Chapter 8 Details of Page-based Memory Management
(Revision number 20)

In this chapter, we will focus on page-based memory management. This is fundamental to pretty much all processors and operating systems that support virtual memory. As we mentioned already, even processors that support segmentation (such as Intel Pentium) use paging to combat external fragmentation.

8.1 Demand Paging

As we mentioned in Chapter 7, the memory manager sets up the page table for a process on program startup. Let us first understand what fraction of the entire memory footprint of the program the memory manager should allocate at program startup. Production programs contain the functional or algorithmic logic part of the application as well as the non-functional parts that deal with erroneous situations that may occasionally arise during program execution. Thus, it is a fair expectation that any well-behaved program will need only a significantly smaller portion of its entire memory footprint for proper execution. Therefore, it is prudent for the memory manager not to load the entire program into memory on startup. This requires some careful thinking, and understanding of what it means to execute a program that may not be completely in memory. The basic idea is to load parts of the program that are not in memory *on demand*. This technique, referred to as *demand paging* results in better memory utilization.

Let us first understand what happens in hardware and in software to enable demand paging.

8.1.1 Hardware for demand paging

In Chapter 7 (please see Section 7.4.2), we mentioned that the hardware fetches the PFN from the page table as part of address translation. However, with demand paging the page may not be in memory yet. Therefore, we need additional information in the page table to know if the page is in memory. We add a *valid* bit to each page table entry. If the bit is 1 then the PFN field in this entry is valid; if not, it is invalid implying that the page is not in memory. Figure 8.1 shows the PTE to support demand paging. The role of the hardware is to recognize that a PTE is invalid and help the operating system take corrective action, namely, load the missing page into memory. This condition (a PTE being invalid) is an unintended program interruption since it is no fault of the original program. The operating system deliberately decided not to load this part of the program to conserve memory resources. This kind of program interruption manifests as a *page fault exception or trap*.

Figure 8.1: Page Table Entry

The operating system handles this fault by bringing in the missing page from the disk. Once the page is in memory, the program is ready to resume execution where it left off. Therefore, to support demand paging the processor has to be capable of restarting an instruction whose execution has been suspended in the middle due to a page fault.

Figure 8.2 shows the processor pipeline. The IF and MEM stages are susceptible to a page fault since they involve memory accesses.

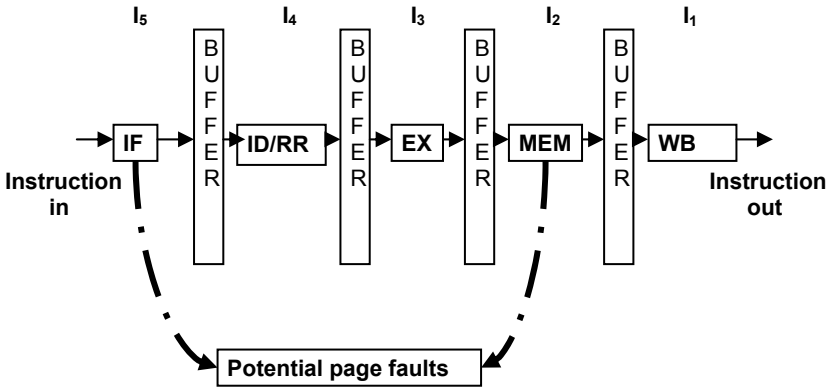


Figure 8.2: Potential page faults in a processor pipeline

Hardware for instruction re-start: Let us first understand what should happen in the hardware. Consider that instruction I_2 in the MEM has a page fault. There are several instructions in partial execution in the pipeline. Before the processor goes to the INT state to handle the interrupt (refer to Chapter 4 for details on how interrupts are handled in hardware in the INT state), it has to take care of the instructions in partial execution already in the pipeline. In Chapter 5, we briefly mentioned the action that a pipelined processor may take to deal with interrupts. The page fault exception that I_2 experiences in the MEM stage is no different. The processor will let I_1 to complete (draining the pipe) and squash (flushing the pipeline) the partial execution of instructions I_3-I_5 before entering the INT state. The INT state needs to save the PC value corresponding to instruction I_2 for re-starting the instruction after servicing the page fault. Note that there is no harm in flushing instructions I_3-I_5 since they have not modified the permanent state of the program (in processor registers and memory) so far. One interesting and important side effect manifesting from page faults (as well as any type of exceptions): The pipeline registers (shown as buffers in Figure 8.2) contain the PC value of the instruction in the event there is an exception (arithmetic in the EX stage or page fault in the MEM stage) while executing this instruction. We discussed the hardware ramifications for dealing with traps and exceptions in a pipelined processor in Chapter 5.

8.1.2 Page fault handler

The page fault handler is just like any other interrupt handler that we have discussed in earlier chapters. We know the basic steps that any handler has to take (saving/restoring state, etc.) that we have seen already in Chapter 4. Here, we will worry about the actual work that the handler does to service the page fault:

1. Find a free page frame

2. Load the faulting virtual page from the disk into the free page frame
3. Update the page table for the faulting process
4. Place the PCB of the process back in the ready queue of the scheduler

We will explore the details of these steps in the next subsection.

8.1.3 Data structures for Demand-paged Memory Management

Now, we will investigate the data structures and algorithms needed for demand paging. First, let us look at the data structures. We already mentioned that the page table is a per-process data structure maintained by the memory manager. In addition to the page tables, the memory manager uses the following data structures for servicing page faults:

1. **Free-list of page frames:** This is a data structure that contains information about the currently unused page frames that a memory manager uses to service a page fault. The free-list does not contain the page frames themselves; each node of the free-list simply contains the page frame number. For example (referring to Figure 8.3), page frames 52, 20, 200, …, 8 are currently unused. Therefore, the memory manager could use any page frame from this list to satisfy a page fault. To start with when a machine boots up, since there are no user processes, and the free-list contains all the page frames of the user space. The free-list shrinks and grows as the memory manager allocates and releases memory to satisfy the page faults of processes.

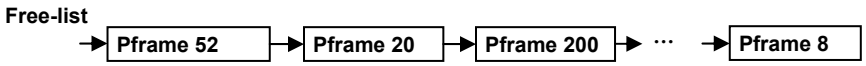


Figure 8.3: Free-list of page frames

2. **Frame table (FT):** This is a data structure that contains the reverse mapping. Given a frame number, it gives the Process ID (PID) and the virtual page number that currently occupies this page frame (Figure 8.4). For example, page frame 1 is currently unallocated while the virtual page 0 of Process 4 currently occupies page frame 6. It will become clear shortly how the memory manager uses this data structure.

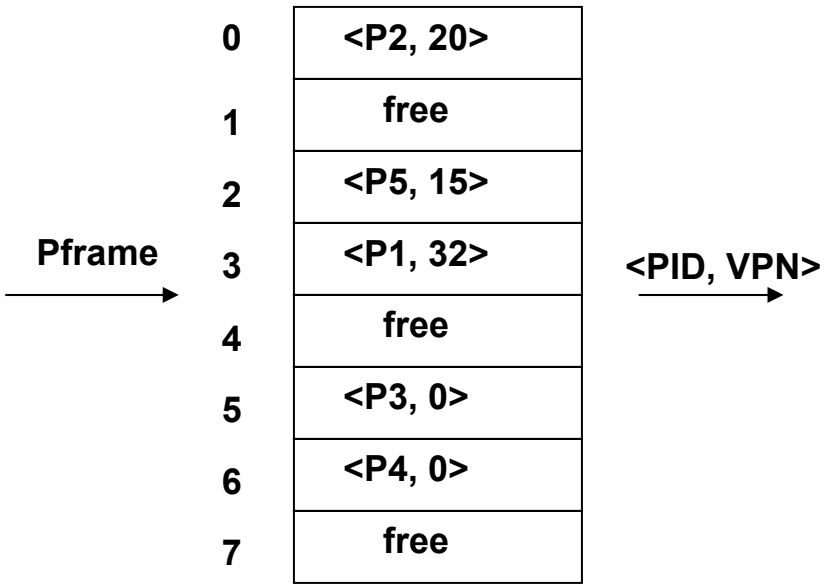


Figure 8.4: Frame Table

3. **Disk map (DM):** This data structure maps the process virtual space to locations on the disk that contain the contents of the pages (Figure 8.5). This is the disk analog of the page table. There is one such data structure for each process.

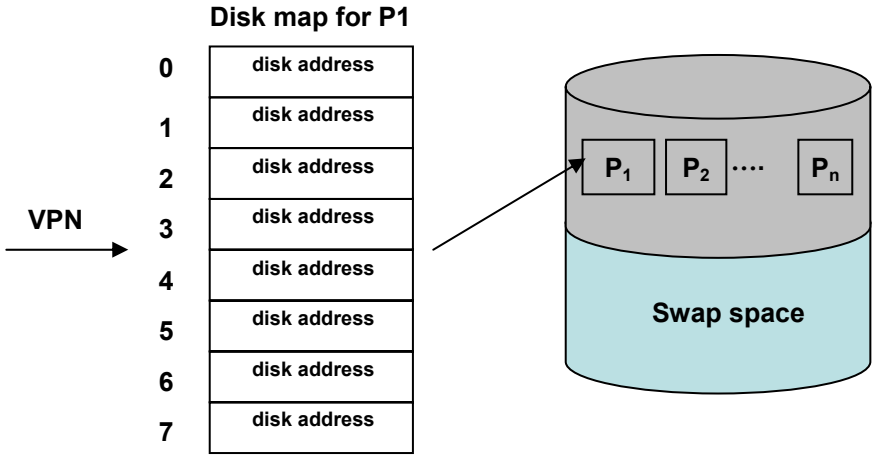


Figure 8.5: Disk map for Process P1

For clarity of discussion, we have presented each of the above data structures as distinct from one another. Quite often, the memory manager may coalesce some of these data structures for efficiency; or have common data structures with multiple views into them to simulate the behavior of the above functionalities. This will become apparent when we discuss some of the page replacement policies (see Section 8.3).

8.1.4 Anatomy of a Page Fault

Let us re-visit the work done by the page fault handler upon a page fault using the data structures above.

1. **Find a free page frame:** The page fault handler (which is part of the memory manager) looks up the *free-list* of page frames. If the list is empty, we have a problem. This means that all the physical frames are in use. However, for the faulting process to continue execution, the memory manager has to bring the faulting page from the disk into physical memory. This implies that we have to make room in the physical memory to bring in the faulting page. Therefore, the manager selects some physical frame as a *victim* to make room for the faulting page. We will describe the policy for selecting a victim in Section 8.3.
2. **Pick the victim page:** Upon selecting a victim page frame, the manager determines the victim process that currently owns it. The *frame table* comes in handy to make this determination. Here, we need to make a distinction between a *clean* page and a *dirty* page. A clean page is one that has not been modified by the program from the time it was brought into memory from the disk. Therefore, the disk copy and the memory copy of a clean page are identical. On the other hand, a dirty page is one that has been modified by the program since bringing from the disk. If the page is clean, then all that the manager needs to do is to set the PTE corresponding to the page as *invalid*, i.e., the contents of this page need not be saved. However, if the page is dirty, then the manager writes the page back to the disk (usually referred to as *flushing* to the disk), determining the disk location from the *disk map* for the victim process.
3. **Load the faulting page:** Using the *disk map* for the faulting process, the manager reads in the page from the disk into the selected page frame.
4. **Update the page table for the faulting process and the frame table:** The manager sets the mapping for the PTE of the faulting process to point to the selected page frame, and makes the entry *valid*. It also updates the *frame table* to indicate the change in the mapping to the selected page frame.
5. **Restart faulting process:** The faulting process is ready to be re-started. The manager places the PCB of the faulting process in the scheduler's ready queue.

Example 5:

Let us say process P1 is currently executing, and experiences a page fault at VPN = 20. The *free-list* is empty. The manager selects page frame PFN = 52 as the victim. This frame currently houses VPN = 33 of process P4. Figures 8.6a and 8.6b show the *before-after* pictures of the *page tables* for P1 and P2 and the *frame table* as a result of handling the page fault.

Answer:

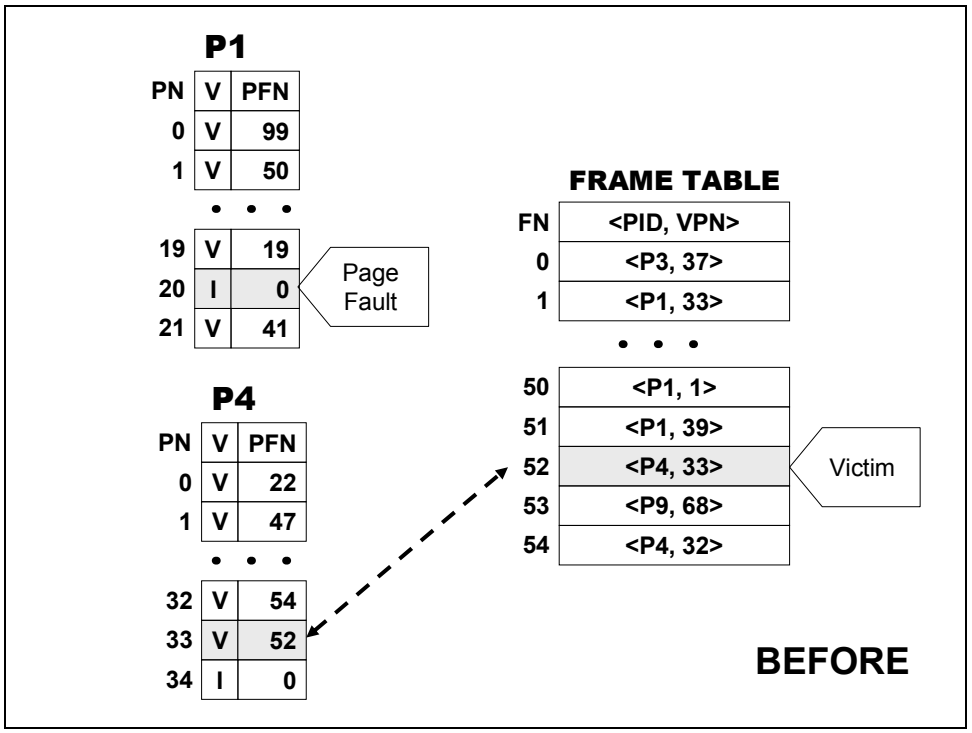


Figure 8.6a: **Process P1 is executing and has a page fault on VPN 20**

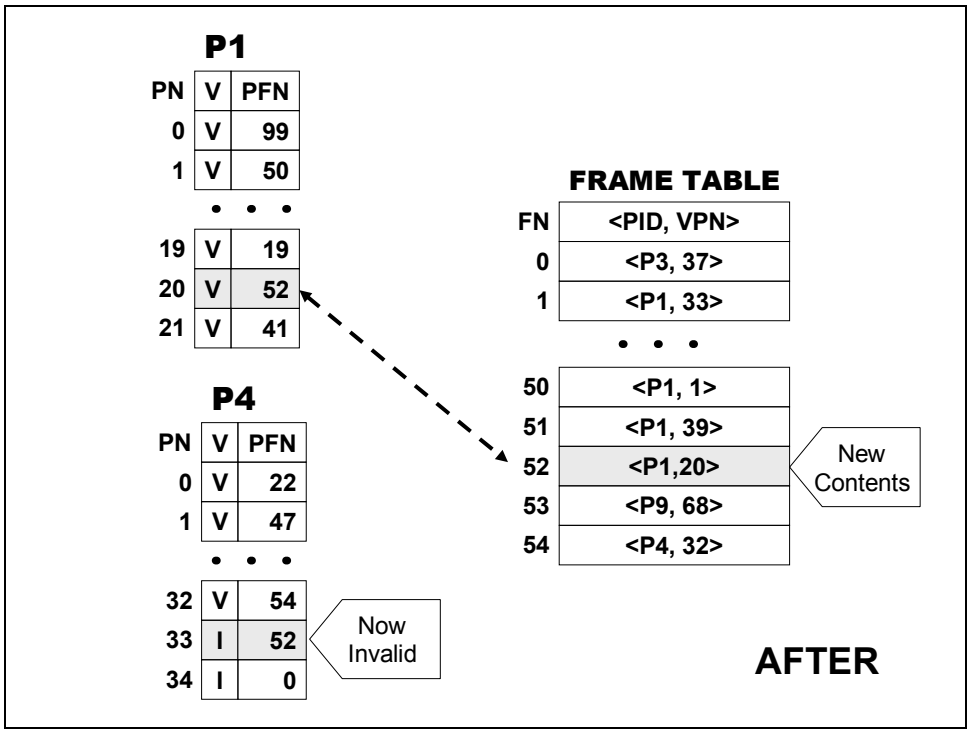


Figure 8.6b: Page fault service for Process P1 complete

Example 6:

Given the before picture of the page manager's data structure as shown below, show the contents of the data structures after P1's page fault at VPN = 2 is serviced. The victim page frame chosen by the page replacement algorithm is PFN = 84. Note that only relevant entries of the Frame Table are shown in the Figures.

BEFORE THE PAGE FAULT

	PFN	V
VPN = 0	50	V
VPN = 1	52	V
VPN = 2	--	I
VPN = 3	60	V

P1's PT

	PFN	V
VPN = 0	70	V
VPN = 1	80	V
VPN = 2	85	V
VPN = 3	84	V

P2's PT

0	....
	....
50	<P1, 0>
52	<P1, 1>
60	<P1, 3>
70	<P2, 0>
80	<P2, 1>
84	<P2, 3>
85	<P2, 2>

Frame Table

SHOW THE CONTENTS OF THE DATA STRUCTURES AFTER THE PAGE FAULT FOR P1 at VPN = 2.

Answer:

	PFN	V
VPN = 0	50	V
VPN = 1	52	V
VPN = 2	84	V
VPN = 3	60	V

P1's PT

	PFN	V
VPN = 0	70	V
VPN = 1	80	V
VPN = 2	85	V
VPN = 3	--	I

P2's PT

0	....
	....
50	<P1, 0>
52	<P1, 1>
60	<P1, 3>
70	<P2, 0>
80	<P2, 1>
84	<P1, 2>
85	<P2, 2>

Frame Table

Remember that the page fault handler is also a piece of code just like any other user process. However, we cannot allow the page fault handler itself to page fault. The operating system ensures that certain parts of the operating system such as the page fault handler are always memory resident (i.e., never paged out of physical memory).

Example 7:

Five of the following seven operations take place upon a page fault when there is no free frame in memory. Put the five correct operations in the right temporal order and identify the two incorrect operations.

---

a) use the frame table to find the process that owns the faulting page
b) using the disk map of faulting process, load the faulting page from the disk into the victim frame
c) select a victim page for replacement (and the associated victim frame)
d) update the page table of faulting process and frame table to reflect the changed mapping for the victim frame
e) using the disk map of the victim process, copy the victim page to the disk (if dirty)
f) look up the frame table to identify the victim process and invalidate the page table entry of the victim page in the victim page table
g) look up if the faulting page is currently in physical memory

Answer:

Step 1: c
Step 2: f
Step 3: e
Step 4: b
Step 5: d ***
(*** Note: It is OK if this is shown as step 3 or 4 so long as the other steps have the same relative order)

Operations (a) and (g) do not belong to page fault handling.

8.2 Interaction between the Process Scheduler and Memory Manager

Figure 8.7 shows the interaction between the CPU scheduler and the memory manager. At any point of time, the CPU is executing either one of the user processes, or one of the subsystems of the operating system such as the CPU scheduler, or the memory manager. The code for the scheduler, the memory manager, and all the associated data structures are in the kernel memory space (Figure 8.7). The user processes live in the user memory space. Once the CPU scheduler dispatches a process, it runs until one of the following events happen:

1. The hardware timer interrupts the CPU resulting in an upcall (indicated by 1 in the figure) to the CPU scheduler that may result in a process context switch. Recall that we defined an upcall (in Chapter 6) as a function call from the lower levels of the

system software to the higher levels. The CPU scheduler takes the appropriate action to schedule the next process on the CPU.

2. The process incurs a page fault resulting in an upcall (indicated by 2 in the figure) to the memory manager that results in page fault handling as described above.
3. The process makes a system call (such as requesting an I/O operation) resulting in another subsystem (not shown in the figure) getting an upcall to take the necessary action.

While these three events exercise different sections of the operating system, all of them share the PCB data structure, which aggregates the current state of the process.

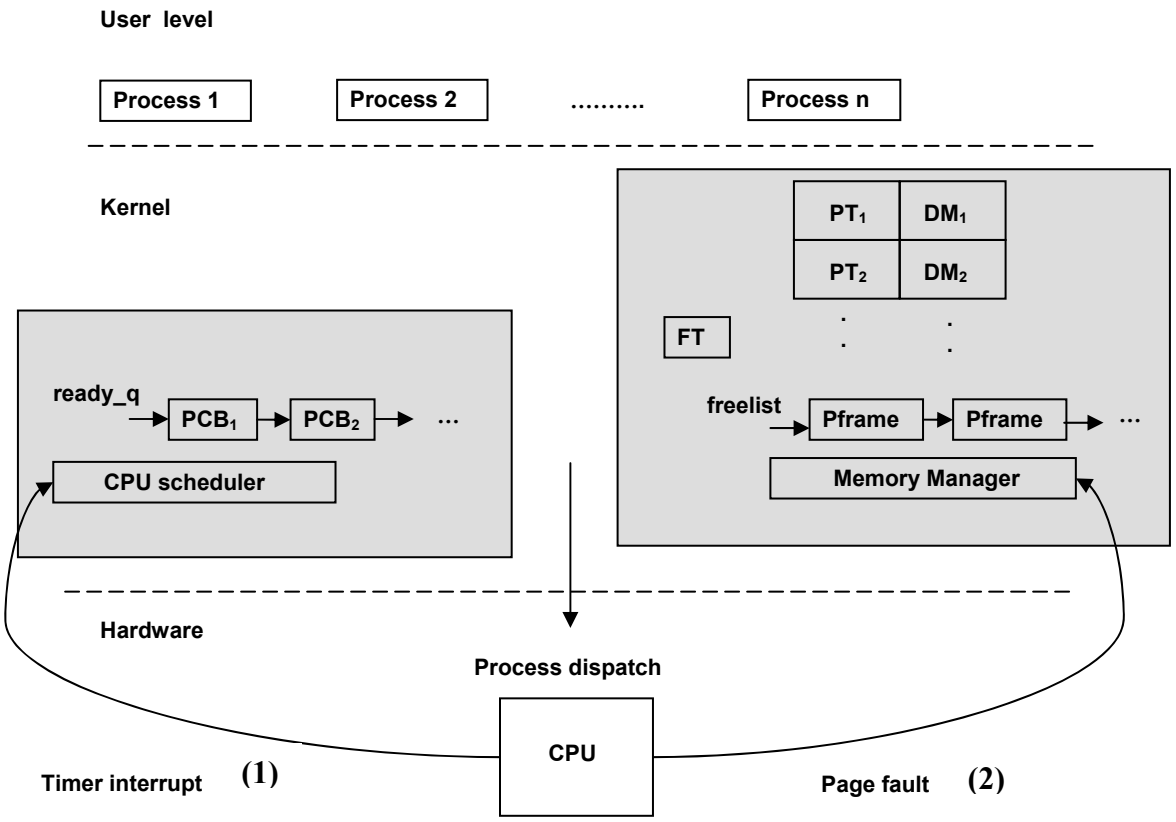


Figure 8.7: Interaction between the CPU scheduler and the memory manager

8.3 Page Replacement Policies

Now we will discuss how to pick a victim page to evict from the physical memory when there is a page fault and the free-list is empty. The process of selecting a victim page to evict from the physical memory is referred to as the *page replacement policy*. Let us understand the attributes of a good page replacement policy.

1. For a given string of page references, the policy should result in the least number of page faults. This attribute ensures that the amount of time spent in the operating system dealing with page faults is minimized.

2. Ideally, once a particular page has been brought into physical memory, the policy should strive not to incur a page fault for the same page again. This attribute ensures that the page fault handler respects the reference pattern of the user programs.

In selecting a victim page, the memory manager has one of two options:

- **Local victim selection:** The idea is to steal a physical frame from the faulting process itself to satisfy the request. Simplicity is the main attraction for this scheme. For example, this scheme eliminates the need for a *frame table*. However, this scheme will lead to poor memory utilization.
- **Global victim selection:** The idea is to steal a physical frame from any process, not necessarily the faulting one. The exact heuristic used to determine the victim process and the victim page depends on the specifics of the algorithm. This scheme will result in good utilization due to the global selection of the victim page.

The norm is to use global victim selection to increase memory utilization. Ideally, if there are no page faults the memory manager never needs to run, and the processor executes user programs most of the time (except for context switching). Therefore, *reducing the page fault rate* is the goal of any memory manager. There are two reasons why a memory manager strives to reduce the page fault rate: (1) the performance of a program incurring a page fault is adversely affected since bringing in a page from secondary storage is slow, and (2) precious processor cycles should not be used too often in overhead activities such as page replacement.

In the rest of the discussion, the presumption is global page replacement policy though the examples will focus on the paging behavior of a single process to keep the discussion simple. The memory manager bases its victim selection on the paging activity of the processes. Therefore, whenever we refer to a page we mean a virtual page. Once the memory manager identifies a virtual page as a victim, then the physical frame that houses the virtual page becomes the candidate for replacement. For each page replacement policy, we will identify the hardware assist (if any) needed, the data structures needed, the details of the algorithm, and the expected performance in terms of number of page-faults.

8.3.1 Belady's Min

Ideally, if we know the entire string of references, we should replace the one that is not referenced for the *longest time in the future*. This replacement policy is not feasible to implement since the memory manager does not know the future memory references of a process. However, in 1966 Laszlo Belady proposed this *optimal replacement algorithm*, called *Belady's Min*, to serve as a gold standard for evaluating the performance of any page replacement algorithm.

8.3.2 Random Replacement

The simplest policy is to replace a page randomly. At first glance, this may not seem like a good policy. The upside to this policy is that the memory manager does not need any hardware support, nor does it have to keep any bookkeeping information about the current set of pages (such as timestamp, and referential order). In the absence of

knowledge about the future, it is worthwhile understanding how bad or good a random policy would perform for any string of references. Just as Belady's Min serves as an upper bound for performance of page replacement policies, a random replacement algorithm may serve as a lower bound for performance. In other words, if a page replacement policy requires hardware support and/or maintenance of bookkeeping information by the memory manager, then it should perform better than a random policy as otherwise it is not worth the trouble. In practice, memory managers may default to random replacement in the absence of sufficient bookkeeping information to make a decision (see Section 8.3.4.1).

8.3.3 First In First Out (FIFO)

This is one of the simplest page replacement policies. FIFO algorithm works as follows:

- Affix a timestamp when a page is brought in to physical memory
- If a page has to be replaced, choose the *longest resident* page as the victim

It is interesting to note that we do not need any hardware assist for this policy. As we will see shortly, the memory manager remembers the order of arrival of pages into physical memory in its data structures.

Let us understand the data structure needed by the memory manager. The memory manager simulates the "timestamp" using a *queue* for recording the order of arrival of pages into physical memory. Let us use a circular queue (Figure 8.8) with a *head* and a *tail* pointer (initialized to the same value, say 0). We insert at the tail of the queue. Therefore, the longest resident page is the one at the head of the queue. In addition to the queue, the memory manager uses a *full* flag (initialized to *false*) to indicate when the queue is full. The occupancy of the queue is the number of physical frames currently in use. Therefore, we will make the queue size equal the number of physical frames. Each index into the circular queue data structure corresponds to a unique physical frame. Initially, the queue is empty (*full* flag initialized to *false*) indicating that none of the physical frames are in use. As the memory manager demand-pages in the faulting pages, it allocates the physical frames to satisfy the page faults (incrementing the tail pointer after each allocation). The queue is full (*full* flag set to *true*) when there are no more page frames left for allocation (*head* and *tail* pointers are equal). This situation calls for page replacement on the next page fault. The memory manager replaces the page at the *head,* which is the longest resident page. The interesting point to note is that this one data structure, *circular queue*, serves the purpose of the *free list* and the *frame table* simultaneously.

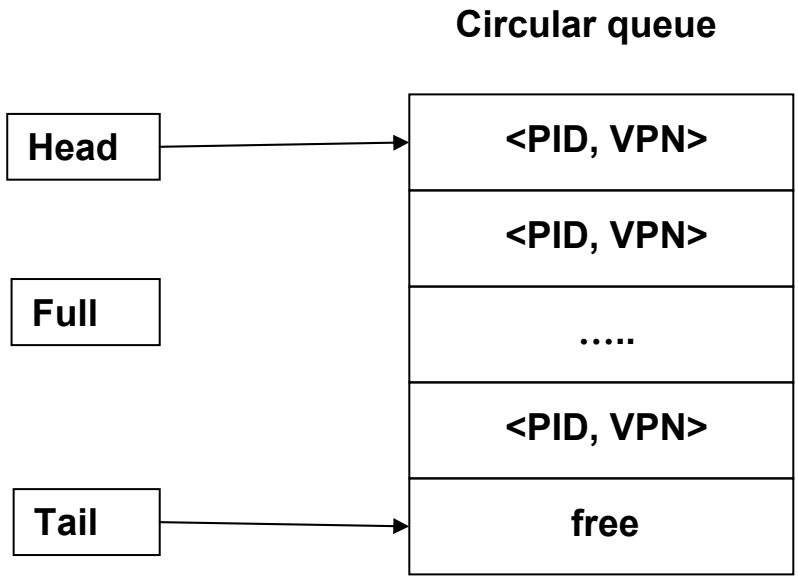


Figure 8.8: Circular queue for FIFO page replacement algorithm. Tail points to first free physical frame index. Each queue entry corresponds to a physical page frame, and contains the <PID, VPN> housed in that frame. Head contains the longest resident page.

Example 8:

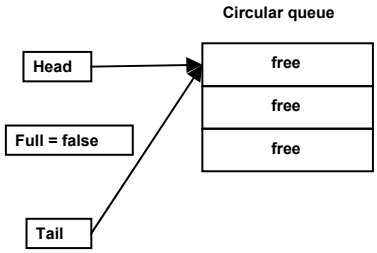
Consider a string of page references by a process:

Reference number:	1	2	3	4	5	6	7	8	9	10	11	12	13
Virtual page number:	9	0	3	4	0	5	0	6	4	5	0	5	4

Assume there are 3 physical frames. Using a circular queue similar to Figure 8.8, show the state of the queue for the first 6 references.

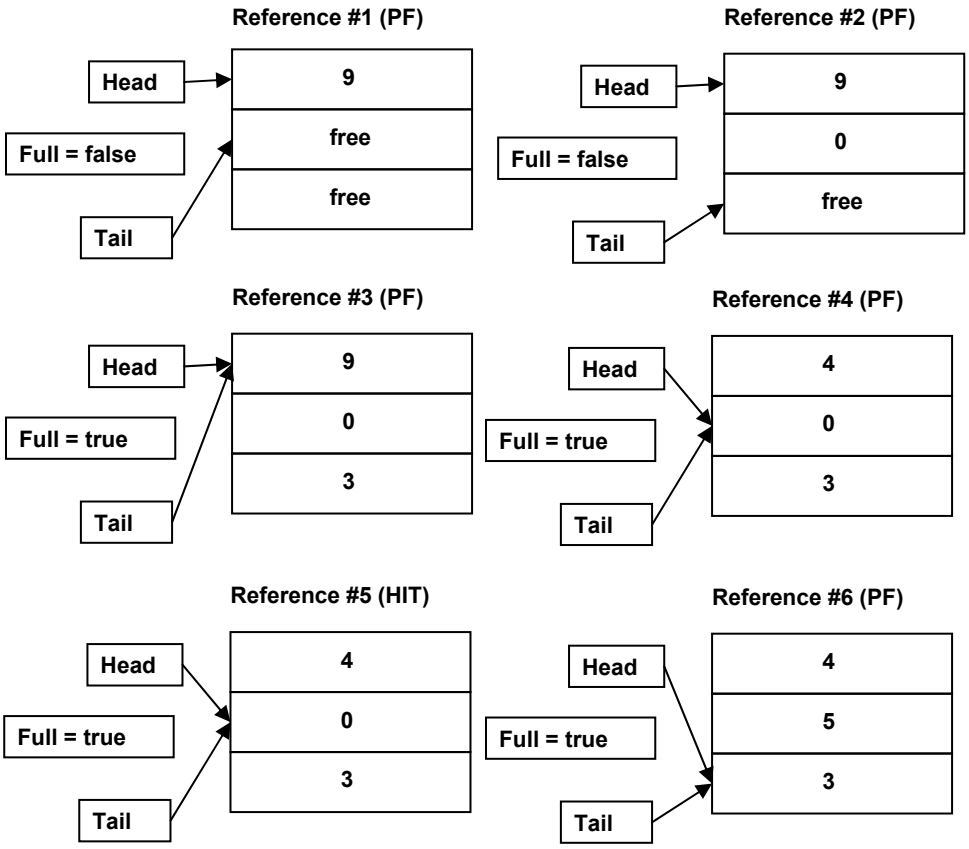
Answer:

Initially the circular queue looks as follows:



Upon a page fault, insertion of a new page happens at the entry pointed to by the tail. Similarly, victim page is one pointed to by the head, since the head always points to the FIFO page. Once selected, the head pointer moves to point to the next FIFO candidate. When the queue is full, both head and pointers move to satisfy a page fault. The

following snapshots of the data structure show the state of the queue after each of the first six references (PF denotes page fault; HIT denotes the reference is a hit, that is, there is no page fault):



The above sequence shows the anomaly in the FIFO scheme. Reference #6 replaces page 0 since that is the longest resident one to make room for page 5. However, page 0 is the immediate next reference (Reference #7).

Looking at the sequence of memory accesses in the above example, one can conclude that page 0 is *hot*. An efficient page replacement policy should try not to replace page 0. Unfortunately, we know this fact post-mortem. Let us see if we can do better than FIFO.

8.3.4 Least Recently Used (LRU)

Quite often, even in real life, we use the past as the predictor of the future. Therefore, even though we do not know the future memory accesses of a process, we do know the accesses made thus far by the process. Let us see how we use this information in page replacement. *Least Recently Used (LRU)* policy makes the assumption that if a page has not been referenced for a long time there is a good chance it will not be referenced in the future as well. Thus, the victim page in the LRU policy is the page that has not been used for the longest time.

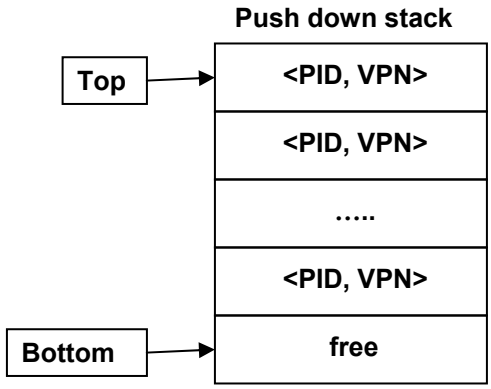


Figure 8.9: Push down stack for LRU replacement. The most recently used page is at the top of the stack. The least recently used page is at the bottom of the stack.

Let us understand the hardware assist needed for the LRU policy. The hardware has to track every memory access from the CPU. Figure 8.9 shows a stack data structure. On every access, the CPU pushes the currently accessed page on the top of the stack; if that page currently exists anywhere else in the stack the CPU removes it as well. Therefore, the bottom of the stack is the least recently used page and the candidate for replacement upon a page fault. If we want to track the references made to every page frame then the size of the stack should be as big as the number of frames.

Next we will investigate the data structure for the LRU policy. The memory manager uses the hardware stack in Figure 8.9 to pick the page at bottom of the stack as the victim. Of course, this requires some support from the instruction-set for the software to read the bottom of the stack. The hardware stack is in addition to the page table that is necessary for the virtual to physical translation.

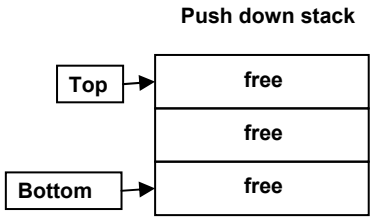
Notice that the memory manager has to maintain additional data structures such as the *free-list* and *frame table* to service page faults.

Example 9:

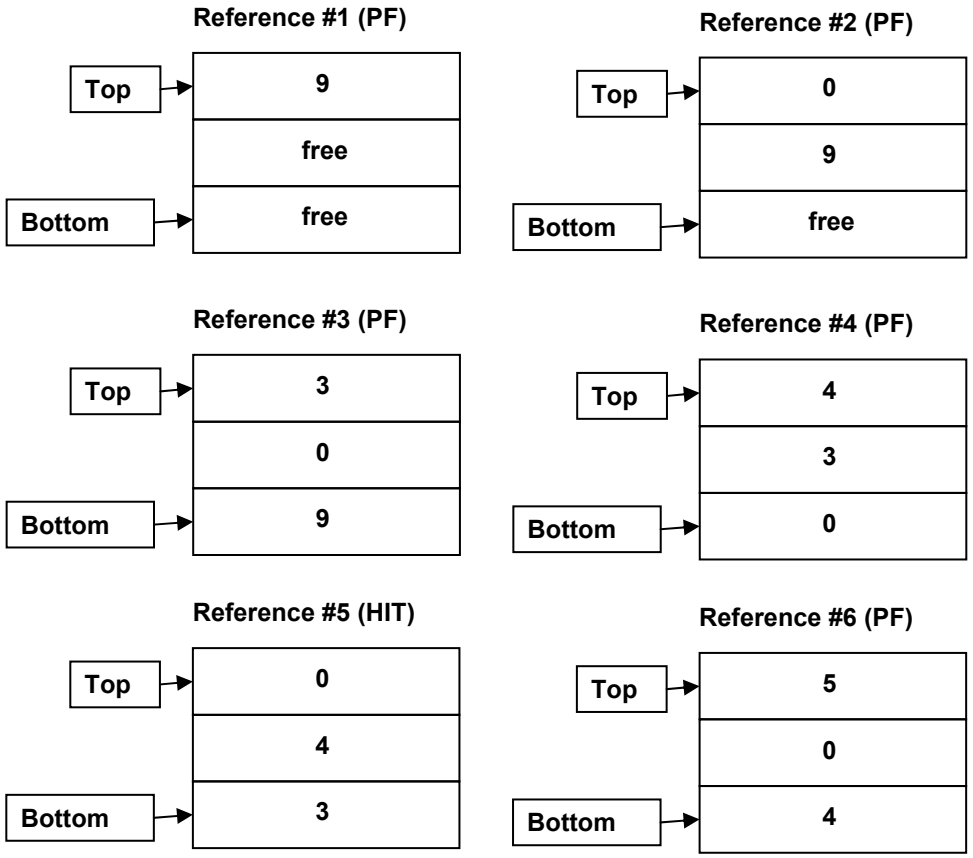
We will consider the same string of page references by a process as in the FIFO example:

Reference number:	1	2	3	4	5	6	7	8	9	10	11	12	13
Virtual page number:	9	0	3	4	0	5	0	6	4	5	0	5	4

Assume there are 3 physical frames. Initially the stack looks as follows.



The following snapshots show the state of the stack after each of the first six references (PF denotes page fault; HIT denotes the reference does is a hit, that is, there is no page fault):



It is interesting to compare Examples 4 and 5. Both of them experience 5 page faults in the first 6 references. Unfortunately, we cannot do any better than that since these pages (9, 0, 3, 4, and 5) are not in physical memory and therefore these are page faults are unavoidable with any policy. However, notice that Reference #6 replaces page 3 in the LRU scheme (not page 0 as in the FIFO example). Thus, Reference #7 results in a hit for the LRU scheme. In other words, LRU is able to avoid the anomalous situation that we encountered with the FIFO policy.

8.3.4.1 Approximate LRU #1: A Small Hardware Stack

The LRU scheme, while conceptually appealing, is simply not viable from an implementation point of view for a couple of reasons:

1. The stack has as many entries as the number of physical frames. For a physical memory size of 4 GB and a pagesize of 8 KB, the size of the stack is 0.5 MB. Such large hardware structures in the datapath of a pipelined processor add enormous latencies to the clock cycle time of the processor. For this reason, it is not viable to have such a huge hardware stack in the datapath of the processor.
2. On every access, the hardware has to modify the stack to place the current reference on the top of the stack. This expensive operation slows down the processor.

For these reasons, it is not feasible to implement a *true* LRU scheme. A more serious concern arises because true LRU in fact can be detrimental to performance in certain situations. For example, consider a program that loops over N+1 pages after accessing them in sequence. If the memory manager has a pool of N page frames to deal with this workload, then there will be a page fault for every access with the LRU scheme. This example, pathological as it sounds, is indeed quite realistic since scientific programs manipulating large arrays are quite common.

It turns out that implementing an approximate LRU scheme is both feasible and less detrimental to performance. Instead of a stack equal in size to the physical memory size (in frames), its size can be some small number (say 16). Thus, the stack maintains the history of the last 16 unique references made by the processor (older references fall off the bottom of the stack). The algorithm would pick a random page that is not in the hardware stack as the victim. Basically, the algorithm protects the most recent N referenced pages from replacement, where N is the size of the hardware stack.

In fact, some simulation studies have found that true LRU may even be worse than approximate LRU. This is because anything other than Belady's Min is simply guessing the page reference behavior of a process, and is therefore susceptible to failure. In this sense, a purely random replacement policy (Section 8.3.2), which requires no sophisticated mechanisms in either hardware or software, has been shown to do well for certain workloads.

8.3.4.2 Approximate LRU #2: Reference bit per page frame

It turns out that tracking every memory access is just not feasible from the point of view of implementing a high-speed pipelined CPU. Therefore, we have to resort to some other means to approximate the LRU scheme.

One possibility is to track references at the page level instead of individual accesses. The idea is to associate a *reference bit* per page frame. Hardware sets this bit when the CPU accesses any location in the page; software reads and resets it. The hardware orchestrates accesses to the physical memory through the page table. Therefore, we can have the reference bits in the page table.

Let us turn our attention to choosing a victim. Here is an algorithm using the reference bits.

1. The memory manager maintains a *bit vector* per page frame, called a *reference counter*.

2. Periodically the memory manager reads the reference bits of all the page frames and dumps them in the *most significant bit (MSB)* of the corresponding per frame *reference counters*. The counters are right-shifted to accommodate the reference bits into their respective MSB position. Figure 8.10 shows this step graphically. After reading the reference bits, the memory manager clears them. It repeats this step every time quantum. Thus each counter holds a snapshot of the references (or lack thereof) of the last *n* time quanta (n = 32 in Figure 8.10).

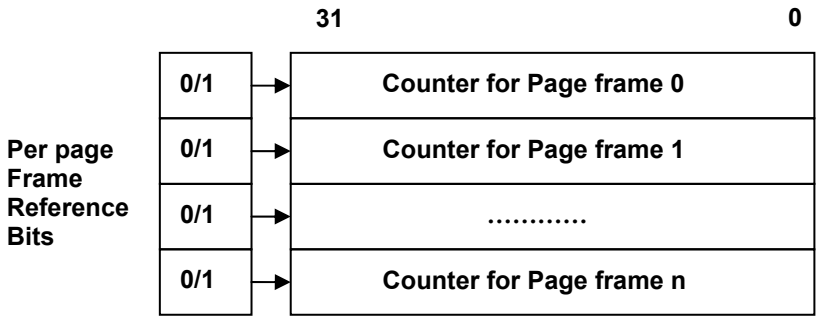


Figure 8.10: Page frame reference counters; The reference bit for each page may be 0 or 1

3. The reference counter with the largest absolute value is the most recently referenced page. The reference counter with the smallest absolute value is the least recently referenced page and hence a candidate for selection as a replacement victim.

A *paging daemon* is an entity that is part of the memory manager that wakes up periodically (as determined by the time quantum) to carry out the steps of the above algorithm.

8.3.5 Second chance page replacement algorithm

This is a simple extension to the FIFO algorithm using the reference bits. As the name suggests, this algorithm gives a second chance for a page to be **not** picked as a victim. The basic idea is to use the reference bit set by the hardware as an indication that the page deserves a second chance to stay in memory. The algorithm works as follows:

1. Initially, the operating system clears the reference bits of all the pages. As the program executes, the hardware sets the reference bits for the pages referenced by the program.
2. If a page has to be replaced, the memory manager chooses the replacement candidate in FIFO manner.
3. If the chosen victim's reference bit is set, then the manager clears the reference bit, gives it a *new arrival time* and repeats step 1. In other words, this page is moved to the end of the FIFO queue.
4. The victim is the first candidate in FIFO order whose reference bit is not set.

Of course, in the pathological case where all the reference bits are set, the algorithm degenerates to a simple FIFO.

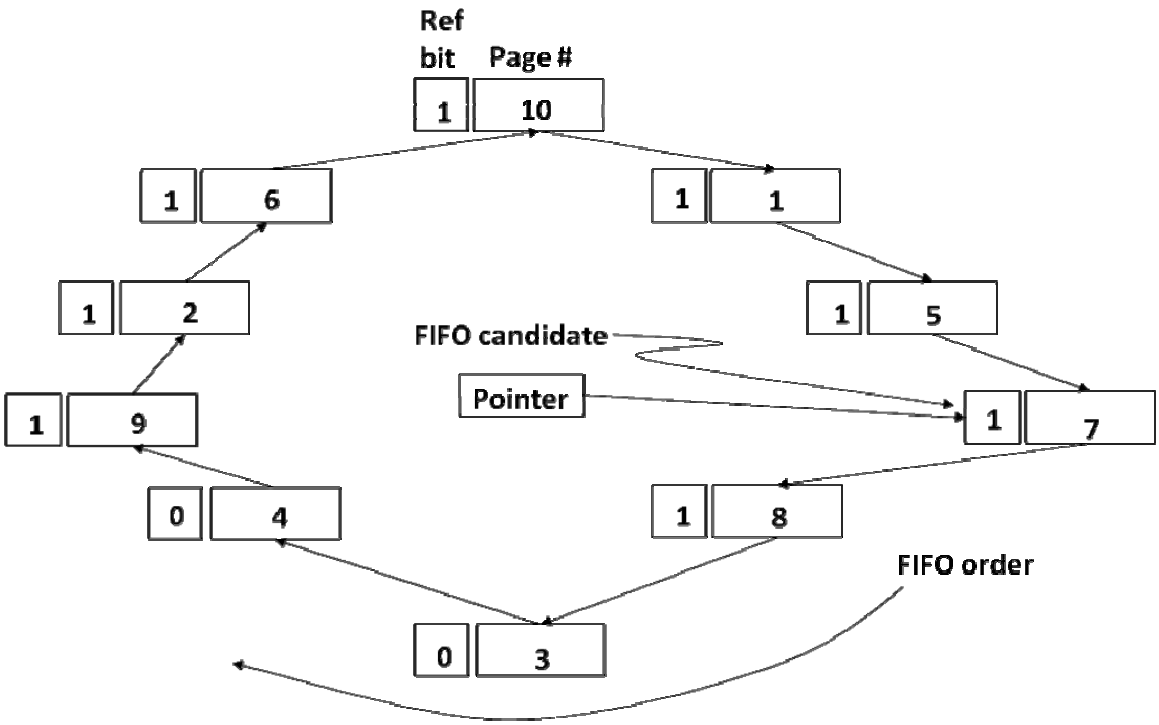


Figure 8.11a: Second chance replacement – the memory manager keeps a software pointer that points to the FIFO candidate at the start of the algorithm. Note the frames that have reference bits set and the ones that have their reference bits cleared. The ones that have their reference bits set to 1 are those that have been accessed by the program since the last sweep by the memory manager.

A simple way to visualize and implement this algorithm is to think of the pages forming a circular queue with a pointer pointing to the FIFO candidate as shown in Figure 8.11a. When called upon to pick a victim, the pointer advances until it finds a page whose reference bit is not set. As it moves towards the eventual victim, the memory manager clears the reference bits of the pages that it encounters. As shown in Figure 8.11a, the FIFO candidate is page 7. However, since its reference bit is set, the pointer moves until it gets to page 3 whose reference bit is not set (first such page in FIFO order) and chooses that page as the victim. The algorithm clears the reference bits for pages 7 and 8 as it sweeps in FIFO order. Note that the algorithm does not change the reference bits for the other pages not encountered during this sweep. It can be seen that the pointer sweep of the algorithm resembles the hand of a clock moving in a circle and for the same reason this algorithm is often referred to as the *clock* algorithm. After choosing page 3 as the victim, the pointer will move to the next FIFO candidate (page number 4 in Figure 8.11b).

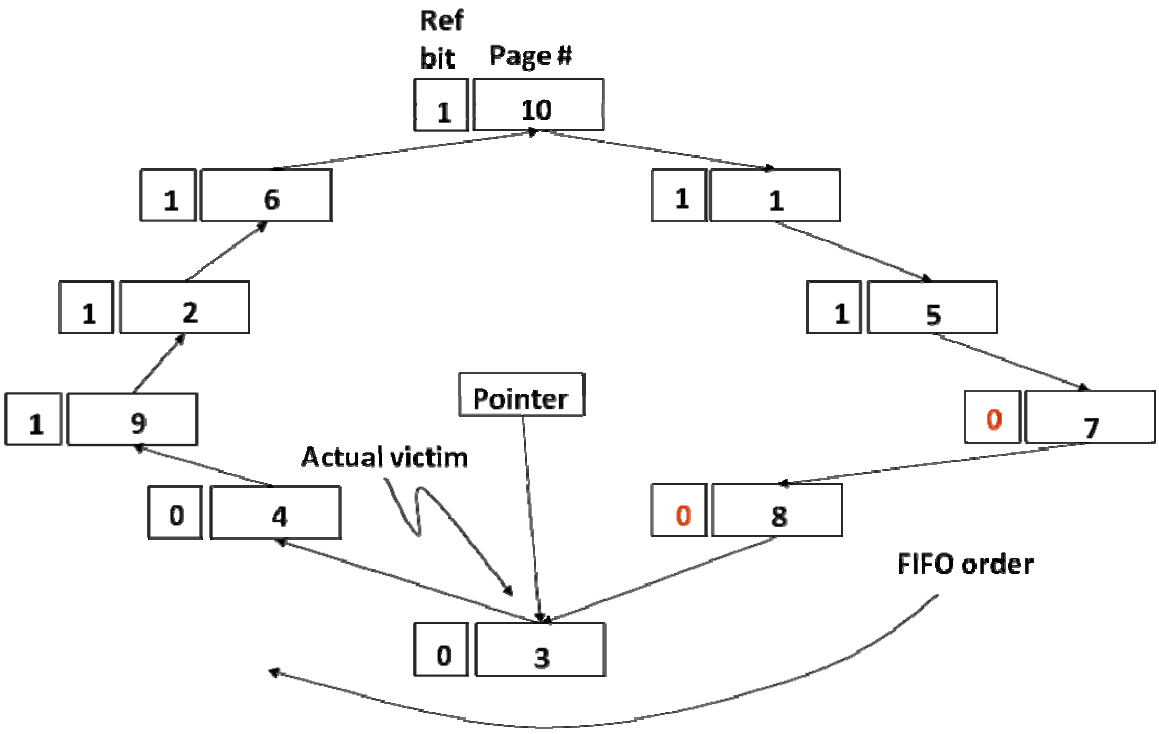


Figure 8.11b: Second chance replacement – algorithm passes over the frames whose reference bits are set to 1 (clearing them during the sweep) until it finds a frame that does not have its reference bit set, the actual victim frame.

Example 10:

Suppose that we only have three physical frames and that we use the second-chance page replacement algorithm. Show the virtual page numbers that occupy these frames for the following sequence of page references.

Reference number:	1	2	3	4	5	6	7	8	9	10
Virtual page number:	0	1	2	3	1	4	5	3	6	4

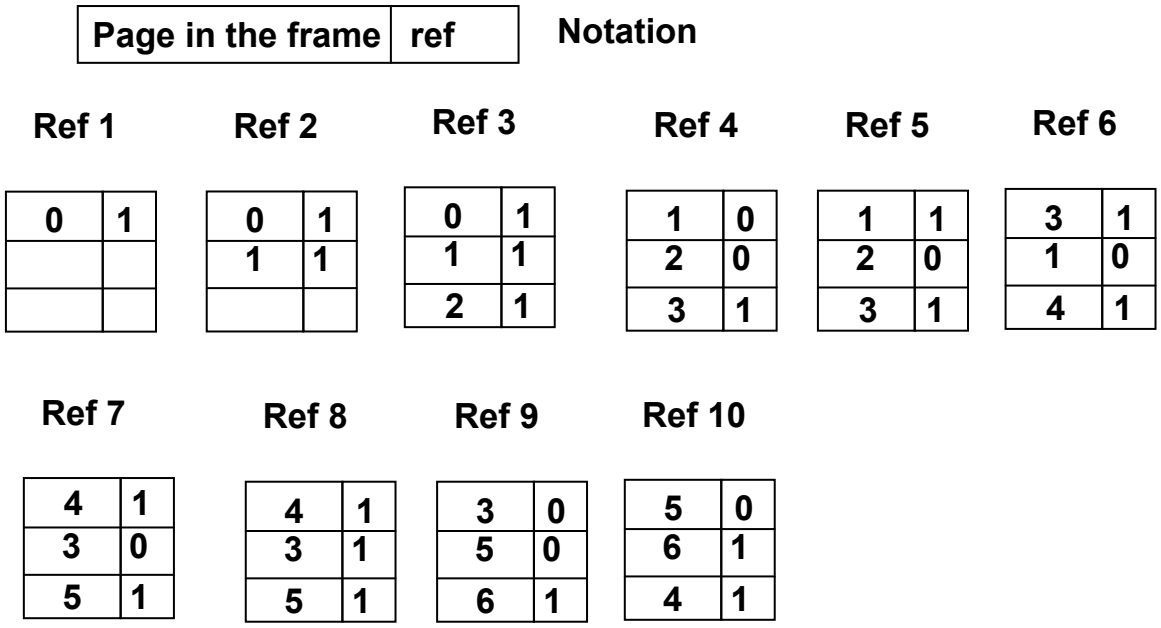
Answer:

The following sequence of figures show the state of the page frames and the associated reference bits **AFTER** each reference is satisfied. The top entry is always the normal FIFO candidate. Note that the reference bit is set when the page is brought into the page frame.

To understand the choice of the victim in a particular reference, please see the state of the pages shown after the previous reference.

References 1-3: no replacement
Reference 4: Page 0 is the victim (all have ref bits set, so FIFO candidate is the victim)
Reference 5: no replacement (page 1's reference bit set)
Reference 6: Page 2 is the victim (Page 1 the FIFO candidate gets a second chance)

Reference 7: Page 1 is the victim (Page 3 the FIFO candidate gets a second chance)
Reference 8: no replacement (page 3's reference bit set)
Reference 9: Page 4 is the victim (all have ref bits set, so FIFO candidate is the victim)
Reference 10: Page 3 is the victim (FIFO candidate and its ref bit is off)



8.3.6 Review of page replacement algorithms

Table 8.1 summarizes the page replacement algorithms and the corresponding hardware assists needed. It turns out that approximate LRU algorithms using reference bits do quite well in reducing the page fault rate and perform almost as well as true LRU; a case in point for engineering ingenuity that gets you most of the benefits of an exact solution.

PAGE REPLACEMENT ALGORITHM	HARDWARE ASSIST NEEDED	BOOKKEEPING INFORMATION NEEDED	COMMENTS
Belady's MIN	Oracle	None	Provably optimal performance; not realizable in hardware; useful as an upper bound for performance comparison
Random Replacement	None	None	Simplest scheme; useful as a lower bound for performance comparison
FIFO	None	Arrival time of a virtual page into physical memory	Could lead to anomalous behavior; often performance worse than random
True LRU	Push down stack	Pointer to the bottom of the LRU stack	Expected performance close to optimal; infeasible for hardware implementation due to space and time complexity; worst-case performance may be similar or even worse compared to FIFO
Approximate LRU #1	A small hardware stack	Pointer to the bottom of the LRU stack	Expected performance close to optimal; worst-case performance may be similar or even worse compared to FIFO
Approximate LRU #2	Reference bit per page frame	Reference counter per page frame	Expected performance close to optimal; moderate hardware complexity; worst-case performance may be similar or even worse compared to FIFO
Second Chance Replacement	Reference bit per page frame	Arrival time of a virtual page into physical memory	Expected performance better than FIFO; memory manager implementation simplified compared to LRU schemes

Table 8.1: Comparison of page replacement algorithms

8.4 Optimizing Memory Management

In the previous sections, we presented rudimentary techniques for demand-paged virtual memory management. In this section, we will discuss some optimizations that memory managers employ to improve the system performance. It should be noted that the optimizations are not specific to any particular page replacement algorithm. They could be used on top of the basic page replacement techniques discussed in the preceding section.

8.4.1 Pool of free page frames

It is not a good idea to wait until a page fault to select a victim for replacement. Memory managers always keep a *minimum* number of page frames ready for allocation to satisfy page faults. The paging daemon wakes up periodically to check if the pool of free frames on the *free-list* (Figure 8.3) is below this minimum threshold. If so, it will run the page replacement algorithm to free up more frames to meet this minimum threshold. It is possible that occasionally the number of page frames in the free list will be way over the minimum threshold. When a process terminates, all its page frames get on the free-list leading to such a situation.

8.4.1.1 Overlapping I/O with processing

In Section 8.1.4, we pointed out that before the memory manager uses a page frame as a victim, the manager should save the current contents of the page that it houses to the disk if it is dirty. However, since the manager is simply adding the page frame to the *free-list* there is no rush to do the saving right away. As we will see in a later chapter, high-speed I/O (such as to the disk) occurs concurrently with the CPU activity. Therefore, the memory manager simply schedules a write I/O for a dirty victim page frame before adding it to the *free-list*. Before the memory manager uses this dirty physical frame to house another virtual page, the write I/O must be complete. For this reason, the memory manager may skip over a dirty page on the *free-list* to satisfy a page fault (Figure 8.11). This strategy helps to delay the necessity to wait on a write I/O at the time of a page fault.

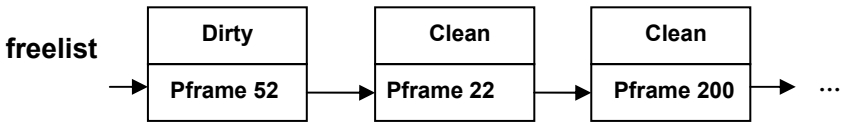


Figure 8.11: Free-list; upon a page fault, the manager may choose pframe 22 over pframe 52 as the victim since the former is clean while the latter is dirty.

8.4.1.2 Reverse mapping to page tables

To meet the minimum threshold requirement, the paging daemon may have to take away page frames currently mapped into running processes. Of course, the process that lost this page frame may fault on the associated page when it is schedule to run. As it turns out, we can help this process with a little bit of additional machinery in each node of the free-list. If the memory manager has not yet reassigned the page frame to a different process, then we can retrieve it from the *free-list* and give it to the faulting process. To enable this optimization, we augment each node in the *free-list* with the reverse mapping (similar to the *frame table*), showing the virtual page it housed last (See Figure 8.12).

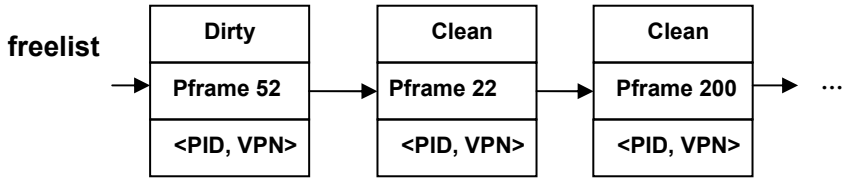


Figure 8.12: Reverse mapping for the page frames on the Free-list

Upon a page fault, the memory manager compares the <PID, VPN> of the faulting process with entries in the *free-list*. If there is a match, the manager simply re-establishes the original mapping in the page table for the faulting process by using the page frame of the matching entry to satisfy the page fault. This optimization removes the need for a read I/O from the disk to bring in the faulting page.

It is interesting to note that this enhancement when used in tandem with the second change replacement algorithm, essentially gives a *third* chance for a page before it is kicked out of the system.

8.4.2 Thrashing

A term that we often come across with respect to computer system performance is *thrashing*, which is used to denote that the system is not getting useful work done. For example, let us assume that the degree of multiprogramming (which we defined in Chapter 6 as the number of processes coexisting in memory and competing for the CPU) is quite high but still we observe low *processor utilization*. One may be tempted to increase the degree of multiprogramming to keep the processor busy. On the surface, this seems like a good idea, but let us dig a little deeper.

It is possible that all the currently executing programs are I/O bound (meaning they do more I/O than processing on the CPU) in which case the decision to increase the degree of multiprogramming may be a good one. However, it is also possible that the programs are CPU bound. At first glance, it would appear odd that the processor utilization would be low with such CPU bound jobs. The simple answer is too much paging activity. Let us elaborate this point. With a demand-paged memory management, a process needs to have sufficient memory allocated to it to get useful work done. Otherwise, it will page fault frequently. Therefore, if there are too many processes coexisting in memory (i.e., a high degree of multiprogramming), then it is likely that the processes are continuously paging against one another to get physical memory. Thus, none of the processes is making forward progress. In this situation, it is incorrect to increase the degree of multiprogramming. In fact, we should reduce it. Figure 8.13 shows the expected behavior of CPU utilization as a function of the degree of multiprogramming. The CPU utilization drops rapidly after a certain point.

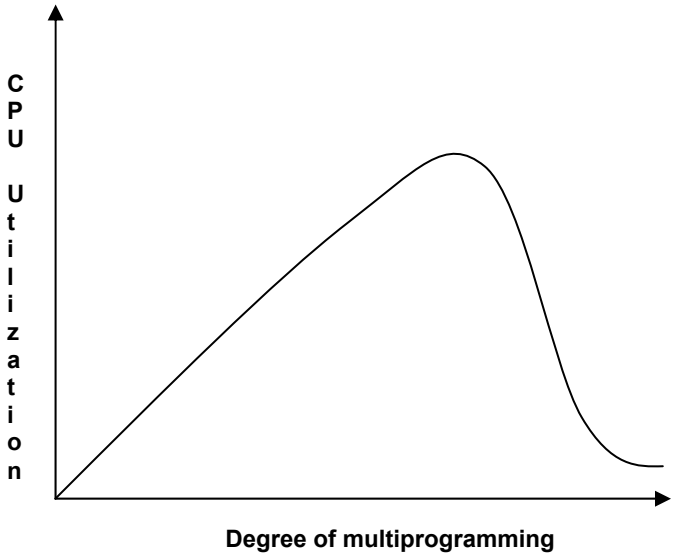


Figure 8.13: CPU Thrashing Phenomenon

This phenomenon, called *thrashing*, occurs when processes spend more time paging than computing. Paging is *implicit I/O* done on behalf of a process by the operating system. Excessive paging could make a process, which is originally compute bound into an I/O bound process. An important lesson to take away from this discussion is that CPU scheduling should take into account memory usage by the processes. It is erroneous to have a CPU scheduling policy based solely on processor utilization. Fortunately, this is a correctable situation by cooperation between the CPU scheduling and memory management parts of the operating system.

Let us investigate how we can control thrashing. Of course, we can load the entire program into memory but that would not be an efficient use of the resources. The trick is to ensure that each process has *sufficient* number of page frames allocated so that it does not page fault frequently. The *principle of locality* comes to our rescue. A process may have a huge memory footprint. However, if you look at a reasonable sized *window of time*, we will observe that it accesses only a small portion of its entire memory footprint. This is the principle of locality. Of course, the locus of program activity may change over time as shown in Figure 8.14. However, the change is gradual not drastic. For example, the set of pages used by the program at time t1 is {p1, p2}; the set of pages at t2 is {p2, p3}; and so on.

We do not want the reader to conclude that the locus of activity of a program is always confined to consecutive pages. For example, at time t7 the set of pages used by the program is {p1, p4}. The point to take away is that the locus of activity is confined to a small subset of pages (see Example 11).

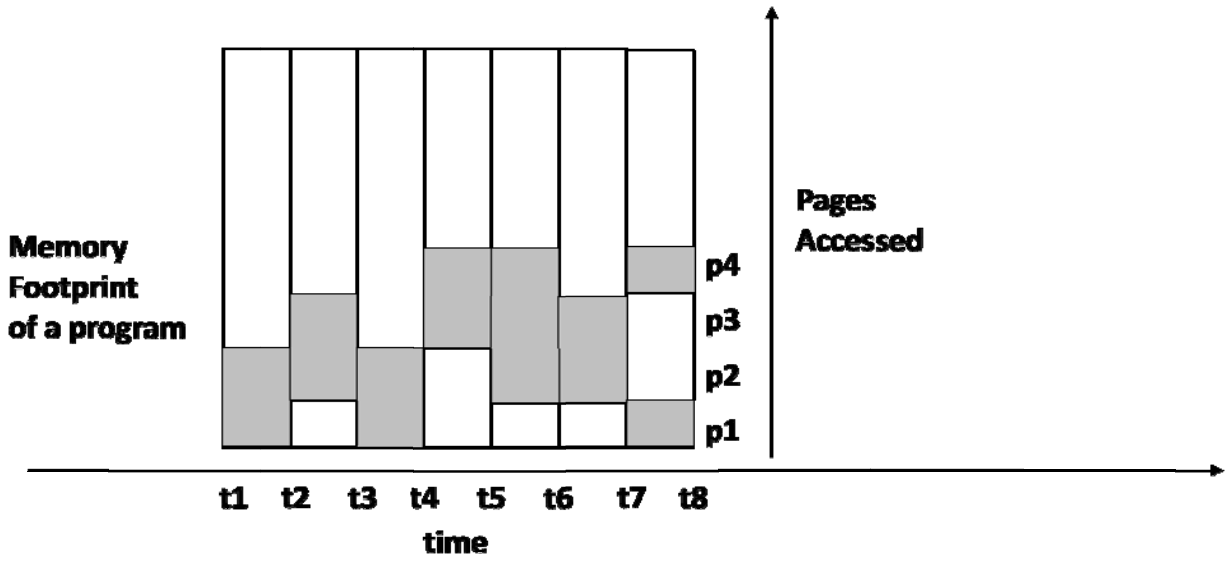


Figure 8.14: Changing loci of program activity as a function of time; the set of pages in use by the program is captured at times t1, t2, etc.; assume that the locus of program activity remains unchanged in the time intervals such as t1-t2, t2-t3, and so on

It is straightforward to use this principle to reduce page faults. If the current locus of program activity is in memory then the associated process will not page fault until the locus changes. For example, if the set of pages accessed by the program remains unchanged between t1 and t2, and if p1 and p2 are in physical memory then the program will not experience any page fault between t1 and t2.

8.4.3 Working set

To determine the locus of program activity, we define and use a concept called *working set*. Working set is the set of pages that defines the locus of activity of a program. Of course, the working set does not remain fixed since the locus of program activity changes over time.

For example, with reference to Figure 8.14,

Working $set_{t1\text{-}t2}$ = {p1, p2}
Working $set_{t2\text{-}t3}$ = {p2, p3}
Working $set_{t3\text{-}t4}$ = {p1, p2}
Working $set_{t4\text{-}t5}$ = {p3, p4}
Working $set_{t5\text{-}t6}$ = {p2, p3, p4}
……..

The *working set size* (WSS) denotes the number of distinct pages touched by a process in a window of time. For example, the WSS for this process is 2 in the interval t1-t2, and 3 in the interval t5-t6.

The *memory pressure* exerted on the system is the summation of the WSS of all the processes currently competing for resources.

$$\text{Total memory pressure} = \sum_{i=1}^{i=n} WSSi \qquad (1)$$

Example 11:

During the time interval t1 - t2, the following virtual page accesses are recorded for the three processes P1, P2, and P3, respectively.

P1: 0, 10, 1, 0, 1, 2, 10, 2, 1, 1, 0

P2: 0, 100, 101, 102, 103, 0, 101, 102, 104

P3: 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5

a) What is the **working set** for each of the above three processes for this time interval?
b) What is the **cumulative memory pressure** on the system during this interval?

Answer:

a)

P1's Working set = {0, 1, 2, 10}

P2's Working set = {0, 100, 101, 102, 103, 104}

P3's Working set = {0, 1, 2, 3, 4, 5)

b)

Working set size of P1 = 4
Working set size of P2 = 6
Working set size of P3 = 6
Cumulative memory pressure = sum of the working sets of all processes
= 4 + 6 + 6
= 16 page frames

8.4.4 Controlling thrashing

1. If the total memory pressure exerted on the system is greater than the total available physical memory then the memory manager decreases the degree of multiprogramming. If the total memory pressure is less than the total available physical memory then the memory manager increases the degree of multiprogramming.
   One approximate method for measuring the WSS of a process uses the reference bits associated with the physical frames. Periodically say every Δ time units, a daemon

wakes up and samples the reference bits of the physical frames assigned to a process. The daemon records the page numbers that have their respective reference bits turned on; it then clears the reference bits for these pages. This recoding of the page numbers allows the memory manager to determine the working set and the WSS for a given process for any interval t and $t + \Delta$.

2. Another simple method of controlling thrashing is to use the observed page fault rate as a measure of thrashing. The memory manager sets two markers, a *low water mark*, and a *high water mark* for page faults (See Figure 8.15). An observed page fault rate that exceeds the high water mark implies excessive paging. In this case, the memory manager reduces the degree of multiprogramming. On the other hand, an observed page fault rate that is lower than the low water mark presents an opportunity for the memory manager to increase the degree of multiprogramming.

The shaded region in Figure 8.15 shows the preferred sweet spot of operation for the memory manager. The paging daemon kicks in to increase the pool of free physical frames when the page fault rate goes higher than the high water mark, decreasing the degree of multiprogramming, if necessary.

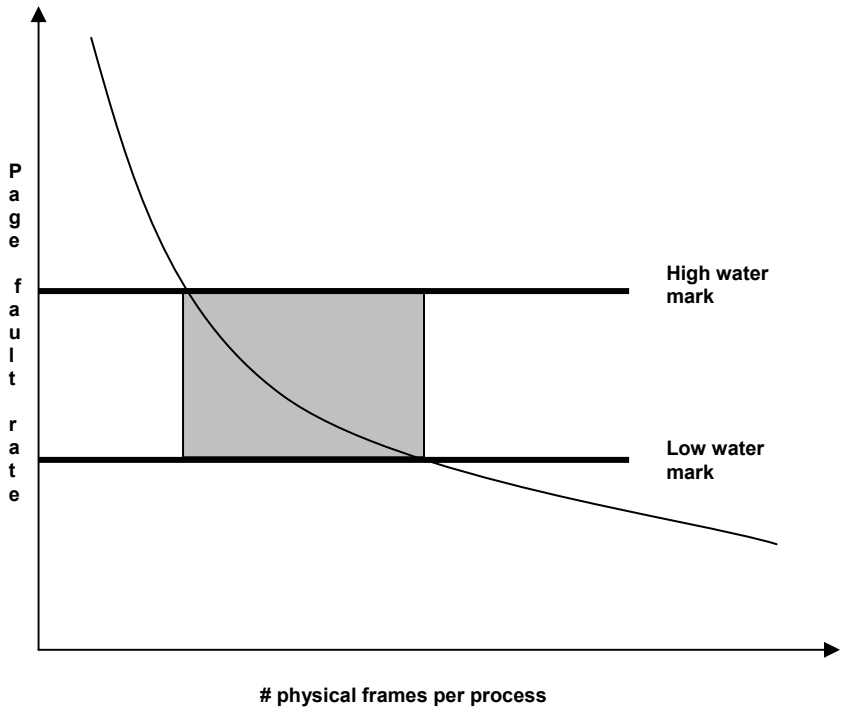


Figure 8.15: Page fault rate

8.5 Other considerations

The operating system takes a number of other measures to reduce page faults. For example, when the memory manager swaps out a process (to decrease the degree of multiprogramming) it remembers the current working set of the process. The memory manager brings in the corresponding working set at the point of swapping in a process. This optimization referred to as *pre-paging* reduces disruption at process start-up.

I/O activity in the system occurs simultaneously and concurrently with the CPU activity. This leads to interesting interaction between the memory and the I/O subsystems of the operating system. For example, the I/O subsystem may be initiating a transfer from a given physical frame to the disk. Simultaneously, the paging daemon may select the same physical frame as a potential victim for servicing a page fault. Just as the CPU scheduler and memory manager work in concert, the I/O and the memory subsystems coordinate their activities. Typically, the I/O subsystem will *pin a page* in physical memory for the duration of the data transfer to prevent the paging daemon from selecting that page as a victim. The page table serves as a convenient communication area between the I/O subsystem and the memory manager for recording information such as the need to pin a page in physical memory. We will discuss I/O in much more detail in a later chapter (see Chapter 10).

8.6 Translation Lookaside Buffer (TLB)

The discussion thus far should make one point very clear. Page faults are disruptive to system performance and the memory manager strives hard to avoid them. To put things in perspective, the context switch time (from one process to another) approximates several tens of instructions; the page fault handling time (without disk I/O) also approximates several tens of instructions. On the other hand, the time spent in disk I/O is in the millisecond range that approximates to perhaps a million instructions on a GHz processor.

However, even if we reduce the number of page faults with a variety of optimizations, it remains that every memory access involves two trips to the memory: one for the address translation and one for the actual instruction or data.

USER/KERNEL	VPN	PFN	VALID/INVALID
U	0	122	V
U	XX	XX	I
U	10	152	V
U	11	170	V
K	0	10	V
K	1	11	V
K	3	15	V
K	XX	XX	I

Figure 8.16: Translation Look-aside Buffer (TLB)

This is undesirable. Fortunately, we can cut this down to one with a little bit engineering ingenuity. The concept of paging removes the user's view of contiguous memory footprint for the program. However, it still is the case that *within* a page the memory locations are contiguous. Therefore, if we have performed the address translation for a page, then the same translation applies to *all* memory locations in that page. This suggests adding some hardware to the CPU to remember the address translations. However, we know that programs may have large memory footprints. The principle of locality that we introduced earlier comes to our rescue again. Referring to Figure 8.14, we may need to remember only a few translations *at a time* for a given program irrespective of its total memory footprint. This is the idea behind *translation look-aside buffer (TLB)*, a small hardware table in which the CPU holds *recent* address translations (Figure 8.16). An address translation for a page needs to be done at least once. Therefore, none of the entries in the table is valid when the processor starts up. This is the reason for the *valid* bit in each entry. The *PFN* field gives the physical frame number corresponding to the *VPN* for that entry.

Notice how the TLB is split into two parts. One part holds the translations that correspond to the user address space. The other part holds the same for the kernel space. On a context switch, the operating system simply invalidates all the user space translations, schedules the new process, and builds up that part of the table. The kernel space translations are valid independent of the user process currently executing on the processor. To facilitate TLB management, the instruction-set provides *purge TLB*, a privileged instruction executable in kernel mode,

8.6.1 Address Translation with TLB

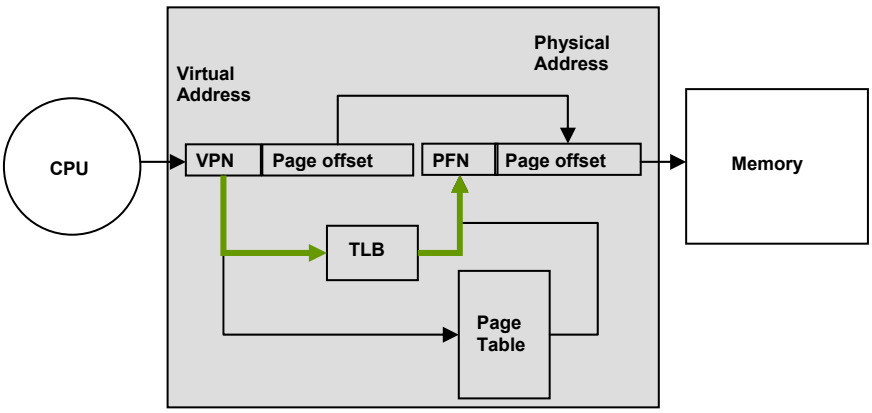


Figure 8.17-a: Address Translation (TLB Hit)

Figure 8.17-a and 8.17-b show the CPU address translation in the presence of the TLB. The hardware first checks if there is a valid translation in the TLB for the CPU generated address. We refer to such a successful lookup of the TLB as a *hit*. It is a *miss* otherwise. On a hit, the PFN from the TLB helps in forming the physical address thus avoiding a trip to the memory for address translation. On a miss, the page table in memory supplies the

PFN. The hardware enters the resulting translation into the TLB for future accesses to the same page.

Of course, it is quite possible that the address translation is done entirely in software. You may be wondering how this would be possible. Basically, the hardware raises a "TLB fault" exception if it does not find the translation it needs in the TLB. At this point, the operating system takes over and handles the fault. The processing that the operating system needs to do to service this fault may escalate to a full-blown page fault if the page itself is not present in the memory. In any event, once it completes the TLB fault handling, the operating system will enter the translation into the TLB so that regular processing could resume. Architectures such as the MIPS and DEC Alpha take this approach of handling the TLB faults entirely in software. Naturally, the ISA for these architecture have special instructions for modifying the TLB entries. It is customary to call such a scheme, *software managed TLB*.

The TLB is a special kind of memory, different from any hardware device we have encountered thus far. Essentially, the TLB is a hash table that contains VPN-PFN matches. The hardware has to look through the entire table to find a match for a given VPN. We refer to this kind of hardware device as *content addressable memory (CAM)* or *associative memory*. The exact details of the hardware implementation of the TLB depend on its organization.

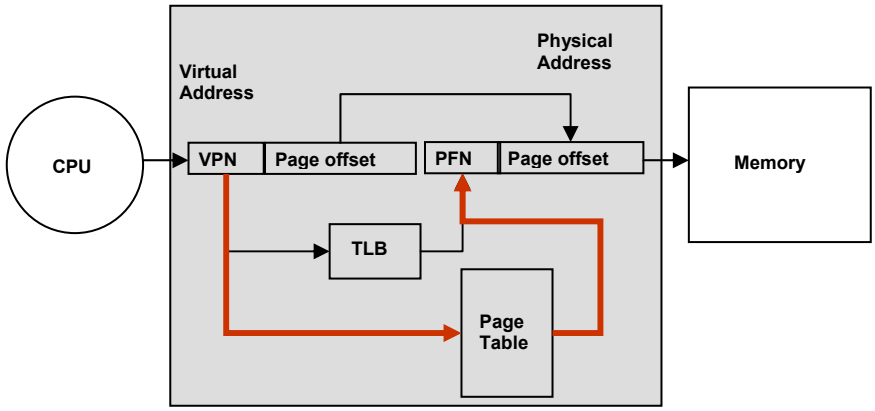


Figure 8.17-b: Address Translation (TLB Miss)

It turns out that TLB is a special case of a general concept, namely, *caching*, which we deal with in much more detail in Chapter 9 in the context of memory hierarchies. Suffice it to say at this point, caching refers to using a small table of items recently referred to by any subsystem. In this sense, this concept is exactly similar to the toolbox/tool-tray analogy we introduced in Chapter 2.

The caching concept surfaces in several contexts. Here are a few concrete examples:

(a) in processor design for keeping recently accessed memory locations in the processor in a hardware device called *processor cache*,
(b) in processor design to keep recent address translations in the TLB,
(c) in disk controller design to keep recently accessed disk blocks in memory (see Chapter 10),
(d) in file system design to keep an in-memory data structure of recently accessed bookkeeping information regarding the physical location of files on the disk (see Chapter 11),
(e) in file system design to keep an in-memory software cache of recently accessed files from the disk (see Chapter 11), and
(f) in web browser design to keep an on-disk cache of recently accessed web pages.

In other words, *caching is a general concept of keeping a smaller stash of items nearer to the point of use than the permanent location of the information*. We will discuss the implementation choices for processor caches in more detail in the next Chapter that deals with memory hierarchy.

8.7 Advanced topics in memory management

We mentioned that the memory manager's data structures include the process page tables. Let us do some math. Assuming a 40-bit byte-addressable virtual address and an 8 KB pagesize, we need 2^{27} page table entries for each process page table. This constitutes a whopping 128 million entries for each process page table. The entire physical memory of the machine may not be significantly more than the size of a single process page table.

We will present some preliminary ideas to get around this problem of managing the size of the page tables in physical memory. A more advance course in operating systems will present a more detailed treatment of this topic.

8.7.1 Multi-level page tables

The basic idea is to break a single page table into a multi-level page table. To make the discussion concrete consider a 32-bit virtual address with a 4 KB page size. The page table has 2^{20} entries. Let us consider a 2-level page table shown in Figure 8.18. The VPN of the virtual address has two parts. VPN1 picks out a unique entry in the first level page table that has 2^{10} entries. There is a second level page table (indexed by VPN2) corresponding to each unique entry in the first level. Thus, there are 2^{10} (i.e., 1024) second level tables, each with 2^{10} entries. The second level tables contain the PFN corresponding to the VPN in the virtual address.

Let us understand the total space requirement for the page tables. With a single-level conventional structure, we will need a page table of size 2^{20} entries (1 M entries). With a two-level structure, we need 2^{10} entries for the first level plus 2^{10} second-level tables,

each with 2^{10} entries. Thus, the total space requirement for a 2-level page table is 1K entries (size of first level table) more than the single-level structure[1].

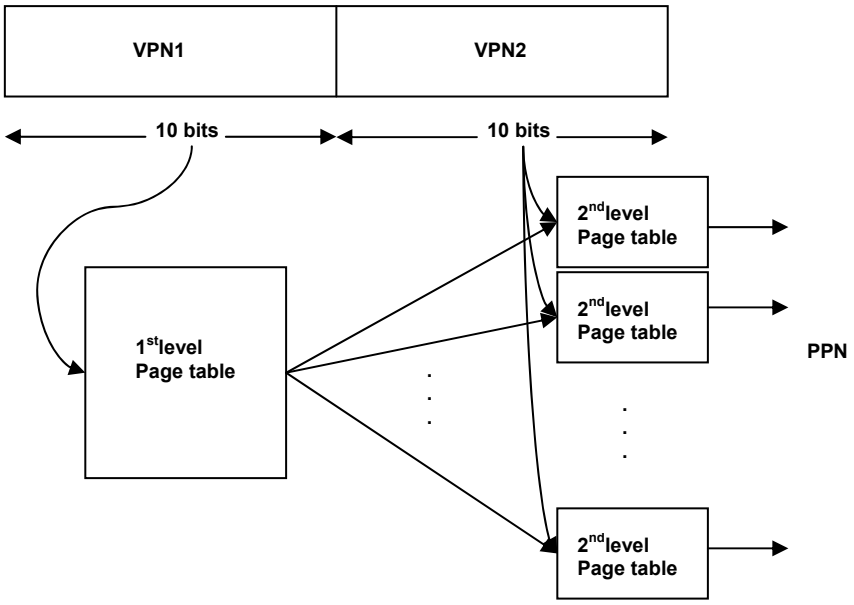


Figure 8.18: A 2-level page table

Let us understand what we have we gained by this two-level structure. The 1st level page table has 1K entries per process. This is a reasonable sized data structure to have per process in physical memory. The 2nd level tables need not be in physical memory. The memory manager keeps them in virtual memory and demand-pages in the 2nd level tables based on program locality.

Modern operating systems catering to 64-bit processor architectures implement multi-level page tables (i.e., more than 2 levels). Unfortunately, with multi-level page tables, a memory access potentially has to make multiple trips to the physical memory. While this is true, fortunately, the TLB makes subsequent memory accesses to the same page translation-free.

Example 13:

Consider a memory system with 64-bit virtual addresses and with an 8KB page size. We use a five-level page table. The 1st level page table keeps 2K (2048) entries for each process. The remaining four levels, each keep 1K (1024) entries.

a) Show the layout of a virtual address in this system with respect to this multi-level page table structure.
b) What is the total page table space requirement for each process (i.e., sum

[1] Note that the size of the page table entries in the 2-level page table is different from that of the single-level structure. However, to keep the discussion simple, we do not get into this level of detail. Please refer to review question 9 for an elaboration of this point.

of all the levels of the page tables)?

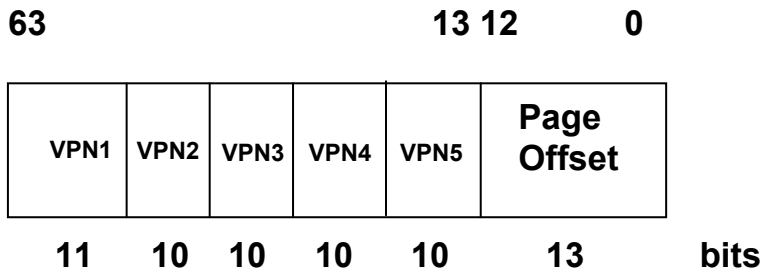
Answer:

a)
With 8KB pagesize, the number of bits for page offset = 13.

Therefore the VPN = 64-13 = 51 bits

Number of bits for the first level page table (2048 entries) = 11
Number of bits for each of the remaining 4 levels (1024 entries each) = 10

Layout of the virtual address:



b)
Number of entries in the 1st level table = 2^{11}
Number of entries in each 2nd level table = 2^{10}
(Similar to Figure 8.18) There are 2^{11} such 2nd level tables (one for each first level table entry)
So total number of entries at the second level = 2^{21}
Number of entries in each 3rd level table = 2^{10}
There are 2^{21} such 3rd level tables (one for each second level table entry)
So total number of entries at the third level = 2^{31}
Number of entries in each 4th level table = 2^{10}
There are 2^{31} such 4th level tables (one for each third level table entry)
So total number of entries at the fourth level = 2^{41}
Number of entries in each 5th level table = 2^{10}
There are 2^{41} such 5th level tables (one for each fourth level table entry)
So total number of entries at the fifth level = 2^{51}

So total page table size for this virtual memory system
= $\mathbf{2^{11} + 2^{21} + 2^{31} + 2^{41} + 2^{51}}$

A single-level page table for this virtual memory system would require a page table with 2^{51} entries.

8.7.2 Access rights as part of the page table entry

A page table entry usually contains information in addition to the PFN and the valid bit. For example, it may contain access rights to the particular page such as *read-only*, *read-write*, etc. This is extremely important from the point of meeting the expectations laid out in Chapter 7 on the functionalities provided by the memory manager (please see Section 7.1). In particular, we mentioned that the memory manager has to provide memory protection and isolation for processes from one another. Further, a process has to be protected against itself from erroneous behavior due to programming bugs (e.g., writing garbage unintentionally across the entire memory footprint due to a bug in the program). Lastly, processes have to be able to share memory with each other when needed. Access rights information in a page table entry is another illustration of the cooperation between the hardware and software. The memory manager sets, in the page table, the access rights for the pages contained in the memory footprint of a program at the time of process creation. For example, it will set the access rights for pages containing code to be read-only and data to be read-write. The hardware checks the access rights as part of the address translation process on every memory access. Upon detection of violation of the access rights, the hardware takes corrective action. For example, if a process attempts to write to a read-only page, it will result in an *access violation trap* giving the control to the operating system for any course correction.

The page table entry is also a convenient place for the operating system to place other information pertinent to each page. For example, the page table entry may contain the information needed to bring in a page from the disk upon a page fault.

8.7.3 Inverted page tables

Since the virtual memory is usually much larger than the physical memory, some architectures (such as the IBM Power processors) use an *inverted page table*, essentially a frame table. The inverted page table alleviates the need for a per-process page table. Further, the size of the table is equal to the size of the physical memory (in frames) rather than virtual memory. Unfortunately, inverted page tables complicate the logical to physical address translation done by the hardware. Therefore, in such processors the hardware handles address translations through the TLB mechanism. Upon a TLB miss, the hardware hands over control (through a trap) to the operating system to resolve the translation in software. The operating system is responsible for updating the TLB as well. The architecture usually provides special instructions for reading, writing, and purging the TLB entries in privileged mode.

8.8 Summary

The memory subsystem in modern operating systems comprises the *paging daemon, swapper, page fault handler,* and so on. The subsystem works in close concert with the CPU scheduler and the I/O subsystem. Here is a quick summary of the key concepts we learned in this chapter:

- Demand paging basics including hardware support, and data structures in the operating system for demand-paging
- Interaction between the CPU scheduler and the memory manager in dealing with page faults

- Page replacement policies including FIFO, LRU, and second chance replacement
- Techniques for reducing the penalty for page faults including keeping a pool of page frames ready for allocation on page faults, performing any necessary writes of replaced pages to disk lazily, and reverse mapping replaced page frames to the displaced pages
- Thrashing and the use of working set of a process for controlling thrashing
- Translation look-aside buffer for speeding up address translation to keep the pipelined processor humming along
- Advanced topics in memory management including multi-level page tables, and inverted page tables

We observed in Chapter 7, that modern processors support page-based virtual memory. Correspondingly, operating systems on such modern processors such as Linux and Microsoft Windows (NT, XP, and Vista) implement page-based memory management. The second-chance page replacement policy, which we discussed in Section 8.3.5, is a popular one due to its simplicity and relative effectiveness compared to the others.

8.9 Review Questions

1. Consider an architecture wherein for each entry in the TLB there is:
   - 1 reference bit (that is set by hardware when the associated TLB entry is referenced by the CPU for address translation)

   - 1 dirty bit (that is set by hardware when the associated TLB entry is referenced by the CPU for a store access).

   These bits are in addition to the other fields of the TLB that we have discussed above.

   The architecture provides three special instructions: one for sampling the reference bit for a particular TLB entry (Sample_TLB(entry_num)); one for clearing the reference bit for a particular TLB entry (Clear_refbit_TLB(entry_num)); and one for clearing the reference bits in all the TLB entries (Clear_all_refbits_TLB()). Come up with a scheme to implement page replacement using this additional help from the TLB. For full credit, you should show data structures and pseudo code that the algorithm maintains to implement page replacement.

2. A process has a memory reference pattern as follows:

   1 3 1 2 3 4 2 3 1 2 3 4

   The above pattern (which are the virtual page numbers of the process) repeats throughout the execution of the process. Assume that there are 3 physical frames.

   Show the paging activity for a True LRU page replacement policy. For full credit show an LRU stack and the page that gets replaced (if any) for the first 12 accesses. Clearly indicate which accesses are hits and which are page faults.

3. A process has a memory reference pattern as follows:

   4 3 1 2 3 4 1 4 1 2 3 4

   The above pattern (which are the virtual page numbers of the process) repeats throughout the execution of the process.

   What would be the paging activity for an optimal page replacement policy for the first 12 accesses? For such a policy indicate which accesses are hits and which are page faults.

4. A processor asks for the contents of virtual memory address 0x30020. The paging scheme in use breaks this into a VPN of 0x30 and an offset of 0x020.

   PTB (a CPU register that holds the address of the page table) has a value of 0x100 indicating that this process's page table starts at location 0x100.

   The machine uses word addressing and the page table entries are each one word long.

   PTBR = 0x100

VPN	Offset
0x30	0x020

   The contents of selected memory locations are:

Physical Address	Contents
0x00000	0x00000
0x00100	0x00010
0x00110	0x00000
0x00120	0x00045
0x00130	0x00022
0x10000	0x03333
0x10020	0x04444
0x22000	0x01111
0x22020	0x02222
0x45000	0x05555
0x45020	0x06666

   What is the physical address calculated?

   What are the contents of this address returned to the processor?

   How many memory references would be required (worst case)?

5. Consider a virtual memory system with 20-bit page number. This exercise is to work out the details of the actual difference in page table sizes for a 1-level and 2-level page table arrangement. For a 2-level page table assume that the arrangement is similar to that shown in Figure 8.18. We are only interested in the page table that has to be memory resident always. For a 2-level page table this is only the first level page table. For the single level page table, the entire page table has to be memory resident. You will have to work out the details of the PTE for each organization (1-level and 2-level) to compute the total page table requirement for each organization.