Virtual Memory

1 Introduction

In an operating system, it is possible that a program is too large to be loaded into the main memory. In theory, a 32-bit program may have a linear space of up to 4 giga bytes, which is larger than almost all computers nowadays. Thus we need some mechanism that allows the execution of a process that is not completely in main memory.

Overlay is one choice. With it, the programmers have to deal with swapping in and out themselves to make sure at any moment that the instruction to be executed next is physically in main memory. Obviously this brings a heavy burden on the programmers. In this chapter, we introduce another solution called *virtual memory*, which has been adopted by almost all modern operating systems.

Virtual memory refers to the technology in which some space in hard disk is used as an extension of main memory so that a user program need not worry if its size extends the size of the main memory. If that does happen, at any time only a part of the program will reside in main memory, and other parts will otherwise remain on hard disk and may be switched into memory later if needed.

This mechanism is similar to the two-level memory hierarchy we once discussed before, including cache and main memory because the *principle of locality* is also a basis here. With virtual memory, if a piece of process that is needed is not in a full main memory, then another piece will be swapped out and the former be brought in. If unfortunately, the latter is used immediately, then it will have to loaded back into main memory right away. As we know, the access to hard disk is time-consuming compared to the access to main memory, Thus the reference to the virtual memory space on hard disks will deteriorate the system performance significantly. Fortunately, the principle of locality holds. That is the instruction and data references during a short period tend to be bounded to one piece of process. So the access to hard disks will not be frequently requested and performed. Thus, the same principle, on the one hand, enables the caching mechanism to increase system performance, and on the other hand avoids the deterioration of performance with virtual memory.

With virtual memory, there must be some facility to separate a process into several pieces

so that they may reside separately either on hard disks or in main memory. Paging or/and segmentation are two methods that are usually used to achieve the goal.

2 Paging

As we stated in the previous chapter, for paging memory management, each process is associated with a page table. Each entry in the table contains the frame number of the corresponding page in the virtual address space of the process. This same page table is also the central data structure for virtual memory mechanism based on paging, although more facilities are needed.

2.1 Control bits

Since only some pages of a process may be in main memory, a bit in the page table entry, **P** in Figure 1 (a), is used to indicate whether the corresponding page is present in main memory or not.

Another control bit needed in the page table entry is a modified bit, **M**, indicating whether the content of the corresponding page have been altered or not since the page was last loaded into main memory. We often say swapping in and swapping out, suggesting that a process is typically separated into two parts, one residing in main memory and the other in secondary memory, and some pages may be removed from one part and join the other. They together make up of the whole process image. Actually the secondary memory contains the whole image of the process and part of it may have been loaded into main memory. When swapping out is to be performed, typically the page to be swapped out may be simply overwritten by the new page, since the corresponding page is already on secondary memory. However sometimes the content of a page may have been altered at runtime, say a page containing data. In this case, the alteration should be reflected in secondary memory. So when the **M** bit is 1, then the page to be swapped out should be written out.

Other bits may also be used for sharing or protection.

2.2 Multi-level page table

Typically, there is only one page table for each process, which is completely loaded into main memory during the execution of the process. However some processes may be so large that even its page table cannot be held fully in main memory. For example, in 32-bit x86 architecture, each process may have up to $2^{32} = 4$G bytes of virtual memory. With pages of $2^9 = 512$

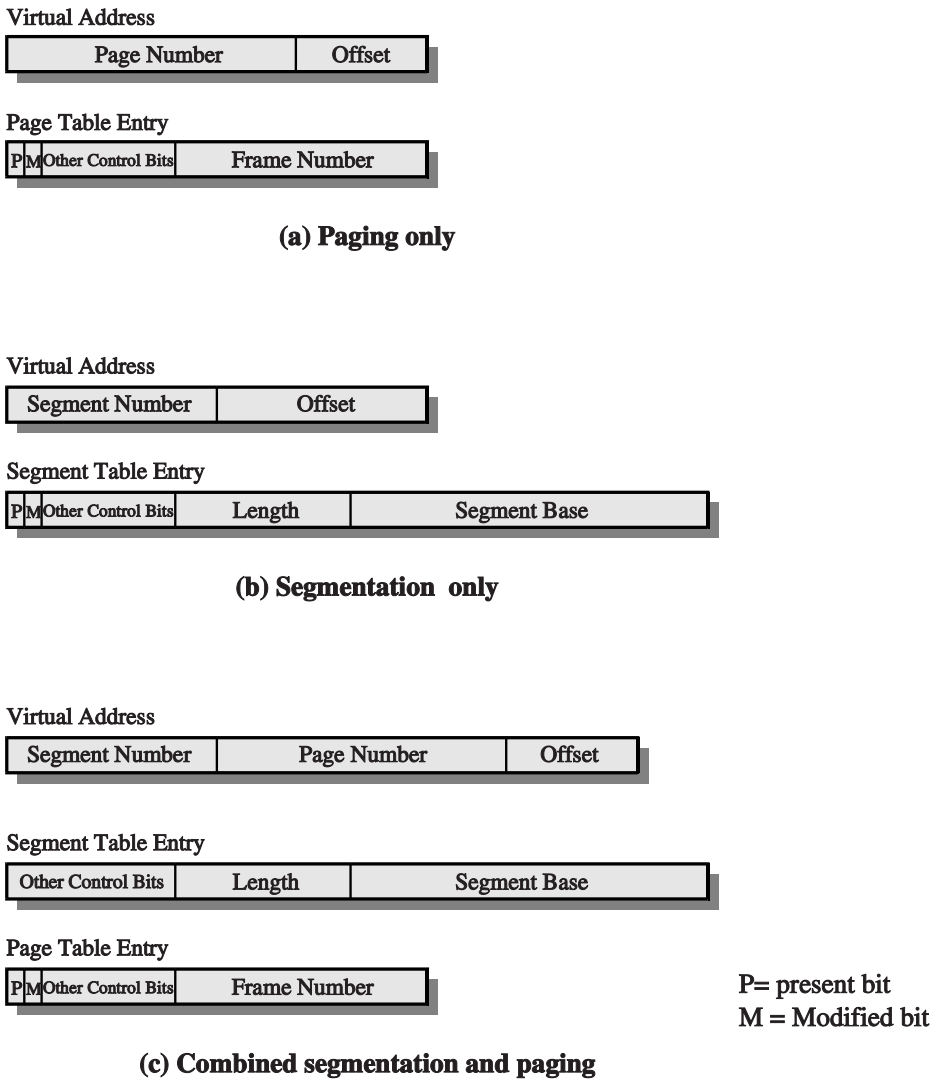

Figure 1: Typical memory management formats

bytes, as many as 2^23 pages are needed as well as a page table of 2^23 entries. If each entry requires 4 bytes, that will be $2^25 = 32$M bytes. Thus some mechanism is needed to allow only part of a page table is loaded in main memory. Naturally we use paging for this. That's page tables are subject to paging just as other pages are, called *multi-level paging*.

Figure 2 shows an example of a two-level scheme with a 32-bit address. If we assume 4K-byte pages, then 4G-byte virtual address is composed of 2^{20} pages. If each page table entry requires 4 bytes, then a user page table of 2^{20} entries requires 4M bytes. This huge page table itself needs 2^10 pages. For paging with it, a root page table of 2^{10} is needed, requiring 4K bytes.

With this two-level paging scheme, the root page table always remains in main memory. The

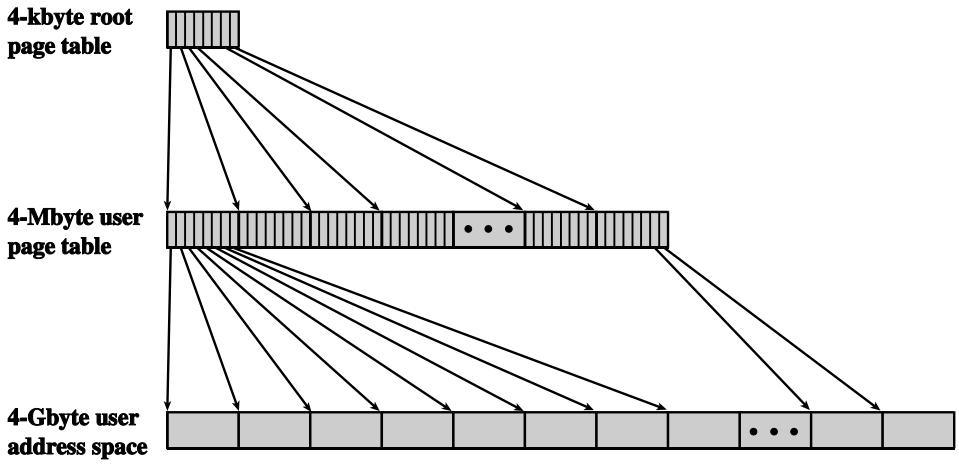

Figure 2: A two-level hierarchical page table

first 10 bits of a virtual address are used to index into the root page table to find an entry for a page of the user page table. If that page is not in main memory, a page fault occurs and the operating system is asked to load that page. If it is in main memory, then the next 10 bits of the virtual address index into the user page table to find the entry for the page that is referenced by the virtual address. This whole process is illustrated in Figure 3.

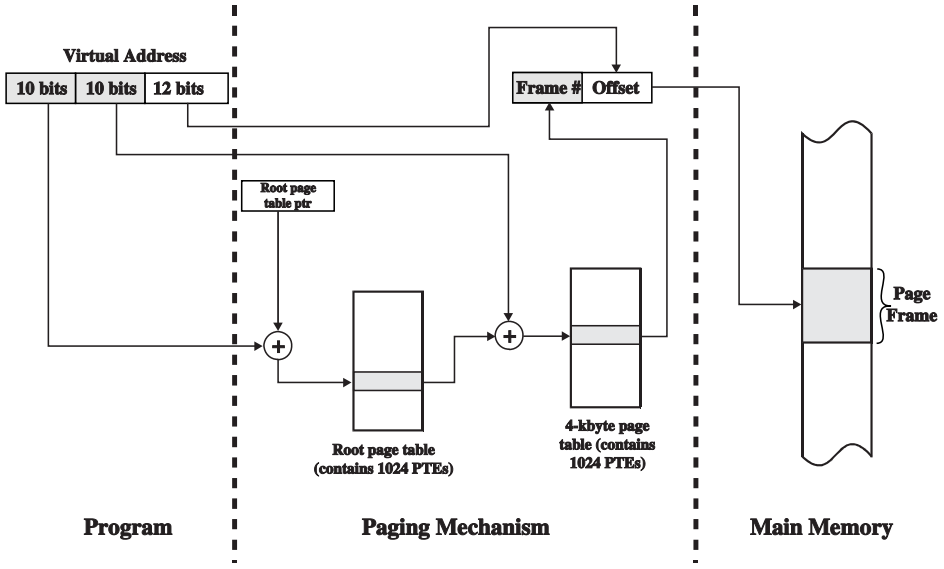

Figure 3: Address translation in A two-level paging system

2.3 Translation lookaside buffer

As we discussed before, a *translation lookaside buffer* (TLB) may be used to speed up paging and avoid frequent access to main memory, which is shown in Figure 4. With multi-level paging scheme, the benefit of TLB will be even more significant.

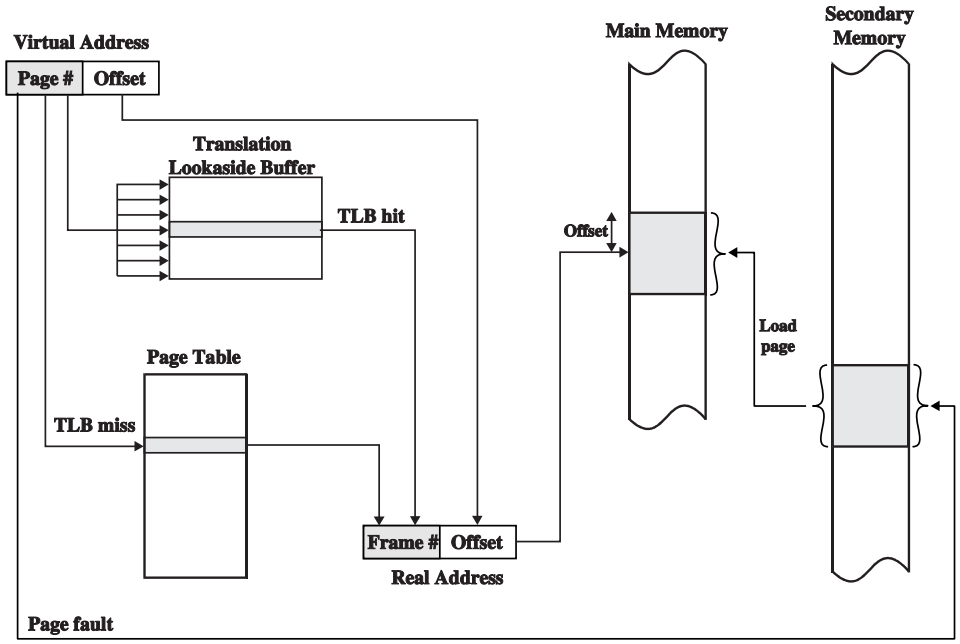

Figure 4: Use of a translation lookaside buffer

It should be noted that the TLB is a cache for a page table while the regular cache we mentioned before is for main memory and these facilities should work together when they are both present in a system. As figure 5 illustrates, for a virtual address consisting of a page number and an offset address, the memory system consults the TLB first to see if the matching page entry is present. If yes, the real address is generated by combining the frame number with the offset. If not, the entry is accessed from a page table. Once the real address is generated, the cache is consulted to see if the block containing that word is present. If so, it is returned to the CPU. If not, the word is retrieved from main memory.

3 Segmentation

Segmentation is another popular method for both memory management and virtual memory. It has a number of advantages:

1. It fits the programmers' need to organize the programs. With segmentation, a program

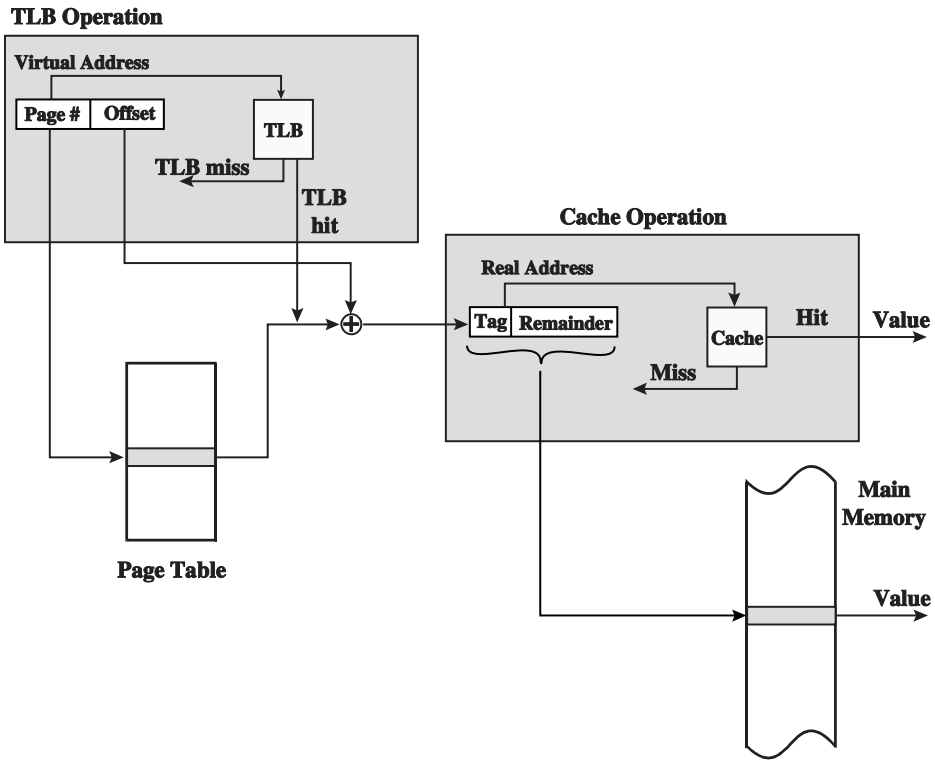

Figure 5: Translation lookaside buffer and cache operation

may be divided into several segments with each for a specific function or instructions relating to each other in some way.

2. Segmentation allows different pieces of a program to be compiled independently. Thus it is not needed to recompile and relink the whole program after a single revision is made.

3. Segmentation eases the control of sharing and protection.

Figure 6 shows the address translation in a segmentation system. Actually similar to paging, the same segment table is used for both virtual memory and main memory management, except that some same control bits are also needed.

4 Combined paging and segmentation

Both paging and segmentation have their strengths, so they are often combined to benefit from the advantages of both.

In a combined paging/segmentatino system, a user program is broken into a number of segments, at the discretion of the programmer, which is in turn broken up into a number of

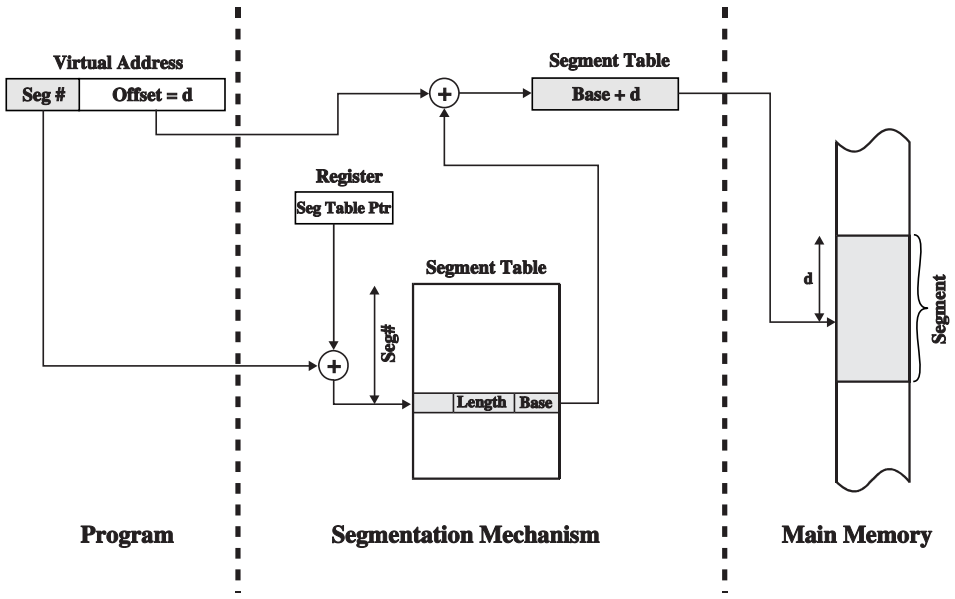

Figure 6: Address translation in a segmentation system

fixed-size pages. From the programmer's point of view, a logical address still consists of a segment number and a segment offset, as before in pure segmentation, while from the system's point of view the segment offset is viewed as a page number and a page offset for a page within the specified segment.

Figure 7 shows the address translation in the combined scheme. Each process is associated with a segment table and a number of page tables, one for each segment. For a running process, a register holds the starting address of the segment table for the process. Presented with a virtual address, the processor uses the segment number portion to index into the segment table to find the page table for that segment. Then the page number portion of the virtual address is used to index the page table and look up the corresponding frame number. It is then combined with the offset portion of the virtual address to produce the desired real address. Figure 1 (c) suggests the segment table entry and page table entry formats.

5 Operating system software

The operating system is the other part that is engaged in virtual memory implementation besides the hardware part in the microprocessor for address translation.

If we have a look at the popular operating systems over the history, we may find out that except for MS-DOS and specialized systems, all important operating systems provide virtual memory support. And they usually use paging or the combination of paging and segmentation for virtual memory subsystem. The reason why pure segmentation is seldom used is

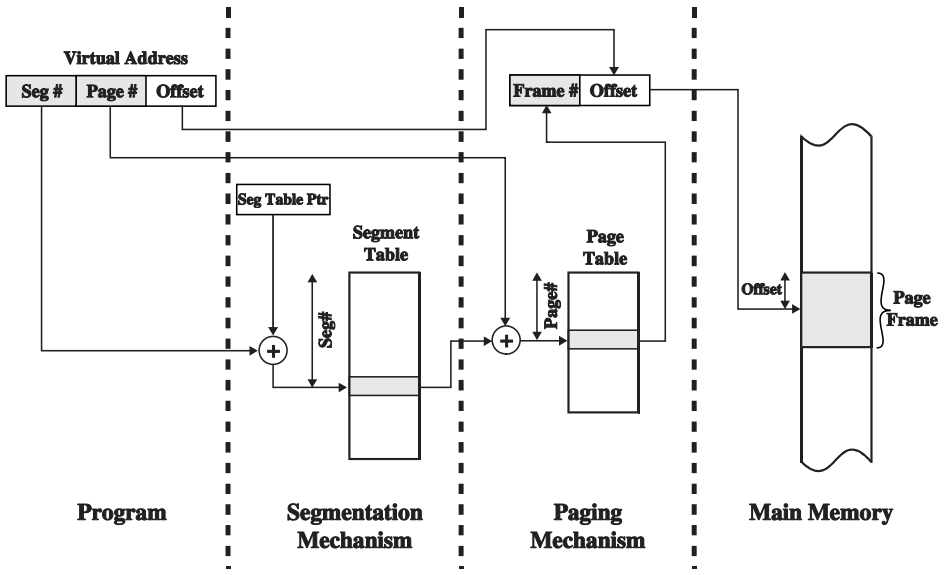

Figure 7: Address translation in a paging/segmentation system

that a segment in a program is still a large unit for memory management, which may lead to large external fragments in main memory.

Since paging is always an issue that the operating system designers have to face, this section will focus on the algorithms for paging. Even when segmentation mechanism is used as well, the segment level usually make sense for sharing or protection. The unit of size that is managed by virtual memory is always a page or a frame.

Before we proceed to various algorithms regarding virtual memory, we should first make it clear what the goal of such a kind of algorithm is. The goal is of course to enable the execution of processes whose sizes are larger than the size of physical main memory. However this is a trivial requirement for those algorithms. To distinguish them from each other, we actually need to tell which is better. Thus the key issue here is performance. In detail, we would like to minimize the rate at which page faults occur. A page fault occurs when a referenced page does not exist in main memory and needs to be brought in.

Why could the page fault rate be the key factor measuring the performance? Because without page faults, no access to secondary memory is needed. And page faults cause considerable software overhead, including deciding which page to replace, and the I/O operation for exchanging pages. Since I/O operations are slow, another process may have to be scheduled to run. The context switching also brings overhead more or less. Hence the goal of the intended algorithm is to minimize the rate of page fault, avoiding the overhead above as much as possible.

This section will discuss the issues involved when a page fault happens, including when

a page should be fetched from secondary memory to main memory, which pages should be replaced for free frames for demanded pages, and when a page should be sent back to secondary memory.

5.1 Fetch policy

The fetch policy determines when a page should be brought into main memory. There are two common alternatives: *demand paging* and *prepaging*.

With demand paging, a page will not be brought into main memory until a reference is made to a location on that page. If this is chosen by virtual memory subsystem, then when a process is first started, there will be a flurry of page faults. And with more and more pages brought in, page faults will less likely happen, since due to the principle of locality, the most recently visited pages will be most likely to be visited again. These pages are already in main memory.

With prepaging, pages may be brought in even when they are not requested. This method usually takes advantage of the characteristic of secondary memory devices. Take a disk as an example, where pages may be stored on a track consecutively. In this case, it is natural to read in pages along the track where the read head of the disk has to pass. This is often effective since related instructions are likely to be neighbors and thus stored adjacently.

Prepaging could be employed especially when a process is first started up, thus the frequent page fault at that stage may be avoided to some extent.

5.2 Placement policy

The placement policy determines where in real memory a process piece is to reside. This is an important issue only in pure segmentation systems, because with paging, a page is the unit for main memory and virtual memory management and placement is usually irrelevant because the address translation hardware and the main memory access hardware can perform their functions for any page-frame combination with equal efficiency.

With pure segmentation, a segment requires a contiguous space whose size is at least same as the size of the segment. Some simple algorithms have been proposed for the job, such as best-fit, first-fit, etc.. *Best-fit* algorithm searches for the smallest contiguous section of memory space that is big enough to hold the segment. *First-fit* algorithm, beginning with the start point of the main memory, searches for the first of those sections that are big enough. Experiments have shown that the first-fit algorithm is not only the simplest but usually the best and fastest as well.

5.3 Replacement policy

The replacement policy deals with the selection of a page in memory to be replaced after a page fault occurs and a new page must be brought in. The goal of any replacement algorithm includes two aspects: (1) The algorithm itself should be simple to implement and efficient to run; and (2) the selection of page should not harm the performance of the virtual system as a whole, or more specifically, the page that is removed should be the page least likely to be referenced in the near future.

5.3.1 Algorithms

Replacement algorithms that have been considered in the literature include:

- **Optimal**

  If we know which page is least likely to be referenced later on, then we simply select that page, which is undoubtedly the optimal solution. However unfortunately this is not realistic because we cannot know exactly what is going to happen in the future. The value of discussing this algorithm is that it may be a benchmark to evaluate the performance of other algorithms.

  Suppose we have a process, which is made up of 5 pages, and 3 frames in main memory are allocated for this process. And we already know the references to those pages are in the order:

  ```
  2 3 2 1 5 2 4 5 3 2 5 2
  ```

  Figure 8 shows how the optimal algorithm works on this example step by step. As we can see from Figure 8, 3 page faults are produced during the process.

- **Least recently used (LRU)**

  Although we do not exactly know the future, we can predict the future to some extent based on the history. Based on the principle of locality, the page that has not been used for the longest time, is also least likely to be referenced. The LRU algorithm thus selects that page to be replaced. And experience tells that the LRU policy does nearly as well as the optimal policy. However since the decision making is based on the history, the system has to keep the references that have been made all the way from the beginning of the execution. The overhead would be tremendous. With the above example, Figure 8 shows there are 4 page faults in the case of the LRU algorithm.

- **First-in-first-out (FIFO)**

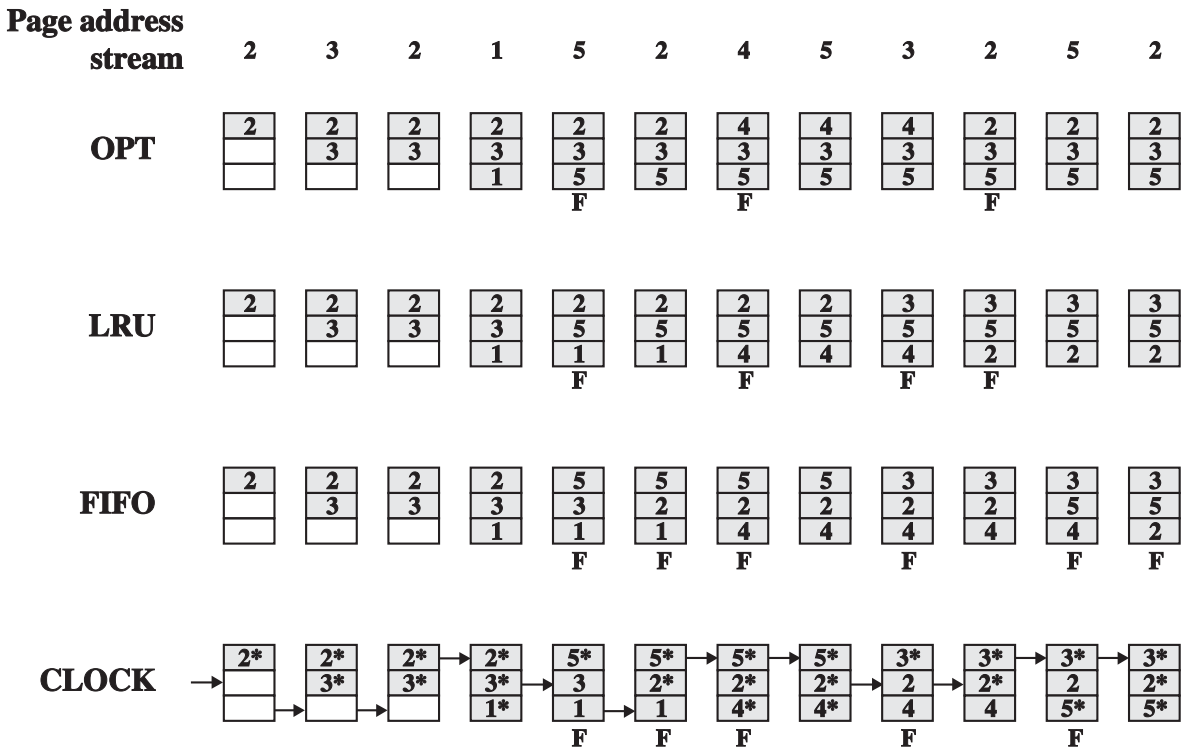

Figure 8: Behavior of four page replacement algorithms

The FIFO policy treats the page frames allocated to a process as a circular buffer, and pages are removed in round-robin style. It may be viewed as a modified version of the LRU policy, and this time instead of the least recently used, the earliest used page is replaced since the page that has resided in main memory for the longest time will also least likely used in the future. This logic may be wrong sometimes if some part of the program is constantly used, which thus may lead to more page faults. The advantage of this policy is that it is one of the simplest page replacement policies to implement, since all that is needed is a pointer that circles through the page frames of the process. As Figure 8 shows, 6 page faults occurs if the FIFO policy is used upon the same reference stream.

- **Clock**

  Each of the above policies has its advantages and disadvantages. Some may need less overhead, and some may produce better results. Thus here is an issue of balance. People have proposed all kinds of algorithms based on different considerations of balance between overhead and performance. Among them, the clock policy is one of the most popular ones.

  The clock policy is basically a variant of the FIFO policy, except that it also considers to some extent the last accessed times of pages by associating an additional bit with each

frame, referred to as the *use bit*. And when a page is referenced, its use bit is set to 1.

As Figure 9 illustrates, the set of frames that might be selected for replacement is viewed as a circular buffer, with which a pointer is associated. When a free frame is needed but not available, the system scans the buffer to find a frame with a use bit of 0 and the first frame of this kind will be selected for replacement. During the scan, whenever a frame with a use bit of 1 is met, the bit is reset to 0. Thus if all the frames have a use bit of 1, then the pointer will make a complete cycle through the buffer, setting all the use bits to 0, and stop at its original position, replacing the page in that frame. And after a replacement is made, the pointer is set to point to the next frame in the buffer.

Figure 9 gives an example of this clock policy. Figure 9 shows the status of the buffer at some moment. Suppose page 727 is referenced but is not available in the buffer which is already full. Thus a page fault occurs and a page needs to be replaced. According to the clock policy, page 556 is replaced. The other updates include the use bits of frame 2 and 3 are set to 0 and the pointer is made pointing to frame 5.

Assume that the number of frames assigned to a process is fixed, experiments have shown that the performance of all the above 4 algorithms are significantly different when the number of frames is small. However when much more frames are allocated, the performance of the clock policy is almost as good as the LRU policy.

A variant of the clock policy additionally associates a modify bit with every frame in main memory, indicating if the page in the frame has been modified or not since it was last loaded into main memory. This bit is included because a modified page has to be written out to secondary memory before overwritten by a new page. With both the use bit and the modify bit considered, any page may fall into one of the following four categories:

- $u = 0$ and $m = 0$: not accessed recently and not modified
- $u = 1$ and $m = 0$: accessed recently and not modified
- $u = 0$ and $m = 1$: not accessed recently and modified
- $u = 0$ and $m = 1$: accessed recently and modified

Accordingly, the algorithm goes as follows:

1. Scan the frame buffer and select the first frame with $u = 0$ and $m = 0$ if any.
2. If step 1 fails, scan again, look for a frame with $u = 0$ and $m = 1$ and select it for replacement if such a frame exists. During the scan, the use bits of 1 are set to 0.
3. If step 2 fails, all the use bits in the buffer are 0. Repeat step 1 and if necessary step 2. This time, a frame will be definitely chosen to be replaced.

Note that the recently accessed pages are given higher priority than those that have been modified, based on the consideration that though the latter need to be written

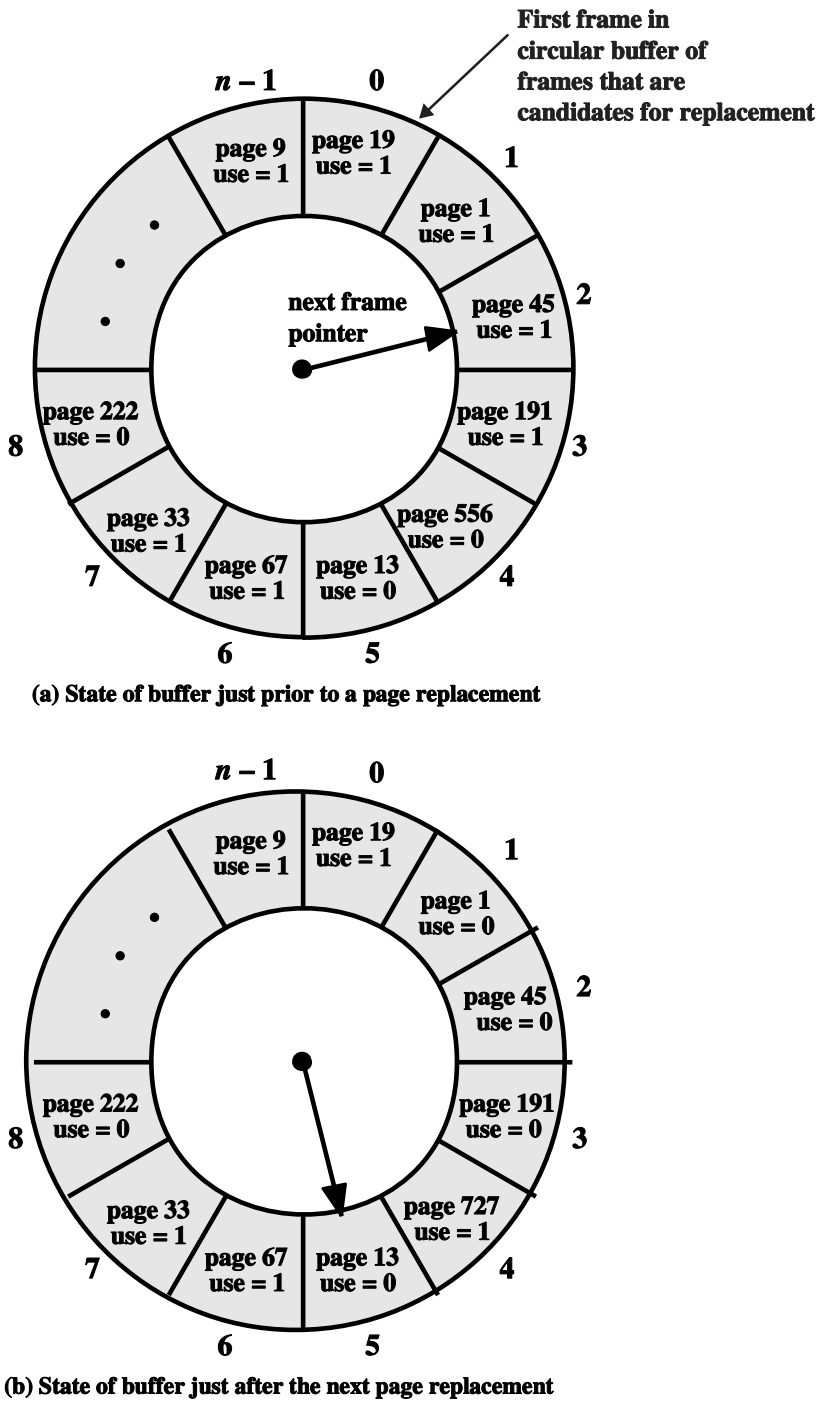

Figure 9: Example of clock policy operation

back to secondary memory first before replacement, they still involve less overhead than probable reloading a recently accessed but replaced page.

5.3.2 Page buffering

Once a page is selected for replacement, it may be either written back to secondary memory if needed or simply thrown away, which seems obvious to us so far. However there is still space to improve the performance at this step. For example, the VAX VMS machine has used page buffering, in which the replaced pages will first go to one of two page lists in main memory, depending whether it has been modified or not. The advantage of this scheme is that if in a short time, the page is referenced again, it may be obtained with little overhead. When the length of the lists surpasses a specific limit, some of the pages that went into the lists earlier will be removed and written back to secondary memory. Thus the two lists actually act as a cache of pages. Another advantage is the modified pages may be written out in cluster rather than one at a time, which significantly reduces the number of I/O operations and therefore the amount of time for disk access.

5.4 Cleaning policy

A cleaning policy is the opposite of a fetch policy. It deals with when a modified page should be written out to secondary memory. There are two common choices:

- **Demand cleaning**: A page is written out only when it has been selected for replacement.
- **Precleaning**: Modified pages are updated on secondary memory before their page frames are needed so that pages can be written out in batches.

Precleaning has advantage over demand cleaning but it cannot be performed too frequently because some pages may be modified so often that frequent writing out turns out to be unnecessary.

5.5 Frame locking

One point that is worth mentioning is that some of the frames in main memory may not be replaced, or may be locked. For example, the frames occupied by the kernel of the operating system, used for I/O buffers and other time-critical areas should always be available in main memory for the operating system to operate properly. This requirement can be satisfied by adding an additional bit in the page table.

5.6 Load control

Another related question is how many processes may be started to run and reside in main memory simultaneously, which is called *load control*. Load control is critical in memory management because, if too few processes are in main memory at any one time, it will be very likely for all the processes to be blocked, and thus much time will be spent in swapping. On the other hand, if too many processes exist, each individual process will be allocated a small number of frames,and thus frequent page faulting will occur. Figure 10 shows that if all other aspects are given, there is a specific point to achieve the highest utilization.

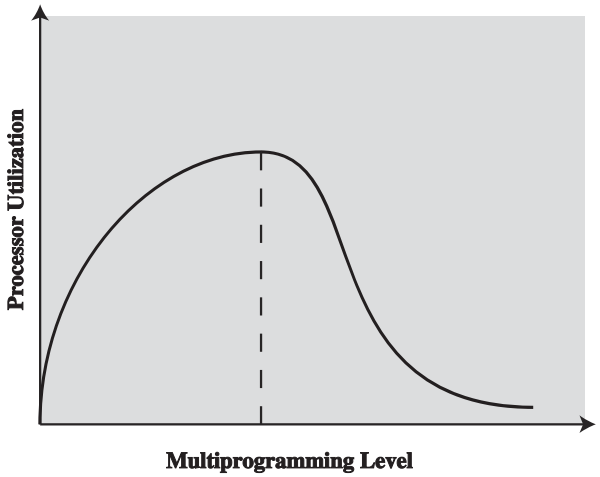

Figure 10: Multiprogramming effects