OPERATING SYSTEMS FOUNDATIONS
MSIT 5170

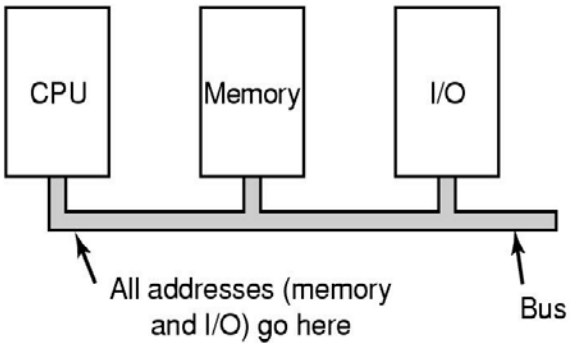
University of Massachusetts Lowell
Department of Computer Science
Spring Semester 2020

Week #6 Lesson

1. The material in this lesson follows the reading assignment for ch 3.1 through ch 3.4 from the Tanenbaum text
2. Please read the text before you read through this lesson.
3. The material in chapter 3 will focus on issues of memory management and address space.
4. We continue to look at system level tools, and I encourage students to experiment with these tools. We will also build some simple applications in the Linux environment to explore some of the dimensions of memory management.

The Need to Manage System Memory:

In our most fundamental view of a Von Neumann style computing system, we envision a processor (CPU), a working memory of some size (RAM), some sort of interface to the outside world (an I/O bridge), and a system bus to tie these basic components together.



Manipulation of data requires that the data be first brought into the CPU (registers) for processing. This is typically done by loading from memory (RAM) or directly from an IO device (like a UART register). Computed results are then generally moved from the CPU (registers) back into memory where they can be accessed either by the CPU again (for further processing) or sent to the outside world through the I/O interface. The machine instructions that a thread will execute to manipulate data must themselves be first moved into memory from the outside world (usually from an executable binary kept somewhere in a file system that is accessed via the I/O interface). Once in memory, the individual machine instructions are brought into the CPU during a fetch cycle, to be executed by the CPU's functional units. We see the CPU during

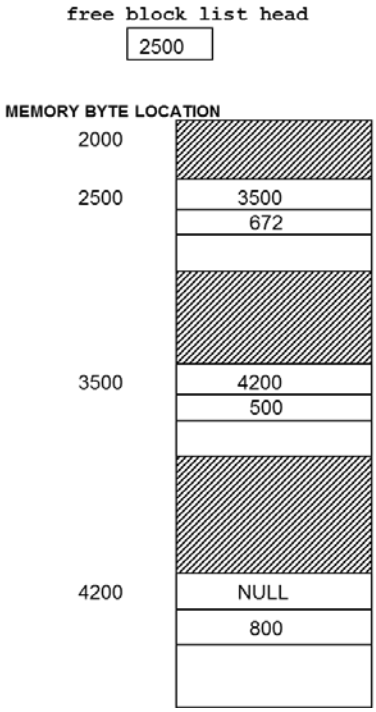
normal execution constantly performing load and store operations to move data objects between its registers and memory (or IO devices), and constantly performing fetch operations to move machine instructions from memory into the CPU instruction interpretation register (IR). Clearly, memory is a central and critical resource, and must be managed carefully in order to extract peak performance from a given system.

An operating system is generally concerned with three basic aspects of memory management:

- **When** to bring something into physical RAM
  - should entire programs be loaded into RAM when we want to run them, or should we only bring in what's demanded at any given time
- **Where** should we place something into physical memory
  - placement is critical to using the available space most efficiently
  - good placement is intended to limit fragmentation
- When and **what should be evicted** to make room for incoming code and data
  - removing things that have been "used up" and will not be demanded again for the life of an application is the goal
  - how can we know what has been "used up" if the owning application is still alive?
  - if we choose poorly, we may remove something that will be needed again soon, and will then have to evict something else to recover something we already had in memory once

Managing Free Space:

Placing objects into memory requires some strategy for knowing what parts of memory are available (free) to load into. Several strategies for managing free memory are discussed in the book, including **bit maps** (which are used extensively to manage free disk space in a file system, but not very often to manage RAM) and a number of variations of **linked list management** (linked lists being frequently used to manage RAM in many simple embedded systems). Embedded linked lists are very space efficient, since their control data (meta data) components are kept in the free (otherwise un-assigned) parts of memory.



In the picture shown above, a region of memory beginning at byte location 2000 is being managed by a linked list implementation. There is a head pointer with a value of 2500, indicating that the memory from 2000 to 2499 is in use, but that there is a block of free memory beginning at 2500 that is 672 bytes long. The linked list header element located at 2500 holds the address of the next free block (3500), along with the size of this free block (672 bytes). The rest of this block is free unused memory. Shaded areas have been allocated (as we mentioned for the space from 2000 to 2499), and the next free block is at 3500. The linked list header element located here identifies this block as being 500 bytes long, and points to the next free block at location 4200. The block at 4200 is the last free block in the managed memory area, spanning some 800 byes (its next pointer of NULL indicates the end of the list).

Using Linked Lists:

There are three basic algorithms used to find a free block in a linked list managed memory area:

- **First Fit**
  - knowing what size is required, a search begins at the head of the list and continues until the first block that is big enough to satisfy the allocation request is found
  - the starting address of this area will be returned to the caller, and unless the block is a perfect fit, the residual space (external fragment) will have a new header built into it and will be linked into the rest of the list
  - for example, if a request for 300 bytes was made from the system shown above:
    - the first free block is used since it is big enough
    - the returned address will be 2500
    - the space at 2500 + 300 = 2800 will have a new header created
    - the next pointer will still be 3500, but size field will now be 672 - 300 = 372
    - the head pointer will have to change to point to this new external fragment at 2800

- **Best Fit**
  - the search begins at the head of the list, but does not stop until it has checked each element on the list to see which element is closest in size to the request
  - the objective is to leave the **smallest external fragment possible**, hoping that this strategy will be more space efficient than first fit (i.e. for a given initial free area, will be able to satisfy more allocation requests before it runs out of memory)
  - using the same example, if a request for 300 bytes was made from the system shown above:
    - the free block containing 500 bytes will be selected, since it will leave the smallest external fragment
    - the returned address will be 3500
    - the space at 3500 + 300 = 3800 will have a new header created
    - the next pointer will still be 4200, but size field will now be 500 - 300 = 200
    - the previous block at 2500 will have its next pointer now set to 3800
- **Worst Fit**
  - the idea here is to leave the **largest external fragment**, assuming its large size will be usable to satisfy subsequent requests
  - as with best fit, the search must examine all elements before a choice can be made, and the hope is that this strategy will be more space efficient than the others (i.e. for a given initial free area, will be able to satisfy more allocation requests before it runs out of memory)
  - using the same example, if a request for 300 bytes was made from the system shown above:
    - the free block containing 800 bytes will be selected, since it will leave the largest external fragment
    - the returned address will be 4200
    - the space at 4200 + 300 = 4500 will have a new header created
    - the next pointer will still be NULL, but size field will now be 800 - 300 = 500
    - the previous block at 3500 will have its next pointer now set to 4500

Managing the list is further complicated by the need to deal with space that is returned by a user when the user has finished with it. Free space must be put back into the list, but it must be **coalesced** if appropriate with free neighbors above it or below it, or both.

Research has shown that **first fit** is, on average, as space efficient as either of the others, but that its average run time is half of the others, so first fit is most often the choice for linked list management.

Linked list mechanisms are appropriate when allocations are allowed for any arbitrary sized chunk of memory extracted from some contiguous range of free memory, in a fashion similar to the way in which the **malloc()** dynamic memory allocator routine works in a **C program** to manage the so-called heap space. This view of a contiguous range of free space, however, would be difficult to maintain if we were operating directly against physical memory. We would have to break up physical memory into varying sized chunks, which would cause a fragmentation nightmare. Since we need to project the flat, contiguous memory model (address space) to a program, and the process it runs in, we need a **low-level strategy** to deal with physical RAM efficiently. This underlying model provided by an operating system is called the **virtual memory model**.

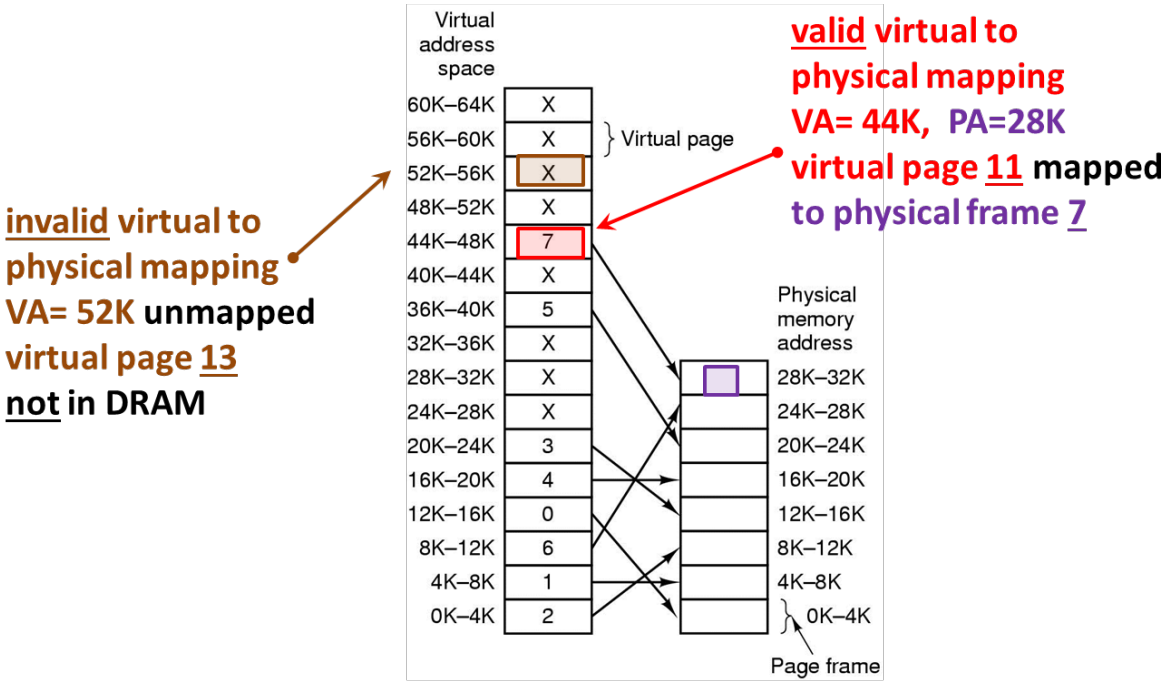
Virtual Memory:

While most contemporary processors view RAM as a contiguous array of byte addressable locations at their lowest level of operation, just about everything except the simplest CPUs support memory management functionality to model RAM as a collection of **fixed sized pages** for higher levels of operation. This paging model is built around a physical memory allocation policy of a fixed sized contiguous chunk of RAM called a **page**. In paged systems, physical RAM is always allocated in this fixed sized page unit.

The Intel x86 family of processors, for example, do not power-up in paged mode, but once they load and begin running an operating system like Windows or Linux, these operating systems quickly enable paged mode, and remain in paged mode as long as the system continues to run. Page sizes vary from processor to processor, and are adjustable in some processors. **The x86 family uses a 4 KiB** (2^{12} = 4096 byte) page size (along with some larger choices like 2 MiB and 1 GiB), while the HP Alpha uses an 8 KiB page and certain ARM processors can vary their page size from 1 KiB to 64 KiB. The obvious question to ask at this point is "how do we satisfy a request from a running thread for say 1 MiB of memory, if the operating system can only allocate 4 KiB at a time". The operating system could try and find enough 4 KiB free pages in contiguous order to make the allocation, but this may be impossible if different groups of pages have been given to other requests and no contiguous run of 1 MiB remains. There may be much more than 1 MiB of memory available, but not in contiguous order.

To solve this problem, operating systems have adopted a high level technique of memory management called **virtual memory management**. As we've seen in chapter 2, each process in a Linux or Windows system is provided a private address space. Now we can see that this address space is not immediately carved out of physical RAM, but is instead, an **abstraction** in the form of a virtual address space. The virtual address space represents a **contiguous region of byte addressable storage** to any thread running in it, but the operating system views it as a collection of virtual pages, any one of which can only be used if and when that virtual page is **connected to a physical RAM page**. Connecting a virtual page to an actual physical RAM page is called a **mapping operation**, and typically occurs to resolve what is known as a **page fault**. Page faults occur when a running thread attempts to reference a virtual address that has no physical mapping. The CPU hardware reacts to a memory reference (to fetch, load or store) that is not physically mapped by raising an **exception**. The exception handler, which is part of the resident operating system, then reacts to this event by determining if the memory reference is to a valid object, and if so, will find a **free physical page,** and place the appropriate content into this page (i.e. the exception handler loads needed code or data from the executable file for example), and then **completes the mapping of the virtual to physical address** to allow a retry of the faulting memory reference instruction. The faulting address will, of course, be a part of some page, and the page fault mechanism will handle these events one page at a time. The newly mapped page will resolve the current fault, and because of the **locality** attributes of programs, will hopefully serve to fulfill other references in near proximity.

The mapping of virtual pages to physical pages is managed for each process using a process specific data structure called a **page table**. The operating system maintains a **discrete page table** for each process in the system, and this table is a key component of the process address space we've previously discussed. The book provides a simple example of a page table mapping a collection of virtual pages to physical pages (page frames are 4 KiB):



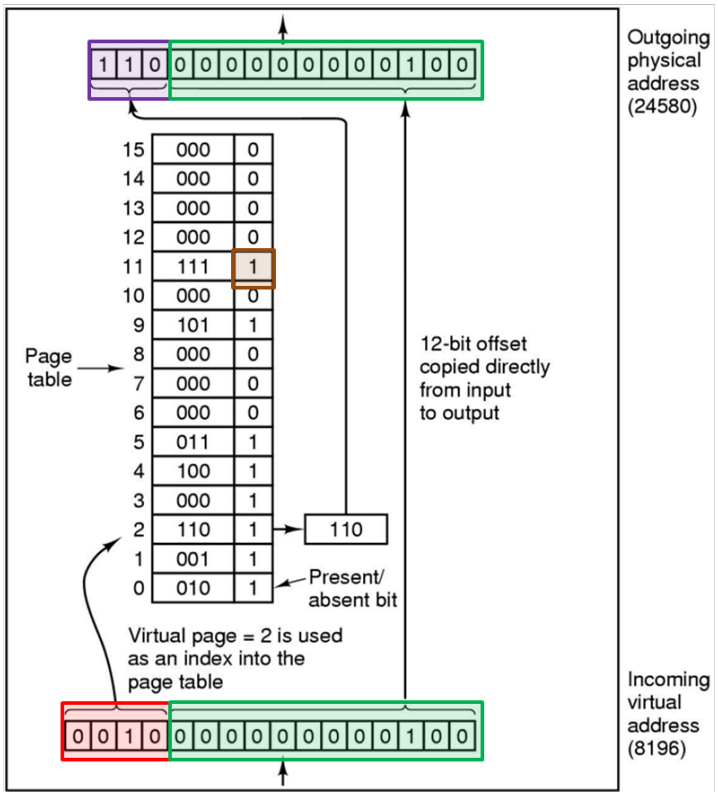
A Page Table is an array of Page Table Entries (PTEs) that either map a virtual page to a physical page if a mapping exits, or indicate that no current mapping exits

The virtual address space shown above consists of **16 virtual pages**, with a 4 KiB page size. Physical memory, as shown here, has only **8 physical pages**. The entire virtual address space could never fit into the physical address space at the same time in this example, but we can map as many as 8 virtual pages at any given time. The operating system will do its best to make sure that the **"best" 8 pages** are mapped at any time, meaning the 8 pages that a running thread will need to reference the most during a given time interval. Since each physical frame shown is currently mapped, if a thread running against the given virtual address space suddenly make a reference to an address in the range of 28 to 32 K (this range is shown un-mapped), a page fault would occur, and the operating system would have to select an existing mapping to evict, such that the freed frame could be used to satisfy the **current fault**.

Choosing a victim to **evict** when a page fault occurs against a full physical memory is one of the most critical parts of page fault management. If we evict a page that will be needed again soon, we will have wasted a lot of effort in removing the page, only to have to fault it right back in again. We need to select victims that are unlikely to be needed again to achieve optimal performance. Section 3.4 from the text examines some replacement algorithms.

The page table entries (PTEs) shown above include either a physical page number, if the virtual page is mapped, or the letter X, if the virtual page is not mapped. In a real system, PTEs normally include enough bits to point to any existing physical page, along with some **additional bits** to indicate a valid mapping and the **type of access permitted** to the target physical page. The number of additional bits is dictated by a given CPU, but there will always be at least one bit to indicate if a mapping is **valid** or not. This bit is called the valid (V) or present (P) bit, and may be accompanied by other bits that describe how the mapping can be used (read page only, read/write page, cache parts of page, page modified state, etc.). For the simple system shown above, the bit level view of the page table would look like:

An example of a **16 virtual page** address space mapped into an **8 frame physical DRAM space**. PTEs have a **"present or valid"** bit to show **map state**. The 12 bit **"offset"** portion of an address points to a byte on a **4KB** page/frame

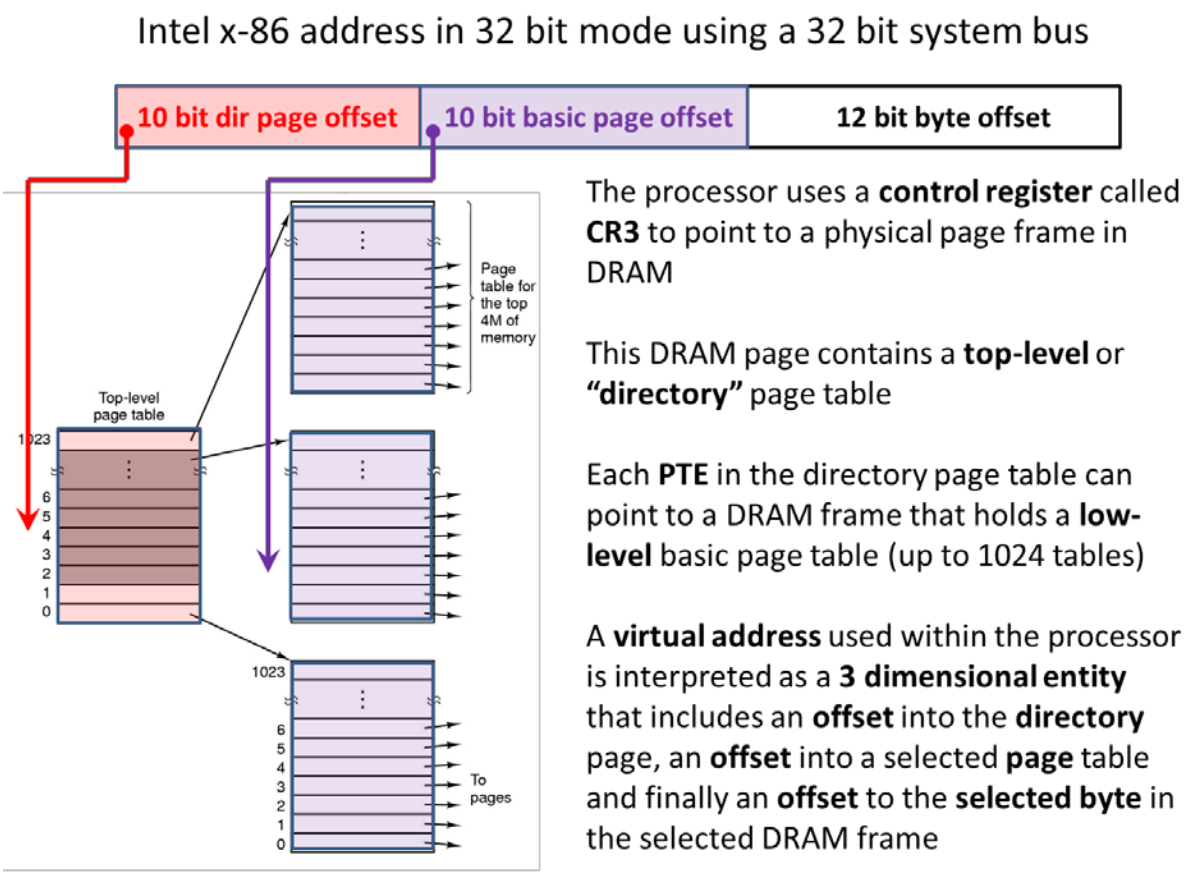


Notice how the incoming 16 bit virtual address of 8196_{10} ($0010\ 000000000100_2$) is **translated**. The full virtual address space discussed here is 2^{16} = 64 KiB (16 pages * 4 KB/page), so the given 16 bit virtual address is broken into a page component (the most significant 4 bits point to a virtual page, since 2^4 = 16) and a byte offset (the next 12 bits point to some byte location on a 4 KiB page, since 2^{12} = 4096, AKA 4K). The page bits bring us to a PTE (0010 send us to slot 2 in this case), and from there we can trade our virtual page bits (0010) for the corresponding physical page bits (here 110). The trade is allowed, because the **P** (present) bit is set to 1 (TRUE). If our inbound page bits were set to say 0111 (slot 7), we could not have made the trade for virtual to physical page bits because slot 7 has the **P** bit set to 0 (FALSE), and this would have resulted in an exception to the CPU (a page fault). This translation involves changing the original 16 bit virtual address into a 15 bit physical address (because physical memory is only 2^3 = 8 pages * 2^{12} = 4 KiB/page = 2^{15} = 32 KiB).

In general, the translation of a virtual to physical address involves **replacing the virtual page bits** in the inbound address with the appropriate **physical page bits** in the outbound address. The byte offset portion of the address just gets carried along, since the byte location is the same whether we consider the page to be virtual or physical. In some cases, a virtual address may be smaller than a physical address (e.g. the x86 architecture in 32 bit mode only allows a virtual address space per process of 2^{32} or 4 GiB, but the processor's physical bus supports at least 2^{36} or 64 GiB of physical real estate). In other cases, the virtual address space may be (much)

larger than the physical address space (e.g. the x86 architecture in 64 bit mode allows a virtual address space per process of 2^{48} or 256 TiB, but the largest physical bus presently available supports 2^{46} or 64 TiB of physical real estate).

Virtual memory translation requires a **page table per process** (address space), as we've seen. The larger the virtual address space, the larger these tables must become. Considering the x86 architecture in 32 bit mode, the 4 GiB virtual address space that the processor supports requires 1M (2^{20}) page table entries for full mapping (i.e. 2^{20} (1 M PTEs) * 2^{12} (4 KiB/page) = 2^{32}, or 4GiB). In this case, each PTE is 4 bytes long, meaning that the full mapping page table for just **one process address space** would be **4 MiB big**, and we know that we need such a map for each process that is alive in a system like Windows or Linux. It's not practical to require such a large table to be kept **resident** in RAM for each process (the problem is much worse for 64 bit systems as you can envision), so page table structures are generally **broken into two or more hierarchically related tables**, only the highest level of which is required to be memory resident at all times. The x86 processor has several special registers that manage memory, one of which is called **CR3** (control register #3). The CR3 register is one of the few registers in the CPU that actually has a **physical address** in it (once the operating system has started, all addresses used in the rest of the CPU for normal computations are virtual). The address in CR3 always points to a **physical 4 KiB page** that represents the **top level** (directory level) of the page table hierarchy of whatever process is currently using that CPU (remember, there must be a thread on every CPU running at all times, and every thread belongs to some process, and runs within that process address space). When the running thread presents the CPU with a reference to a virtual address (to fetch an instruction or load/store a piece of data), the CPU must translate that virtual address into it corresponding physical address. It does this by using **CR3** to find the **directory level page table**, and then locates a **PTE** in this table by using the most significant 10 bits of the given virtual address (since the table is in a 4 KiB page and each entry is 4 bytes, there are 4096/4 = 1024 = 2^{10} = 1K entries, and 10 bits can index 1024 entries). An entry in this directory level page, if valid, will contain the **physical address of a 4 KiB page acting as a low-level page table.** A low-level page table also has 1024 PTEs, so this table is indexed by the **next 10 bits** in the virtual address. The selected entry in this table, if valid, will have the **actual physical address** of the page that holds the referenced instruction or piece of data, and completes virtual to physical translation. This can be seen for the x86 32 bit deployment in the diagram below:



Notice that each entry in the top-level table can point to a physical page with 1024 entries to target physical pages. This two level strategy provides 1024 * 1024 PTEs, the required 1 M entries. The x86 CPU requires a top-level (directory) table to be **resident** in memory for **each living process** in the system, and each time that a **heavy weight context switch** occurs on a CPU, the CR3 register is reset to point to the directory page of the inbound thread's process directory table. Only the directory table is required to be memory resident, the low-level page table pages are only paged into memory **when they are needed**, lessening the burden of page tables on the physical memory system.

Page Replacement Algorithms:

Virtual memory implementations allow an application to see a contiguous range of (virtual) memory addresses, but allow the system to locate any one of the needed virtual pages wherever in memory there is an available free physical page. If an application needs 512 MiB of contiguous memory space to build a data structure in, it can get that space using a malloc() request. The malloc() routine returns a starting address for this region to the calling application, and the application is **assured** of having this **contiguous** space available. As the application references different parts of this address chunk, the operating system will **find physical pages** to map to the referenced virtual addresses. The connected physical pages will have **little or no contiguity** among one another, but the application will see its space as a contiguous range of addresses as required. Virtual memory systems allow us to use physical memory very efficiently, by **eliminating the need for allocating physically contiguous memory ranges**.

We can imagine that at boot time, the operating system has a large number of physical memory frames (4 KiB pages) that are not immediately in use (**the free list of pages**). As applications begin to run and **fault** pages into their address spaces, the **free list dwindles**. Since page faults happen frequently in a busy system, the operating system must keep a **supply of free pages** in order to handle faults when they occur. If the operating system detects that the size of the free page list is getting **too small,** it will run a thread called the **page-purger**, that will begin to **take pages away** from various processes that have previously faulted these pages into physical frames. This is called **page replacement**, and requires the page-purger to **decide** what pages to take away from which processes. In general, we would like the purger to **reclaim pages** that are **unlikely to be needed** any further by the processes that own them. Removing a page from a process address space involves finding the page table mapping for the target page and updating that entry to be **invalid** (turn off the valid or present bit). If the selected page frame is **dirty** (has been modified since the owning process first mapped it), then that page will have to be **backed up to disk somewhere,** so if the owning process ever needs it again we can retrieve its content.

Several algorithms are discussed in the text, but in practice, page replacement algorithms tend to be various approximations of the **Least Recently Used** (LRU) strategy. The **LRU** strategy attempts to keep a small amount of information about each physical page frame, regarding when it was **last referenced** (used). The assumption is that if a page has not been referenced for a long time, the owner of that page has probably finished using it and has moved on to other parts of its program, thus making its **reclamation efficient**. If, on the other hand, the owner of the reclaimed page demands it back right after we've taken it away, then our replacement algorithm will **not be very efficient**.

Whatever algorithm is deployed in a given operating system, you can see that what we need to make page replacement efficient is:

- an algorithm that can **quickly** make a decision about what page to reclaim
- an algorithm that **limits the overall system page fault rate** by making **"good"** decisions about what page to reclaim.

Evaluating the efficacy of a page replacement algorithm is often done by using a post analysis on a particular workload with the **Optimal Replacement Algorithm** (OPT). The page reference pattern of a particular workload (benchmark) is collected on an instrumented system. This data consists of recording every memory reference that occurs during the execution of the benchmark work load. The data are then analyzed with the **OPT** algorithm, which determines what the **minimum number of page faults** for this workload could possibly be for a **given sized physical memory**. OPT provides a lower bound on page faults for the benchmark, but it is a post analysis tool, and is of no value to a running system. Once the benchmark data has been analyzed, the benchmark is re-run under an operating system that is attempting to evaluate its page replacement algorithm. The number of faults in this real system is collected, and compared to the OPT result. For example, two different replacement algorithms might yield 230,000,000 and 210,000,000 faults respectively. The second would appear superior to the first, but if OPT analysis shows a lower bound of 90,000,000, then neither algorithm is really any good, and it's back to the drawing board.

We will consider some additional aspects of paging systems in the next lesson, but keep in mind that this paging mechanism is used by all but the simplest embedded operating systems to enable efficient memory management. When systems underperform, thrash or cause endless delays to users, it's very likely that system memory management is the underlying problem. Contemporary virtual memory implementations can utilize memory very efficiently, but even the

most efficient management of a resource will falter when the load simply outstrips the capacity. Adding more physical memory to a system is almost always a remedy for performance woes.

Week #6 Summary:

We think of memory as one of the critical resources of a computing system. A CPU cannot execute an instruction or manipulate a piece of data unless that object has first found its way into some kind of physical memory.

In contemporary systems, memory management occurs at different levels of abstraction, beginning with a view of memory as a contiguous range of byte sized storage elements. When presented with this view of memory, we need to consider management strategies that deal with the allocation and accounting of variable sized chunks of contiguous memory to be allocated on request. We looked at several **linked list approaches** to this type of allocation, including **first-fit, best-fit and worst-fit**, considering implementation and performance aspects of both allocating and then freeing contiguous memory.

Managing physical memory with contiguous allocation policies, however, leads to serious **fragmentation** issues, so this level of management needs to be layered on top of a more efficient mechanism. **Virtual memory** provides a solution by giving each process the illusion of a contiguous address space to work in, but that space is virtual, and requires a lower level management technique to work efficiently.

Below the virtual address space, is a **paging system** that manages physical memory in **fixed sized allocation units called pages.** When a running thread references a part of its address space that is not currently realized as physical, a **page fault exception** is delivered to the operating system, which then connects the virtual page that includes the referenced address to a physical page frame and allows the reference to proceed.

Because free pages are in constant demand on a busy system that is experiencing many page faults per second, the operating system maintains a thread called the **page-purger.** The purger is responsible for running the **page replacement algorithm** of the system, during which it **reclaims pages** from existing processes to add to the free list. The replacement algorithm is a critical element of overall system performance. A good replacement implementation will keep the system page fault rate as close to the theoretical minimum as possible for the least amount of execution time possible. In other words, it will make **good decisions** about what pages to reclaim, and will make those decisions **very quickly**.