**Week 3 Interactive Assignment: Outline the objectives and functions of memory management in operating systems.**

in computers, effective memory management is extremely important, considering the operating system shares memory space with user processes. To increase performance and optimize CPU utilization, multi-programming, the ability to execute multiple programs simultaneously, is facilitated in many modern computers through concurrent and/or parallel processing (Silberschatz, Galvin, & Gagne, 2014). To achieve successful multi-programming, a memory management system is required in a computer. The objective of the memory management system is to handle allocation, protection, relocation, sharing, logical organization, and physical

organization (Hand, 2010).

The memory management unit (MMU) must be able to map logical addresses to physical addresses (explained further later). Allocation refers to assigning free memory space to a process. Protection, refers to the operating system being protected from users, user processes are protected from one another, and a program is protected from memory overlaps and over-writing (Staff, n.d.). The functionality of a memory management system that allows a program to be moved around within a memory space to create more free blocks of available memory is relocation. Sharing is the ability for different processes to share data in a protected way. Logical and physical organization is how logical and physical address spaces are mapped to one another, and how processes that occupy a large logical address space can be executed with a smaller physical address space.

**Compare and contrast the physical address space with the virtual address space as they relate to different memory mapping techniques in operating systems.**

One reason for having both logical (virtual) and physical address spaces is, when the code is written, the programmer is unaware of where in memory the program will be loaded (Hand, 2010). Thus, logical addresses generated by the CPU when a program begins execution need to be mapped by the MMU into physical address spaces. This process is called address-binding, and can occur during compiling, loading, or execution. If completed during compiling, absolute code is generated and the code may need to be recompiled before running. If done during loading, the physical address was not known at compile time, so the compiler generated relocatable code and final binding does not occur until load time. If completed during execution, the process was likely moved during execution, so binding does not occur until run-time (Silberschatz, Galvin, & Gagne, 2014). The MMU utilizes base and limit registers while mapping the physical address that correspond to a logical (virtual) address. Base and limit registers define the logical address space and determine the range of addresses for a process. Once the base logical address is offset to calculate the corresponding physical address, the limit register is used to calculate the maximum physical address (Silberschatz, Galvin, & Gagne, 2014).

Memory can be allocated in several ways through the MMU. One model for allocation is contiguous allocation. In this model, memory allocation has two partitions, low memory and high memory. The OS resides in low memory, while user processes occupy high memory. Memory can be separated into fixed-size partitions made of contiguous physical address space, and each partition contains only one process. When a partition frees up, a new process is loaded into the address space (Hand, 2010). Extra space inside a fixed partition (internal fragmentation) cannot be used by a new process. Another model of partitioning is variable partitioning. With variable partitioning, the memory space is one large hole, in which, operations are allocated partitions exactly equal to the size of the process. Once multiple processes are added to the space, holes begin to form in the remaining space as processes finish execution and get removed from the address space, and new processes are added (Silberschatz, Galvin, & Gagne, 2014). Dynamic partitioning is then utilized to allocate processes to the remaining physical address space.

With dynamic partitioning, three common algorithms are used in determining where new processes gets loaded into the address space, first fit, best fit, and worst fit. First fit loads a program into an address space as soon as a hole (or vacancy) large enough to fit the process has been found. Best fit is a method that searches until it finds the smallest hole that will fit a program, and then loads the program. Worst fit is a search algorithm that finds the largest memory hole, and then load the program there. While first fit and best fit are better than worst fit in terms of decreasing time and storage utilization, contiguous allocation can still cause problems such as fragmentation (Silberschatz, Galvin, & Gagne, 2014). Fragmentation is the small fragments of memory in between the contiguous address spaces of other processes that become wasted when they are not large enough to load a new process contiguously (Hand, 2010). With a fixed-partitioning memory allocation scheme, the space created by internal fragmentation within a partition gets wasted, because each partition can contain only one process. In a variable-partitioned environment, internal fragmentation can sometimes be reduced because it can be re-allocated to a new process if the process is small enough. In any case, internal fragmentation can be resolved through compaction (reorganizing memory to create larger holes from the smaller fragments), paging, or segmentation (Silberschatz, Galvin, & Gagne, 2014). Paging and segmentation are memory allocation models that aid in resolving the issue of fragmentation due to utilizing non-contiguous memory addresses. Rather than searching for one large block big enough to fit the whole program, the physical memory is divided into smaller fixed-sized blocks called frames, and logical memory is divided into fixed sized blocks called pages (Hand, 2010). A page tables are used to track the pages in virtual memory and the offset values that correspond to a frame in physical memory. This solution makes memory allocation easier, reducing external fragmentation, and keeping the user and system views of memory usage separate. However, there is added overhead for context switching when a page table entry is not presented for the referenced page, and the page table takes up space in main memory, so there can still be internal fragmentation (when a frame is larger than the address space required for a process) (Hand, 2010). Segmentation is like paging except segments of a process can be varying sizes, rather than being a fixed size.