Lecture Notes for CS347: Operating Systems

Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

7. Memory Management

7.1 Basics of Memory Management

- What does main memory (RAM) contain? Immediately after boot up, it contains the memory image of the kernel executable, typically in "low memory" or physical memory addresses starting from byte 0. Over time, the kernel allocates the rest of the memory to various running processes. For any process to execute on the CPU, the corresponding instructions and data must be available in main memory. The execution of a process on the CPU generates several requests for memory reads and writes. The memory subsystem of an OS must efficiently handle these requests. Note that contents of main memory may also be cached in the various levels of caches between the CPU and main memory, but the OS largely concerns itself with CPU caches.

- The OS tries to ensure that running processes reside in memory as far as possible. When the system is running low on memory, however, running processes may be moved to a region identified as **swap space** on a secondary storage device. Swapping processes out leads to a performance penalty, and is avoided as far as possible.

- The memory addresses generated by the CPU when accessing the code and data of a running process are called **logical addresses**. The **logical/virtual address space of a process** ranges from 0 to a maximum value that depends on the CPU architecture (4GB for 32-bit addresses). Code and data in the program executable are assigned logical addresses in this space by the compiler. On the other hand, the addresses used by the actual memory hardware to locate information are called **physical addresses**. The **physical address space of a system** spans from 0 to a maximum value (determined by how much RAM the machine has). The OS plays a big role in mapping logical addresses of a process to physical addresses.

- When a new process is created, there are two ways of allocating memory to it: contiguous allocation or non-contiguous allocation. With contiguous allocation, the kernel tries to find a contiguous portion of physical memory to accommodate the new process. The kernel may use a best-fit or worst-fit or first-fit heuristic to identify a portion of unused memory. The problem with this type of allocation is (external) fragmentation of memory: sometimes, there is enough free memory in the system, but it is fragmented over several locations, so that a contiguous block cannot be found to satisfy a request. To avoid external fragmentation with contiguous allocation, the OS must periodically relocate running processes to other regions of memory.

- With contiguous allocation, the mapping between logical and physical addresses is straightforward. The OS maintains the starting physical address (or *base*) and the size of the memory image (or *limit*) for every process it places in memory. This base address value is added to the logical addresses to translate it to a physical address. While contiguous allocation has the benefit of simplicity, it is rarely used in modern operating systems due to the issues around memory fragmentation.

- The most common way of allocating memory to processes in modern operating systems is a type of non-contiguous allocation called **paging**. The logical address space of every process is divided into fixed-size (e.g., 4KB) chunks called **pages**. The physical memory is divided into fixed size chunks called **frames**, which are typically the same size as pages. Every process is allocated some free physical memory frames, and its logical pages are *mapped* to these physical frames. A **page table** of a process maintains this mapping from logical page numbers to physical frame numbers. The operating system maintains a separate page table for every process it creates. With paging, the issue of external fragmentation is completely eliminated, because any free frame can be allocated to any process. However, a smaller problem of internal fragmentation (the last page being partially filled) arises.

- Paging adds a level of indirection between logical and physical addressing, and this provides several benefits. There is now a clean separation between logical addresses and physical addresses: the compiler can assign addresses to code and data without worrying about where the program would reside in memory. Paging also makes sharing of memory between processes easier: two logical pages can point to the same physical frame in the page tables. However, paging adds the additional overhead of address translation.

- How are virtual addresses mapped to physical addresses with paging? The virtual address is first split into a page number and an offset within the page. The page number is mapped to the physical frame number by looking up the page table of the process. The physical address is then obtained from the physical frame number and the offset within the frame. Who does this translation from CPU-generated logical addresses to physical addresses? The OS takes the responsibility of constructing and maintaining page tables during the lifecycle of a process; the PCB contains a pointer to the page table. The OS also maintains a pointer to the page table of the current process in a special CPU register (CR3 in x86), and updates this pointer during context switches. A specialized piece of hardware called the **memory management unit / MMU** then uses the page table to translate logical addresses requested by the CPU to physical addresses using the logic described above.

- Segmentation is another way of allocating physical memory to a process. With segmentation, the process memory image is divided into segments corresponding to code, data, stack, and so on. Each segment is then given a contiguous chunk of physical memory. A segment table stores the mapping between the segments of a process and the base/limit addresses of that segment. Most modern operating systems, however, use paging, or a combination of segmentation and paging. Unix-like operating systems make minimal use of segmentation.

- Segmentation can be used to create a hierarchical address space, where the segment number forms the most significant bits of the logical address. However, Unix-like operating systems mostly use a flat address model on modern CPU architectures in user mode. Most C compilers for Linux on x86 today, for example, generate logical addresses from 0 to 4GB, and all segment registers are set to zero. The values of segment registers only change when moving from user mode to kernel mode and vice versa.

- At boot time, the first pieces of code that executes must work only with physical addresses. Once the boot loader executes code to construct page tables and turn the MMU on, the kernel

code will start working with virtual addresses. Therefore, the booting process has to deal with some complexities arising from this transition.

- On all memory accesses, the memory hardware checks that the memory access is indeed allowed, and raises a trap if it detects an illegal access (e.g., user mode process accessing kernel memory or the memory of another process). With contiguous memory allocation, it is easy to check if the memory address generated by the CPU indeed belongs to the process or not: before accessing the memory, one can check if the requested address lies in the range [base, base+limit). With paging, every page page has a set of bits indicating permissions. During address translation, the hardware checks that the requested logical address has a corresponding page table entry with sufficient permissions, and raises a trap if an illegal access is detected.

- With a separation of virtual address space and physical address space in modern operating systems, each process can have a large virtual address space. In fact, the combined virtual address space of all processes can be much larger than the physical memory available on the machine, and logical pages can be mapped to physical frames only on a need basis. This concept is called **demand paging**, and is quite common in modern operating systems. With demand paging, the memory allocated to a process is also called **virtual memory**, because not all of it corresponds to physical memory in hardware.

7.2 Address Space of a Process

- The virtual address space of a process has two main parts: the user part containing the code/data of the process itself, and the kernel code/data. For example, on a 32-bit x86 system, addresses 0-3GB of the virtual address space of a process could contain user data, and addresses 3-4GB could point to the kernel. The page table of every process contains mappings for the user pages and the kernel pages. The kernel page table entries are common for all processes (as there is only one physical copy of the kernel in memory), while the user page table entries are obviously different.

- Note that every physical memory address that is in use will be *mapped* into the virtual address space of at least one process. That is, the physical memory address will correspond to a virtual address in the page table of some process. Physical memory that is not mapped into the address space of any process is by definition not accessible, since (almost) all accesses to memory go via the MMU. Some physical memory can be mapped multiple times, e.g., kernel code and data is mapped into the virtual address space of every process.

- Why is the kernel mapped into the address space of every process? Having the kernel in every address space makes it easy to execute kernel code while in kernel mode: one does not have to switch page tables or anything, and executing kernel code is as simple as jumping to a memory location in the kernel part of the address space. Page tables for kernel pages have a special protection bit set, and the CPU must be in kernel mode to access these pages, to protect against rogue processes.

- The user part of the address space contains the executable code of the process and statically allocated data. It also contains a heap for dynamic memory allocation, and a stack, with the heap and stack growing in opposite directions towards each other. Dynamically linked libraries, memory-mapped files, and other such things also form a part of the virtual address space. By assigning a part of the virtual address space to memory mapped files, the data in these files can be accessed just like any other variable in main memory, and not via disk reads and writes. The virtual address space in Linux is divided into memory areas or maps for each of the entities mentioned above.

- The kernel part of the address space contains the kernel code and data. For example, it has various kernel data structures like the list of processes, free pages to allocate to new processes, and so on. The virtual addresses assigned to kernel code and data are the same across all processes.

- One important concept to understand here is that most physical memory will be mapped (at least) twice, once to the kernel part of the address space of processes, and once to the user part of some process. To see why, note that the kernel maintains a list of free frames/pages, which are subsequently allocated to store user process images. Suppose a free frame of size N bytes is assigned a virtual address, say V, by the kernel. Suppose the kernel maintains a 4-byte pointer to this free page, whose value is simply the starting virtual address V of the free page. Even though the kernel only needs this pointer variable to track the page, note that it cannot assign

the virtual addresses $[V, V + N)$ to any other variable, because these addresses refer to the memory in that page, and will be used by the kernel to read/write data into that free page. That is, a free page blocks out a page-sized chunk of the kernel address space. Now, when this page is allocated to a new process, the process will assign a different virtual address range to it (say, $[U, U + N)$), from the user part of its virtual address space, which will be used by the process to read/write data in user mode. So the same physical frame will also have blocked out another chunk of virtual addresses in the process, this time from the user part. That is, the same physical memory is mapped twice, once into the kernel part of the address space (so that the kernel can refer to it), and once into the user part of the address space of a process (so that the process can refer to it in user mode).

- Is this double consumption of virtual addresses a problem? In architectures where virtual address spaces are much larger than the physical memory, this duplication is not a problem, and it is alright to have one byte of physical memory block out two or more bytes of virtual address space. However, in systems with smaller virtual address spaces (due to smaller number of bits available to store memory addresses in registers), one of the following will happen: either the entire physical memory will not be used (as in the case of xv6), or more commonly, some part of user memory will not be mapped in the kernel address space all the time (as in the case of Linux). That is, once the kernel allocates a free page to a process, it will remove its page table mappings that point to that physical memory, and use those freed up virtual addresses to point to something else. Subsequently, this physical memory will only be accessible from the user mode of a process, because only the user virtual addresses point to it in the page table. Such memory is called "high memory" in Linux, and high memory is mapped into the kernel address space (i.e., virtual addresses are allocated from the kernel portion of the virtual memory) only on a need basis.

7.3 Paging

- Paging is a memory management technique that allows non-contiguous memory allocations to processes, and avoids the problem of external fragmentation. Most modern operating systems use paging. With paging, the physical memory is divided into frames, and the virtual address space of a process is divided into logical pages. Memory is allocated at the granularity of pages. When a process requires a new page, the OS finds a free frame to map this page into, and inserts a mapping between the page number and frame number in the page table of the process.

- Suppose the size of the logical address space is 2^m bytes, and the size of the physical memory is 2^n bytes. That is, logical addresses are m bits long, and physical memory addresses are n bits long. Suppose the size of each logical page and physical frame is 2^k bytes. Then, out of the m bits in the virtual address, the most significant $m-k$ bits indicate the page number, and the least significant k bits indicate the offset within the page. A page table of a process contains 2^{m-k} **page table entries (PTEs)**, one for each logical page of the process. Each PTE stores, among other things, the physical frame number which this page corresponds to. Now, since there are a total of 2^{n-k} physical frames, it would take $n-k$ bits to identify a physical frame. Thus, each of the PTEs must be at least $n-k$ bits in size, putting the total size of the page table of a process at at least $2^{m-k} \times (n-k)$ bits. Note that PTEs contain a bit more information than just the physical frame number, e.g., permissions to access the page, so PTEs are slightly longer than $n-k$ bits.

- For example, consider a system with 32-bit virtual addresses (i.e., the virtual address space is of size 2^{32} bytes = 4GB). Suppose the size of the physical memory is 64 GB, and pages/frames are of size 4KB each. The total number of pages, and hence the number of PTEs in a page table, is 2^{20} for each process. The number of physical frames in the system is 2^{24}. Therefore, each PTE must be at least 24 bits, or 3 bytes long. Assume that a PTE is 4 bytes long. Therefore, the size of the page table of every process is $2^{20} * 4$ bytes = 4MB.

- To translate a virtual address to a physical address, the MMU uses the most significant $p = m-k$ bits of the virtual address to offset into the page table, and replace the most significant p bits with the $f = n-k$ bits of the physical frame number. Assuming pages and frames are the same size, the lower k bits that indicate the offset in the logical page will still map to the offset in the physical frame.

- Note that a process need not always use up its entire virtual address space, and some logical pages can remain unassigned. In such cases, a bit in the PTE can indicate whether the page is valid or not, or the size of the page table can itself be trimmed to hold only valid pages. Accesses to unmapped/invalid logical addresses will cause the MMU to raise a trap to the OS. The OS can sometimes leave certain addresses unmapped on purpose, e.g., at the end of the user stack, to detect processes writing beyond page boundaries. The valid bit in the PTE, and the read/write permissions bits are powerful tools for the OS to enforce memory access control at the granularity of a page.

- A subtle point to note is that when a process is executing on the CPU, memory access requests from the CPU directly go to the memory via the MMU, and the OS is not involved in inspecting

or intercepting every request. If the OS wishes to be informed of access to any memory page, the only way to enforce it would be to set appropriate bits in the PTEs which will cause a memory request to trap and reach the kernel. For example, consider how an OS would implement copy-on-write during fork. If the OS has only one memory copy for the child and parent processes, how does it know when one of them modifies this common image, so that it can make a copy? The OS can set all PTEs in the common memory image to read only, so that when either the parent or the child tries to write to the memory, a trap is raised by the MMU. This trap halts the execution of the process, moves it to kernel mode, and transfers control to the kernel. The kernel can then make a copy the memory image, update the page table entries to point to the new image, remove the read only constraint, and restart the halted process.

- How are pages sized? In general, smaller page sizes lead to lesser internal fragmentation (i.e., space being unused in the last page of the process). However, smaller pages also imply greater overhead in storing page tables. Typical pages sizes in modern operating systems are 4-8KB, though systems also support large pages of a few MB in size. Most modern operating systems support multiple page sizes based on the application requirements.

- With smaller virtual address spaces, the page table mappings could fit into a small number of CPU registers. But current page tables cannot be accommodated in registers. Therefore, page tables are stored in memory and only a pointer to the starting (physical) address of a page table is stored in CPU registers (and changed at the time of a context switch). Thus, accessing a byte from memory first requires mapping that byte's virtual address to a physical address, requiring one or more accesses of the page table in memory, before the actual memory access can even begin. This delay of multiple memory accesses imposes an unreasonable overhead. Therefore, most modern systems have a piece of hardware called a **translation look-aside buffer (TLB)**.

7.4 TLB cache

- A TLB is a small, fast cache that stores the most recently used mappings from virtual to physical addresses. TLB is an associative memory that maps keys (virtual addresses) to values (physical addresses). Most architectures implement the TLB in hardware, and the TLB is transparent to the OS. That is, the OS only manages the page tables and does not control the TLB, and caching in the TLB is done automatically by the hardware. However, some architectures do provide a software TLB with more complicated features.

- If a virtual address is found in the TLB (a TLB hit), a memory access can be directly made. On the other hand, if a TLB miss occurs, then one or more memory accesses must first be made to the page tables before the actual memory access can be made. A healthy TLB hit ratio is thus essential for good performance. Further, the TLB reach (or the amount of address space it covers) should also be large.

- Let T_c denote the time to check if an address is mapped in the TLB, and let T_m denote the time to access main memory. Further, suppose a TLB miss requires n additional memory accesses to page tables before the actual memory access. Then, the expected time for a memory access in a system with a TLB hit ratio of f is given by $f \times (T_c + T_m) + (1 - f) \times (T_c + (n + 1)T_m)$.

- LRU (least recently used) policy is often used to evict entries from the TLB when it is running out of space. However, some entries (e.g., those corresponding to kernel code) are permanently stored in the TLB. Further, most TLB entries become invalidated after a context switch, because the mappings from virtual to physical addresses are different for different processes. To avoid flushing all TLB entries on a context switch, some TLB designs store a tag of the process ID in the TLB entry, and access only those TLB entries whose tag matches that of the current running process.

7.5 Page table design

- How are page tables stored in memory? Since the CPU uses just one register to store the starting address of the page table, one way is to store the page table contiguously beginning from the start address. However, it is hard to store page tables as large as 4MB contiguously in memory. Therefore, page tables are themselves split up and stored in pages. Suppose a PTE is of size 2^e bytes and pages are of size 2^k bytes. Then each page can hold 2^{k-e} PTEs. Now, the 2^{m-k} page table entries in the *inner* page table will be spread out over $\frac{2^{m-k}}{2^{k-e}} = 2^{m+e-2k}$ pages. And an *outer* page table stores the frame numbers of the pages of the inner page tables. That is, given the virtual address, the least significant k bits provide an offset in a page, the middle $k-e$ bits will be used to offset into the inner page table to obtain the frame number, and the most significant $m+e-2k$ bits will be used to index into the outer page table to find the location (frame number) of the inner page table itself. For example, for 32-bit virtual addresses, 4KB pages, and 4-byte page table entries, the most significant 10 bits are used to index into the outer page table, and next 10 bits are used to index into the inner page table, to find the frame number. For this example, the outer page table fits snugly in a page. However, if the outer page table size gets larger, it further may need to be split, and several levels of hierarchy may be required to store the page table. This concept is called **hierarchical paging**, and quickly gets inefficient for large virtual address spaces (e.g., 64 bit architectures).

- An alternative design for large address spaces is a **hashed page table**. In this design, the virtual address (key) is hashed to locate its corresponding value entry. For multiple virtual addresses that hash to the same location, a linked list or some such mechanism is used to store all the mappings.

- Another alternative design for page tables in systems with large virtual address spaces is to have a table with entries for every physical frame, and not for every logical page. This is called an **inverted page table**. Every entry of the inverted page table maps a physical frame number to the process identifier, and the logical page number in the address space of that process. To translate a virtual address to a physical address, one would have to search through all the entries in the inverted page table until a matching logical page number is found. So, while inverted page tables save on size, lookup times are longer. Also, inverted page tables make it hard to share pages between processes.

- Some operating systems use a combination of segmentation and paging. The logical address space is divided into a small number of segments, and the logical address is the combination of the segment number and an offset within that segment. The CPU can only specify the offset part of the logical address, and based on which segment of the memory image it is executing in, the complete logical address can be generated. This logical address is then mapped to the physical address via paging. For example, Linux uses segmentation with paging on architectures that have support for segmentation (e.g., Intel x86). Linux uses a small number of segments: one each of user code, user data, kernel code, kernel data, stack segment, and so on. A segment table stores the starting (logical) addresses of each of these segments in the virtual address space of a process, and implements protection in terms of who can access that segment. For example, access to the kernel segments is allowed only when the CPU is in kernel mode.

7.6 Demand Paging

- When are physical frames allocated to logical pages? In a simplistic case, all logical pages can be assigned physical frames right at the beginning when a process is created and brought into memory. However, this is wasteful for a number of reasons. For example, a process may access only parts of its memory during a run. Therefore, modern operating systems use a concept called demand paging. With demand paging, a logical page is assigned a physical frame only when the data in the page needs to be accessed. At other times, the data in the page will reside on secondary storage like a hard disk. A bit in the page table indicates whether the page is in memory or not. With demand paging, a process is allocated only as much physical memory as it needs, thereby allowing us to accommodate many more processes in memory.

- Suppose the CPU tries to access a certain logical address and the page table shows that the page containing that address has not been mapped into memory. Then a **page fault** is said to occur. A page fault traps the OS. The process moves into kernel mode. The OS then handles the page fault as follows: a free physical frame is identified, the required page is requested to be read from the disk, and the process that generated the page fault is context switched out. The CPU then starts running another process. When the data for the page has been read from disk into the free frame, an interrupt occurs, and is handled by whichever process currently has the CPU in its kernel mode. The interrupt service routine in the OS updates the page table of the process that saw the page fault, and marks it as ready to run again. When the CPU scheduler runs the process again, the process starts running at the instruction that generated the page fault, and continues execution normally.

- Note that an instruction can be interrupted midway due to a page fault, and will be restarted from the beginning after the page fault has been serviced. Care should be taken in saving the context of a process to ensure that any temporary state that needs to be preserved is indeed preserved as part of the saved context. The instruction set of the underlying architecture must be carefully examined for such issues when adding paging to the OS.

- Let T_m be the time to access main memory and T_f be the time to service a page fault (which is typically large, given that a disk access is required). If p is the page fault rate, then the effective memory access time is $(1 - p) \times T_m + p \times T_f$. Systems with high page fault rate see a high memory access time, and hence worse performance.

- Note that most operating systems do not allow the kernel code or data to be paged out, for reasons of performance and convenience. For example, the operating system could never recover from a page fault, if the code to handle the page fault itself is paged out, and causes a page fault when accessed.

- Some operating systems implement **prepaging**, where a set of pages that are likely to be accessed in the future are brought into memory before a page fault occurs. The success of prepaging depends on how well the page access pattern can be predicted, so as to not bring pages that will never be used into memory.

- Note that an inverted page table design will be inefficient when demand paging is used, because one also needs to lookup pages that are not in memory, and an inverted page table does not have entries for such pages. Therefore, operating systems that use inverted page tables must also keep a regular page table around to identify and service page faults.

- Demand paging allows for an over-commitment of physical memory, where the aggregate virtual addresses of all processes can be (much) larger than the underlying physical address space. Over-commitment is generally acceptable because processes do not tend to use all their memory all the time. In fact, most memory accesses have a locality of reference, where a process runs in one part of its virtual address space for a while (causing memory accesses in a small number of pages), before moving on to a different part of its address space. Therefore, over-commitment, when done carefully, can lead to improved performance and an ability to support a large number of processes. Operating systems that over-commit memory to processes are said to provide virtual (not physical) memory to processes. Note that virtual addressing does not necessarily imply virtual memory, though most modern operating systems implement both these ideas.

- To implement virtual memory, an OS requires two mechanisms. It needs a **frame allocation algorithm** to decide how many physical frames to allocate to each process. Note that every process needs a certain minimum number of frames to function, and an OS must strive to provide this minimum number of frames. Further, when all physical frames have been allocated, and a process needs a new frame for a page, the OS must take a frame away from one logical page (that is hopefully not being actively used) and use it to satisfy the new request. A **page replacement algorithm** decides which logical page must be replaced in such cases.

- A process memory image is initially read from the file system. As the process runs, some pages may be modified and may diverge from the content in the file systems. Such pages that are not backed by a copy in the file system are called **dirty** pages. When a dirty page is replaced, its contents must be flushed to disk (either to a file or to swap space). Therefore, servicing a page fault may incur two disk operations, one to swap out an unused page to free up a physical frame, and another to read in the content of the new page. The page table maintains a dirty bit for every page, along with a valid bit.

7.7 Page Replacement Algorithms

- Below are some popular page replacement algorithms. Modern operating systems like Linux use some variants of these simple algorithms.

    1. **FIFO** (First In First Out) is the simplest page replacement algorithm. With FIFO, logical pages assigned to processes are placed in a FIFO queue. New pages are added at the tail. When a new page must be mapped, the page at the head of the queue is evicted. This way, the page brought into the memory earliest is swapped out. However, this oldest page maybe a piece of code that is heavily used, and may cause a page fault very soon, so FIFO does not always make good choices, and may have a higher than optimal page fault rate. The FIFO policy also suffers from *Belady's anomaly*, i.e., the page fault rate may not be monotonically decreasing with the total available memory, which would have been the expectation with a sane page replacement algorithm. (Because FIFO doesn't care for popularity of pages, it may so happen that some physical memory sizes lead you to replace a heavily used page, while some don't, resulting in the anomaly.)
    2. What is the **optimal** page replacement algorithm? One must ideally replace a page that will not be used for the longest time in the future. However, since one cannot look into the future, this optimal algorithm cannot be realized in practice.
    3. The **LRU** (least recently used) policy replaces the page that has not be used for the longest time in the past, and is somewhat of an approximation to the optimal policy. It doesn't suffer from Belady's anomaly (the ordering of least recently used pages won't change based on how much memory you have), and is one of the more popular policies. To implement LRU, one must maintain the time of access of each page, which is an expensive operation if done in software by the kernel. Another way to implement LRU is to store the pages in a stack, move a page to the top of the stack when accessed, and evict from the bottom of the stack. However, this solution also incurs a lot of overhead (changing stack pointers) for every memory access.

       LRU is best implemented with some support from the hardware. Most memory hardware can set a *reference bit* when a page is accessed. The OS clears all reference bits initially, and periodically checks the reference bits. If the bit is set, the OS can know that a page has been accessed since the bit was last cleared, though it wouldn't know when it has been accessed during that period. An LRU-like algorithm can be built using reference bits as follows: the OS samples and clears the reference bit of every page every epoch (say, every few milliseconds), and maintains a history of the reference bit values in the last few epochs in the page table. Using this history of references, the OS can figure out which pages were accessed in the recent epochs and which weren't.
    4. Finally, several variants of the simple FIFO policy can be created using the hardware support of reference bits. In the **second chance** algorithm, when the OS needs a free page, it runs through the queue of pages, much like in FIFO. However, if the reference bit is set, the OS skips that page, and clears the bit (for the future). The OS walks through the list of pages, clearing set bits, until a page with no reference bit set is found. The OS keeps this pointer to the list, and resumes its search for free pages from the following page

for subsequent runs of the algorithm. In addition to the reference bit, the second chance algorithm can also look at the dirty bit of the page. Pages that have not been recently accessed or written to (both reference and dirty bits are not set) are ideal candidates for replacements, and pages that have both bits set will not be replaced as far as possible.

- The OS can also proactively evict pages and maintain a small pool of free frames in the background, or when the secondary storage is free, in order not to incur the overhead of disk access when a page is requested. Also, the OS can keep the contents of such freed pages intact, so that the content can be reused without going to disk, should the old page be accessed again before the frame is allocated to another logical page. Several such optimizations are possible over the simple policies mentioned above.

- Some operating systems provide the option of locking some pages in memory so that they are never evicted. For example, it does not make sense to evict a page that is brought into memory to service a page fault before the process that faulted gets a chance to run on the CPU again. When a memory buffer has been allocated for an I/O operation such as a disk read, the page should not be evicted before the I/O completes. For all such cases, a lock bit may be set to instruct the kernel to not evict the page.

- Finally, page replacement algorithms can be local or global. That is, when a process requests a page, the OS could evict one of the pages of that process itself. Or, the OS could treat the pool of pages as a global resource, and evict pages from any other process, possibly even taking process priority into account. With global page replacement algorithms, lower priority processes are likely to see a higher page fault rate, and may have no control over it.

7.8 Frame Allocation Policies

- Now, how many frames must be allocated to a process? Every process must be allocated a certain minimum number of frames to let it function properly. For example, every process must get enough frames to hold the data required for any single instruction. Otherwise, the instruction will halt midway due to a page fault, and will page fault when rerun again. Note that some instructions require accessing many pages (e.g., due to indirect addressing), so care must be taken to carefully allocate this minimum number for every instruction set.

- It is generally good to give each process more frames than this bare minimum number, so as to avoid having a high page fault rate. If the system has too many running processes, with too little memory to go around, then most processes spend a large portion of their time servicing page faults, and spend very little time doing actual work. This phenomenon is called **thrashing**. When a system thrashes, some processes must be terminated, so that the remaining processes can get more memory.

- While every process can be allocated a theoretical maximum of all available memory frames, most operating systems impose a limit. For example, one could have an **equal allocation** policy, where each of the N running processes get $1/N$-th of the total frames. Or one could have a **proportional allocation** policy, where each process gets frames in proportion to its memory requirement. Or, every process can get enough frames to accommodate its **working set**, which is defined as the number of memory pages being accessed in a certain specified time window in the past. Normally, processes execute in one region of their virtual address space for a while, before moving on to another region, and thus have a well defined working set. If the memory allocated to a process is large enough to accommodate its current working set, then the process will see very few page faults, except when moving across regions.

7.9 Kernel Memory Allocation

- All the memory management techniques described so far deal with user processes. However, the kernel itself would need to allocate and free up memory for its own functioning, e.g., for various kernel data structures. While it is alright to allocate memory at the granularity of pages for user space processes and risk internal fragmentation, the kernel must have a lower memory footprint, and hence must be more careful in allocating memory to itself. Further, some parts of the kernel are constrained to be in certain contiguous portions of memory (so that hardware devices can locate it). For all these reasons, kernel allocates memory to itself using mechanisms that are different from what it uses for user processes. The kernel can of course use the regular page-based memory allocation scheme when appropriate. However, for smaller memory requests, several other memory allocation schemes are available to the kernel.

- One of the older kernel memory allocation algorithms is called the **buddy allocation system**, where memory is always allocated in sizes that are powers of 2. For example, a 25KB memory request is satisfied with a 32KB block. With such allocation sizes, a free block could be split into smaller power-of-2 blocks, or adjacent blocks could be coalesced to form a bigger power-of-2 block, reducing the problem of memory fragmentation. However, the buddy system does suffer from the internal fragmentation problem.

- A more recent technique for kernel memory allocation that is used in Linux is called the **slab allocation**. This allocation scheme is particularly useful when allocating memory for various kernel objects. The kernel initially allocates *slabs*, which are contiguous pages of memory, and preallocates kernel objects in these slabs to form *caches*. Each type of kernel object has its own cache, which is built from a set of slabs. When a new object needs to be allocated, the kernel tries to find a free cache entry in one of the partially filled slabs. If none exists, a new empty slab is used. When an object is released, it is marked as free in the cache, and is available for immediate reallocation. The slab allocator does not cause any fragmentation by design. Memory requests can also be quickly created because kernel objects are preallocated and reused.