**Allocation of Frames**

**Minimum Number of Frames:**::

Each process needs be allocated a minimum number of frames. As the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution. In addition, remember that, when a page fault occurs before an executing instruction is complete, the instruction must be restarted. Consequently, we must have enough frames to hold all the different pages that any single instruction can reference. The minimum number of frames is defined by the computer architecture.

Whereas the minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory.

**Allocation Algorithms:**

:

**Equal allocation:** The easiest way to split m frames among n processes is to give everyone an equal share, m/n frames. For instance, if there are 93 frames and five processes, each process will get 18 frames. The three leftover frames can be used as a free-frame buffer pool.

**Proportional allocation**– Allocate free frames according to the size of process.

Let the size of the virtual memory for process *pi* be *si* , and define

*S* =\_*si* .

Then, if the total number of available frames is *m,* we allocate *ai* frames to process *pi*, where *ai* is approximately

*ai* = *si*/*S* × *m*.

With proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since

10/137 × 62 ≈ 4, and

127/137 × 62 ≈ 57.

**Global Versus Local Allocation:**

Another important factor in the way frames are allocated to the various processes is page replacement

With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement**.

Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.

Local replacement requires that each process select from only its own set of allocated frames.

With a local replacement strategy, the number of frames allocated to a process does not change. With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it (assuming that other processes do not choose *its* frames for replacement).

One problem with a global replacement algorithm is that a process cannot control its own page-fault rate. The set of pages in memory for a process depends not only on the paging behavior of that process but also on the paging behavior of other processes. Therefore, the same process may perform quite differently (for example, taking 0.5 seconds for one execution and 10.3 seconds for the next execution) because of totally external circumstances. Such is not the case with a local replacement algorithm. Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process. Local replacement might hinder a process, however, by not making available to it other, less used pages of memory. Thus, global replacement generally results in greater system throughput and is therefore the more common method.

**Thrashing:** a process is busy swapping pages in and out.

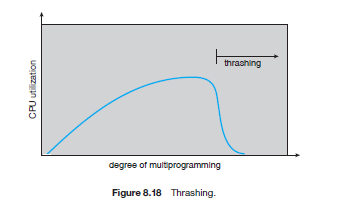
**Cause of Thrashing:**

The operating system monitors CPU utilization. If CPU utilization 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 8.18, 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

even 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.

****

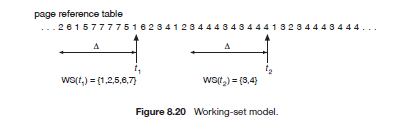
**Solution for Thrashing:**

We can limit the effects of thrashing by using a **local replacement algorithm** With local replacement, 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 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”? There are several techniques. The working-set strategy starts by looking at how many frames a process is actually using. This approach defines the **locality model** of process execution.

**Working-set model:**

The **working-set model** is based on the assumption of locality. This model uses a parameter, ∆, to define the **working-set window**. The idea is to examine the most recent ∆page references. The set of pages in the most recent ∆page references is the **working set** (Figure 8.20). 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 ∆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 8.20, if ∆= 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 ∆. If ∆is too small, it will not encompass the entire locality; if ∆is too large, it may overlap several localities. In the extreme, if ∆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, *WSSi* , for each process in the system, we can then consider that

*D* =∑*WSSi* ,

where *D*is the total demand for frames. Each process is actively using the pages in its working set. Thus, process *i* needs *WSSi* 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.