to prevent two processes from accessing the same memory block. Whilst there is a check performed at the time of memory allocation to ensure that a memory allocation block that is already allocated will not be reallocated, it could be possible in a multithreaded or multiprocessor environment that two processes could be allocated the same memory block and this would inadvertently allow processes to modify the memory contents of another process. A better and safer strategy would be to continuously check, whenever the memory is attempted to be accessed, to ensure that the memory is allocated to the process that is attempting to access it.

The strategy implemented for resources is stronger than the policy implemented for memory, because it allows each resource to be aware of the process to which it has been assigned. This makes it more difficult for a process to interact with a resource belonging to another process. This strategy was implemented because of the one-to-many relationship between resources and processes, unlike the strict one-to-one relationship existing for memory blocks. Care needs to be exercised with this strategy, however, as if a process terminates unexpectedly, it may be possible that the lock on a resource is not released and hence the resource would not be freed and as such would be unusable by other processes.

The memory allocation process could be improved by supporting virtual memory, so that processes that are waiting for resources could be swapped out to disk to allow for a higher throughput of processes. The memory allocation process would also be improved if the host hardware MMU supported paging.