Now suppose there are 15,000,000 computer installations in the world, and each has 20 Gbytes of disk storage. How much virtual address space (in bits of virtual address) would be required to map all of that storage into your virtual address space?

33. The IBM 801 has only 16 segment registers, even though 4096 segments are possible. Give some reasons why they did not provide more segment registers.

34. Suppose you wanted to extend the file mapping interface to allow you to modify the length of a file that is mapped. What would you add to the existing system calls, and what new system calls would you add to do this? Be sure it is clear how they are used to change the length of a mapped file.

chapter

12

Virtual Memory Systems

In the last chapter, we looked at the mechanisms of virtual memory in an operating system. In this chapter, we will look at page replacement algorithms and at some extensions and variations of the paging idea.

12.1 PAGE REPLACEMENT

placement

In a dynamic memory allocation system, the principle problem was *placement*—where would we allocate a block. In a paging system, placement is trivial, since all page frames are the same size and interchangeable. In a virtual memory paging system, the principal issue is *replacement*—which page should we remove from memory so we can bring in a new page. What we need is a *page replacement algorithm*, an algorithm that will tell us which page to replace. A large number of page replacement algorithms have been proposed and investigated over the years.

replacement

page replacement algorithm

An important early decision to make in deciding on a page replacement algorithm is whether it would replace only another page belonging to the process that needs the new page, *local replacement*, or should all the pages in memory be considered for replacement, *global replacement*. Global replacement is simpler and is more commonly used, so we will discuss it first.

local replacement

global replacement

12.2 GLOBAL PAGE REPLACEMENT ALGORITHMS

12.2.1 MEASURING THE PERFORMANCE OF A PAGE REPLACEMENT ALGORITHM

We are going to look at several page replacement algorithms, and so the question comes up, how do you decide among them? The main performance measure is the number of page faults they generate. To compare two algorithms, you compare them on a particular sequence of page references and see which one produces fewer page faults. Of course, you have to do this for a range of page reference sequences that are typical of the processes you want to run under this paging algorithm. Unfortunately, there is a wide range of paging behavior among programs, and it is hard to say what typical behavior is.

Another performance question is not how two page replacement algorithms compare with each other, but how an algorithm compares to the best that can be done with this particular page reference sequence. This brings us to our first page replacement algorithm.

12.2.2 OPTIMAL PAGE REPLACEMENT

In order to make judgments on the effectiveness of a page replacement algorithm, it is useful to have a bound on how well any algorithm can do. So we want to think about an *optimal page replacement algorithm*, that is, an algorithm that produces the fewest possible page faults.

optimal page replacement algorithm

We claim that an algorithm that replaces the page that will not be used for the longest time is an optimal algorithm. The way to reduce the page fault rate is to increase the average time between page faults. The optimal algorithm puts the page fault on the page being replaced as far in the future as possible.

Replacing the page that will be unused for the longest time is the optimal algorithm, but this is not a generally useful algorithm since it requires knowledge of the future behavior of the program, and we do not generally have that knowledge.

Sometimes, however, we might actually have that knowledge. If we run a program once and observe its page reference sequence, then we can predict the page reference sequence the next time we run the program. So, in certain special cases, we might have future knowledge.

The optimal algorithm is still interesting, however, because it gives us a lower bound on page faults and allows us to see how well another algorithm is doing compared to the optimal one. If we have an algorithm that is within a few percent of the optimal algorithm, it may not be worth our trouble to try to find a better one.

Optimal page replacement is not realizable but is a standard to compare other page replacement algorithms against.

12.2.3 Theories of Program Paging Behavior

theory of program behavior

All global page replacement algorithms try to emulate the optimal algorithm and replace the page that will not be used for the longest time; in other words, all page replacement algorithms try to predict the future. They do this by looking at the past behavior of the programs running in the system, and use this information to predict future behavior. Each page replacement algorithm embodies a theory about how programs behave. The theory says that if a program has been acting a certain way in the past, it will act a certain way in the future. If the theory is accurate, then the page replacement algorithm will work well; otherwise it will not.

When discussing each algorithm, we will discuss its theory of program behavior.

Page replacement algorithms have a theory of how past program behavior predicts future program behavior.

12.2.4 Random Page Replacement

The optimal algorithm required knowledge of the future to work. The other extreme would be an algorithm that requires no knowledge at all (of the future or of the past) to work. The *random page replacement algorithm* is such an algorithm. It does not take anything about the program's paging behavior into account, but simply picks a page at random to replace. Figure 12.1 shows one view of random page replacement.

random page replacement algorithm

The theory of the random page replacement algorithm is that the past is of no use in predicting the future and should not be considered.

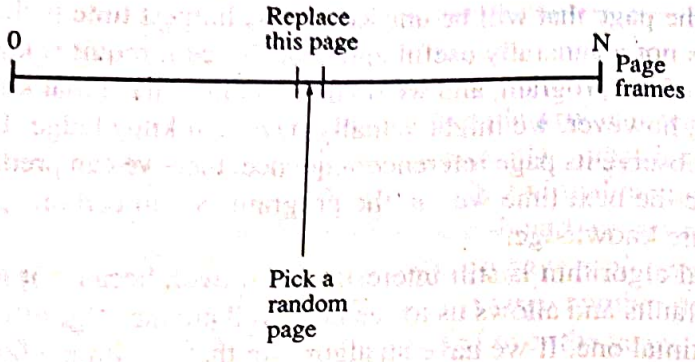


Figure 12.1 Random page replacement

This algorithm is easy to implement and is not the worst possible page replacement algorithm. One advantage of random page replacement is that its performance is predictable. Other algorithms make some assumptions about how programs will behave. If those assumptions are valid, the algorithm will work well, but if they are not valid, then the algorithm will work badly, worse than random replacement. Let's think of this in another way. Suppose the program deliberately tries to make the paging algorithm perform badly. It does this by accessing pages in such a way that the page replacement algorithm's choices always turn out wrong. In other words, it acts as an adversary. But a program cannot do this against random page replacement because its choices are unpredictable and there is no way for the program to "defeat" the random page replacement algorithm. There is no algorithmic way a program can act to make random page replacement do any worse than it would with a well-behaved program.

Just as the optimal algorithm is the upper bound on algorithm performance, the random replacement algorithm should be the lower bound of performance. It is possible for algorithms to be worse than the random algorithm, but any one that is should not be used.

Random replacement is useful where you think that worst cases are likely.

12.2.5 First-In, First-Out Page Replacement

FIFO page replacement

In *first-in, first-out (FIFO) page replacement* you replace the page that has been in memory the longest. The implementation is easy: you just keep a pointer that starts at the first page frame and continually cycles through all the page frames, replacing each page as the pointer passes it.

The theory behind FIFO is that old pages are not referenced as much as new pages, that programs keep moving on to new program and data areas and do not go back to old code or data that much. This theory does not seem to be true in practice, and the FIFO page replacement algorithm works poorly. The problem is that

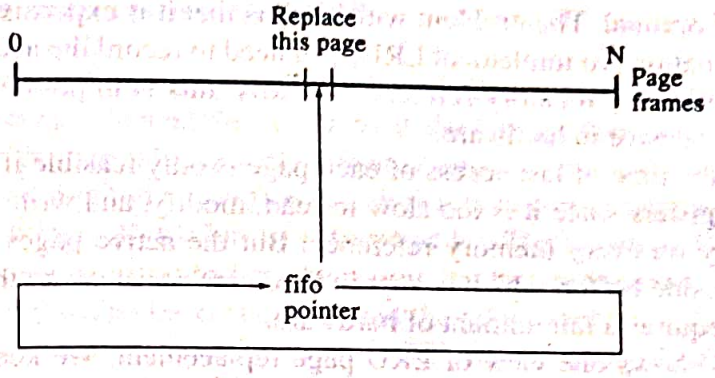


Figure 12.2 FIFO page replacement

programs tend to have certain pages of instructions and data that they use over and over all through the execution of the program. Even though these pages are heavily used, under FIFO they eventually get old and are paged out. Generally they get referenced again very soon, and so the page written out has to be read in again right away. Figure 12.2 shows one view of FIFO page replacement.

Many stores are stocked using a FIFO theory. Take a bookstore for example. When someone wants to buy a book, it is probably a book that was published recently. If you wanted to buy an old book, you probably would have bought it already, so most customers want new books. Therefore, bookstores stock mostly books that have been published recently.

FIFO is not a good paging algorithm.

12.2.6 Least-Recently Used Page Replacement

LRU page replacement

Least-recently used (LRU) page replacement keeps track of the last time each page was accessed and replaces the page that has not been accessed in the longest time.

The theory behind LRU is that pages are in active use for a while, and once they stop being actively used they are not used for a long time. Another way to say this is that the recent past is a good predictor of the near future. LRU predicts that the future will be a reflection of the past around the present. That is, recently accessed pages will be accessed again soon, and pages that have not been accessed for a while will not be accessed for a while in the future. Figure 12.3 shows this idea pictorially. In the extreme, the optimal page to replace, the one that will not be accessed for the longest time in the future, is the page that has not been accessed for the longest time in the past. Pages that have been used recently will be used again soon, and pages that have not been used recently will not be used again soon.

LRU is a very good theory and is one of the best global paging algorithms. Simulation studies (Coffman and Varian, 1968) have shown LRU to be within 30 percent

to 40 percent of optimal. The problem with LRU is that it is expensive to record the necessary information. To implement LRU, you need to record the access time of the accessed page on every memory reference. Clearly, this is impossible in software, and it is even expensive in hardware.

Recording the time of last access of each page is only feasible if the page table entries are in registers since it is too slow to read, modify, and write the page table entry in memory on every memory reference. But the active pages will be in the translation lookaside buffer, which is kept in hardware registers, so this approach is possible but it requires a fair amount of hardware.

Figure 12.4 shows one view of LRU page replacement. We keep a list of the page frames in order of their most recent use. A page reference moves a page frame to the front of the list (and moves the other page frames down one slot). On a page fault, the last page frame in the list is removed from memory.

Any software implementation must be an approximation of LRU.

LRU is an excellent paging algorithm, but it must be approximated by the implementation.

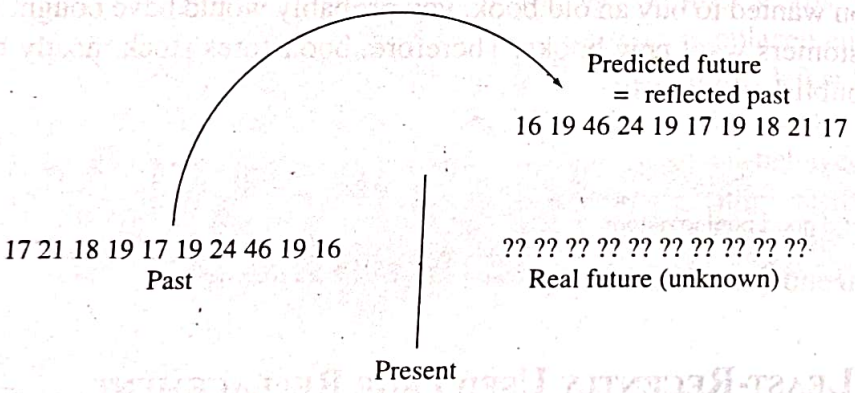


Figure 12.3 LRU model of the future

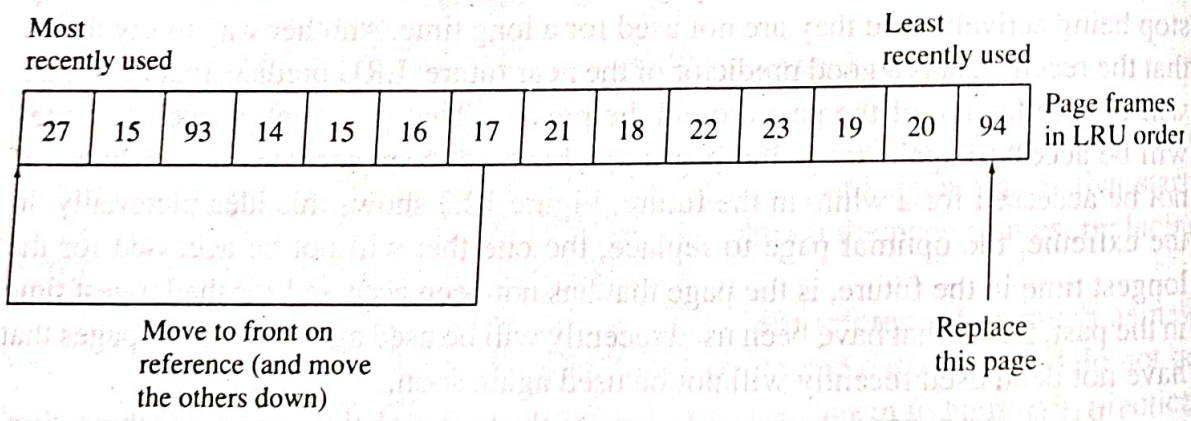


Figure 12.4 LRU page replacement

12.2.7 Approximations of LRU

Most LRU approximations use a combination of software and hardware. They use some simple hardware mechanism to provide the basic information, and then use software to develop approximate LRU information.

The hardware required most often is a bit that tells you whether a page has been referenced or not. We call this bit the *referenced bit*. The referenced bit is a field in the page table entry that is set to 1 by the paging hardware whenever that page is accessed. There are machine instructions to read the referenced bit and to set it to zero.

referenced bit

Here is how our first approximation of LRU works. Every time we get a page fault, we can scan the pages and see which ones have their referenced bit set. These pages have been referenced recently. We pick a page to replace that has a referenced bit of 0, and then set all the referenced bits back to 0. The referenced bits serve to divide the pages into two groups, those that have been reference since the last page fault and those that have not. This is a very crude approximation of LRU.

We can improve the approximation with more effort in the software. Suppose we keep a counter for each page. This counter is initialized to zero. Every so often (say every second), an LRU daemon will wake up and check the referenced bit of all the pages in memory. For each one that has been referenced in the last second, it will reset the counter back to zero, and for the other pages it will increment the counter by one.

When we get a page fault, the counter for each page tells you how many seconds have gone by since the page was referenced. We pick the one with the highest count as the least-recently used page. Typically this counter will be small, say only eight bits. When it gets to 255, we just leave it there and lump all pages that have not been referenced for 255 or more seconds in the same group.

In true LRU, each page has a unique reference time, and so all pages are ordered from most-recently used to least-recently used. If there are N pages, there are N categories. Our first LRU approximation reduced the number of categories to only 2. All pages referenced since the last page fault were in one category, and all the pages that were not referenced since the last page fault were in the other category. Our second approximation of LRU divided the N pages into 256 categories, which should provide a much closer approximation of LRU. Figure 12.5 shows the LRU approximations.

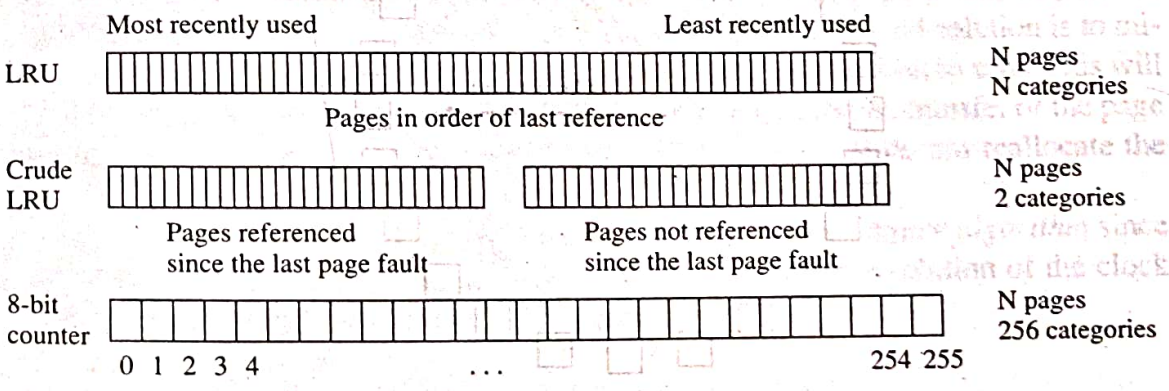


Figure 12.5 LRU and its approximations

12.2.7 Approximations of LRU

Most LRU approximations use a combination of software and hardware. They use some simple hardware mechanism to provide the basic information, and then use software to develop approximate LRU information.

The hardware required most often is a bit that tells you whether a page has been referenced or not. We call this bit the *referenced bit*. The referenced bit is a field in the page table entry that is set to 1 by the paging hardware whenever that page is accessed. There are machine instructions to read the referenced bit and to set it to zero.

referenced bit

Here is how our first approximation of LRU works. Every time we get a page fault, we can scan the pages and see which ones have their referenced bit set. These pages have been referenced recently. We pick a page to replace that has a referenced bit of 0, and then set all the referenced bits back to 0. The referenced bits serve to divide the pages into two groups, those that have been reference since the last page fault and those that have not. This is a very crude approximation of LRU.

We can improve the approximation with more effort in the software. Suppose we keep a counter for each page. This counter is initialized to zero. Every so often (say every second), an LRU daemon will wake up and check the referenced bit of all the pages in memory. For each one that has been referenced in the last second, it will reset the counter back to zero, and for the other pages it will increment the counter by one.

When we get a page fault, the counter for each page tells you how many seconds have gone by since the page was referenced. We pick the one with the highest count as the least-recently used page. Typically this counter will be small, say only eight bits. When it gets to 255, we just leave it there and lump all pages that have not been referenced for 255 or more seconds in the same group.

In true LRU, each page has a unique reference time, and so all pages are ordered from most-recently used to least-recently used. If there are N pages, there are N categories. Our first LRU approximation reduced the number of categories to only 2. All pages referenced since the last page fault were in one category, and all the pages that were not referenced since the last page fault were in the other category. Our second approximation of LRU divided the N pages into 256 categories, which should provide a much closer approximation of LRU. Figure 12.5 shows the LRU approximations.

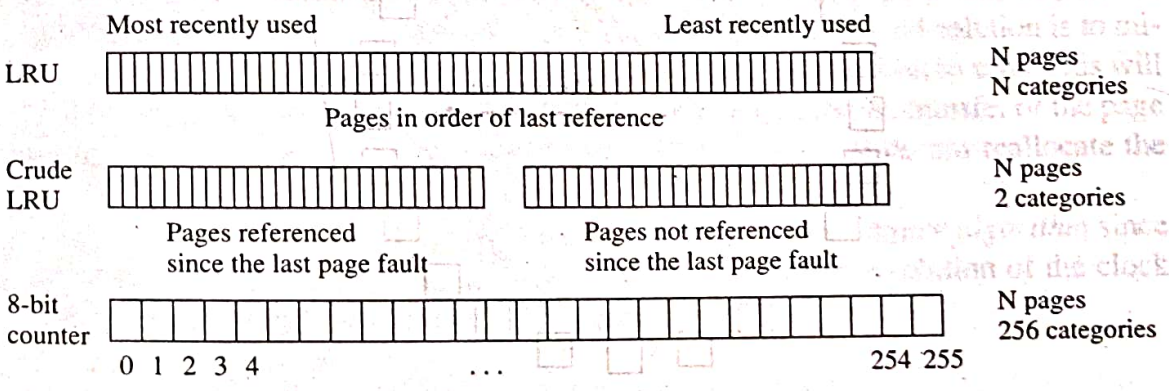


Figure 12.5 LRU and its approximations

these pages induced by their location around the clock face. This ordering means that you do not replace a random page that has not been referenced recently, but you replace the one that has had the longest time to get referenced.

The basic clock algorithm is really a modified FIFO algorithm. In fact, when it was first developed, it was called first-in, not-used, first-out (FINUFO). It is also called *not recently used (NRU).*

not recently used

Just to give you a rough idea of the scales involved, experiments have shown that one clock algorithm had to look at an average of 13 page frames in order to find one to replace.

Using Modified Bits Besides a referenced bit, most paging hardware supports another bit in the page table entry called the *modified bit.* This bit is set to 1 whenever the page is written into. This bit is sometimes called the *dirty bit.*[1]

modified bit
dirty bit

The modified bit is important since a page that has been modified must be written out to disk before it can be replaced. If a page has not been modified, then the copy in the swap area is still correct and there is no need to write it out. Replacing a modified page takes twice as long since there is a disk write to write the old page out, and then a disk read to read the new page in. Because of this, some page replacement algorithms try to avoid writing out modified pages. There is a variation of the clock algorithm that does this. It is shown in Figure 12.7 and described below.

This clock algorithm looks at the page under the clock hand and looks at both its referenced bit and its modified bit. If the referenced bit and the modified bit are 0, then the page is replaced. If the referenced bit is 1, it is set to 0 and the page is passed over and given another chance. If the referenced bit is 0 and the modified bit is 1, then the modified bit is set to 0, the transfer of the page to disk is begun (or at least requested), and the page is passed over.

How can a page be modified and not referenced? Well, normally it can't, but if a page is modified and referenced and the clock hand passes over it the referenced bit is set to 0. The next time the clock hand passes the referenced bit is 0 and the modified bit is 1 (unless it is read or written in the meantime).

Another problem here is that if you set the modified bit to 0, how will you know the page has been modified when it comes time to replace it? There are two solutions to this problem. One is to keep a separate bit array which contains the "real" modified bit. When you set a hardware modified bit to zero, you set the modified bit in this array to remember that the page really has been modified. A second solution is to initiate a transfer of the page to disk as soon as you set the modified bit to zero. This will ensure that the dirty page is written out to disk. When we start the transfer of the page to disk, we remember that this transfer is in progress, and we do not reallocate the page until the transfer is completed.

This clock algorithm variation is also called the *second chance algorithm* since modified pages are given a second chance (that is, a second revolution of the clock hand) to get referenced before they are replaced.

second chance algorithm

[1] Not to be confused with Monty Python's concept of the "naughty bits."

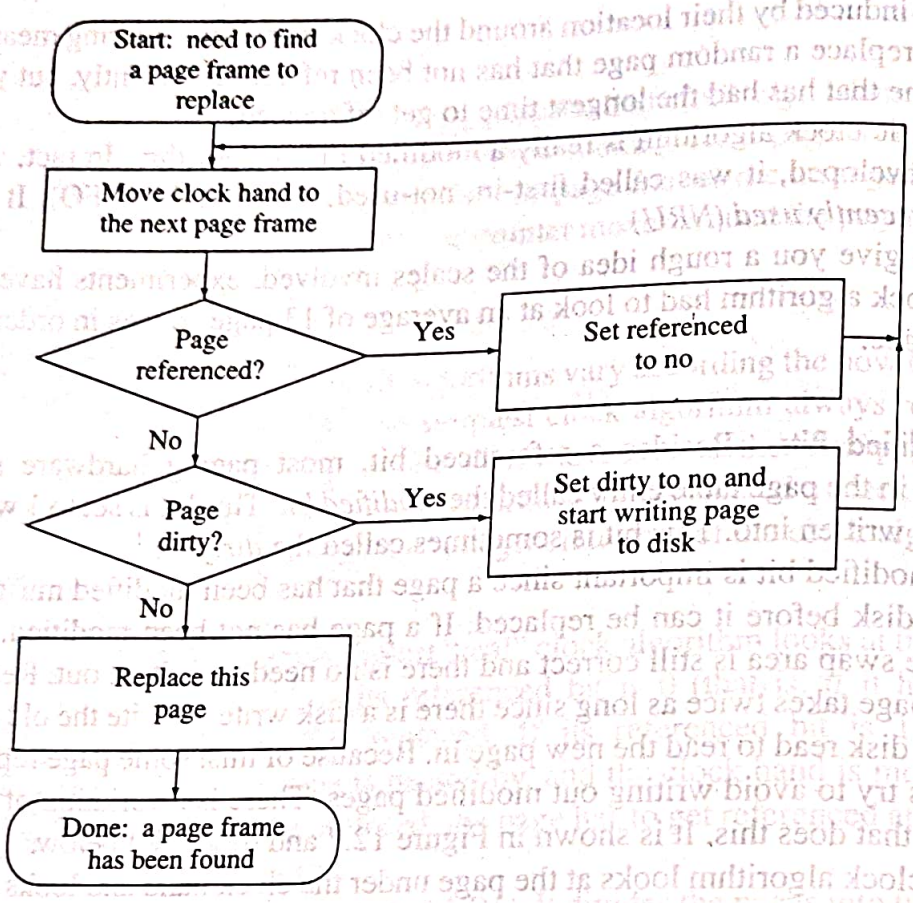


Figure 12.7 *Flow chart for the clock algorithm*

Here is a description of the general clock algorithm:

1. Test the page frame pointed to by the clock hand.
2. If it fails the test, then replace this page and the algorithm is completed.
3. If it passes the test, then:
   a. Modify the page frame information.
   b. Move the clock hand to the next page frame.
   c. Go back to step 1.

This is the algorithm for all the clock variations. Here is a table of the appropriate test and modification for each specific clock algorithm.

Variant	Test	Modification
FIFO	always fail	none
Clock	ref=1?	ref←0
Second chance	ref=1 or mod=1?	if ref=1 then ref←0 else mod←0

Clock algorithms are versatile and effective.

12.3 PAGE REPLACEMENT EXAMPLES

A typical program will have hundreds or thousands of pages. Pages are referenced at least once each instruction (one for the instruction fetch and another for the data fetch, if any), so a typical program execution will contain many millions of page references. Page faults only occur about once every million page references. Because of these large numbers, it is difficult to give realistic examples of page replacement. Instead, we will give radically simplified examples which show the basic idea of the page replacements but which do not represent real program behavior. Since we want to show page replacement in response to page faults, our examples will have lots of page faults, like one every two or three page references. A real program that page faulted at that rate would run 50,000 times slower than it would with real memory.

In the following page fault diagrams, we will show the same page reference string and number of page frames for five algorithms: optimal, LRU, clock, FIFO, and random. The labels on the left side are page frame numbers, and the labels on the top are the page reference numbers. The items in the matrix are the page numbers in each page frame *after* the page reference and subsequent page replacement (if any). An entry is followed by an asterisk if there was a page fault and this was the page just brought in. An entry is followed by a plus sign if the page was referenced and was found in memory. Thus each column has either an asterisk or a plus following one of the page numbers in that column.

The page fault count following the name of the page replacement algorithm does not include the initial four page faults to fill the available page frames with pages.

Let's take a page reference string and see how it would be handled by our different page replacement algorithms. Suppose we have a program that consists of eight pages and we have four page frames of physical memory for it. This means that half of the program will be in memory at any one time. The page reference string will be 12325634637315363424345l. Here are the results:

Optimal (7 page faults)

	1	2	3	2	5	6	3	4	6	3	7	3	1	5	3	6	3	4	2	4	3	4	5	1
0	1*	1	1	1	1	1	1	1	1	1	1	1	1+	1	1	1	1	1	2*	2	2	2	2	1*
1	-	2*	2	2+	2	6*	6	6	6+	6	6	6	6	6	6	6+	6	4*	4	4+	4	4+	4	4
2	-	-	3*	3	3	3	3+	3	3	3+	3	3+	3	3	3+	3	3+	3	3	3	3+	3	3	3
3	-	-	-	-	5*	5	5	4*	4	4	7*	7	7	5*	5	5	5	5	5	5	5	5	5+	5

LRU (10 page faults)

	1	2	3	2	5	6	3	4	6	3	7	3	1	5	3	6	3	4	2	4	3	4	5	1
0	1*	1	1	1	1	6*	6	6	6+	6	6	6	6	5*	5	5	5	5	2*	2	2	2	2	1*
1	-	2*	2	2+	2	2	2	4*	4	4	4	4	1*	1	1	1	1	4*	4	4+	4	4+	4	4
2	-	-	3*	3	3	3	3+	3	3	3+	3	3+	3	3	3+	3	3+	3	3	3	3+	3	3	3
3	-	-	-	-	5*	5	5	5	5	5	7*	7	7	7	7	6*	6	6	6	6	6	6	5*	5

12.3 PAGE REPLACEMENT EXAMPLES

A typical program will have hundreds or thousands of pages. Pages are referenced at least once each instruction (one for the instruction fetch and another for the data fetch, if any), so a typical program execution will contain many millions of page references. Page faults only occur about once every million page references. Because of these large numbers, it is difficult to give realistic examples of page replacement. Instead, we will give radically simplified examples which show the basic idea of the page replacements but which do not represent real program behavior. Since we want to show page replacement in response to page faults, our examples will have lots of page faults, like one every two or three page references. A real program that page faulted at that rate would run 50,000 times slower than it would with real memory.

In the following page fault diagrams, we will show the same page reference string and number of page frames for five algorithms: optimal, LRU, clock, FIFO, and random. The labels on the left side are page frame numbers, and the labels on the top are the page reference numbers. The items in the matrix are the page numbers in each page frame *after* the page reference and subsequent page replacement (if any). An entry is followed by an asterisk if there was a page fault and this was the page just brought in. An entry is followed by a plus sign if the page was referenced and was found in memory. Thus each column has either an asterisk or a plus following one of the page numbers in that column.

The page fault count following the name of the page replacement algorithm does not include the initial four page faults to fill the available page frames with pages.

Let's take a page reference string and see how it would be handled by our different page replacement algorithms. Suppose we have a program that consists of eight pages and we have four page frames of physical memory for it. This means that half of the program will be in memory at any one time. The page reference string will be 12325634637315363424345l. Here are the results:

Optimal (7 page faults)

	1	2	3	2	5	6	3	4	6	3	7	3	1	5	3	6	3	4	2	4	3	4	5	1
0	1*	1	1	1	1	1	1	1	1	1	1	1	1+	1	1	1	1	1	2*	2	2	2	2	1*
1	-	2*	2	2+	2	6*	6	6	6+	6	6	6	6	6	6	6+	6	4*	4	4+	4	4+	4	4
2	-	-	3*	3	3	3	3+	3	3	3+	3	3+	3	3	3+	3	3+	3	3	3	3+	3	3	3
3	-	-	-	-	5*	5	5	4*	4	4	7*	7	7	5*	5	5	5	5	5	5	5	5	5+	5

LRU (10 page faults)

	1	2	3	2	5	6	3	4	6	3	7	3	1	5	3	6	3	4	2	4	3	4	5	1
0	1*	1	1	1	1	6*	6	6	6+	6	6	6	6	5*	5	5	5	5	2*	2	2	2	2	1*
1	-	2*	2	2+	2	2	2	4*	4	4	4	4	1*	1	1	1	1	4*	4	4+	4	4+	4	4
2	-	-	3*	3	3	3	3+	3	3	3+	3	3+	3	3	3+	3	3+	3	3	3	3+	3	3	3
3	-	-	-	-	5*	5	5	5	5	5	7*	7	7	7	7	6*	6	6	6	6	6	6	5*	5

Each global page replacement algorithm assumes a theory of process behavior, but the global algorithms do not treat each process individually. Instead, they lump them all together in one group and look at the paging characteristics of the entire group together. We can do better page replacement if we treat each process individually.

Researchers have been working on developing more accurate models of program paging behavior since the 1960s. This development is an excellent example of the *science* in computer science. At each stage, a mathematical model of program behavior was proposed. Then the model was validated by comparing it to the behavior of real programs. Usually the model was found to be insufficient to explain the behavior of real programs, and so a more complex model was tried. The currently accepted model of program behavior is a phase model of working sets.

12.4.1 What Is a Working Set?

working set

At any one time, a process is only using a subset of its pages. This is the principle of locality that makes paging feasible. Intuitively, the *working set* of a process is the set of pages that it is currently using. If all these pages are in memory, then the process will not get any page faults. This description of the working set is vague, and since it is the set of pages a process is currently using, it involves predicting the future again. To formulate the concept into something we can compute, we will have to define the working set in terms of past page references. To do this, we will need the concept of *virtual time* in a process. Each page reference is one unit of virtual time, and we will number the time units from 1 to T (the present time). At each unit of virtual time, the process makes a reference to a page. The *reference string* is the sequence of pages $r_1, r_2, r_3, \ldots, r_T$ where r_t is the page referenced at virtual time t. We will look back at a certain number, θ, of page references, and the set of pages referenced in that interval will be the working set. So the working set is:

virtual time

reference string

$$W(T, \theta) = \{p | p = r_t \text{ where } T - \theta < t \leq T\}$$

Like LRU, the working set model uses the theory that the recent past will be like the near future, that is, the set of pages that have been recently used is the set of pages we will be using in the near future.

The working set theory is that a program needs all the pages in its working set in memory in order to run efficiently, and so the operating system should only run a process when all the pages in its working set are loaded into memory.

12.4.2 Program Phases

A working set is, informally, the subset of its pages that a program is currently using and, formally, the subset of its pages that has been referenced in the last θ page references. In each case, the important fact is that the working set is a subset of

the process' pages. Since a program will, in most executions, reference all its pages, this implies that the working set will change as time goes on.

Originally, researchers held the idea that the working set of a program slowly changed over time as the program entered new phases of its work. Measurements of real program behavior have shown that this "slow drift" model of program phase changes is incorrect. Instead, programs change phases abruptly, and the working set changes radically in a short time. Each working set is called a *program phase*. Each transition between phases causes a lot of page faults.

program phase

The currently accepted model of program behavior is that a program goes through a series of phases. Within each phase, the working set is fairly stable, and page-referencing behavior can be modeled with any one of several simple probabilistic models of program behavior. But the transitions between these phases are short and sudden.

Measurements show that 95 percent to 98 percent of the virtual time of a process is spent in phases where the working set is stable for a period of time. Hence, only 5 percent to 2 percent of a process' time is spent in transitions between phases. However, 40 percent to 50 percent of the page faults occur in these transitions, and page fault rates are 100 to 1000 times higher in the transitions than they are within the phases.

Let's take a specific example of a program with phases. We will use a program that has very distinct phases to make the ideas show up clearly. Suppose a program has three arrays and it is going to sort them one after the other. Each array uses 10 pages of memory and it takes 200,000 page references to sort one of the arrays. We will use a value of $\theta = 1000$, that is, we will look at the last 1000 page references to compute the working set. We will only look at references to data pages, not program pages.

The following table shows the real working set for this program and what is computed by $W(t, \theta)$.

t	Number of Pages in the Working Set	Number of Pages in $W(t, 1000)$
0–199,999	0–9	0–9
200,000–200,999	10–19	0–19
201,000–399,999	10–19	10–19
400,000–400,999	20–29	10–29
401,000–599,999	20–29	20–29

We see that $W(t, \theta)$ is a good approximation of the working set except in the transitions between phases where W takes too much of the past into account. The measured working sets in the transitions are twice as big as the stable working sets within phases. All of the page faults will occur in these transitions.

the process' pages. Since a program will, in most executions, reference all its pages, this implies that the working set will change as time goes on.

Originally, researchers held the idea that the working set of a program slowly changed over time as the program entered new phases of its work. Measurements of real program behavior have shown that this "slow drift" model of program phase changes is incorrect. Instead, programs change phases abruptly, and the working set changes radically in a short time. Each working set is called a *program phase*. Each transition between phases causes a lot of page faults.

program phase

The currently accepted model of program behavior is that a program goes through a series of phases. Within each phase, the working set is fairly stable, and page-referencing behavior can be modeled with any one of several simple probabilistic models of program behavior. But the transitions between these phases are short and sudden.

Measurements show that 95 percent to 98 percent of the virtual time of a process is spent in phases where the working set is stable for a period of time. Hence, only 5 percent to 2 percent of a process' time is spent in transitions between phases. However, 40 percent to 50 percent of the page faults occur in these transitions, and page fault rates are 100 to 1000 times higher in the transitions than they are within the phases.

Let's take a specific example of a program with phases. We will use a program that has very distinct phases to make the ideas show up clearly. Suppose a program has three arrays and it is going to sort them one after the other. Each array uses 10 pages of memory and it takes 200,000 page references to sort one of the arrays. We will use a value of $\theta = 1000$, that is, we will look at the last 1000 page references to compute the working set. We will only look at references to data pages, not program pages.

The following table shows the real working set for this program and what is computed by $W(t, \theta)$.

t	Number of Pages in the Working Set	Number of Pages in $W(t, 1000)$
0–199,999	0–9	0–9
200,000–200,999	10–19	0–19
201,000–399,999	10–19	10–19
400,000–400,999	20–29	10–29
401,000–599,999	20–29	20–29

We see that $W(t, \theta)$ is a good approximation of the working set except in the transitions between phases where W takes too much of the past into account. The measured working sets in the transitions are twice as big as the stable working sets within phases. All of the page faults will occur in these transitions.

more than the working set, then those extra pages will not be referenced and that memory is not used productively.

Since working sets change as the program makes a transition to another phase, the size of the working set, and hence the resident set, should change during the execution of a program. So the size of the resident set should change as the program changes phases.

The global page replacement algorithms do change the resident set size of the programs, but in an indirect way. A more direct approach is to directly measure the working set of a program and make sure the resident set is exactly the working set. This means that a local page replacement scheme will vary the number of page frames allocated to a process as its working set changes.

12.4.4 The Working Set Paging Algorithm

working set

The *working set* concept can be translated into a local page replacement algorithm. We monitor the working set for each process by computing W as described above. For each process, we keep its working set in memory while it is running. As the computed working set (determined by W) changes, we will change the resident set of the process to reflect those changes. Sometimes we will increase the size of the resident set and sometimes we will reduce it. By implication, a process cannot be run unless its entire working set will fit into memory. If the working set of a process will not fit into memory, then the process is removed from memory completely until its working set will fit.

The working set algorithm is a very good algorithm, and some believe (Denning, 1980) that it is close to the optimal realizable page replacement algorithm. Unfortunately, it is hard to implement the working set algorithm efficiently.

The problem with implementing the working set algorithm comes down to computing W. Computing W is theoretically possible, since it does not require predicting the future, but practically it is not feasible to compute W directly. It is like LRU in that to compute it would require extensive hardware support. What is generally used is a software approximation of W.

12.4.5 Approximating the Working Set

One hardware requirement for implementing the working set algorithm or an approximation of the working set algorithm is a timer that records the virtual time of each process. Fortunately, the process management part of the operating system is probably already doing this. When a process is dispatched, the dispatcher notes the time on a high-resolution (one microsecond or shorter) timer. When the process is interrupted, the elapsed time is added to the accumulated total execution time in the process descriptor. The accumulated total execution time is used as the virtual time of the process. This will not be exactly the reference number used in defining the working set, but it is sufficient for our purposes.

Here is an algorithm that will approximate the working set page replacement algorithm.

DESIGN TECHNIQUE: WORKING SET CONCEPT

Some ideas from operating systems have close analogies in other areas of computer science or even beyond. For example, the idea of *thrashing* is familiar to all of us who have too many things to do. We find that the overhead of switching between tasks ends up taking up a lot of our time, and we find it more efficient to go hide somewhere and get one task done without interruption.

The working set model has been applied to user interfaces in an interesting way. A working set is the minimum number of page frames a program needs to run without excessive page faulting. The idea can be applied to screen space and windows. Certain tasks require information from several sources and so need a window on each source. If the screen space is not sufficient to display all these windows, then the user will waste a lot of time switching between and rearranging windows. This idea was used to develop the idea of multiple workspaces, each of which holds the working set of windows for a particular task.

Anyone who has tried to get certain types of work done on an airplane will have experienced the same phenomenon. The tray table is quite small, and if you have several documents to work on, you will end up spending a lot of time switching documents and trying to figure out where to put a document you are going to need again in a minute.

Here again, we use models to inspire ideas rather than to prove them. The idea of working sets and thrashing (when the working set is not present) can inspire the idea for a window manager with multiple workspaces, each holding the working set of windows for one task. This analogy does not prove that the multiple workspace idea is a good one. This is proved by trying it out and seeing if it improves the way people work with windows.

Models and analogies are for inspiration, not proof.

1. When a page fault occurs, scan through all the pages that process has in memory.
2. If the referenced bit is set, then set the "time of last use" of that page to the current virtual time of the process.
3. If the referenced bit is not set, then check the "time of last use" field of the page. If the page has not been used within θ (the working set parameter) time units, then remove the page.
4. Bring in the new page and add it to the resident set of the program.

Step 3 removes a page from the resident set if it has not been used within the working set window (θ). Step 4 adds a new page to the resident set since the page fault indicates that it is now part of the working set. So the size of the resident set (that is, the measured working set) may move either up or down or may stay the same after each page fault.

This algorithm requires a scan of all the resident pages each time there is a page fault. This could be a large overhead, and so even this approximation of the working set algorithm may be too slow for practical use.

12.4.6 WSClock Paging Algorithm

There is a variant of the clock algorithm that does a good job of simulating the working set algorithm and is very efficient as well. The algorithm is based on the observation

that a page is in the working set of a process if it has been referenced in the working set window. When we look at a page, we see if it has been referenced in that window, and remove it from memory if it has not.

WSClock algorithm

Here is the *working set clock (WSClock) algorithm.*

1. Examine the referenced bit of the page frame the clock hand is currently pointing to. Say page N is in this page frame.
2. If the referenced bit is on, then:
   a. Turn off the referenced bit.
   b. Set LastUsedTime(N) to the virtual execution time of the process that owns the page.
   c. Move the clock hand to the next page frame.
   d. Go to step 1.
3. If the referenced bit is off, compare (where P is the process the page belongs to):

$$\text{CurrentVirtualTime}(P) - \text{LastUsedTime}(N) < \theta$$

   a. If the difference is less than θ (the size of the working set window), then:
      (1) Move the clock hand to the next page frame.
      (2) Go to step 1.
   b. If the difference is more than θ and the page is not modified, then use this page frame to hold the page that the process just faulted on.
      (1) Remove this page from the resident set.
      (2) Use this page frame to hold the page that the process just faulted on.
      (3) Exit the algorithm.
   c. If the difference is more than θ and the page is modified, then:
      (1) Schedule the page for page out and continue looking for a page to replace.
      (2) Go to step 1.

The page frames are in a circular list, as they are in all clock algorithms. When you need to find a page to replace, you look at the next page frame. If the referenced bit is on, then you turn it off and record the virtual time of the process that owns that page as the "last used time" of the page. This saved time is an approximation of the last time the page was referenced. If the referenced bit is off, then you compare the current virtual process time with the saved "last used time" for the page in that page frame. If this difference is larger than the working set window size, then we can conclude that the page has not been referenced in the working set window and so is not in the working set, so we replace it. Otherwise we move on to the next page frame. We may examine pages from several different processes while looking for a page to replace.

The important difference between this algorithm and the working set approximation we described above is that we do not scan all the resident pages when we get a page fault, but just enough of them to find a page to replace. We may not remove pages from the resident set as quickly as the original working set algorithm, but we do remove them when we need a new page.

The working set clock algorithm looks like a global algorithm, but we are keeping information about the working set of each process. The thing that is "global" about the algorithm is that we decide on a global basis which pages to look at to test whether they are within the working set window for their process.

If all the page frames are examined and none can be replaced (even after waiting for modified pages to be paged out), then memory is overcommitted. We pick a process to swap out and reclaim all of its page frames. This is how WSClock ensures that processes are only run if their entire working set can be resident. WSClock works even better with LT/RT load control (see Section 12.6.6).

This is an efficient and reasonably accurate approximation of the working set algorithm, and simulations (Carr and Hennessey, 1981) show it performs as well as working set.

*12.5 EVALUATING PAGING ALGORITHMS

Now that we have seen several paging algorithms, we need a way to compare them and decide which one to use. As we noted before, each paging algorithm is based in a model of program paging behavior, and if that model is accurate the algorithm will perform well. The problem is that an operating system runs a lot of different programs with different paging behaviors. An operating system will use a single paging algorithm, and that may work well on some programs and poorly on others. We want to use an algorithm that will work well on the largest range of programs.

There are three approaches to the evaluation of paging algorithms: measurement, simulation, and mathematical modeling. In the measurement approach, you implement the algorithm in an operating system and measure its performance while programs are running. This approach is quite expensive and lengthy. More commonly, the measurement approach is used to get example page reference traces to use in simulation approaches to the evaluation of paging algorithms.

The mathematical modeling approach involves finding a mathematical model that describes the behavior of a computer system doing paging. This requires a model of program behavior. We have already seen some models of program behavior. The problem with the mathematical model approach is that models that are simple enough to be able to be analyzed mathematically are usually not accurate enough in the way they model program behavior. We will discuss the simulation approach in the next section.

12.5.1 Methodology for Paging Simulation

In the simulation approach, we use simulation techniques to measure the performance of paging algorithms. Figure 12.9 shows the stages you have to go through in order to do paging simulation experiments.

The first thing to do in using simulation is to get a suite of "typical" programs, that is, programs that represent the kind of programs that you will be using. In addition to

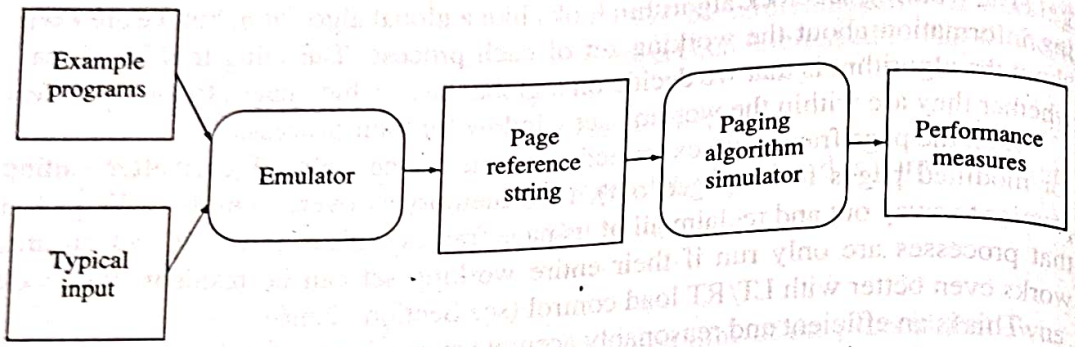


Figure 12.9 Steps in the simulation analysis of page replacement algorithms

getting typical programs, you need to get typical input data to drive the programs. This is not an easy thing to do, and what is typical changes over time anyway, but there is no other way to proceed.

Then you run each program with one or more sets of typical data, using a program that records the address reference string of the program. Such a program will contain an *emulator* for the hardware architecture the program is compiled to run on. The emulator will interpret and execute each machine instruction and record all the page references. Emulation of this sort causes a slowdown of from 50 to 500 times, so gathering this data requires a lot of machine time. Since programs make many millions of memory references, this trace data is very large and requires a lot of disk space to store it.

emulator

A *page reference string* is the sequence of pages a program references during a particular execution. It contains one reference for each memory reference the program makes. The result of the emulator step is a number of page reference strings representing typical program behavior.

page reference string

The next tool you need is a program that can read in a page reference string and compute how many page faults a particular paging algorithm would generate with this reference string. That is, you simulate the operation of the paging algorithm. By examining the page reference string, you can figure out which pages would be in memory, when page faults will occur, and what page will be replaced. You must write a simulator for each paging algorithm you want to evaluate and compare the number of page faults each algorithm generates.

The problem with this comparison is that there are three variables you need to consider. For each test run, you have to specify:

- The page reference string to use.
- The size of the page.
- The number of page frames available.

You want to use representative page reference strings. That means very long strings, and a wide range of them. Each simulation runs a long time, and you have to run a lot of them to get representative results. You will want to repeat all these runs

for a range of page sizes and numbers of page frames. It should be clear by now that the simulation approach is very resource intensive, but this is generally true of all uses of simulation.

12.5.2 Some Page Simulation Results

The simulation approach is the one most commonly used to evaluate paging algorithms. We have quoted a number of results about page replacement algorithms in the text already and most of these have come from simulation experiments. One of the surprising things is that most of the paging algorithms that have been proposed perform about the same. The graph in Figure 12.10 shows the ranges of performance that you typically see. There are two things to note about this chart. The first is that all of the "reasonable," realizable algorithms perform about the same, and pretty close to the optimal algorithm. The second thing to observe is the sharp "knee" in the chart. If a process does not have enough page frames, then it will fault excessively no matter what page replacement algorithm you choose. Conversely, if a process has enough page frames, then all of the page replacement algorithms will perform well.

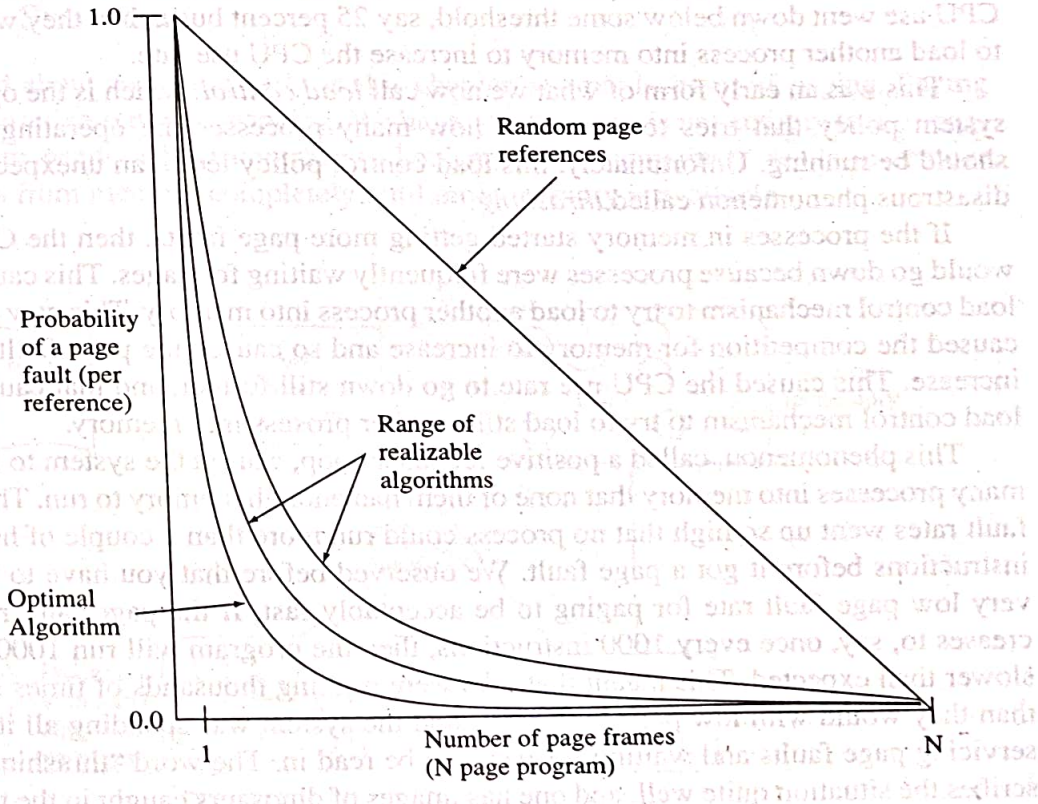


Figure 12.10 Range of performance of page replacement algorithms

The point here is that there are two things a paging system must decide: how many page frames to give a process, and what pages to keep in those page frames. Figure 12.10 indicates that the second decision is fairly easy to make (since the algorithms do not vary much) and the first decision is critical to performance.

The way to ensure that processes have enough page frames to run efficiently (without excessive page faulting) is to be sure that there are not too many processes in memory at the same time. Deciding how many processes to have in memory is called load control, and we will discuss it in the next section.

12.6 THRASHING AND LOAD CONTROL

Early paging systems experienced a form of catastrophic performance failure called thrashing. It was caused by incorrect load control.

12.6.1 How Thrashing Occurs

One purpose of paging is to pack more processes into memory so that there will always be some process ready to run and not waiting for I/O. This way we keep the CPU busy. In the early paging systems, keeping the CPU busy was an important goal, and so they were set up to detect when the CPU busy percentage got too low. When CPU use went down below some threshold, say 25 percent busy, then they would try to load another process into memory to increase the CPU use rate.

This was an early form of what we now call *load control,* which is the operating system policy that tries to determine how many processes the operating system should be running. Unfortunately, this load control policy led to an unexpected and disastrous phenomenon called *thrashing*.

thrashing

If the processes in memory started getting more page faults, then the CPU use would go down because processes were frequently waiting for pages. This caused the load control mechanism to try to load another process into memory. This new process caused the competition for memory to increase and so caused the page fault rate to increase. This caused the CPU use rate to go down still further, and that caused the load control mechanism to try to load still another process into memory.

This phenomenon, called a positive feedback loop, caused the system to load so many processes into memory that none of them had enough memory to run. The page fault rates went up so high that no process could run more than a couple of hundred instructions before it got a page fault. We observed before that you have to have a very low page fault rate for paging to be acceptably fast. If the page fault rate increases to, say, once every 1000 instructions, then the program will run 1000 times slower than expected. This meant that jobs were running thousands of times slower than they would with low page fault rates, and the system was spending all its time servicing page faults and waiting for pages to be read in. The word "thrashing" describes the situation quite well, and one has images of dinosaurs caught in the tar pits whose thrashing around to get free just caused them to sink more rapidly into the tar.

12.6.2 Load Control

load control

What was learned from observing the phenomenon of thrashing was the necessity for better *load control*, that is, the operating system has to monitor the paging rate and reduce the level of multiprogramming if the paging rate is too high. Virtual memory is not magic, and you cannot run as many programs as you want simultaneously or else thrashing will occur. The early systems had a form of load control, but it was based on CPU utilization, which is the wrong measure. Load control must be based on the paging rate.

The idea of load control is quite simple. You monitor the page fault rate by incrementing a counter every time there is a page fault. Every second or so, you check the page fault rate to see if it is over some threshold. If it is, you remove one process from memory, that is, you swap it out and give all its page frames to other processes. A new process can be started or a swapped-out process can be swapped in only if the page fault rate is low enough.

Figure 12.11 shows the difference between the tasks of a page replacement algorithm and a load control algorithm. They do logically different things, but they are closely enough related that they have to communicate with each other (load control uses information that the page replacement algorithm collects). The working set algorithm is a page replacement algorithm that includes load control.

12.6.3 Swapping

We talked about swapping early in this chapter as a predecessor of paging. Paging was supposed to replace swapping. We have seen that this is not entirely true. A paging system needs a load control mechanism, and it sometimes needs to remove processes from memory completely until more memory is available.

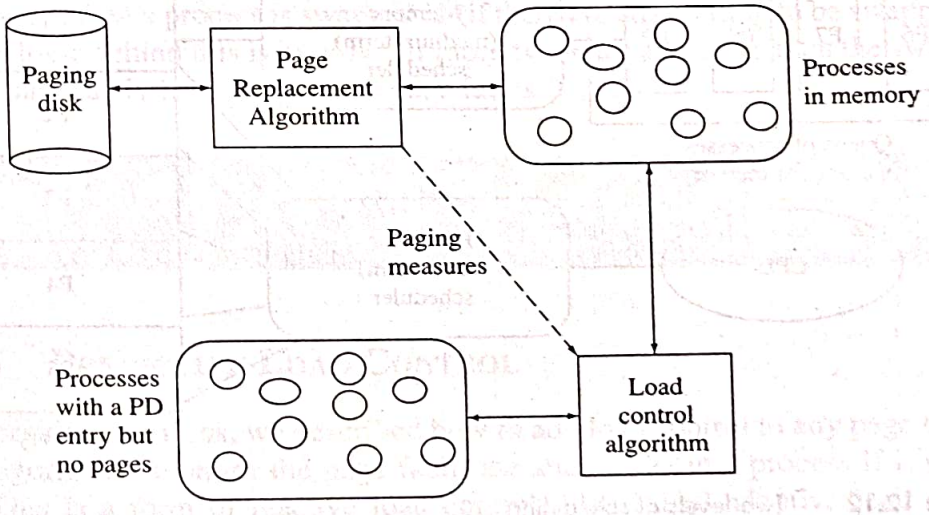


Figure 12.11 Load control and page replacement

In swapping, we write the entire process out to disk. In a paging system, a copy of the process is already on disk. To swap a process out, all we have to do is to release all the page frames it is using, and mark it "swapped out" in its process descriptor. This will cause the dispatcher to ignore it, and it will stop getting processor time. Its page frames can be used by other processes. To swap a process in, it is only necessary to change its status in its process descriptor to runnable again. Soon the dispatcher will dispatch it, and it will start running again and competing for page frames with the other processes.

12.6.4 Scheduling and Swapping

Swapping creates a scheduling problem. You have to decide which programs should be in memory, when they should be swapped out, and what other program(s) should be swapped in. The memory scheduling has to be coordinated with the processor scheduling.

medium-term scheduler

short-term scheduler

What you need is a two-level scheduling system where the levels are called short-term scheduling and medium-term scheduling. The *medium-term scheduler* (or memory scheduler) decides what jobs will be in memory, and the *short-term scheduler* (or processor scheduler) decides which of the jobs in memory will get to use the processor. (See Figure 12.12.) These two schedulers have to be coordinated since it is important that, as much of the time as possible, at least one of the processes in memory be able to use the processor. That way, at least one process will be making progress. In general, the medium-term scheduler should schedule a mix of processes into memory that will utilize all the resources of the system.

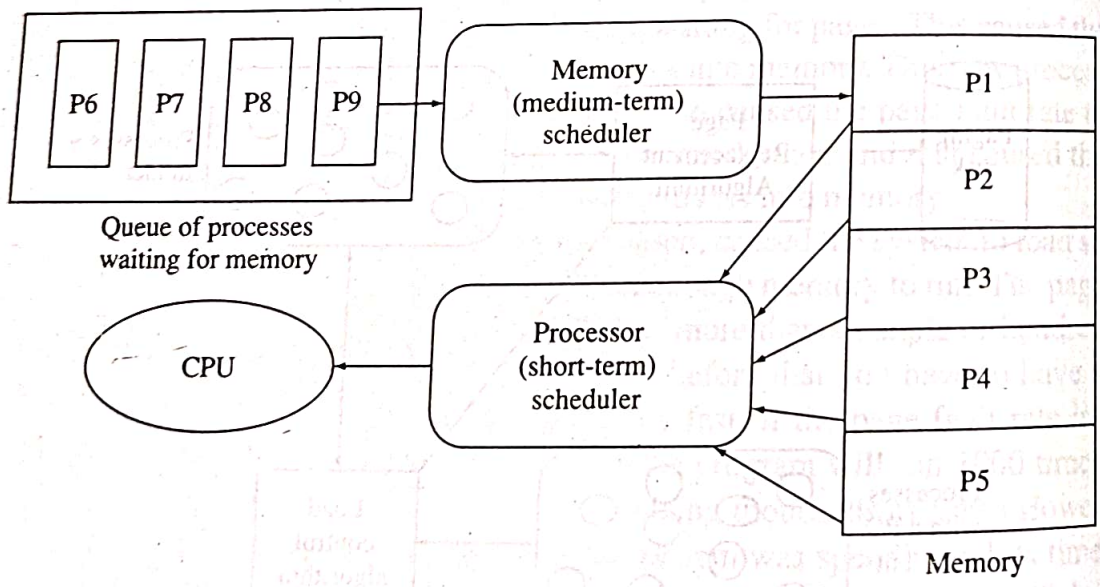


Figure 12.12 Two levels of scheduling

12.6.5 Load Control and Paging Algorithms

Generally load control is an additional mechanism that we add to a page replacement algorithm. The page replacement algorithm is in charge of trying to keep the most useful pages in memory, and the load control algorithm is responsible for deciding how many processes should be in memory. The two mechanisms are independent, but they do interact.

Some Measures of Load Some of the paging algorithms we have encountered have natural load measures. The clock algorithm has a clock hand that moves around to all the page frames. If the clock hand is moving at a low rate (in revolutions per second), then either there are not many page faults or the clock hand does not have to move very far to find a page to replace. In this case, more processes should be brought into memory. If the clock hand is rotating quickly, then there are too many page faults or it is too hard to find a page to replace. This means the memory is overcommitted and we should remove a process from memory.

Page Fault Frequency Load Control If we do not want to use some load measure from the page replacement algorithms (or it does not have any natural ones), we can do more generic load control by monitoring the overall page fault rate to determine whether the load is too high. This is called *page fault frequency load control*. The purpose of load control is to keep the page fault rate at a reasonable level, and so a direct measurement of the page fault rate seems like a good measure. When the page fault rate gets too high, we reduce the level of multiprogramming by swapping out a process.

page fault frequency load control

But how do we decide what paging rates are "too high"? One measure that has been proposed is the $L = S$ criterion. We define L as the mean time between page faults and S as the mean time to service a page fault (assuming no waiting for other processes to use the disk or other time in queues). The $L = S$ load control method tries to keep L equal to S. If L gets higher than S, then a process is swapped out. If L gets lower than S, then a process is swapped in (if there are any waiting to be swapped in).

The logic behind this is intuitive. If those two rates are equal, then the swapping system can exactly keep up with the page faults.

Load control is necessary in any paging system.

12.6.6 Predictive Load Control

In the previous sections, we described how to add load control to any page replacement algorithm—monitor the page fault rate and swap out a process if it gets too high. This is a form of reactive load control (also called adaptive load control)

predictive load control

where we detect that the load is too high and reduce it. The working set algorithm is a *predictive load control* method. It tries to predict what loads will be too high and prevents them from happening. One would expect that predictive load control would be better (if it predicts accurately) because it will not have to suffer reduced performance until the overload is detected.

There is a predictive method of load control that can be added to any paging algorithm, and it is called LT/RT (load time/run time) load control. The idea is based on the facts about phase transitions we talked about in Section 12.4.5. We observed that the page fault rates for programs in transition were hundreds to thousands of times higher than for programs within a program phase. The most predictable transition is when a program is first loaded (or swapped in after being completely swapped out). Then it has to load in all the pages of its working set.

The LT/RT algorithm divides processes into loading processes and running processes. Any process that has just been activated and whose paging rate is still high is called a loading process. The load control strategy is to only allow one loading process (per paging disk) to be executing at a time.

If two loading processes are competing, then the loading phase of each one of them will be lengthened. During this longer loading phase, other programs may finish and allow even more programs to start. This positive feedback can cause the paging device to be saturated, even though memory is not really overcommitted. Page faults during loading indicate loading problems, not too little memory allocated to the process. Page fault frequency load control can be fooled by loading processes.

LT/RT load control has been shown in simulation studies (Carr and Hennessey, 1981) to be very effective—more effective than the working set load control.

12.6.7 Preloading of Pages

A swapped-out process will lose all its page frames, and so when it is swapped back in again it will get a page fault on every page. It seems like we might as well bring in some of its pages right away, rather than have a flurry of page faults while it loads in the pages it needs.

demand paging

Up to now, we have used only what we call *demand paging*, which means that we do not read in a page unless we get a page fault on the page. This is generally a good strategy, since it avoids bringing in pages before they are really needed. An alternative policy is called *prepaging*, which involves reading in pages before they are requested by the program. We only want to prepage a page if we have a good reason to believe it will be accessed in the very near future.

prepaging

But how can we predict which pages will be used? Well, the page replacement algorithms have been doing that all along, and we can use their predictions for prepaging. When a process is swapped out, we remember which pages it had in memory. These are the pages the page replacement algorithm thinks it needs. When the process is swapped in again, we prepage in all the pages it had before.

It is harder to prepage a new process because we have no idea which pages it will need.

12.7 DEALING WITH LARGE PAGE TABLES

In early paging systems, the logical address space was small enough that they could have the page table entirely in hardware registers. This became infeasible with larger address spaces, and so the page tables were moved out of hardware registers into memory to avoid the problems of saving and restoring so many registers. What happens when the page tables get so big they do not easily fit into physical memory? We have organized our paging hardware on the premise that the page tables are always in memory. This is not really feasible, even with 32-bit logical address spaces. We will look at some solutions to this problem.

12.7.1 WHAT IS THE PROBLEM?

First, let's look at the extent of the problem. Suppose we have a 32-bit address space (which is pretty much the standard these days) and 12-bit page offsets (4 Kbyte pages). This leave 20 bits of page number, over one million pages. Each page table entry takes four bytes, so 4 Mbytes of memory is required to hold a full page table. This is for a process that is as large as possible, 4 Gbytes. Few processes will be that big, but programs are getting larger and larger and the sum total of the page tables for 100–200 processes can add up to many megabytes, even if none of them used the total address space possible. As programs get larger, we will have to do something about the size of the page tables.

12.7.2 TWO-LEVEL PAGING

two-level paging

One solution is to reuse the paging idea and page the page tables themselves. This idea is called *two-level paging*, and it is an elegant solution to the problem of large page tables. We will start with an example of a specific two-level paging system.

master page number
secondary page number

Suppose we have a 32-bit virtual address. This will be divided up into 10 (high-order) bits of *master page number,* 10 (middle) bits of *secondary page number,* and 12 (low-order) bits of offset into the (4 Kbyte) page. The hardware page table register points to a master page table containing 1024 entries.(See Figure 12.13.) The master page table is also called the *primary page table*. Each entry is four bytes, and so the master page table is 4 Kbytes long (which conveniently fits exactly into one page frame). The master page number is used to look up the entry in the master page table. The master page table entry contains the address of a secondary page table, which also contains 1024 entries and is 4 Kbytes long. The secondary page number is used to look up an entry in the secondary page table. The secondary page table entry contains the base address of the page, which is added to the offset to get the physical address of the word being accessed.

primary page table

Figure 12.14 gives another visualization of two-level paging. Both of these diagrams only show one secondary page table, but you should note that there will be many secondary page tables, one for each entry in the master page table.

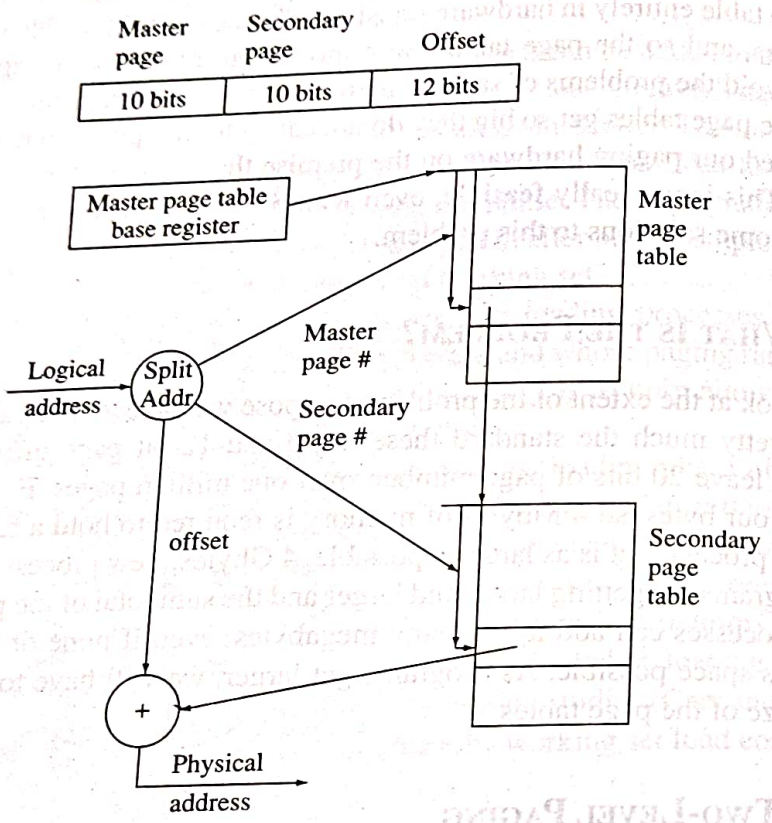


Figure 12.13 Two-level paging

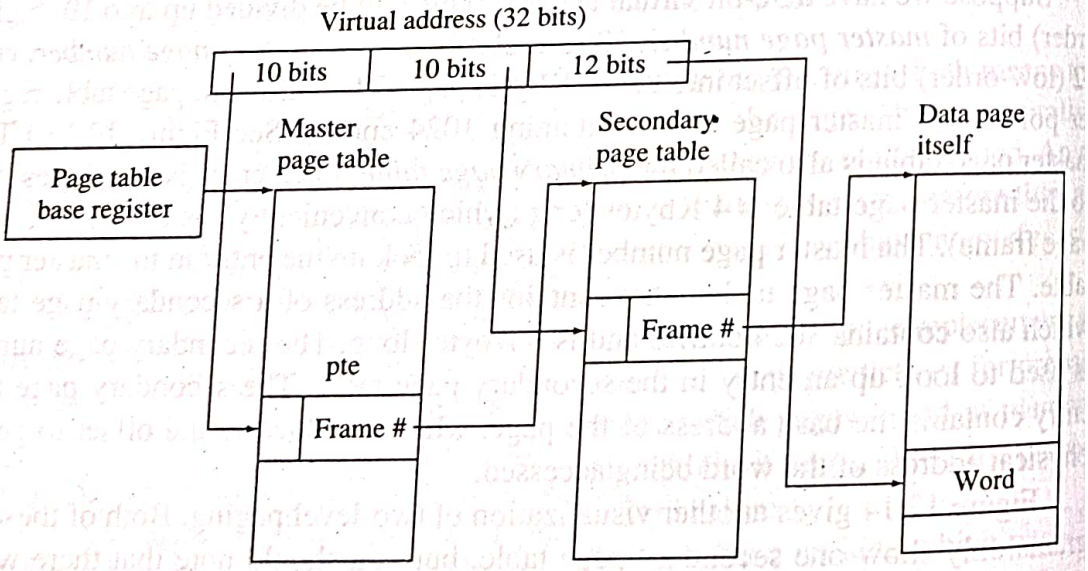


Figure 12.14 Two-level paging (another view)

Each secondary page table entry maps one 4K page, so an entire secondary page table maps 4 Mbytes of memory. One master page table entry maps one secondary page table, or 4M of memory. Thus a program of 4 Mbytes or less will only require two pages of page tables, a one-page master page table and a one-page secondary page table. In general, page tables in a two-level paging system are one page larger than they would be in a one-level paging system.

12.7.3 Benefits of Two-Level Paging

If we use two-level paging, small programs will still have small page tables and large programs will still have large page tables, but we are better off for two reasons. First, the page pages do not have to be in contiguous physical memory. The master page table and the secondary page tables are all one page long and they can be anywhere in memory.

The second benefit of two-level paging is that we can allow page faults when fetching the secondary page tables. That is, the master page table must always be in memory, but we can page the secondary page tables in and out as needed. So a program with a very large page table need not have the entire page table in memory at one time. Because of locality, we can afford to keep a substantial portion of the secondary page tables on disk without generating excessive page faults.

The main benefit of two-level paging comes in larger programs. A large program will only need to have part of its page tables in main memory, and the page table of an inactive program (large or small) will be paged out to disk and use very little main memory (only a page frame for the master page table). We are again exploiting the locality of page references in typical programs.

This method is especially good if we have lots of inactive processes. A process that is not active for a long time will have all of its secondary page tables paged out, and so will consume very few system resources. It is handy for users to keep lots of inactive processes around because it is easy to start them up again. Two-level page tables mean that they can do this without putting much of a load on the system memory.

Another benefit of this organization is that we can have large holes in the logical address space. For example, suppose we call the 4 Mbytes of address space mapped by one secondary page table a *segment*. We could have the code in one segment, the data in another segment, and the stack in another segment. If each segment begins on a 4M boundary, we will have one master page table entry and one secondary page table for each segment. The invalid pages between the segments would allow the hardware to detect when we overran a segment (like the stack).

12.7.4 Problems with Two-Level Paging

One problem with two-level paging is that you have to look up in two page tables for every memory reference. Without caching hardware, that would mean three memory cycles for each memory access. Clearly, a translation lookaside buffer is necessary to avoid these extra memory references.

But, more importantly, two-level paging does not solve the other problems of very large address spaces: the page tables still have to be initialized and the pages still have to exist on disk. Such large page tables will take a long time to initialize, and the large address space will consume a lot of disk space. Finally, two-level paging still does not handle really large address space such as the 64-bit address spaces that are now becoming more prevalent.

three-level paging

If the address space is *really* large, we could go to *three-level page tables*, which extends this idea in the obvious way. (See Figure 12.15). With address spaces this large, we might go to other methods such as inverted page tables. We will look at this in the next section.

12.7.5 Software Page Table Lookups

Paging is normally handled by paging hardware. The operating system sets up the page tables and the appropriate hardware registers, and the hardware handles the page table lookups. As we have observed, however, the hardware does not usually have to look in the page table for the page table entry. Instead, the hardware will normally find the page table entry in the translation lookaside buffer (TLB). Only when the page is not in the TLB does the hardware have to do an actual lookup in the page tables. We want this to happen very seldom, since it requires an extra

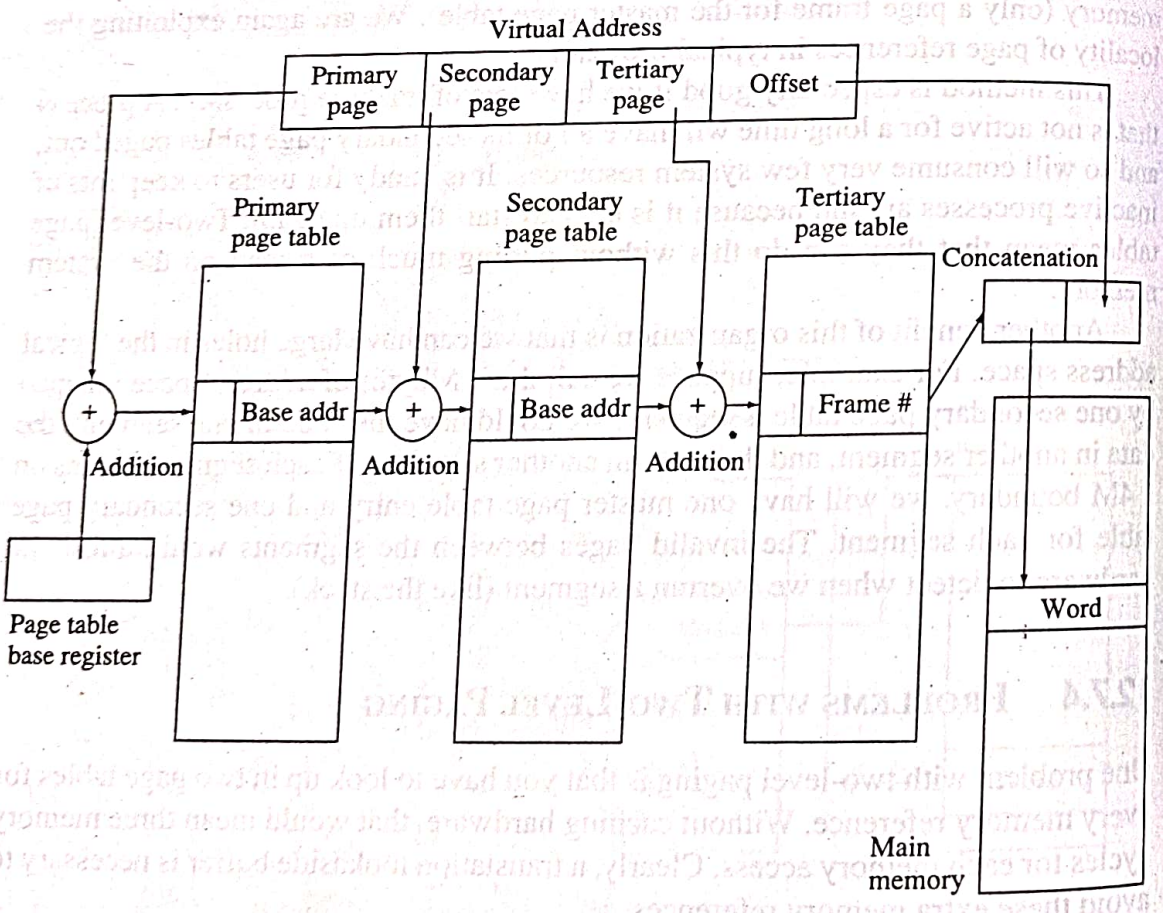


Figure 12.15 Three-level paging

memory cycle (two with two-level paging and three with three-level paging) to fetch the page table entry. If we have a large TLB, we might be able to get the cache miss rate down very low.

So there are three levels of performance and they are shown in the following table which shows the time penalties of three events in four paging schemes.

Event	1-Level Paging	2-Level Paging	3-Level Paging	Software Paging
TLB hit	1	1	1	1
TLB miss	2	3	4	10–50
Page fault	100,000	100,000	100,000	100,000

If the page table entry is in the cache, then the memory word is fetched directly by the hardware and it takes one memory cycle to get the word. If the page table entry is not in the cache, the hardware looks it up in the page table (in memory) and then fetches the memory word. This is called a cache miss, and it takes two memory cycles to fetch the word (three in a two-level paging system). If the page itself is not in memory, then a page fault occurs, and the operating system software takes care of reading the page into memory. This third case takes a very long time because it requires disk I/O. The first two cases are handled by the hardware because both cases are common enough so that they have to be handled very quickly. Page faults are rare enough that we can afford to let the software handle them.

If we provide a large enough TLB, we can reduce the cache miss rate so that cache misses are uncommon. In this case, we could afford to let the operating system software handle cache misses as well as page faults, and the hardware will be simplified. In addition, we can use more sophisticated methods that would be too complex, too expensive, or too experimental to put into hardware. This idea is called *software page table lookups*.

software page table lookups

So here is the idea. The hardware looks up the page in the TLB and fetches the word if it finds the page. If not, the hardware generates a TLB miss interrupt, which is handled by the operating system. The operating system finds the correct page table entry, loads it into the TLB, and restarts the instruction. (See Figure 12.16.) The hardware does not have to handle page table lookups and it does not have to handle cache management.

Inverted Page Tables Since the hardware does not handle paging, it does not dictate the structure of the page tables, so the operating system is free to use any data structures it wants to represent the page tables. A structure that works very well for very large address spaces is an *inverted page table*. An ordinary (noninverted) page table has an entry for each page in the virtual address space and is used to look up the page frame that that page is in. With extremely large address spaces, this becomes a very sparse table since physical memories are not as large as virtual address spaces. An inverted page table has one entry for each page frame in physical memory and is used to look up which virtual page is in the page frame.

inverted page table

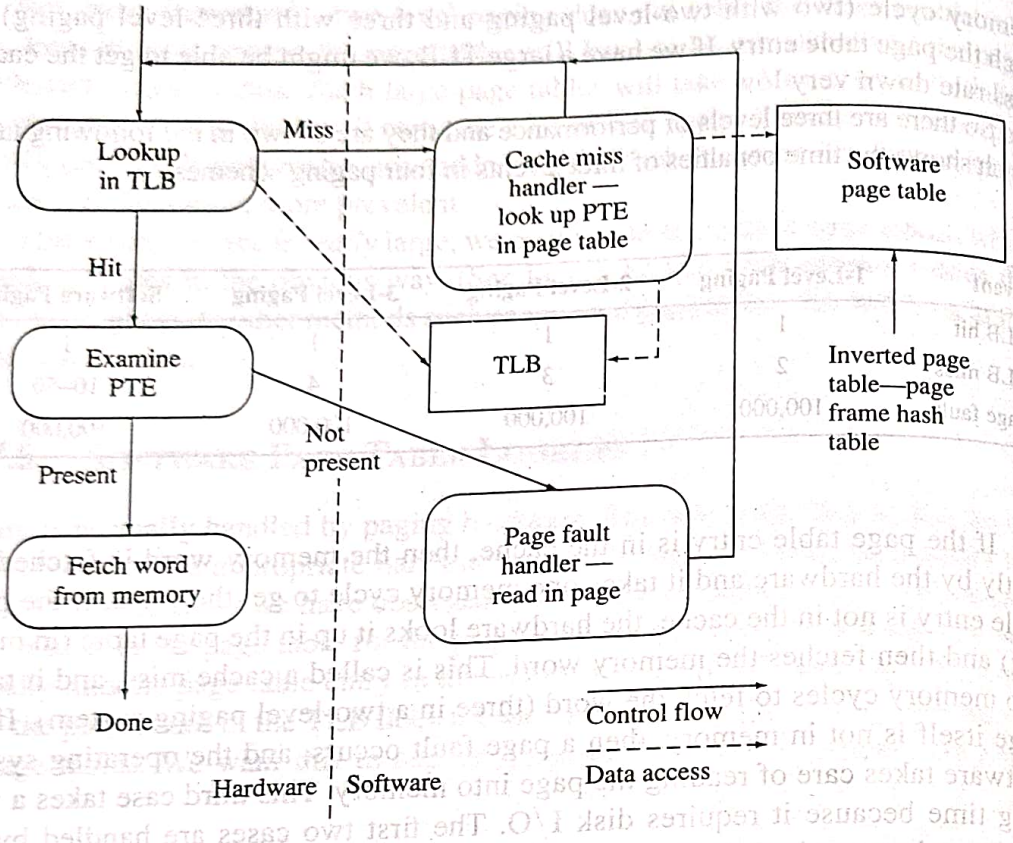


Figure 12.16 Software page table lookup flow chart

This type of page table is "inverted" because it is indexed by the physical page number and contains virtual page numbers, while an ordinary page table is indexed by the virtual page number and contains physical numbers.

The problem with an inverted page table is that it is not fast in the direction we want to go. It is easy to find out which page is in a page frame with an array access, but to find out which page frame a particular page is in requires a search of the entire table. The cache miss will tell us which virtual page is needed, and we have to find the page frame (or really the page table entry). But there are standard data structures to handle lookup problems like this (for example, hash tables, skip lists, binary trees). We can keep a hash table and look up a virtual page very quickly (in constant time).

Figure 12.17 shows normal paging with very large address spaces. Each process has a page table that contains mostly invalid entries that do not provide any information. That is because, with a very large address space, most of it is not in physical memory. The page tables are indexed by the incoming virtual page numbers, and their entries in the page table contain a physical page frame number. The entire page table lookup and access are done in hardware.

Figure 12.18 shows inverted page table paging. There is one inverted page table for the entire system, which is shared by all processes. The virtual page number is found in the inverted page table using a hash table lookup. (This figure shows one rehash to find the virtual page number.) Once this is found, the index of the inverted page table where the virtual page number is found is the physical page frame number

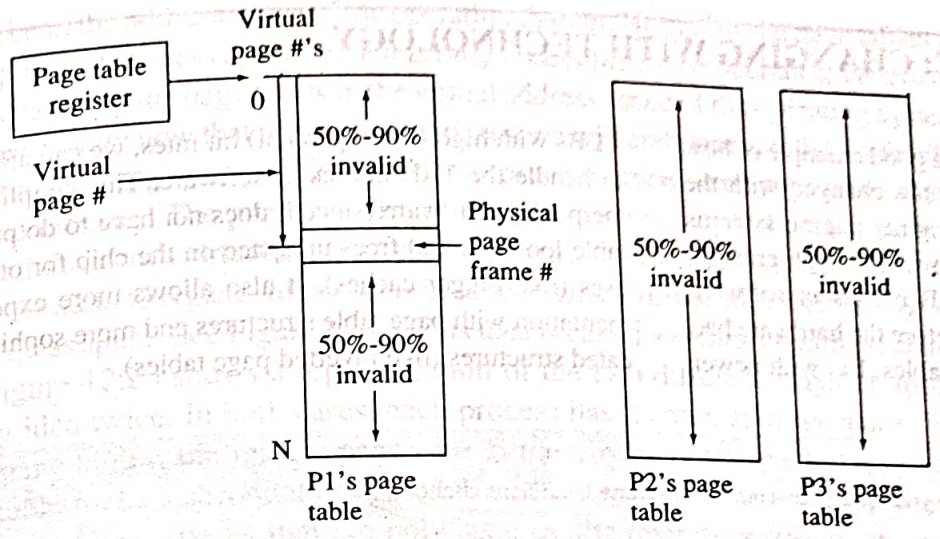


Figure 12.17 Normal page tables

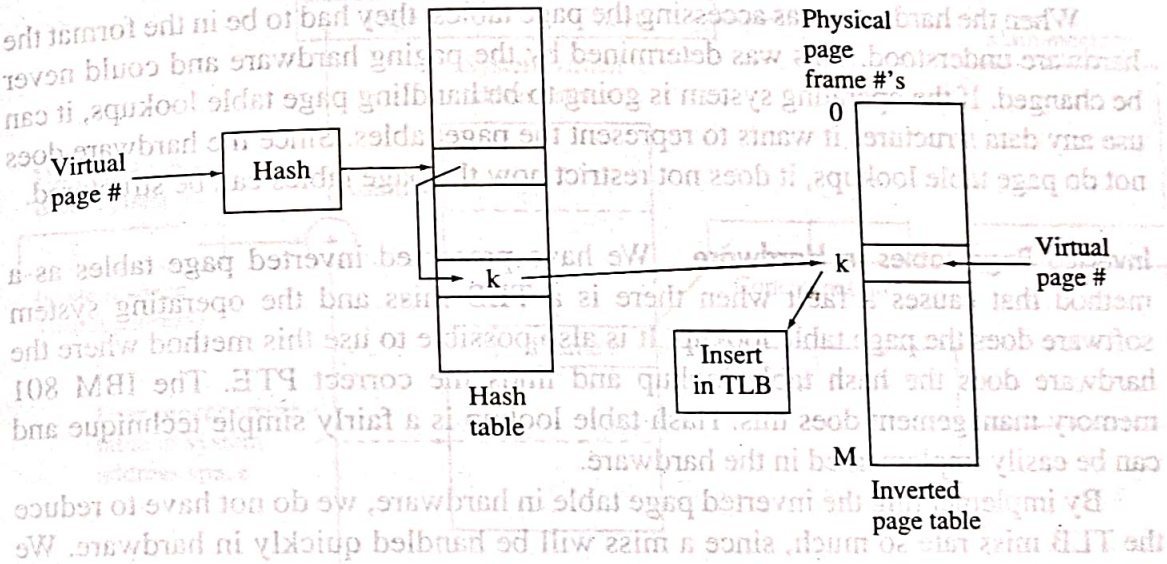


Figure 12.18 Inverted page table

we are looking for. The PTE is reconstructed (from the physical page frame number and the protection bits which are also kept in the inverted page table) and put in the TLB. Then the access is attempted again, and this time there will be a TLB hit. This entire process can be handled in software, although some machines implement hardware lookup in the inverted page table.

Efficiency of Inverted Page Tables Page fault rates have to be extremely low since a page fault requires a disk access which takes a very long time. With software page table lookups, a TLB miss traps to the operating system, which then does the page table lookup. This does not require a disk access, and can be done in less than 50 instructions instead of a few hundred thousand instruction times (like a disk access). So if the TLB miss rate is, say, 0.1 percent, we will have about a 10 percent slowdown.

DESIGN TECHNIQUE: CHANGING WITH TECHNOLOGY

Software page table lookups are a good example of how the design of operating systems has changed with the technology to support it. The original paging systems had small TLBs whose hit rate was not high enough. For these systems, handling TLB misses in software would be too expensive, and therefore the hardware had to handle the mechanics of page tables. But with newer TLBs with high (> 99 percent) hit rates, we can afford to handle the TLB misses in software. This simplifies the paging hardware (since it does not have to do page table lookups) and frees up space on the chip for other uses (like bigger caches). It also allows more experimentation with page table structures and more sophisticated structures (like inverted page tables).

Periodically reexamine and update your design assumptions because conditions change.

When the hardware was accessing the page tables, they had to be in the format the hardware understood. This was determined by the paging hardware and could never be changed. If the operating system is going to be handling page table lookups, it can use any data structures it wants to represent the page tables. Since the hardware does not do page table lookups, it does not restrict how the page tables can be structured.

Inverted Page Tables in Hardware We have presented inverted page tables as a method that causes a fault when there is a TLB miss and the operating system software does the page table lookup. It is also possible to use this method where the hardware does the hash table lookup and finds the correct PTE. The IBM 801 memory management does this. Hash table lookup is a fairly simple technique and can be easily implemented in the hardware.

By implementing the inverted page table in hardware, we do not have to reduce the TLB miss rate so much, since a miss will be handled quickly in hardware. We still get the main advantage of inverted page tables, that is, much smaller page tables. But doing the page table lookup in software has some additional advantages. First, we simplify the hardware, freeing up chip area for more caches or other speedups. Second, we can experiment more easily with different page table organization, different hashing algorithms, different TLB replacement algorithms, etc.

*12.8 RECURSIVE ADDRESS SPACES

Two-level paging works very well, but it is not the only way to apply the idea of paging to page tables. The DEC VAX system used a method that is similar to two-level paging but is different in several crucial respects.

The idea is as follows. The operating system runs in virtual memory also. The virtual memory system creates an address space for each process and one for the the operating system. The page tables should only be accessible to the operating system, and so

should be in the address space of the operating system. Normally, the page tables are kept in the physical address space (which is only accessible to the operating system), but we can instead keep the page tables in the virtual address space of the operating system. Let's examine exactly how that works first, and then we will compare it with two-level paging.

The operating system's virtual address space is mapped with page tables that exist in physical memory. The process's virtual address space is mapped with page tables that exist in the system's virtual address space. The crucial difference is that the hardware page table base address register for a user process contains a virtual address in the system's virtual address space. See Figure 12.19. This idea is called a *recursive address space.*

recursive address space

Figure 12.20 shows a representation of the two different ways of applying the paging idea twice. In both cases, each process has its own address space. With two-level page tables, the master pages are in the physical address space. Each master page table maps a secondary page table address space, one for each process. These are little address spaces that are only seen by the operating system and the paging

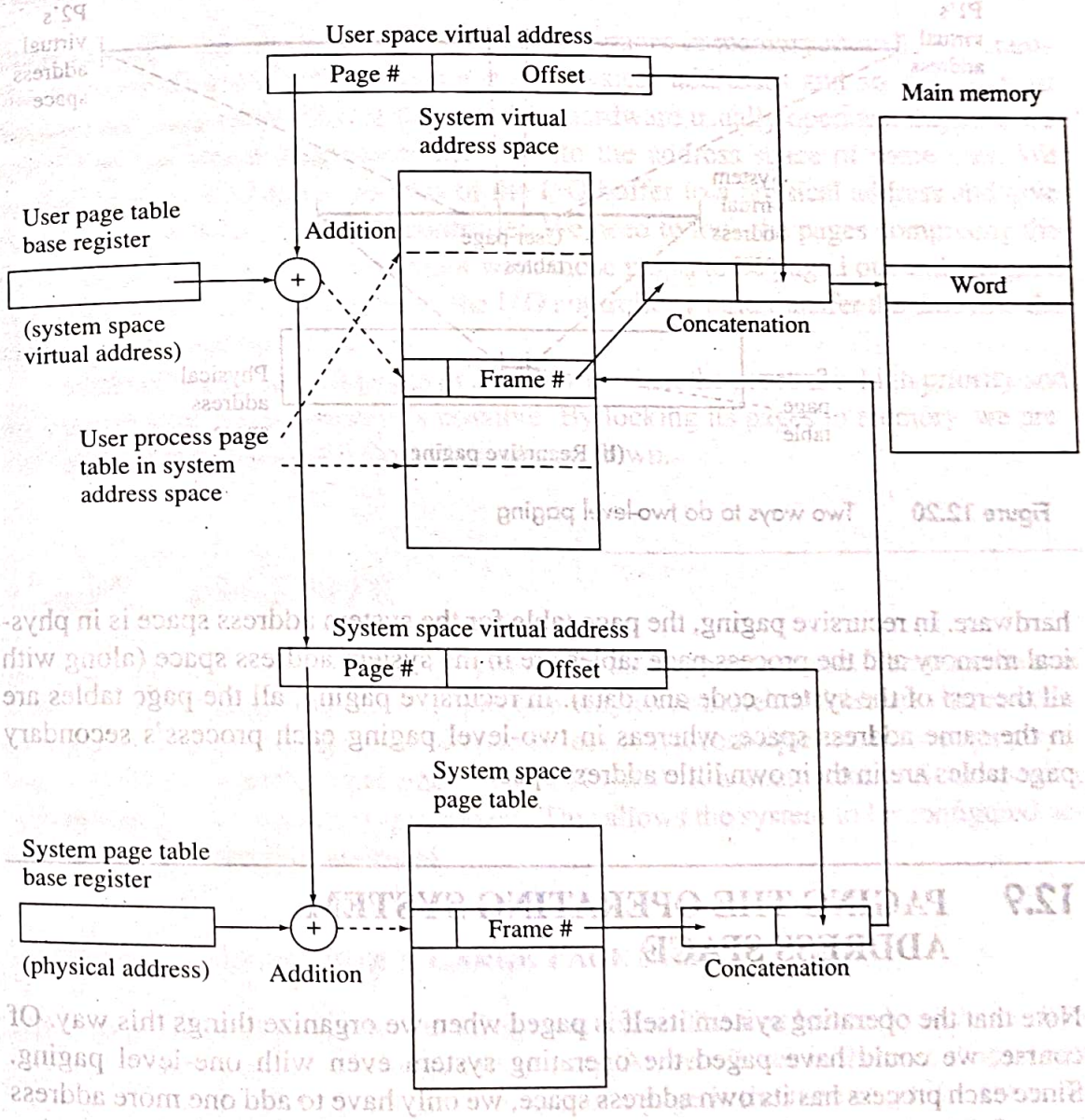


Figure 12.19 Two levels of virtual memory

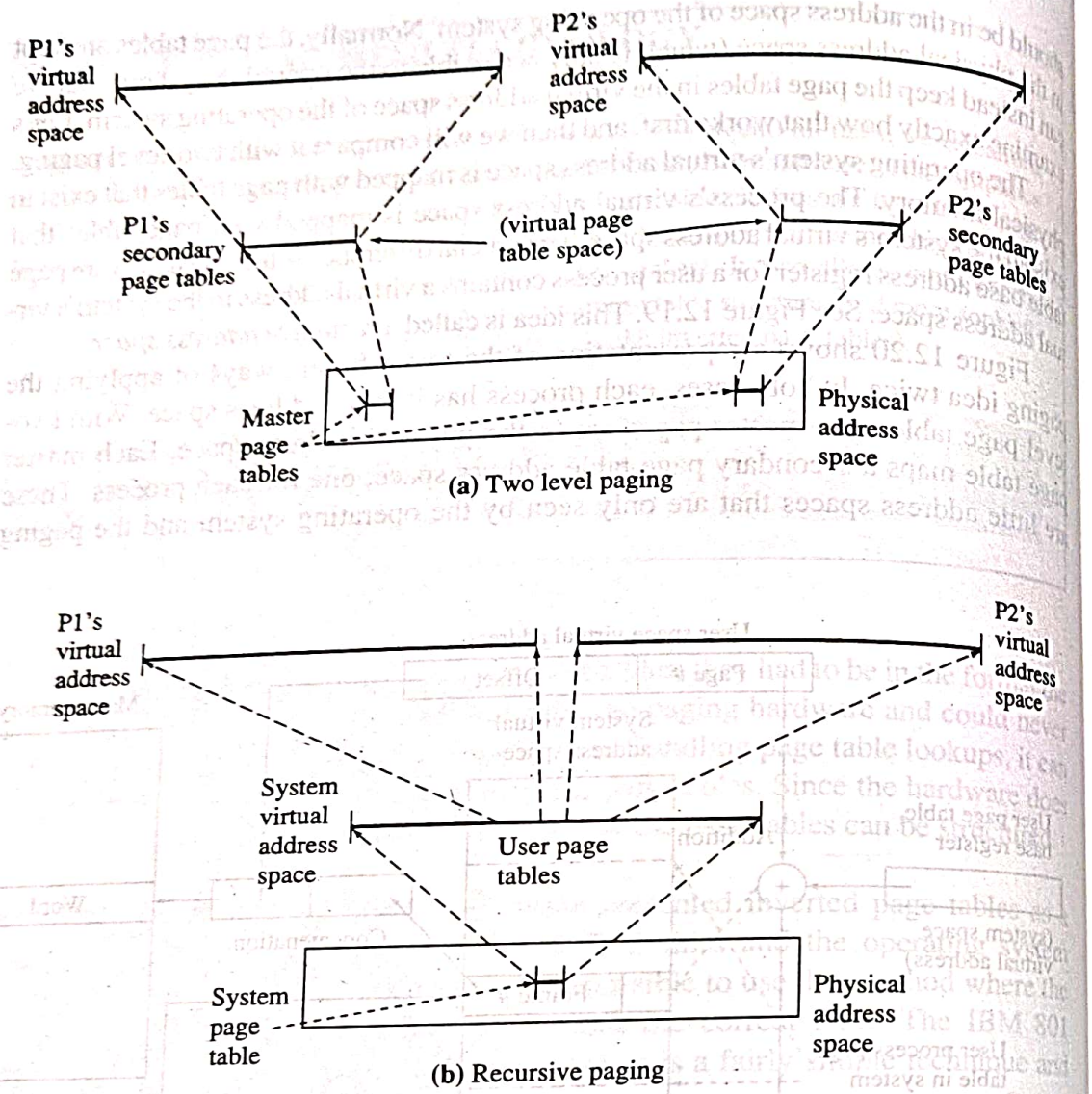


Figure 12.20 Two ways to do two-level paging

hardware. In recursive paging, the page table for the system address space is in physical memory and the process page tables are in the system address space (along with all the rest of the system code and data). In recursive paging, all the page tables are in the same address space, whereas in two-level paging each process's secondary page tables are in their own little address space.

12.9 PAGING THE OPERATING SYSTEM ADDRESS SPACE

Note that the operating system itself is paged when we organize things this way. Of course, we could have paged the operating system even with one-level paging. Since each process has its own address space, we only have to add one more address space for the system and run the operating system with paging on. Since the operating

system can modify its own page tables, it will still be able to access any physical address it wants, although it may have to go to a little more trouble to do so.

But what if we get a page fault and go to the page fault handler, and it gets a page fault? It would seem that we would get into an infinite cycle, and indeed we would. The solution to the problem is "Don't do that." That is, we make sure that certain crucial code, like the page fault handler, never gets paged out. Paging systems provide a way to lock pages in memory and prevent them from being paged out. This might be a flag in the page table entry, or it could be kept in a separate table maintained by the paging system.

What parts of the operating system need to be locked into memory? The paging code, of course, and usually all the interrupt handlers since they need to run quickly and cannot afford to wait for a page to be read in. In general, the critical parts of the operating system that we want to run quickly will be locked in memory.

12.9.1 Locking Pages in Memory

There are other reasons you might want to lock pages in memory as well. For example, suppose that the I/O hardware uses physical addresses and so does not go through the page table. This is the way I/O hardware usually operates. Suppose we wanted to transfer a disk block directly into the address space of some user. We would translate the logical address of the I/O buffer to a physical address and give the physical address to the I/O controller. We need to lock the pages comprising the buffer in memory because we do not want those pages to be paged out and assigned to another process. If that happens, the I/O controller would transfer the data into the memory of another process.

Another reason to lock pages in memory is when the process is high priority and we want it to finish as quickly as possible. By locking its pages in memory, we prevent page faults that would slow the program down.

*12.10 PAGE SIZE

An important decision in a paging system is what page size to use. There are factors that encourage large page sizes (pages of 8 Kbytes or more) and factors that encourage small page sizes (pages of 1 Kbytes or less). As a consequence, most paging systems use some middle-sized page (like 4 Kbytes) as a compromise. Some paging systems allow two (or more) page sizes. This allows the system to be configured according to its expected workload.

12.10.1 Reasons for a Large Page Size

There are two reasons for having a large page size. The first has to do with the characteristics of the disks that are used for paging. As we will see in the next chapter (see Section 14.4), the bulk of the time to read in a page is the time to get to where the page is stored on disk. The time to transfer the data is a small part of the access time.

So it is better to move a lot of data in one access than to have several disk accesses. This consideration might change if we are paging to different devices that have different access time characteristics.

A second reason for large pages is that it takes fewer page faults to bring in a substantial portion of the program. We have observed that half of the page faults occur in loading the working set back again after a process has been swapped out or when a program changes phase to another working set. With large pages, there are fewer page faults to get the working set read in.

12.10.2 Reasons for a Small Page Size

There are two reasons for choosing a small page size. The first is internal fragmentation. On the average, half of the last page will be lost to internal fragmentation. If the page size is small, then this loss is small. On the other hand, smaller pages mean larger page tables, and we get what is sometimes called *table fragmentation*, that is, space wasted on tables.

For example, suppose the average program size is 300 Kbytes. If the page size is 1 Kbyte then the average program will have 300 pages. Each page table entry is four bytes, so the page table will take 1200 bytes. The total fragmentation consists of internal fragmentation (which is, on the average, half the size of the page at the end of each program) plus table fragmentation (which is the size of the page table of the average program). The table in Figure 12.21 shows the internal, table, and total fragmentation for a range of page sizes. Since the two factors vary inversely, we have a minimum. In this case, the smallest total fragmentation is with a 1K or 2K page. Remember that this table only figures in these two kinds of fragmentation, and there are other factors that are affected by the page size.

The second reason for small pages is that they allow us to only have the parts of the process that are really necessary in memory. Suppose you have a 4 Kbyte page which holds two 2 Kbyte procedures, and that one procedure is being heavily used but the other is not being used at all in this phase of the process. The 2 Kbytes occupied by the unused procedure are wasted and should not be in memory. If we used a

Page Size	Internal Fragmentation	Table Fragmentation	Total Fragmentation
8K	4K	150	4.2K
4K	2K	300	2.3K
2K	1K	600	1.6K
1K	512	1200	1.7K
512	256	2400	2.7K
256	128	4800	4.9K

Figure 12.21 Internal and table fragmentation for a range of page sizes

2 Kbyte page, then we could have the useful procedure in memory and the other one could be paged out. Smaller pages allow exactly the parts of the address space we are using to be in memory. Experiments have shown that if the page size is halved, then the average amount of memory required to hold the working set of the program is reduced by 10 percent.

Small pages are good for programs with long phases since we can get the exact working set in memory. Large page sizes are better for programs with short phases since they lower the cost during initial loading and phase transitions. Simulations (Coffman and Varian, 1968) indicate that increasing the number of pages is more effective at reducing page faults than increasing the size of the page.

12.10.3 Clustering Pages

One problem with paging is that, when a program is started, it gets a lot of page faults while it is loading in its pages. This also happens when a process is swapped in after being completely swapped out. In these cases, it has to reload its entire working set into memory. One way to avoid this is to bring in more than one page after a page fault. This idea is called *clustering*. With clustering, we bring in several (two to eight or more) pages at every page fault. The pages around the faulting page are part of the same *cluster*, and they are brought into memory all at the same time. The idea behind this is that page references are localized, and so the pages around a faulting page are likely to be in the working set of the program as well. This is a guess that usually turns out to be correct, and it saves page faults when processes are loading.

clustering

Clustering only affects the loading of pages. The pages are not paged out in a cluster. If we always move the pages in and out in a cluster, we are effectively increasing the page size of the system. This is a different technique (although it is also called clustering) and is used when the page size is too small.

12.11 SEGMENTATION

We have used the word "segment" several times, and now it is time to define it more carefully and discuss the use of segments in memory management.

12.11.1 What Is a Segment?

Paging is an arbitrary division of the logical address space into small fixed-size pieces. Instead of using pages, we could divide the address space of a process into pieces based on the semantics of the program. Such pieces are called *segments*. For example, we could put each procedure or module, array, matrix row, or linked list in a segment. A segment is a division of the logical address space that is visible to the programmer. *Segmentation* leads to a *two-dimensional address space* because each memory cell is addressed with a segment and an offset within that

segment

segmentation

two-dimensional address space

segment. Segments are the size of the logical object they hold, and so are not of any fixed length.

12.11.2 Virtual Memory with Segmentation

Segments can be used as a basis for a virtual memory system that is similar to paging. Figure 12.22 shows segmented virtual memory. Instead of a page table, we have a segment table. Since segments can be of variable length, the segment table entry must contain a length field as well as the segment base address (in addition to protection and presence information). The virtual address is divided into a segment number and an offset into the segment. The segment number is used to index into the segment table to retrieve the segment table entry. The offset is added to the segment base address, and also checked against the segment length. The resulting address is used to access the word in memory. As you can see, the mechanism for segmentation is nearly the same as that for paging.

Segment table entries have a present bit which allows some segments to be kept on disk until they are needed and allows the use of segments for virtual memory.

Segment table entries also have protection bits like page table entries, but they are more useful with segments since everything in a segment is logically related (all

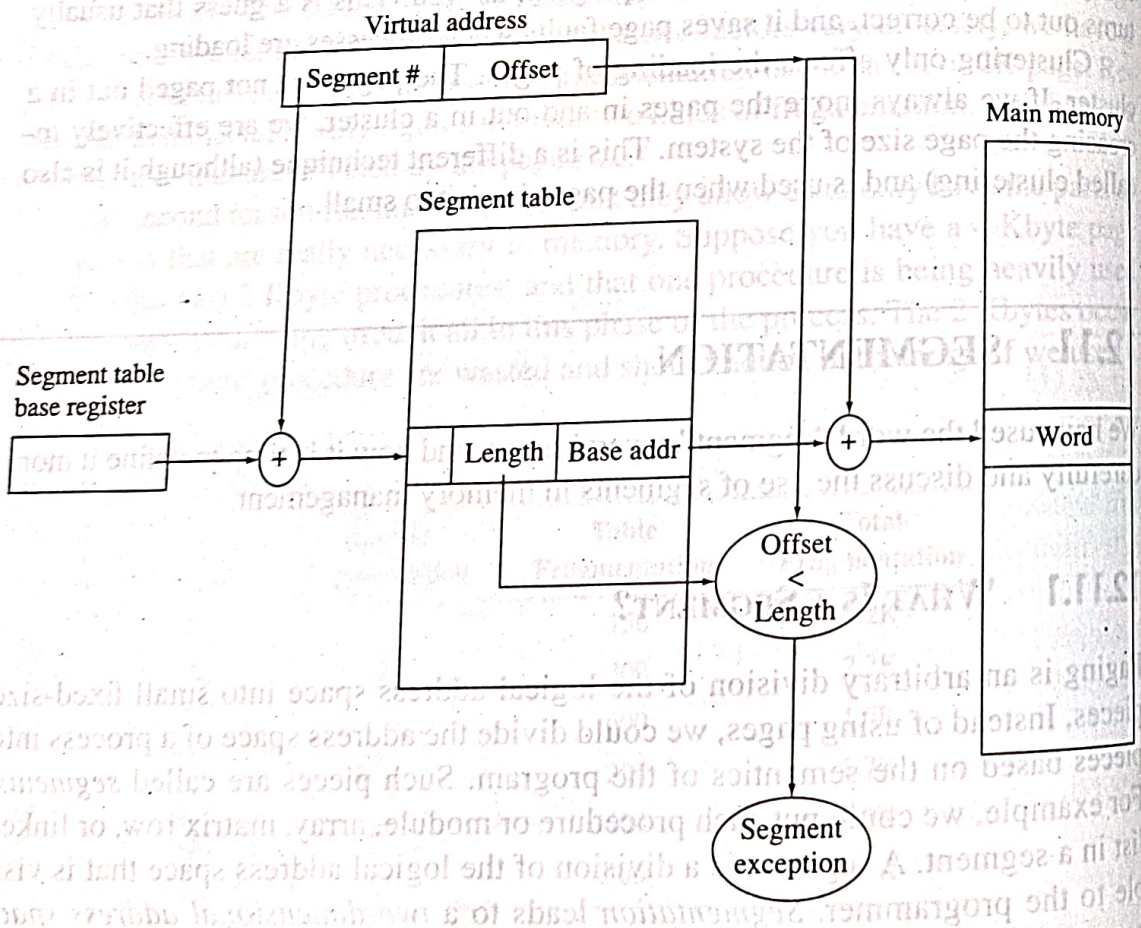


Figure 12.22 Segmentation

one procedure, one array, one list, etc.) and so they will all have the same protection status. In a paged system, a single page may contains parts of two different arrays which should be protected differently.

Are Segments Contiguous to Each Other? Suppose you have a paging system with 1024-byte pages. If you are at byte 1023 of page 12 and you move one byte higher, you get to byte 0 of page 13. Now suppose you have a segmentation system with 64 Kbyte segments, you are at byte 64K-1 of segment 12, and you move one byte higher. In a "true" segmentation system, you will get a segment overflow because the segments are logically distinct and are not contiguous with each other. Segments create a two-dimensional address space where you have a number of linear address spaces. This is what makes the protection of segments work well.

12.11.3 Segmentation with Paging

Segments can be of different lengths, so it is harder to find a place for a segment in memory than a page. With segmented virtual memory, we get the benefits of virtual memory but we still have to do dynamic storage allocation of physical memory. In order to avoid this, it is possible to combine segmentation and paging into a two-level virtual memory system similar to two-level paging. Each segment descriptor points to a page table for that segment. This gives you some of the advantages of paging (easy placement) with some of the advantages of segments (logical division of the program). The problem is that it requires you to use large segments (a number of pages), and that reduces the usefulness of segments to divide the program into logical pieces.

The Burroughs 5000 and 6000 machines had a segmented virtual memory and the average segment size on them was less than 50 bytes. Clearly, paged segments would not have been useful there.

12.11.4 History of Segmentation

In the early days of virtual memory, both paging and segmentation were considered reasonable ways to implement virtual memory. In fact, computer scientists often preferred segmentation because it divided the program into logical segments. This followed the natural partitioning of the program and allowed for more precise protection of the parts of a program.

But segmentation was only used on a few commercial machines, mainly the ones from Burroughs, and pure segmentation is no longer used in any commercial machines. There are several reasons why this is so.

First, paging is transparent to the user, so you can add a paging system to a computer without changing any programs. Paging divides up the logical address space into pages without the user knowing anything about it. Segmentation changes the addressing structure that is visible to the program. All the compilers have to be rewritten to accommodate segmentation. This made it much easier to change to paging.

Second, segmentation requires dynamic allocation of variably sized blocks, which led to external fragmentation and extra time and complexity for the operating system. Paging divides physical memory into equally sized page frames and makes dynamic allocation of space simple.

The only commercial systems using segmentation now are systems that use paged segments. But paged segments nullify some of the advantages of segmentation. You can achieve nearly the same result with two or three levels of page tables.

12.11.5 Segment Terminology

So far, we have used the word "segment" in several different ways. This section talks about the classical use of the word segment, as in a segmented virtual memory. These segments are of varying length and contain some logical program unit. In Section 12.7.3, we used the word "segment" to refer to the 4 Mbyte block of memory mapped by a master page table entry. This was a fixed-length segment, but in the spirit of the word since we talked about putting some logical unit of the program (like the stack) in a segment and isolating it from other parts of the address space using

DESIGN TECHNIQUE: FIXED- AND VARIABLE-SIZED OBJECTS

Dynamic storage allocation works, but it loses some of the space to external fragmentation and it is fairly complex. Because of these problems with dynamic storage allocation, we went to a lot of trouble to reduce the problem to that of allocating fixed-size pages. This illustrates a basic truth of programming: fixed-size objects are easy to deal with, and variable-sized objects are hard to deal with. It is always to your advantage to use fixed-size objects if you possibly can. For example, dynamic allocation of fixed-size objects is trivial, but for variable-sized objects it is much harder.

It is worth your while to convert variable-sized objects into fixed-size objects. If your operating system uses objects of size 30 bytes and size 50 bytes, it might be better to always allocate 50 bytes and just not worry about the wasted 20 bytes in the smaller structures.

Suppose you are allocating strings of variable size and they will vary from 1 to 5000 bytes long. You might consider a block method. Have blocks of 32 bytes—28 bytes of string and four bytes of pointer. The pointer is to the next block in the string. You break the string up in 28-byte sections, and store one section in each block. You will waste an average of 14 bytes (half a section) for each string (unless there are many short strings, in which case you should choose a smaller block size), and you will waste the four bytes of pointer in each string. But you will gain overall because you will be allocating fixed-size objects. You will not have to worry about reclaiming the space of freed strings. All the free blocks can be on a list. You free blocks by adding them to the list, and allocate blocks by removing them from the list. There will be no external fragmentation of storage. The simplicity of the storage allocation may be worth the extra storage used by this method (due to internal fragmentation and pointers).

Avoid variable-sized objects if at all possible.

DESIGN TECHNIQUE: ALLOCATION OF MEMORY FOR OBJECTS

Many library functions return an object of unpredictable length, for example, the system call that returns the full path name of the current working directory. This path name could be quite long or it might be fairly short. The question is, where does the memory come from to hold the path name? There are three possible strategies to use:

- The library routine can allocate the storage, and the caller is responsible for freeing the storage when it is no longer needed. In this case, the call would look something like

  ```
  char * getcwd( void );
  ```

  and a pointer to the current working directory is returned.
- The library routine can have one static block of storage where it puts the path name. In this case, the call would look something like

  ```
  char * getcwd( void );
  ```

  The call looks the same, but the semantics are different. The storage where the path name is kept is not owned by the caller, and another call to getcwd will overwrite it.
- The user can provide the storage and pass it to the library function. In this case, the call would look something like

  ```
  int getcwd( char * buffer, int sizeOfBuffer );
  ```

  In this case, the user sends in the storage and indicates how long it is. The return value is used for error codes. One possible error would be that the storage provided was not long enough to hold the entire path name.

Each of these methods has good and bad points. The method where the library function allocates the storage means that you do not have to allocate it yourself. The defect is that you have the responsibility of freeing the storage. The other problem is that you might only want to look at the string and not save it, and so you will free it immediately. This incurs the overhead of a dynamic storage allocate and free for no purpose. The static string method is the fastest, since no storage allocation needs to be done. The problem is that, if you need to keep a copy of the string, it involves one copy into the static area and then another copy into an allocated string. This method will only work if there is a known maximum size that strings can be, since the space must be permanently allocated and be enough for all situations. The third solution is more trouble for the system caller since it has to provide the storage. This storage can be static, global, local, or dynamically allocated, so this option is the most flexible for the system caller.

Of course, in a language with garbage collection this would not be an issue, since deallocation is automatic and not the responsibility of the programmer.

unmapped addresses. In Section 11.2.2, we used the word "segment" to refer to a variable-sized chunk of the logical address space. When you hear the word "segment," you can be sure that it refers to a part of address space that is either a logical part of a program or is variable in length (sometimes both).

Names and Concepts Often you will have systems that are called "segmentation" systems where the segments are contiguous. The point here is not whether it is valid to use the name "segment" for these objects. The point is that there are two logically different ways to handle segments, contiguous and noncontiguous. Whether you call them by the same name or different names will not change the fact that there are two different concepts, each having its own advantages and disadvantages. When you hear the word being used, do not assume either one, but find out which it is.

12.12 SHARING MEMORY

Our basic memory model so far has been to give each process its own address space. This is consistent with the virtual processor model we have been using. Each process is running in its own virtual computer that is separate from all the other virtual computers. Processes that want to communicate do it through an IPC protocol like messages that is similar to two computers communicating over a network.

But sometimes it is convenient for two processes to share memory. We have seen one example of that with the idea of threads, where two threads share the same address space. But we would like to do this with parts of the address space instead of the entire address space.

There are several reasons why we would like to share memory. One is space efficiency. Suppose two programs were each using a word processor. It would be nice if they could both share the code but each have their own data space. Or two programs working on the same data might want to share the data but use different code.

There is no reason why the same page cannot be mapped into two different address spaces. The page is in a page frame, so we just point two different page table entries to that page frame. This works especially well if the pages are code pages that are not modified. If data pages are shared, the two processes have to worry about conflicting changes and may need mutual exclusion protocols. Figure 12.23 illustrates the idea. This figure shows two processes sharing the word processor code, but each process has its own data page. They are running the same program, but on different data.

If the pages are mapped into the same page numbers in both processes, then things will work fine. Even if the pages are mapped into different page numbers in the two processes, things will be fine unless the pages contain addresses. Then the question is, do they contain the correct addresses for the first process or for the second

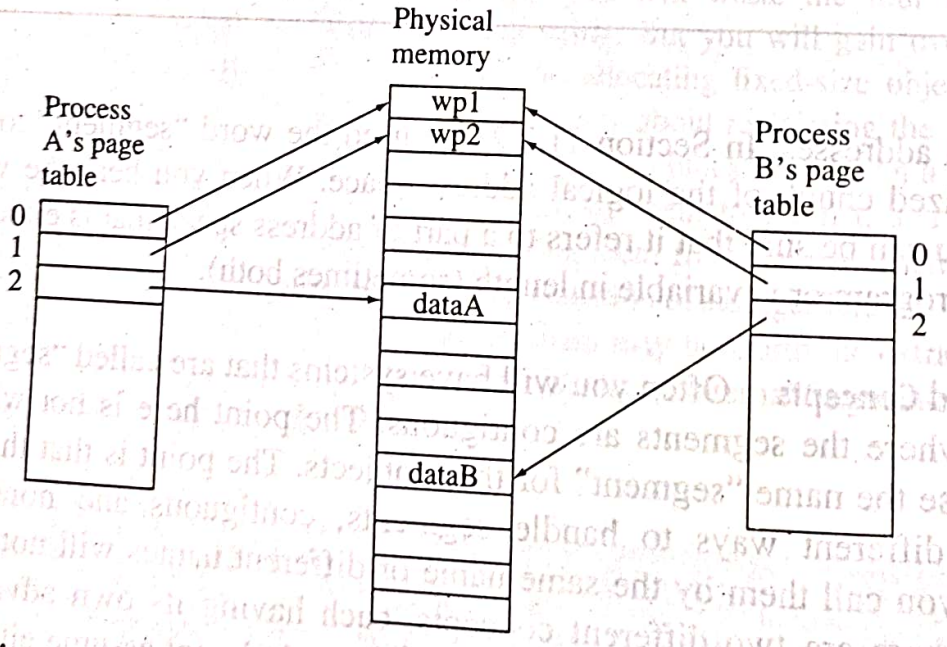


Figure 12.23 Two processes sharing code

process? Most computer architectures allow code to be written so that it is *position independent*, that is, all absolute addresses are kept in registers and not in the code so the code can be moved around in the address space and still work.

Segments are also easy to share. The same addressing problems can come up if the segment is mapped into two different segment numbers in the two processes sharing the segment, and the same set of solutions are possible.

12.12.1 Reasons for Sharing Memory

There are two main reasons for sharing memory. The first reason is to save space in memory by avoiding having two copies of shared pages in memory. If executable code is shared, then we can have one copy of it in the physical memory, and those pages can be mapped into the address space of all the processes that are executing that code. It is mainly code that is shared, but it is also possible to share data. If the memory is writable, then sharing is often not possible since each process needs its own copy.

The second reason to share memory is to transfer large amounts of information between processes. In this case, we usually have read-write memory being shared. Normally it is written by one process and read by the other. It is faster to pass information in shared memory than to send it via messages. Passing information by shared memory does not require any copies.

Some message systems allow you to pass a section of memory along with a message to avoid copying data for large messages. The memory can be mapped into the receiver's address space. Actually, there are two possible ways a large message could be passed. It could be passed as shared memory, or the memory could be unmapped from the sender's address space and mapped into the receiver's address space. The second alternative would transfer the memory, and so it would never be shared simultaneously by the sender and receiver but rather it would be shared serially.

Some instances of shared memory can be handled with threads. To do this, we combine the two processes into one process with two threads. The thread alternative is not always the best solution, since often the two processes are logically separate and only want to share some memory, not all their address space. Also, this cannot be easily done if the two processes are on different machines in a network. So threads and shared memory are two solutions to the problem of sharing large amounts of data, and there are situations where each is preferable.

12.12.2 Shared Memory System Calls

Let us look at some typical system calls to use shared memory. The first issue to address is how to identify shared memory. A section of shared memory is similar to a message queue or pipe, and we have the same naming options:

1. System-assigned names that are returned by a system call and must be passed to other processes by some form of IPC or to child processes.

2. User-assigned names that are arbitrary integers. The sharing processes must agree somehow on the identifier. This method is used in the UNIX shared memory system calls.
3. Shared segments that are named by the file-naming system.

The second and third options are similar. The main difference is whether the name is a character string or an integer. Let us assume the second method.

- `void * AttachSharedMemory( int smid, void * smaddr, int flags )`—This call creates a section of shared memory with identifier `smid` (if one does not exist already). If a section with this identifier already exists, then that one is attached. The memory is placed at the address `smaddr` in the address space of the system caller. If `smaddr` is 0, then the system will decide where it should be placed. In either case, the address of the shared memory is returned by the system calls. The `flags` determines whether the shared memory is read-only and other details of the mapping.
- `void * DetachSharedMemory( void * smaddr )`—This call detaches the shared memory from the process. The shared memory is freed when the last process detaches it.

Two processes wishing to share memory will attach the memory using the same shared memory identifier. Then the memory is in both address spaces until they detach the shared memory segment.

Processes sharing memory might also need semaphores or messages to synchronize the use of the shared memory.

12.13 EXAMPLES OF VIRTUAL MEMORY SYSTEMS

Some of the concepts we have discussed so far have been simplifications to make the explanations easier to understand. Real operating systems use many optimizations when these ideas are implemented.

In this section, we will discuss the virtual memory systems of several existing operating systems. Some features will be present in two or more of these systems, and in those cases we will discuss the feature in general first and then refer to that discussion when we talk about individual operating systems.

12.13.1 SWAP AREA

swap area

A *swap area* is a part of a disk reserved solely for swapping. It is not managed by the file system. This makes swap I/O faster since the file system creates overhead on I/O requests.

The swap area is usually two or three times the size of main memory. A new process cannot start unless there is available swap space for it. That is why you sometimes get an "out of memory" message from a virtual memory system. This message really means "out of swap space."

Some systems will reserve main memory for processes and not require swap space. The idea is that the memory is always filled with pages, and so those pages do not need swap space. This allows the system to runs more processes in the same amount of swap space. This is called *virtual swap space.*

virtual swap space

It is also possible to swap to a regular file instead of to a special swap area. This is not as efficient, since it incurs file system overhead, but it does not require the system to reserve a lot of swap space that cannot be used for other purposes. Many systems allow swapping to a swap area or to ordinary files. This allows them to use an optimistic strategy for swap space management. If they run out of swap space, they can swap to a file instead.

12.13.2 Page Initialization

A process starts with no pages in memory, and so it will page in a number of pages very quickly. Pages are often treated differently the first time they are paged in than other times.

For example, the code pages will usually be read in directly from the load module. This avoids the step of copying the code pages from the load module to the swap area. The first time a code page is swapped out, it can be written to the swap area. This is not necessary, however, because it can always be read in from the load module. Most modern systems read initially from the load module, but some swap code to the swap area because it is faster to swap in pages from the swap area than from a file. If file mapping is used, the load module will always be the disk backup for the code pages.

The same thing is true of initialized data pages, although they have to be in the swap area unless they are read-only.

The uninitialized data and stack pages are initially marked as *zero-fill* pages, meaning that the first time they are used, a page frame is allocated and filled with zeros. The first time they are paged out, they are written to the swap area.

Figure 12.24 shows how pages are initialized. Code and initialized data are initialized from the load module. Other pages are just allocated as zero-filled free pages. So none of the pages actually have to be initialized in the swap area. They only go into the swap area after the initial page-out.

12.13.3 Page Sharing

If a process runs a program that another process is also running, they can share all the read-only pages. This includes all the code pages and the read-only data pages as well.

It is even possible to share writable pages (at least for a while) with a technique called *copy-on-write*. Suppose a process calls `fork`. This requires the operating system to create an exact duplicate of the address space of the process. This can be a lot of copying if the process is large. One solution is to just copy the page table, that is, to share all the pages. But in both page tables, the writable pages are marked as read-only. If a protection violation occurs on one of these pages, we detect that this is a

copy-on-write

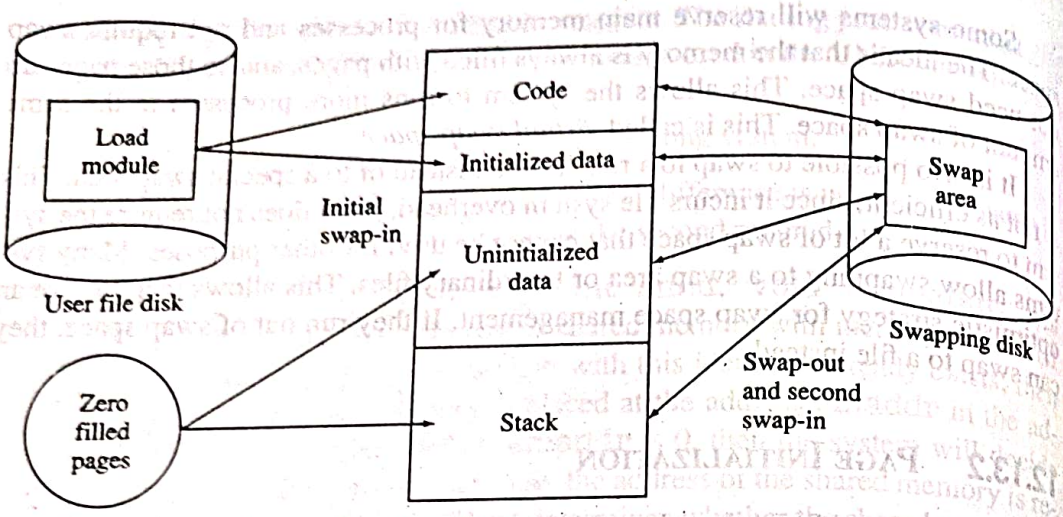


Figure 12.24 Initialization of process pages

write to a copy-on-write page. Then we copy this one page and fix up both page tables so they each point to their own copy of the page, and both are writable. If we are lucky, we will only have to copy a few pages and we will avoid copying the entire address space.

The copy-on-write technique is most useful for efficiently implementing the UNIX `fork` system call, but it can be used in other places where you want to avoid copying of an object as long as possible. The technique is also known as a *lazy copy*.

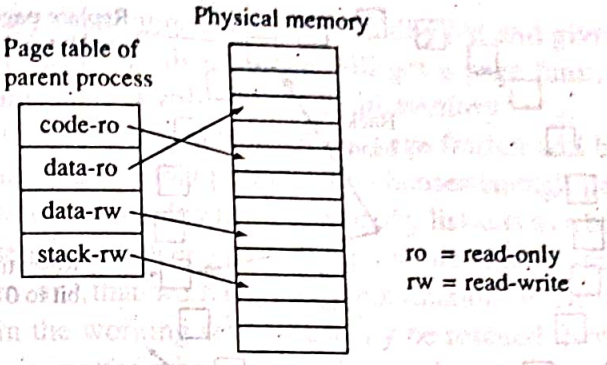
lazy copy

Figure 12.25 shows how copy-on-write works. In Figure 12.25(a), there is a process with four pages, two of which can be written. After it forks (Figure 12.25(b)), the child process has its own page table but shares all the pages. All the pages have been changed to read-only, so we can detect writes into copy-on-write pages. Somewhere else (not shown in the figure), the operating system remembers that those pages are copy-on-write pages that are supposed to be read-write. When the child writes the stack page, it will get a protection interrupt. The operating system will see that it is not a real protection violation, but a write to a copy-on-write page. At that point, the operating system will make a copy of the page and fix up both page tables. The parent's page table entry is set back to read-write, and the child's page table entry is set to point to the copy of the page and is set to read-write.

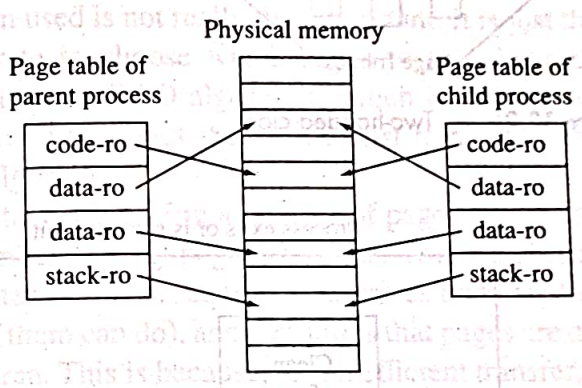
12.13.4 Two-Handed Clock Algorithm

The clock algorithm is a commonly used page replacement algorithm in real operating systems. This approximates LRU and does pretty well. Some operating systems use a *two-handed clock* where the second hand follows the first hand by some distance and allows pages to be reclaimed sooner than waiting for a full revolution of the clock hand. Figure 12.26 shows how a double handed clock works.

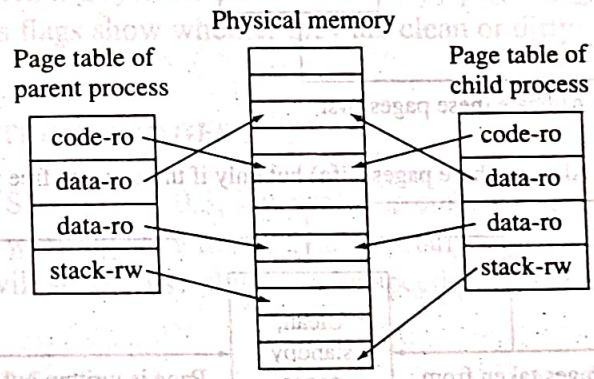
two-handed clock



(a) Before the fork



(b) After the fork



(c) After the child writes the stack

Figure 12.25 Implementing copy-on-write

12.13.5 Standby Page Lists

Many operating systems keep several lists of page frames. Figure 12.27 shows these lists. First they keep track of which pages are dirty (modified) because they cannot be allocated immediately. Dirty pages are scheduled for cleaning and moved to the clean page list after they have been written out.

Some free page frames come from processes that exit or are swapped out. These page frames will not be used again. These page frames are allocated first when a free page frame is needed.

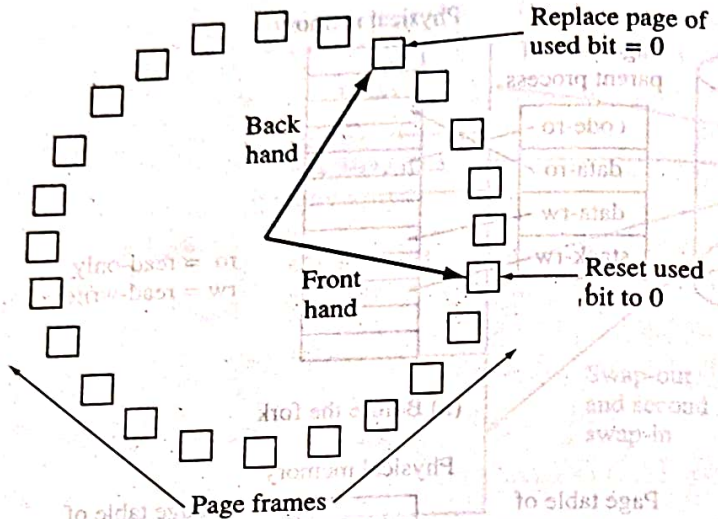


Figure 12.26 Two-handed clock

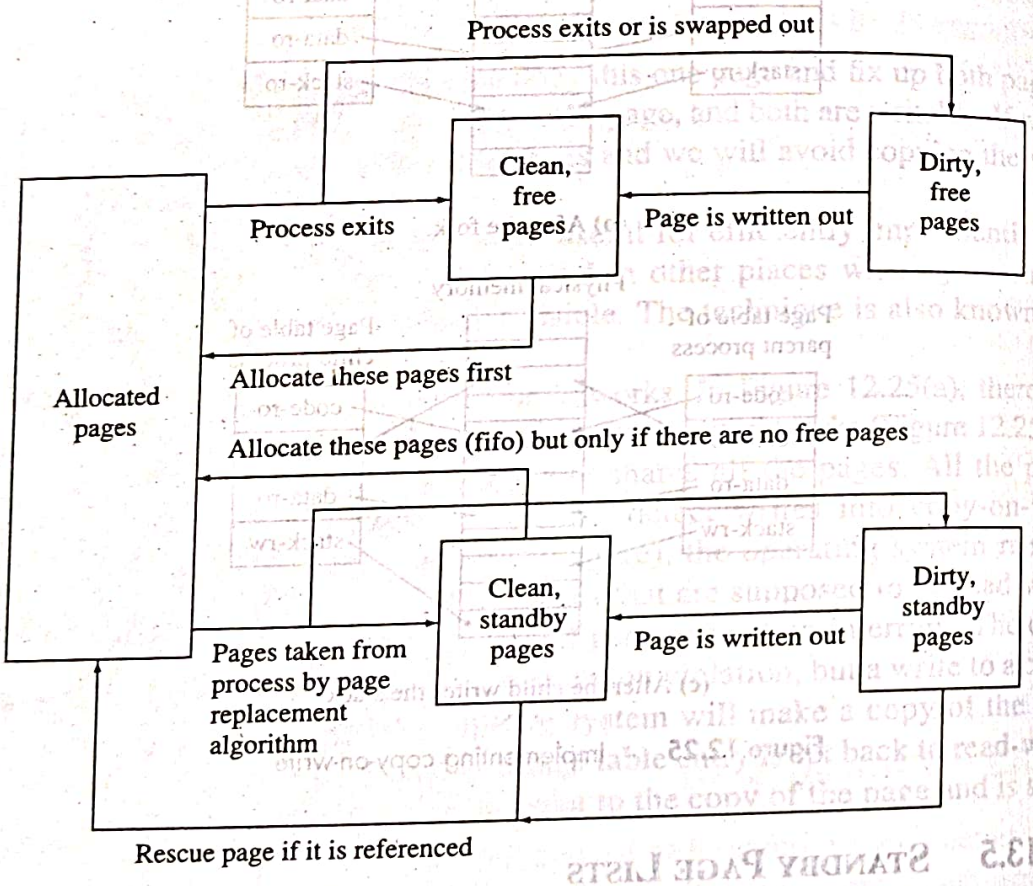


Figure 12.27 Standby page lists

Some page frames are taken from processes by the page replacement algorithm. These page frames might be used again soon. These pages are put in a standby list. When a page frame needs to be allocated to a process and no free pages are available, then a page is taken from the standby list. Pages on the standby list are used in a first-in, first-out order so that a page may remain on this list for some time. If a page on the standby list is referenced while it is on the list,

then it is "rescued" by removing it from the standby list and giving it back to the process that referenced it. In this case, we will get a page fault, but we will not have to read in a page since it will already be in memory.

Usually about 1 percent to 2 percent of the page frames will be on the standby list. There is a paging daemon that periodically chooses enough pages to replace to keep this many pages in the standby list. The standby list acts as a buffer where pages that have been chosen for replacement wait for a while to see if they are referenced again. This buffer means that we have an approximation of the working set algorithm, since pages in the working set will usually be rescued. Note that we get this working set effect no matter what actual page replacement algorithm is used to choose pages to go into the standby list. What this means is that the actual page replacement algorithm used is not really that important. It is just the first of two mechanisms that cooperate to choose which pages to actually reuse. In fact, several operating systems use the FIFO algorithm, which is known to be a poor page replacement algorithm. The use of the standby list transforms this into a very good page replacement algorithm.

Another advantage of keeping a number of pages in the standby list is that you can write out a number of dirty pages at a time. This makes for more efficient I/O transfers (see Chapter 14). This requires disk drives that can do scatter/gather transfers (which most of them can do), and it requires that pages are dynamically allocated space in the swap area. This is because, for an efficient transfer, the pages have to go to the same place on the disk.

In an actual operating system, the clean and dirty pages might actually be on the same list, and status flags show whether they are clean or dirty.

12.13.6 Clustering Pages

As we observed in Section 12.10.3, it is useful to cluster pages for swap-in so that a process loading its working set will not incur so many page faults. Most of the operating systems we will survey use clustering to speed process loading.

12.13.7 File Mapping

We discussed file mapping in Section 11.11. File mapping has so many advantages that almost all modern operating systems use it. One important benefit of file mapping is that the virtual memory system takes care of disk caching (see Section 15.7).

12.13.8 Portable Virtual Memory Systems

It is desirable for an operating system to run on several different hardware platforms. There is not just one dominant computer architecture, and an operating system that can run on all the major ones has an advantage. Because of this, designers have tried to make operating systems portable between architectures. One aspect of this is to make the virtual memory system portable. This has been hard to do since the virtual memory hardware varies widely, even within the same processor architecture. But

over the years, operating system designers have developed techniques that allow them to write portable virtual memory systems.

The main technique used for portability is to specify a small abstract machine that is similar to most real hardware. Most of the virtual memory system is written assuming this abstract machine and the set of operations it supports. To move the operating system to a new machine, it is necessary to implement this small abstract machine on this new hardware. This is the only part of the virtual memory system that is hardware dependent, and so once this is written the rest of the virtual memory system will work on the new hardware.

12.13.9 Sparse Address Space

In Section 11.2.3, we noted that it is inconvenient for a process to be given a noncontiguous logical address space. But this applies only to the initial address space used for the code and initialized data because this is laid out by the compiler, and the compiler assumes the address space is contiguous. It is no problem for dynamically allocated storage to be noncontiguous with other storage; in fact, it can be an advantage. Suppose we allocate a large array of, say, 1,000,000 integers taking 4 Mbytes. If we allocate the array in some remote region of the address space and make sure that the address space around it is not allocated, then we will be able to tell if we try to go beyond the bounds of the array because we will get an unallocated memory error. We will use the term *region* for a contiguous span of pages in the address space.

region

The stack should be a region, but it does not have to be contiguous with any other region. In fact, we want to leave a lot of unused address space at the end of the stack so we can expand it into contiguous addresses. Some systems have a special kind of page, called a *guard page*, which they put at the end of a region. If we access the guard page, we get a memory protection interrupt, and that signals the operating system that the region has to be expanded. In this case, when the stack overflows, it will hit the guard page and be automatically expanded.

guard page

This shows a good reason for noncontiguous allocation—to save address space so a region can be extended with contiguous addresses.

12.13.10 OS/2 Version 2.0

OS/2 2.0 uses 4 Kbyte pages and 32-bit virtual addresses. It uses two-level page tables. When a process is swapped out, even its master page table is swapped out.

OS/2 uses a single-handed clock for page replacement, and replaced pages are put on a standby list. Dirty pages that are passed over by the clock hand are scheduled for swap out. The pages in the standby list are used in FIFO order, and pages that are used while on the standby list can be reclaimed for the process.

The virtual address space does not have to be contiguous. Virtual address space is allocated in two stages. First you reserve the virtual addresses without committing any swap space or physical memory. Later you commit the virtual addresses, and this

allocates swap space for the pages and will assign physical page frames to the pages when it gets page faults on them.

OS/2 uses copy-on-write pages to do lazy copies (for example, on forks). Pages from executable files are marked swap-on-write, meaning that they will not be swapped to the swap area unless they are written into. They are always fetched directly from the executable file.

OS/2 uses scatter/gather I/O to write out a number of dirty pages at a time. It uses a bit map to keep track of free page frames in the swap area. A page is not necessarily swapped out to the same place each time it is swapped out. It goes into the most convenient page frame in the swap area. This way, it can swap out a number of pages at the same time to consecutive page frames in the swap area.

12.13.11 Windows NT

Windows NT uses a 4 Kbyte page and 32-bit virtual addresses. It uses two-level page tables.

Windows NT uses the same two-stage allocation of memory as OS/2, that is, a process can reserve virtual address space without reserving swap space for it. Later the process can commit the memory, and this allocates swap space for it. Windows NT does not allocate page tables for uncommitted memory. It uses a tree structure to keep track of the parts of the virtual address space that have been reserved but not yet committed.

Pages are replaced using a local FIFO strategy, but replaced pages are not swapped out and reused immediately but placed in a standby list. If a page in the standby list is accessed, it is rescued from the standby list and no swap in is necessary.

The main virtual memory structure is a section. A *section* is a range of virtual addresses that is backed either by space in the swap area or by a file. A section backed by a file is used for file mapping. A section can be mapped into the virtual address space of a process. A process can map in a *view* of the section, that is, part of the section. Two processes can share memory by mapping the same section into each of their virtual address spaces. Sections are used to map the code and data from load modules into a process. The code and data sections are mapped in using copy-on-write status so that they can be shared until a process modifies them.

Indirect page table entries are used for shared pages. The page table entries for shared pages point to one shared page table entry. This allows a shared page to be swapped out without having to change the page table of every process that is sharing the page.

Processes with special privileges can set the protection on their own pages and can lock pages into memory.

Windows NT uses demand paging, but it clusters pages on initial swap in so that the initial loading of the memory for a process will go faster and not require as many page faults.

Windows NT is built on a *hardware abstract layer* (a small abstract machine) for easy portability.

12.13.12 MACH AND OSF/1

The Mach operating system has an interesting memory architecture that has been influential in UNIX and other operating systems. In this section, we will present the main ideas of the memory system in Mach.

Goals of Mach Memory Architecture Mach had several goals for its memory system:

- *Flexible sharing*—Mach was intended to be a distributed system and they wanted to have two ways of sharing: copy-on-write sharing for fast copies, and read-write sharing for shared memory. They also wanted to be able to share memory between machines.
- *Large, sparse address space*—They wanted to support a large address space and allow memory to be placed anywhere in the address space.
- *Memory-mapped files*—They wanted to be able to map files into the address space.
- *User control of page replacement*—They wanted to allow processes to control their own paging if they felt they could do better than the system.

Basic Objects Mach is an object-oriented system, and so everything in Mach is an object. A *port* is a message queue to send requests to an object. The messages sent to the port are equivalent to method invocations on the object. Objects are implemented by object managers which maintain object state and handle messages on the ports that represent the objects.

memory object

pager

virtual memory object

A *memory object* is a specific source of pages. A memory object can be a file on the local machine, but it could be a file an another machine, space in the swap area, another process, a frame buffer, etc. A *pager* manages a memory object and handles messages from the memory object's port. These messages tell it to get certain pages and read them into memory or to write out a page in memory. An *internal pager* is implemented by Mach and an *external pager* is implemented by a user process. The internal Mach pager allows memory objects that are backed by swap space or by files. The pagers cache some of the pages of a memory object in memory page frames, and these are the resident pages. A *virtual memory object* or *VM object* manages a region of virtual memory in an address space. The VM object sends messages to the pager to read and write pages from the memory object. The address space of a process (called a *task* in Mach) is a collection of regions, each controlled by a VM object.

Using the Basic Objects Each region is a set of contiguous pages in the address space, but the regions can be anywhere in the address space. So the region concept allows for a large, sparse address space. A memory object can be a file, and so that allows for memory-mapped files. An external pager allows a user process to manage its own page replacement. If two processes both map a region into their address space, then they are sharing the memory. A VM object can implement copy-on-write sharing of a region of memory. An external pager could synthesize the pages when

they are requested or get them from another node in the network. This allows the implementation of shared memory distributed over two or more nodes in a network.

Each process has an address space made up of several regions. Each region has an inheritance attribute that can be do-not-share, copy-on-write-share, or read-write-share. When a process forks, the parent and child will share the regions marked read-write-share. The copy-on-write-share segments will be shared until they are written into by either process. The do-not-share regions are not copied at all.

The Mach Memory System Mach uses global FIFO replacement, but, like Windows NT and OS/2, replaced pages are placed on a standby list where they can be rescued if they are accessed before they are reused. Mach tries to keep 1.25 percent of the page frame free (in the free or standby lists).

The Mach virtual memory system is mostly portable and tries to isolate all machine dependencies in part of the code.

Mach uses copy-on-write to avoid copying whenever possible (for example, for forks).

Pages are clustered into groups of 1, 2, 4, or 8 for faster page in when a process is starting up.

SVR4 UNIX has copied most of the features of the Mach memory system. OSF/1 is derived from Mach and uses the same memory system. Windows NT has a concept of sections which is similar to Mach regions.

12.13.13 System V Release 4

SVR4 uses a two-handed clock for page replacement. Replaced pages are put at the end of the free list and can be rescued if they are accessed before being reused.

The virtual address space is made up into *segments*. The segment types are text segment, data segment, stack segment, shared segment, and file mapped segment. Pages are swapped from a local file, a swap area, a remote file (over the network), or a frame buffer. A swap area can be a device or a regular file. Segments allow for mapped files.

The SVR4 virtual memory system is largely hardware independent.

SVR4 also clusters pages for faster process startup.

12.13.14 Other Systems

4.3 BSD uses a two-handed clock page replacement algorithm.

The 4.4 BSD virtual memory is based on the Mach virtual memory architecture.

The DEC VMS operating system uses local FIFO with a standby list.

The Macintosh operating system uses the second-chance clock algorithm.

IBM's MVS uses 4 Kbyte pages and three-level paging. The lowest level consists of segments of up to 256 pages (1 Mbyte). The intermediate level consists of segment tables which are paged by the top level. It uses an LRU approximation for page replacement. At fixed timer intervals, it increments counters for each page that has not been referenced in the last time interval.

12.14 VERY LARGE ADDRESS SPACES

The new generation of processors allows for virtual address spaces larger than 32 bits, and most of them allow for a 64-bit virtual address. This change is necessary because 32 bits is getting too small for some very large programs. There has been speculation on how this change will affect operating systems, and the dominant opinion is that operating systems for very large address space processors will use a single address space for all processes. This will abandon the paradigm of a separate address space for each process and the idea of combining protection with the address space. The result will be that there will be only threads, all sharing the same (very large) address space.

All code and data will appear only once in the address space, and they will never move once they have been placed in the address space. Naturally, the address space will be very sparsely populated.

There are many issues and problems to be worked out for this kind of operating system. See Chase et al. (1994) for more information.

12.15 SUMMARY

The page replacement algorithm is responsible for deciding which page to replace when we need to bring a new page into physical memory. In order to maintain high page hit rates, the page replacement algorithm must pick pages that are not going to be used for a long time. Page replacement algorithms decide which page to replace by looking at the past page use behavior of the process and using that to predict the future page use behavior of the process. The least recently used (LRU) algorithm seems to be the most successful at this prediction, but it is hard to implement. The most common page replacement strategies are approximations of LRU. The clock algorithms are intended to act similarly to LRU and work very well. Both algorithms use the referenced bit to discover when pages are used.

A paging system must avoid high paging rates, and so it must avoid putting too many processes into memory. We can do this by detecting overloading when it occurs or by predicting it and avoiding it. A load control method decides how many processes should be in memory. The working set strategy tries to predict memory overloading and prevent it. It works by trying to predict the pages a program needs to run, that is, the program's working set. It is a very effective page replacement and load control method, but is also very hard to implement and so is usually approximated. The WSClock page replacement algorithm is a good approximation of the working set algorithm. There are also other methods of load control. An effective method is to limit the system to one loading process.

Paging is a good idea, but it starts breaking down with very large address spaces since the page tables start using too much physical memory. Some ways of dealing with the problem are two-level paging, three-level paging, recursive address spaces, and inverted page tables.

Paging is not the only way to achieve virtual memory; segmentation is another method. But segmentation implies that you have to deal with dynamic memory management and fragmentation, and so paging is much more popular.

12.15.1 TERMINOLOGY

After reading this chapter, you should be familiar with the following terms:

- clock page replacement algorithm
- clustering
- copy-on-write
- demand paging
- dirty bit
- emulator
- first-in, first-out (FIFO) page replacement algorithm
- global replacement
- guard page
- inverted page table
- lazy copy
- least-recently used (LRU) page replacement algorithm
- load control
- local replacement
- master page number
- medium-term scheduler
- memory object
- modified bit
- not recently used (FINUFO) page replacement algorithm
- optimal page replacement algorithm
- page fault frequency load control
- page reference string
- page replacement algorithm
- pager
- placement
- predictive load control
- prepaging
- primary page table
- random page replacement algorithm
- recursive address space
- referenced bit
- region
- replacement
- secondary page number
- second chance page replacement algorithm

- segment
- segmentation
- short-term scheduler
- software page table lookups
- swap area
- theory of program behavior
- thrashing
- three-level paging
- two-dimensional address space
- two-handed clock
- two-level paging
- virtual memory object
- virtual swap space
- working set
- WSClock page replacement algorithm

12.15.2 Review Questions

The following questions are answered in the text of this chapter:

1. What is the difference between placement of a block of allocated memory and replacement of a block of allocated memory?
2. Compare local and global page replacement. What are the advantages of each?
3. What is a theory of program behavior? Give two examples.
4. Explain why the best, worst, and average cases for the random page replacement algorithm are all the same.
5. How would you implement FIFO page replacement?
6. Describe an approximation of LRU.
7. Why is a modified bit useful in a paging system?
8. Describe the general clock algorithm.
9. Contrast resident set with working set.
10. Describe the working set algorithm.
11. Why does the working set algorithm require you to swap out a process completely if all of its working set will not fit into memory?
12. Why do you need page reference strings to evaluate page replacement algorithms? What do you do with them?
13. Why do you need to emulate a machine's instruction set in order to generate page reference strings?
14. Why is load control important?

15. Explain how load control is a form of scheduling.
16. When would the preloading of pages be useful?
17. Describe two-level paging.
18. What problem is two-level paging trying to solve?
19. What are the advantages of an inverted page table?
20. Compare two-level paging with recursive address spaces.
21. What are the advantages and disadvantages of paging the operating system address space?
22. Give a reason to lock a page in memory.
23. Give some arguments for a large page size. Give some arguments for a small page size.
24. Compare paging and segmentation.
25. Why would we want to share memory between address spaces?

12.15.3 Further Reading

Denning (1970) provides a review of virtual memory techniques. See Belady (1966), Hatfield and Gerald (1971), Hatfield (1972), Morrison (1973), Mattson et al. (1970) and Coffman and Varian (1968) for studies of page replacement algorithms. See Aho et al. (1971) and Prieve and Fabry (1976) for a discussion of optimal paging algorithms. Goldman (1989) discusses the use of a clock algorithm in the Macintosh operating system. See Madison and Batson (1976) for a discussion of program locality. See Carr and Hennessey (1981) for a discussion of the WSClock page replacement algorithm.

See Denning (1968) for the original paper on the working set model and a discussion of thrashing. Denning (1980) is an excellent review of the development of the working set model over 10 years. Madison and Batson (1976), Denning and Kahn (1975), and Batson (1976) discuss program locality and the program phase model. Chu and Opderbeck (1977) discuss the PFF (page fault frequency) algorithm.

See Chu and Opderbeck (1974) for a discussion of the effects of varying page size. See Batson (1970) for a discussion of average segment size measurements.

Chang and Mergen (1988) discuss the use of inverted page tables in the IBM 801.

Carr and Hennessy (1981) discuss the simulation of paging algorithms and present an improved method of using traces.

12.16 PROBLEMS

1. Explain why we use the word "global" when we talk about global page replacement.
2. Give some reasons why local page replacement is better than global page replacement.
3. Give some reasons why you think LRU is a good page replacement algorithm. Why is its theory correct?

4. Compare the clock page replacement algorithm with the FCFS algorithm with a free page list that is used FCFS. (It acts as a buffer and delays replacement a lot like the clock algorithm does.)

   Consider the average time a page stays in the free page list (before being rescued or reused). Would this be a good load control measure?

5. Consider a variation of the clock algorithm called the "counting" clock algorithm. Each page frame has a counter. The algorithm takes a parameter N. N is in the range 1 to 10.

   Here is the algorithm: Look at a page frame. If the referenced bit is 1, then set it to 0 and set the count to N. If it is 0, then decrement the count. If the count is zero, then replace the page.

   Give your analysis of this paging algorithm. Is it better than clock? Is it a generalization of clock?

6. Consider each of the global page replacement strategies we looked at in this chapter: optimal, random, FIFO, LRU, clock, and second chance. Think about converting each one into a local page replacement strategy. First say if it is makes sense at all to convert it to a local strategy. Then decide whether it would be better, worse, or about the same as a local strategy (compared to the same page replacement algorithm as a global strategy). Give arguments to support your answer. Remember that a local page replacement algorithm has to make decisions about how many page frames a process should be allocated.

7. What would be the worst possible page replacement algorithm? Suppose that the optimal algorithm gives a page fault rate of one in a million for a program with 20 pages and 10 page frames. Estimate the page fault rate of your worst possible page replacement algorithms on this same page reference string. *Hint:* It is not possible to get the page fault rate up too high because of program locality. Remember that a page is replaced only when there is a page fault.

8. Suppose that the hardware keeps an eight-bit shift register for each page frame, and there is a single hardware instruction that would shift the referenced bit for each page frame into the high-order bit of its associated shift register (the low-order bit is discarded). There are instructions to read and set these shift registers. How would you use this hardware to implement an approximation of LRU? Do you think this would be better than the approximations described in the chapter? Why or why not?

9. Suppose you are writing an emulator and you are at the point where you want to emulate a single instruction, one that adds two hardware registers together. Estimate the number of machine instructions that it would take to emulate this instruction.

   Then estimate the number of instructions that one iteration of the basic fetch-execute loop would take. This iteration would fetch the next instruction, increment the PC, decode the instruction, and jump to the code that emulates that instruction.

   Assume that the emulator emulates memory with a large array, the registers with a small array, and the PC with an integer variable. To do the estimates, write the C++ code to do it and then estimate the number of machine instructions each C++ statement will take.

   What does this say about the speed of emulation?

10. Suppose we have a 48-bit virtual address space. Design a two-level paging system for this. How big are the master page tables, the secondary page tables, and the page frames? How many pages total will there be in the logical address space, and how much memory will the page tables take for a program that used the entire address space?

11. Suppose your paging system uses inverted page tables, and you are using a hash table to find the page frame a specific page is in. Assume the hash table takes an average of two probes to find the page or discover that it is not there. Estimate how many machine instructions it will take to handle a TLB miss when the page is in the page table. Be sure to include the time for the TLB replacement algorithm.

12. Suppose we have a computer system with a 44-bit virtual address, page size of 64K, and 4 bytes per page table entry.

    a. How many pages are in the virtual address space?

    b. Suppose we use two-level paging and arrange for all page tables to fit into a single page frame. How will the bits of the address be divided up?

    c. Suppose we have a 4 Gbyte program such that the entire program and all necessary page tables (using two-level paging as in part *b*) are in memory. (*Note:* It will be a *lot* of memory.) How much memory (in page frames) is used by the program, including its page tables?

13. Suppose we have a computer system with a 38-bit virtual address, 16K pages, and 4 bytes per page table entry.

    a. How many pages are in the virtual address space?

    b. Suppose we use two-level paging and arrange for all page tables to fit into a single page frame. How will the bits of the address be divided up?

    c. Suppose we have a 32 Mbyte program such that the entire program and all necessary page tables (using two-level paging as in part *b*) are in memory. How much memory (in page frames) is used by the program, including its page tables?

14. Suppose you have a two-level paging system where the first level uses 10 bits, the second level uses 10 bits, and the page offset uses 12 bits. If you have a program that uses 18 Mbytes of memory, how many page frames will it use for the program and page tables if everything is in memory?

15. Suppose you have a two-level paging system where it takes 10 milliseconds to service a page fault and a regular memory access takes 200 nanoseconds (1/5 microsecond). An average instruction takes 500 nanoseconds. The translation lookaside buffer hits 90 percent of the time, and the user page fault rate is one page fault every 100,000 instructions.

    a. What is the average time for a memory fetch where there are no page faults?

    b. What is the average instruction time for all instructions, including the ones that cause page faults?

6. Suppose a given program consists of a main program which fits into one page, and a long series of procedures which each fit on a page. The main program is a loop that calls several of these procedures each time through the loop, but a different set each time through so no one procedure is called very frequently. The procedures very seldom call one of the other procedures. The main program and the procedures frequently reference global data, which also fits on one page. Suppose a number of processes with these characteristics run simultaneously. (*Note:* They are not running the *same* program, and there is no sharing of pages between processes. They are running programs that *act similarly.*) Consider each of the three page replacement algorithms:

a. First-in, first-out.

b. Least recently used.

c. Clock.

Discuss what kind of performance each algorithm would give, given this group of programs. Assume a global page allocation policy.

17. In this problem, you have to make up some page reference strings that are good for one paging algorithm and bad for another paging algorithm. Come up with two different page reference strings and a page fault chart for each. Here is an example of the format you should use:

Page ref	1	2	3	2	4	2	5	2	3	4	
Page 0	1*	1	1	1	4*	4	4	4	3*	3	FIFO
Page 1	-	2*	2	2	2	2	5*	5	5	4*	8 page
Page 2	-	-	3*	3	3	3	3	2*	2	2	faults
Page 0	1*	1	1	1	4*	4	4	4	3*	3	LRU
Page 1	-	2*	2	2	2	2	2	2	2	2	7 page
Page 2	-	-	3*	3	3	3	5*	5	5	4*	faults
Page 0	1*	1	1	1	4*	4	5*	4	4	4	OPT
Page 1	-	2*	2	2	2	2	2	2	2	2	5 page
Page 2	-	-	3*	3	3	3	3	3	3	3	faults

The first line has the page reference string. The next section has a line for each page frame. (Note the label says "Page" rather than "Page frame." This is just to save space.) Each column lists the page that is in each page frame *after* the page fault (if any) for the reference has been handled according the page replacement algorithm being shown. An asterisk ("*") is placed after the page if there was a page fault on that reference and this was the new page brought in as a result of that page fault. The next section is the same, except it is for the LRU paging algorithm. And the final section is for the optimal page replacement algorithm. On the right, each section is labeled and the total number of page faults is given.

Make up two such charts that each compare LRU, FIFO, and OPT. On the first chart, show a page reference string that is good for LRU and bad for FIFO. On the second chart, show a page reference string that is good for FIFO and bad for LRU. OPT, of course, will always be the best. The main problem here is to make up these page reference strings. Think of the premise of each algorithm and try to make up a reference string that follows one premise but not the other. For each example use three page frames and a reference string of at least nine references. The first three references on each chart should be exactly the same as in the example above. This is the filling of the pages, and it should be the same for each algorithm. The page frame numbers start at 0 and the pages start at 1. Follow this convention in your charts. You should include a sentence or two explaining what you wanted the reference string to do, that is, how it works well with or interacts badly with the paging algorithms.

18. Suppose an instruction takes 1/2 microsecond to execute (on the average), and a page fault takes 250 microseconds of processor time to handle plus 10 milliseconds of disk time to read in the page. How many pages a second can the disk transfer?

Suppose that 1/3 of the pages are dirty. It takes two page transfers to replace a dirty page. Compute the average number of instructions between page faults that would cause the system to saturate the disk with page traffic, that is, for the disk to be busy all the time doing page transfers.

19. Why might it be bad for the page fault rate to be too low?

20. The concept of virtual memory depends on locality. If a program uses a lot of pages, the virtual memory will not work no matter what page replacement algorithm you use. For example, suppose that a program has 5000 (4K) pages (= 20 Mbytes), and goes into a loop where it references one word in page 0, then one word in page 1, then one word in page 2, and so on over and over. It will access all 5000 pages every 5000 instructions. This program requires 20 Mbytes of memory, and no page replacement algorithm can improve that.

A program with locality, however, will not access a lot of pages in a short period of time, but will restrict itself to a few pages. A page replacement algorithm tries to guess which pages the program will access in the future, and keeps those pages in memory.

For each of the following page replacement algorithms, describe the characteristics of the programs that make that algorithm behave very poorly, that is, describe the worst case for each algorithm. Your programs should *not* just be programs that access a lot of pages; no paging algorithm can handle that. Instead, come up with programs that would work well with the optimal paging algorithm but would work much worse with the page replacement algorithm you are constructing a worst case for. *Hint:* Each algorithm makes certain assumptions about how programs behave. The worst-case program will act in exactly the opposite way.

a. First-in, first-out.

b. Least recently used.

c. Clock.

d. Second chance.

e. Working set.

21. Rewrite the hardware virtual memory access algorithm (in Section 11.7.1) for a two-level paging system.

22. Rewrite the hardware virtual memory access algorithm (in Section 11.17.1) for a system where the page tables are paged (as in Section 12.8).

23. Describe the response of the operating system to the four paging-related events (as is done in Section 11.17.2), but for a two-level paging system.

24. Some paging systems use *clustering* when loading pages. This means that, whenever they get a page fault, they load the page that was faulted on and also some other nearby pages (unless they are already loaded). Paging out is not changed by clustering, that is, we use any of the page replacement algorithms with no changes. For example, we might load the faulting page and the next three pages when we get a page fault.

a. Give one advantage of clustering.

b. How can it be the case that some of the pages in the cluster might already be loaded, since they are all loaded together?

c. Why would clustering be hard to implement if we were using an inverted page table?

25. What would be a good load control measure for each page replacement algorithm?

26. Paged segments would seem like an ideal solution that combines the advantages of pages with the advantages of segments. Explain why this is not so.

chapter

13 Design Techniques III

13.1 MULTIPLEXING

13.1.1 OVERVIEW

Multiplexing is a way of sharing a resource between two or more processes. In space multiplexing, you give each process part of the resource. In time multiplexing, you give each process the resource part of the time.

13.1.2 MOTIVATION

Let us take display screen space as an example of a scarce resource. Anyone who uses a window system wishes they had a larger screen; there never seems to be enough space to show all the things you want to show. We can space multiplex the screen by having several windows on the screen at the same time (but not overlapping one another). For example, we might be editing three files at the same time. See Figure 13.1 for an example.

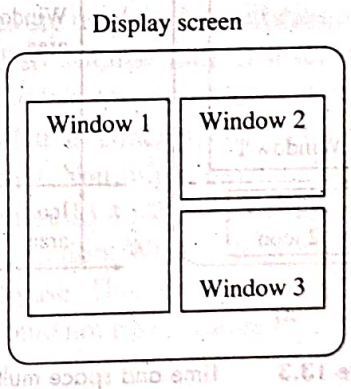


Figure 13.1 Space multiplexing a display