

10.3 Copy-on-Write

In Section 10.2, we illustrated how a process can start quickly by demand-paging in the page containing the first instruction. However, process creation using the `fork()` system call may initially bypass the need for demand paging by using a technique similar to page sharing (covered in Section 9.3.4). This technique provides rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.

Recall that the `fork()` system call creates a child process that is a duplicate of its parent. Traditionally, `fork()` worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However, considering that many child processes invoke the `exec()` system call immediately after creation, the copying of the parent's address space may be unnecessary. Instead, we can use a technique known as **copy-on-write**, which works by allowing the parent and child processes initially to share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. Copy-on-write is illustrated in Figures 10.7 and 10.8, which show the contents of the physical memory before and after process 1 modifies page C.

For example, assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write. The operating system will obtain a frame from the free-frame list and create a copy

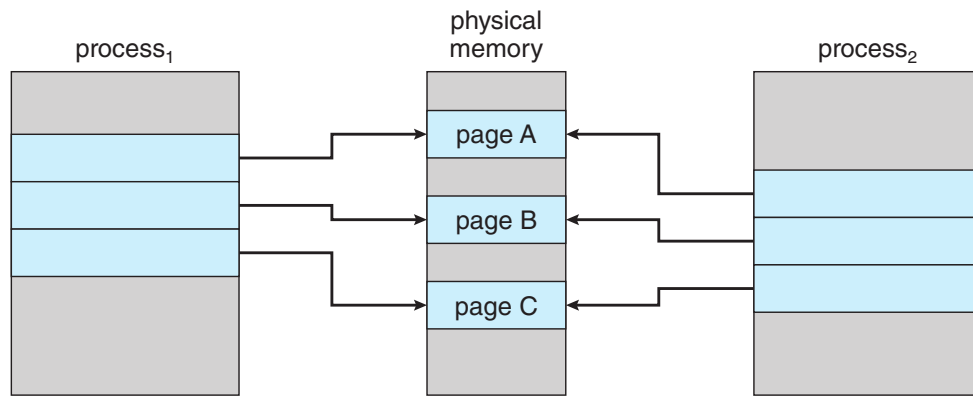


Figure 10.7 Before process 1 modifies page C.

of this page, mapping it to the address space of the child process. The child process will then modify its copied page and not the page belonging to the parent process. Obviously, when the copy-on-write technique is used, only the pages that are modified by either process are copied; all unmodified pages can be shared by the parent and child processes. Note, too, that only pages that can be modified need be marked as copy-on-write. Pages that cannot be modified (pages containing executable code) can be shared by the parent and child. Copy-on-write is a common technique used by several operating systems, including Windows, Linux, and macOS.

Several versions of UNIX (including Linux, macOS, and BSD UNIX) provide a variation of the `fork()` system call—`vfork()` (for **virtual memory fork**)—that operates differently from `fork()` with copy-on-write. With `vfork()`, the parent process is suspended, and the child process uses the address space of the parent. Because `vfork()` does not use copy-on-write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes. Therefore, `vfork()` must be used with caution to ensure that the child process does not modify the address space of the parent. `vfork()` is intended to be used when the child process calls `exec()` immediately after creation.

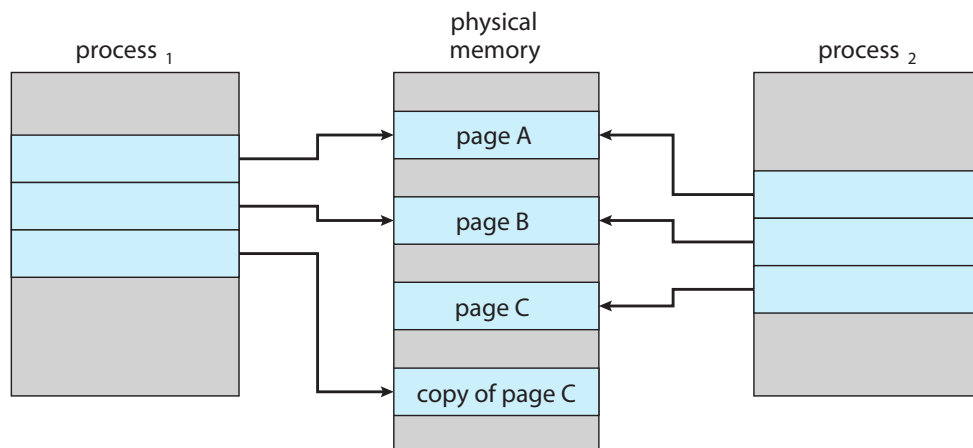


Figure 10.8 After process 1 modifies page C.

10.6 Thrashing

Consider what occurs if a process does not have “enough” frames—that is, it does not have the minimum number of frames it needs to support pages in the working set. The process will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing. As you might expect, thrashing results in severe performance problems.

10.6.1 Cause of Thrashing

Consider the following scenario, which is based on the actual behavior of early paging systems. The operating system monitors CPU utilization. If CPU utiliza-

tion is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization and *increases* the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. The page-fault rate increases tremendously. As a result, the effective memory-access time increases. No work is getting done, because the processes are spending all their time paging.

This phenomenon is illustrated in Figure 10.20, in which CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.

We can limit the effects of thrashing by using a **local replacement algorithm** (or **priority replacement algorithm**). As mentioned earlier, local replacement requires that each process select from only its own set of allocated frames. Thus, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well. However, the problem is not entirely solved. If processes are thrashing, they will be in the queue for the paging device most of the time. The average service time for a page fault will increase

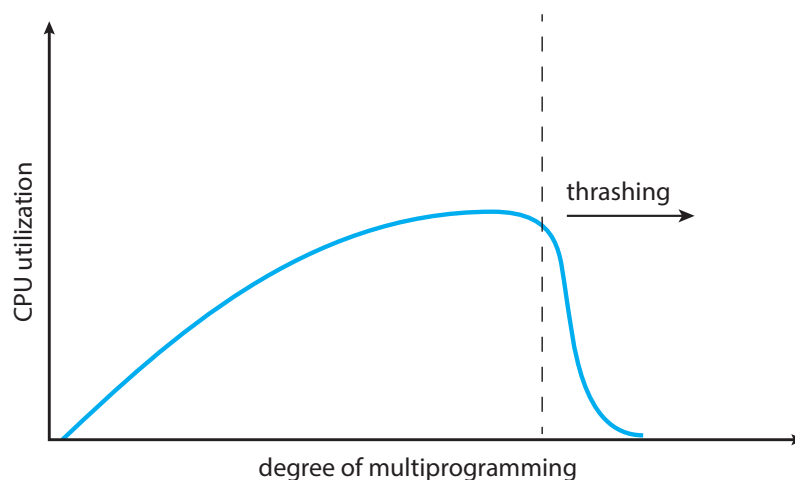


Figure 10.20 Thrashing.

because of the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing.

To prevent thrashing, we must provide a process with as many frames as it needs. But how do we know how many frames it “needs”? One strategy starts by looking at how many frames a process is actually using. This approach defines the **locality model** of process execution.

The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A running program is generally composed of several different localities, which may overlap. For example, when a function is called, it defines a new locality. In this

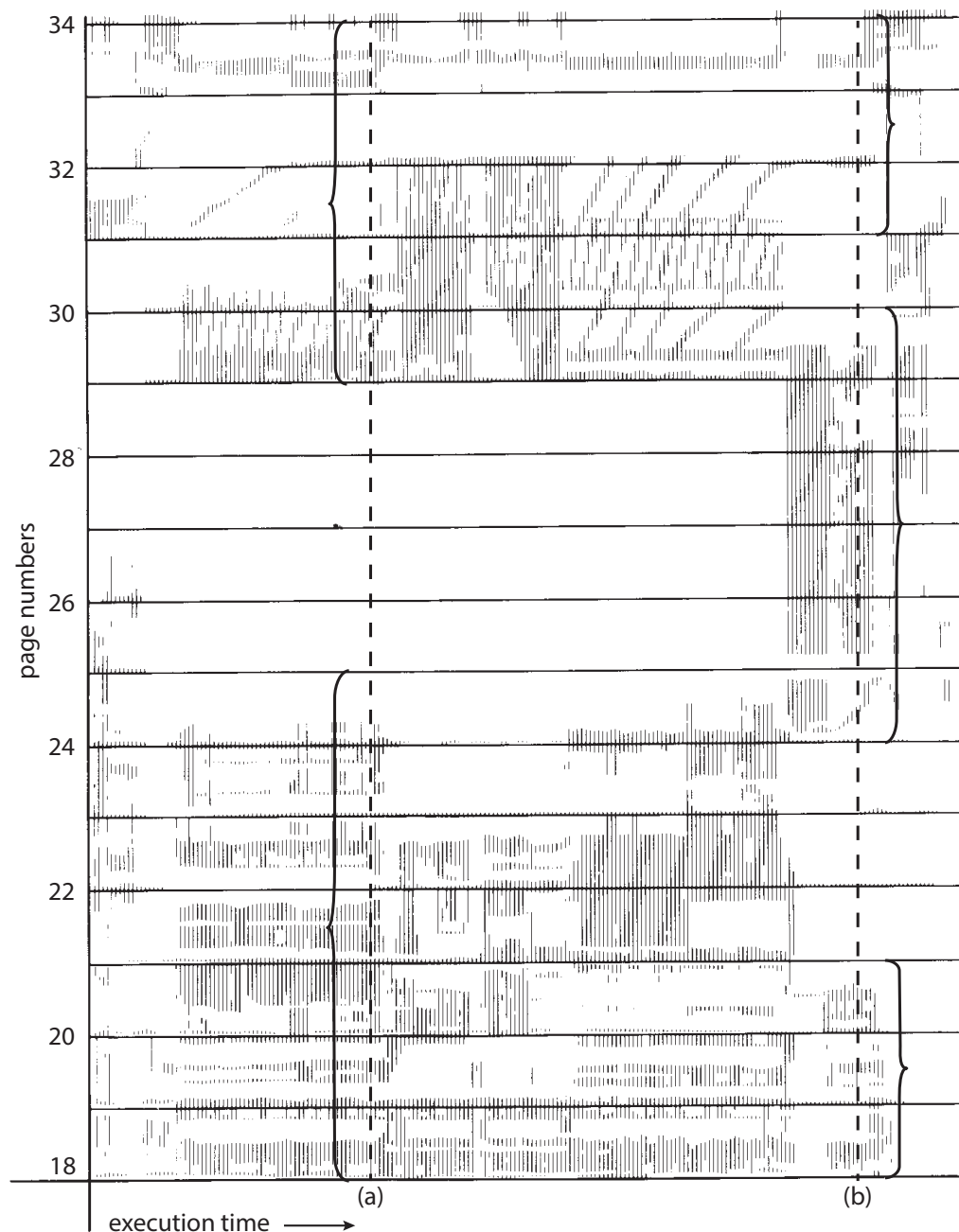


Figure 10.21 Locality in a memory-reference pattern.

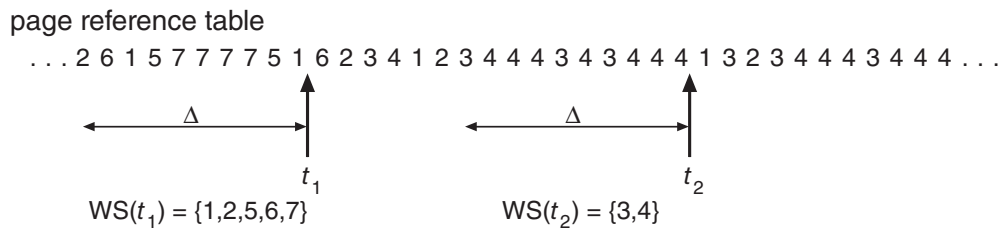


Figure 10.22 Working-set model.

locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use. We may return to this locality later.

Figure 10.21 illustrates the concept of locality and how a process’s locality changes over time. At time (a), the locality is the set of pages {18, 19, 20, 21, 22, 23, 24, 29, 30, 33}. At time (b), the locality changes to {18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33}. Notice the overlap, as some pages (for example, 18, 19, and 20) are part of both localities.

Thus, we see that localities are defined by the program structure and its data structures. The locality model states that all programs will exhibit this basic memory reference structure. Note that the locality model is the unstated principle behind the caching discussions so far in this book. If accesses to any types of data were random rather than patterned, caching would be useless.

Suppose we allocate enough frames to a process to accommodate its current locality. It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities. If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

10.6.2 Working-Set Model

The **working-set model** is based on the assumption of locality. This model uses a parameter, Δ , to define the **working-set window**. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is the **working set** (Figure 10.22). If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference. Thus, the working set is an approximation of the program’s locality.

For example, given the sequence of memory references shown in Figure 10.22, if $\Delta = 10$ memory references, then the working set at time t_1 is {1, 2, 5, 6, 7}. By time t_2 , the working set has changed to {3, 4}.

The accuracy of the working set depends on the selection of Δ . If Δ is too small, it will not encompass the entire locality; if Δ is too large, it may overlap several localities. In the extreme, if Δ is infinite, the working set is the set of pages touched during the process execution.

The most important property of the working set, then, is its size. If we compute the working-set size, WSS_i , for each process in the system, we can then consider that

$$D = \sum WSS_i,$$

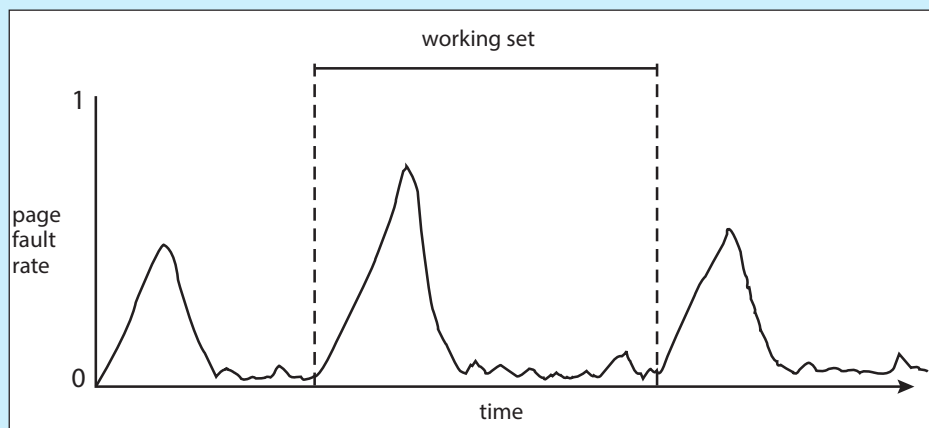
where D is the total demand for frames. Each process is actively using the pages in its working set. Thus, process i needs WSS_i frames. If the total demand is greater than the total number of available frames ($D > m$), thrashing will occur, because some processes will not have enough frames.

Once Δ has been selected, use of the working-set model is simple. The operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size. If there are enough extra frames, another process can be initiated. If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process's pages are written out (swapped), and its frames are reallocated to other processes. The suspended process can be restarted later.

This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization. The difficulty with the working-set model is keeping track of the working set. The

WORKING SETS AND PAGE-FAULT RATES

There is a direct relationship between the working set of a process and its page-fault rate. Typically, as shown in Figure 10.22, the working set of a process changes over time as references to data and code sections move from one locality to another. Assuming there is sufficient memory to store the working set of a process (that is, the process is not thrashing), the page-fault rate of the process will transition between peaks and valleys over time. This general behavior is shown below:



A peak in the page-fault rate occurs when we begin demand-paging a new locality. However, once the working set of this new locality is in memory, the page-fault rate falls. When the process moves to a new working set, the page-fault rate rises toward a peak once again, returning to a lower rate once the new working set is loaded into memory. The span of time between the start of one peak and the start of the next peak represents the transition from one working set to another.

working-set window is a moving window. At each memory reference, a new reference appears at one end, and the oldest reference drops off the other end. A page is in the working set if it is referenced anywhere in the working-set window.

We can approximate the working-set model with a fixed-interval timer interrupt and a reference bit. For example, assume that Δ equals 10,000 references and that we can cause a timer interrupt every 5,000 references. When we get a timer interrupt, we copy and clear the reference-bit values for each page. Thus, if a page fault occurs, we can examine the current reference bit and two in-memory bits to determine whether a page was used within the last 10,000 to 15,000 references. If it was used, at least one of these bits will be on. If it has not been used, these bits will be off. Pages with at least one bit on will be considered to be in the working set.

Note that this arrangement is not entirely accurate, because we cannot tell where, within an interval of 5,000, a reference occurred. We can reduce the uncertainty by increasing the number of history bits and the frequency of interrupts (for example, 10 bits and interrupts every 1,000 references). However, the cost to service these more frequent interrupts will be correspondingly higher.

10.6.3 Page-Fault Frequency

The working-set model is successful, and knowledge of the working set can be useful for prepaging (Section 10.9.1), but it seems a clumsy way to control thrashing. A strategy that uses the **page-fault frequency (PFF)** takes a more direct approach.

The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page-fault rate (Figure 10.23). If the actual page-fault rate exceeds the upper limit, we allocate the process

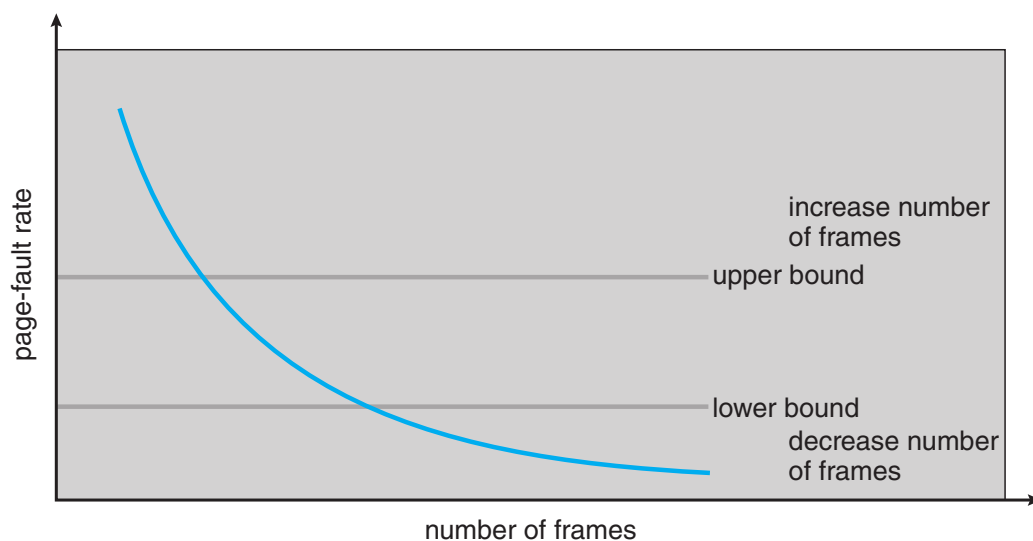


Figure 10.23 Page-fault frequency.