Exam 3, CS – 405, Fall 2020
Iyengar

Please be neat. You may use both sides of the exam sheets.

Describe the concepts of compile time, load time, and run-time address binding. (10)

Compile time address binding is done by the compiler as it generates virtual addresses. It is a static address binding that uses logical addresses and can't be changed. This requires specific interaction with the OS to complete.

Load time address binding is done at load time and converts absolute addresses into relocatable addresses by offsetting each absolute address by whatever the program is offset by in memory. This maintains the benefits of absolute addressing while keeping programs relocatable. It is generally handled by the memory manager of the OS.

Execution time address binding involves using the offsets of variables within compiled code to generate the effective addresses at runtime. This way, variables are also relocatable because they are compiled to only reference memory relatively. Additional math is required to generate the effective address during run-time which does slow the program down.

What are the advantages and disadvantages of Fixed Memory Partitions? (10)

Fixed Memory Partitioning is one of the oldest and simplest partitioning schemes. It involves little overhead and is relatively easy to implement thanks to its simplicity. Unfortunately it means that even the smallest program will still take up the full frame, and this unused free space cannot be used by other programs that do need it, making the scheme quite inefficient in memory use. This also limits the possible size of each program loaded, as well as how many programs can be loaded (and running) at once.

What are the advantages and disadvantages of Variable Partitions? (10)
Variable Partition Partitioning is considerably more complicated but vastly improves memory efficiency over Fixed Partitions. Since memory is allocated only as it is needed, unused space is free for other process to use. This also allows processes to be quite large, and since these frames can be made very small and only pieces of the process need to be loaded in at a time, much more processes can be loaded and run together. This does have considerably higher overhead during runtime as compared to Fixed Memory Partitioning, which is why it is often supplemented with hardware support. When programs finish, there is also the issue of free space 'holes' within memory that can only be used by processes smaller (or that will never load more than) the size of that free space.

What is compaction/defragmentation? When and how often should compaction be done? How does compaction affect binding? (10)
As programs run they unload data no longer necessary, especially when they finish running. This leaves chunks of free space with in memory on both a local (internal) and

memory wide (external) fragmenting. Defragmentation is done to free up any unneeded memory space and combine existing loaded stuff to be closer together. This costs a significant amount of time and cpu processes because address bindings often need to be recalculated relative to where the processes are sitting in memory, but can free up a considerable amount of memory to be used by other processes, and existing loaded processes. Because of the high cost of doing it, it’s not worth doing all the time, but it’s still worth doing quite often. It’s a bit of a balancing act to defragment often enough to maintain peak efficiency within the memory, but not doing it so often that slows everything down too much. It depends largely on how much memory is available, how many processes are trying to run, and how long the compaction takes. Much of this now supplemented by hardware, making the compaction process faster. Together it increases the average efficiency of memory usage at the cost of additional time. If programs are condensed but not moved, binding doesn’t necessarily have to be redone, but if programs are condensed and reorganized within memory (which is usually more effective at freeing up unused memory), then they not only need to be moved and shrunk, but binding must be recalculated.

What is meant by relocatable code? Describe how proper address translation can be done when code is relocated. (5)

Relocatable code, briefly mentioned in the last question, relies heavily on relative addressing. This allows programs to be relocated without an increase in cost of address calculations. A program can be at memory location 10 with a variable at relative 100. This means the variable is at memory location 110. If the process is relocated, which can be a part of memory compaction, it’s still 100 + program location. If the program is moved to location 150, that same variable is now at 250. This increased the time it takes to access a variable because this math must be done when the program is moved or loaded, or this addition must be done each time the variable is requested. Another way to do this is to make relative addresses use the location of the reference instead. If a variable is called at location 600 and the variable resides at location 650, the compiled code will just reference 50. This way it does not need to be recalculated on relocation, but it does need to be recalculated every time it’s run. As such hardware is often used to speed this processes up, and it’s usually considered worth the extra overhead because of the increase in memory efficiency it allows through process relocation.

Describe the concept of paging and how is address translation done when using paging? What is meant by demand paging? (10)

Storage devices (HDDs, SSDs) generally read data chunk (page) by chunk. To avoid having to pull these pages apart, pages are often loaded into memory one at a time. Unfortunately, this can decrease the efficiency of memory usage when a program loaded is smaller than a page, but it increase the efficiency of relocatable code and calculations of address offsets. When these pages are loaded in as non-dynamic in size, the metadata stored to record the relative locations of addresses and programs becomes much easier because it’s simply a multiple of the page size. Dynamic page sizing can make this process more slow and more complicated but tends to increase the efficiency of memory usage. Demand paging only reads in pages from the disk when they are requested, increasing the efficiency of memory usage, but increasing the time required to access data within those pages when they aren't already loaded.

What is the advantage of demand paging? How is demand paging implemented? (10)

Since pages of data are only loaded by request, this means that program data is not loaded into memory unless it is actually needed, leaving memory open and available for other processes. Processes can also be limited to a certain amount of pages loaded at a time, increasing the amount of unloading and loading of pages, but increasing the potential amount of multi-programming and increasing the practical efficiency of memory because pages are only loaded on an as needed basis. This does however require keeping track of how much memory is still available so that a job can still be loaded and run to completion. In our in class examples, a semaphore can be used to keep track of available memory. Each time a process requests a page that isn't loaded, that page is compared to available resources, and loaded if there is enough memory available and enough remaining resources that at least one process can finish. This is done to avoid deadlock between processes.