고려대학교
KOREA UNIVERSITY

KU-The Future

Operating Systems (10th Ed., by A. Silberschatz)

Chapter 10 Virtual Memory

Heonchang Yu
Distributed and Cloud Computing Lab.

Contents

Allocation of Frames

- Consider a simple case of a system with 128 frames.
  - ✓ The OS may take 35KB, leaving 93 frames for the user process.
  - ✓ Under pure demand paging, all 93 frames would initially be put on the free-frame list.
  - ✓ When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th.

Allocation of Frames

– Minimum Number of Frames
  - One reason for allocating at least a minimum number of frames involves performance.
    - ✓ As the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.
    - ✓ 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.
    - ✓ If the move instruction for a given architecture includes more than one word for some addressing modes, the instruction itself may straddle two frames. In addition, if each of its two operands may be indirect references, a total of six frames are required.

Allocation of Frames

- 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 5 processes, each process will get 18 frames. The 3 leftover frames can be used as a free-frame buffering pool.
  - Proportional allocation
    - ✓ Allocate available memory to each process according to its size
    - ✓ Let the size of the virtual memory for process p_i be s_i.
      - ❖ $S = \sum s_i$
    - ✓ If the total number of available frame is m, we allocate a_i frames to process p_i.
      - ❖ $a_i = \frac{s_i}{S} \times m$
    - ✓ 62 frames between one of 10 pages and one of 127 pages
      - ❖ 10/137×62 ≒ 4 and 127/137×62 ≒ 57

Allocation of Frames

- Global versus Local Allocation
  - Global replacement
    - ✓ Allows a process to select a replacement frame from the set of all frames
    - ✓ One problem is that 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.
  - Local replacement
    - ✓ Requires that each process select from only its own set of all allocated frames

Global Page Replacement

- With this approach, we satisfy all memory requests from the free-frame list, but we trigger page replacement when the list falls below a certain threshold.
- The strategy's purpose is to keep the amount of free memory above a minimum threshold.
  - ✓ When it drops below this threshold, a kernel routine (known as **reapers**) is triggered that begins reclaiming pages from all processes in the system.
  - ✓ When the amount of free memory reaches the maximum threshold, the reaper routine is suspended, only to resume once free memory again falls below the minimum threshold.
- Figure 10.18
  - ✓ We see that at point '**a**' the amount of free memory drops below the minimum threshold, and the kernel begins reclaiming pages and adding them to the free-frame list. It continues until the maximum threshold is reached (point '**b**').

Global Page Replacement

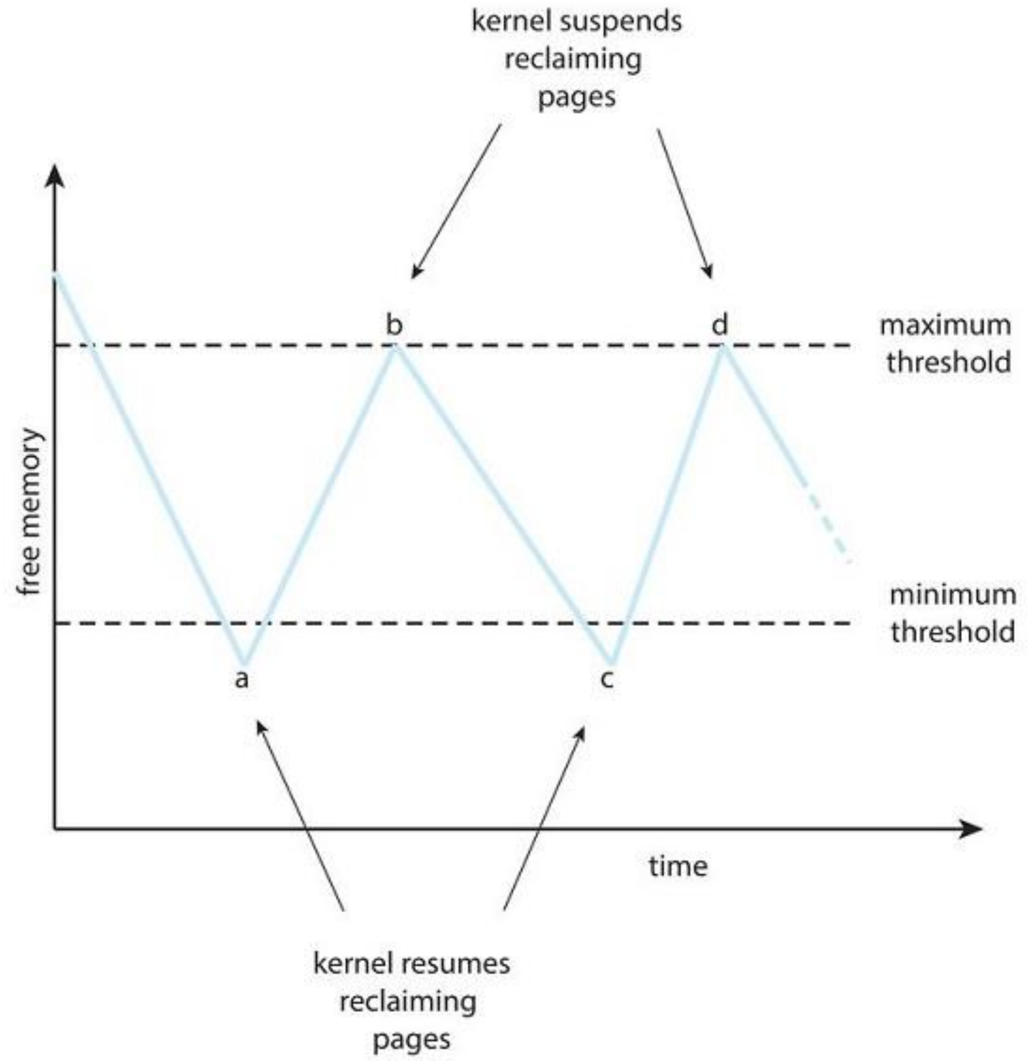
kernel suspends reclaiming pages
b
d
maximum threshold
free memory
minimum threshold
a
c
time
kernel resumes reclaiming pages

Figure 10.18 Reclaiming pages.

Major and Minor Page Faults

- Major Faults (Hard faults)
  - ✓ Occur when a page is referenced and the page is not in memory.
  - ✓ Servicing a major page fault requires reading the desired page from the backing store into a free frame and updating the page table.
- Minor Faults (Soft faults)
  - ✓ Occur when a process does not have a logical mapping to a page, yet that page is in memory. Minor faults can occur for one of two reasons
    - ❖ First, a process may reference a shared library that is in memory, but the process does not have a mapping to it in its page table. In this instance, it is only necessary to update the page table to refer to the existing page in memory.
    - ❖ A second cause of minor faults occurs when a page is reclaimed from a process and placed on the free-frame list, but the page has not yet been zeroed out and allocated to another process.
  - ✓ Resolving a minor page fault is much less time consuming.

Thrashing

- Consider what occurs if a process does not have “enough” frames.
  - ✓ It does not have the minimum number of frames it needs to support pages in the working set.
  - ✓ 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**.

Thrashing

- Cause of Thrashing
  - If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.
  - 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.

Thrashing

- Cause of Thrashing
  - 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.
  - Figure 10.20
    - ✓ 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.
    - ✓ To increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.

Thrashing

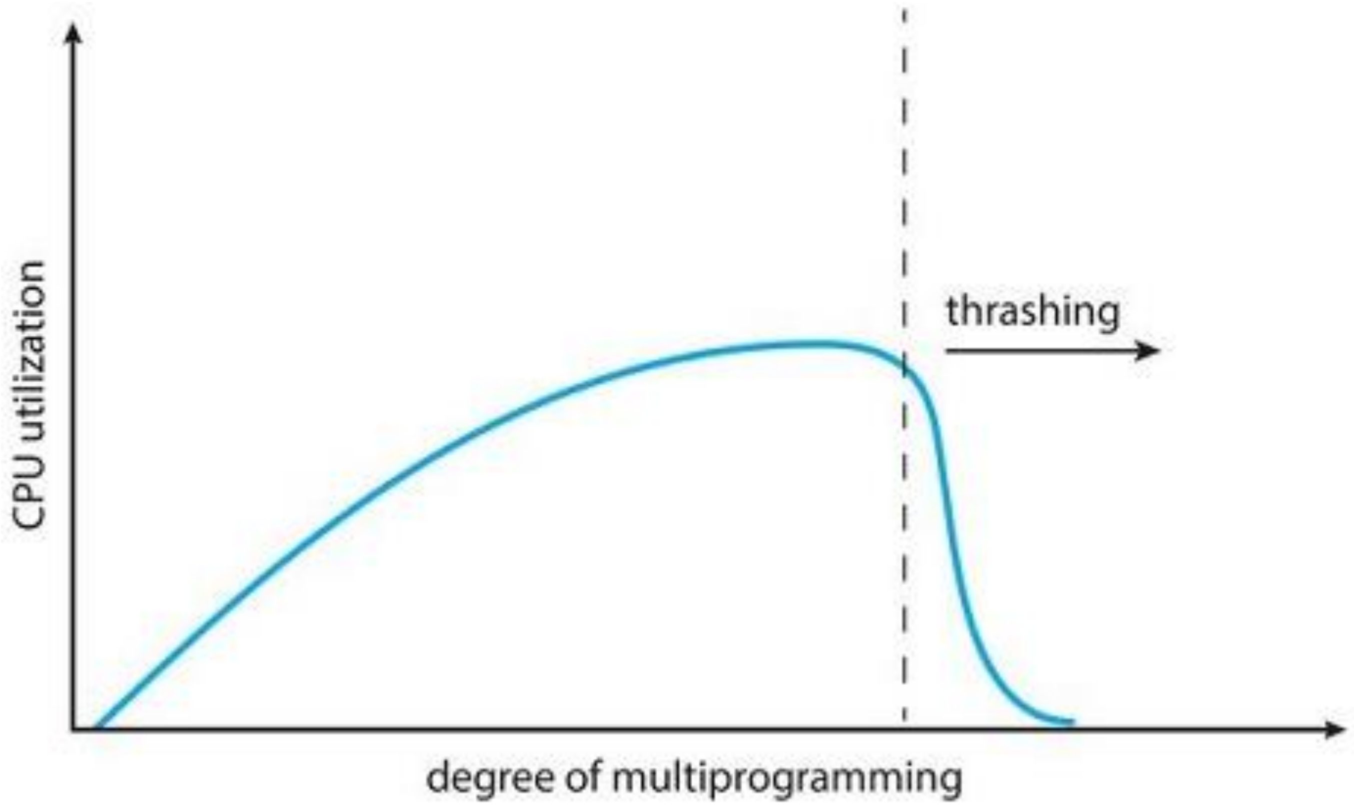

Figure 10.20 Thrashing.

Thrashing

- Cause of 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"?
    - ✓ Working-set strategy defines the locality model.
      - ❖ The locality model states that, as a process executes, it moves from locality to locality.
      - ❖ Figure 10.21 – A locality is a set of pages that are actively used together
        - » When a function is called, it defines a new locality.
        - » When we exit the function, the process leaves this locality.

Thrashing

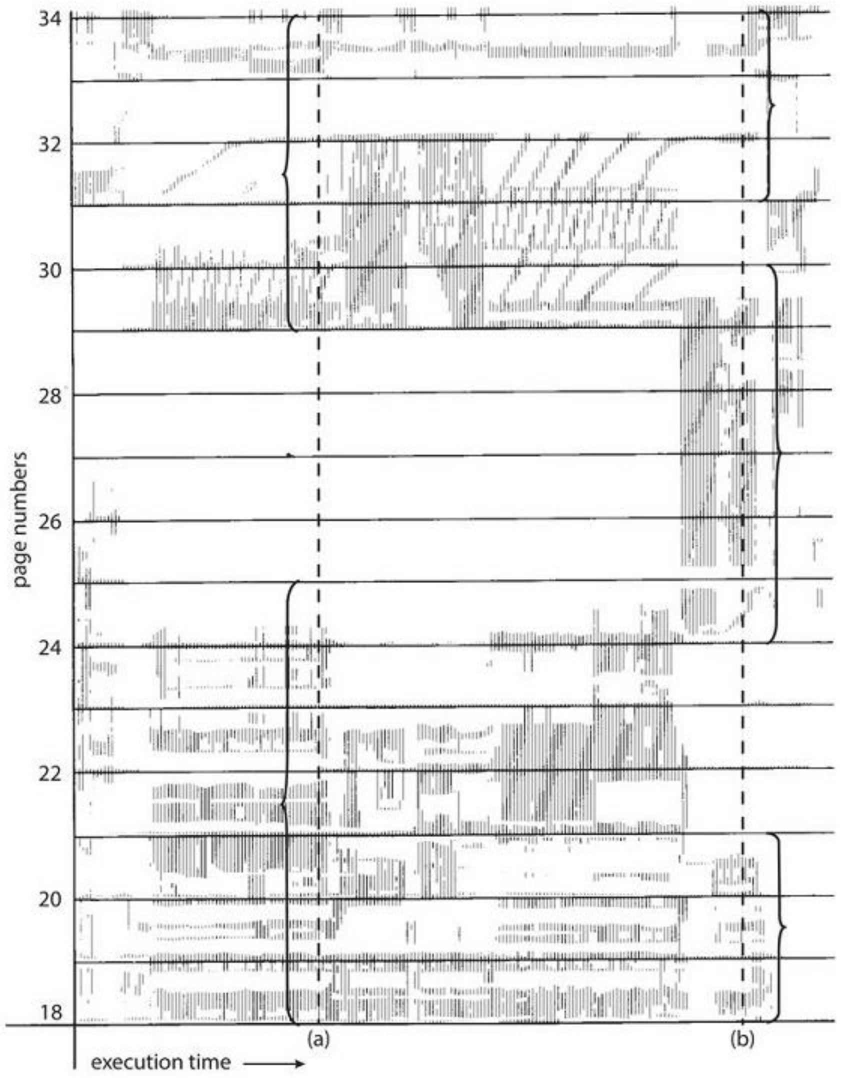

Figure 10.21 Locality in a memory-reference pattern.

Thrashing

- Working-Set Model
  - 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.
    - ✓ Figure 10.22
      - ❖ If Δ = 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}.

Thrashing

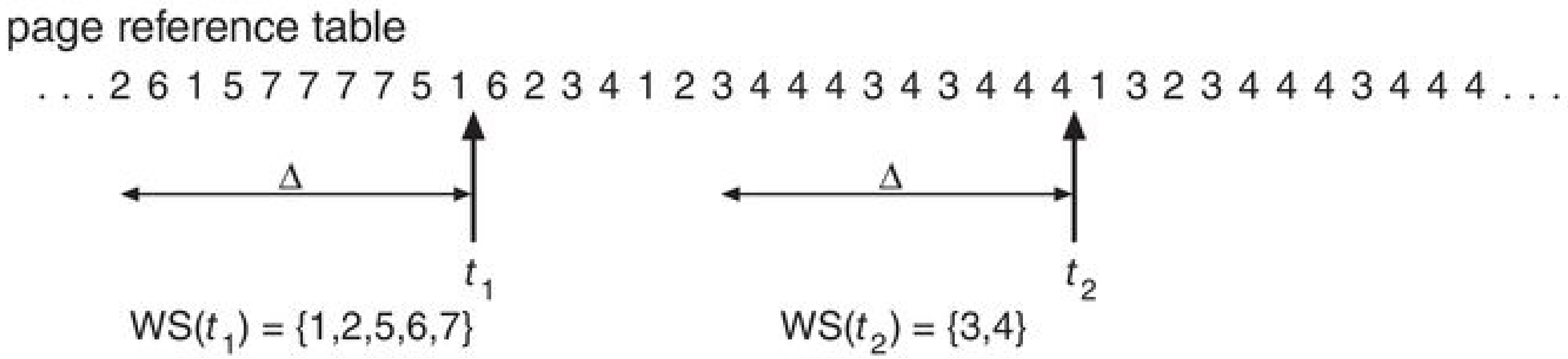

Figure 10.22 Working-set model.

Thrashing

- Working-Set Model
  - 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 is its size. If we compute the working-set size, WSS_i , for each process

      $D = \sum WSS_i$, where D is the total demand for 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.

Thrashing

- Working-Set Model
  - The difficulty with the working-set model is keeping track of the working set.
    - ✓ The 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.

Thrashing

- Page-Fault Frequency
  - The specific problem is how to prevent thrashing.
  - We want to control the page-fault rate.
    - ✓ When it is too high, we know that the process needs more frames.
    - ✓ If the page-fault rate is too low, then the process may have too many frames.
  - Figure 10.23
    - ✓ We can establish upper and lower bounds on the desired page-fault rate.
    - ✓ If the actual page-fault rate exceeds the upper limit, we allocate the process another frame.
    - ✓ If the page-fault rate falls below the lower limit, we remove a frame from the process.

Thrashing

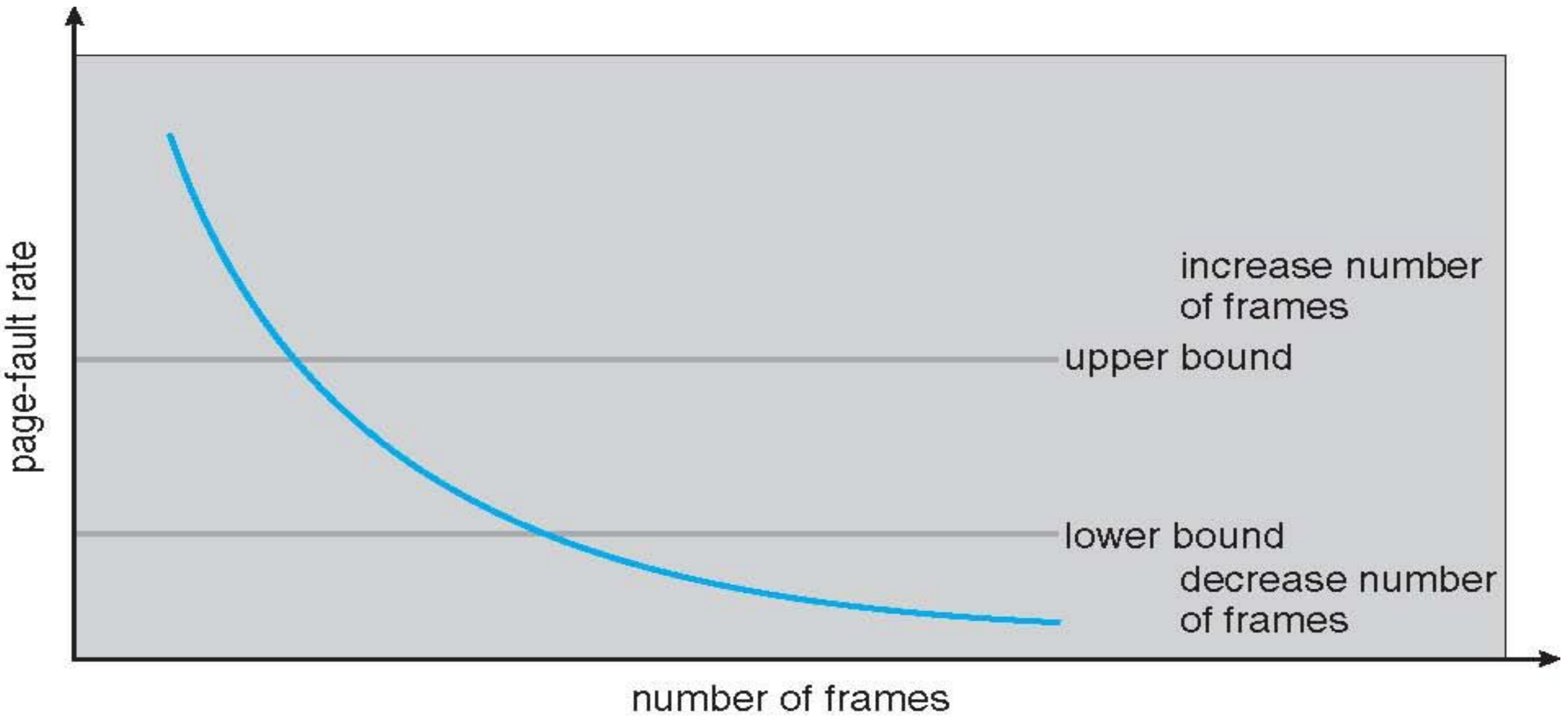

Figure 10.23 Page-fault frequency

Thrashing

- Working Sets and Page-Fault Rate
  - A direct relationship between the working set of a process and its page-fault rate
    - ✓ The working set of a process changes over time as references to data and code sections move from one locality to another
  - Figure
    - ✓ The page-fault rate of the process will transition between peaks and valleys over time
    - ✓ A peak in the page-fault rate occurs when we begin demand-paging a new locality.
    - ✓ Once the working set of this new locality is in memory, the page-fault rate falls.
    - ✓ 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.

Thrashing

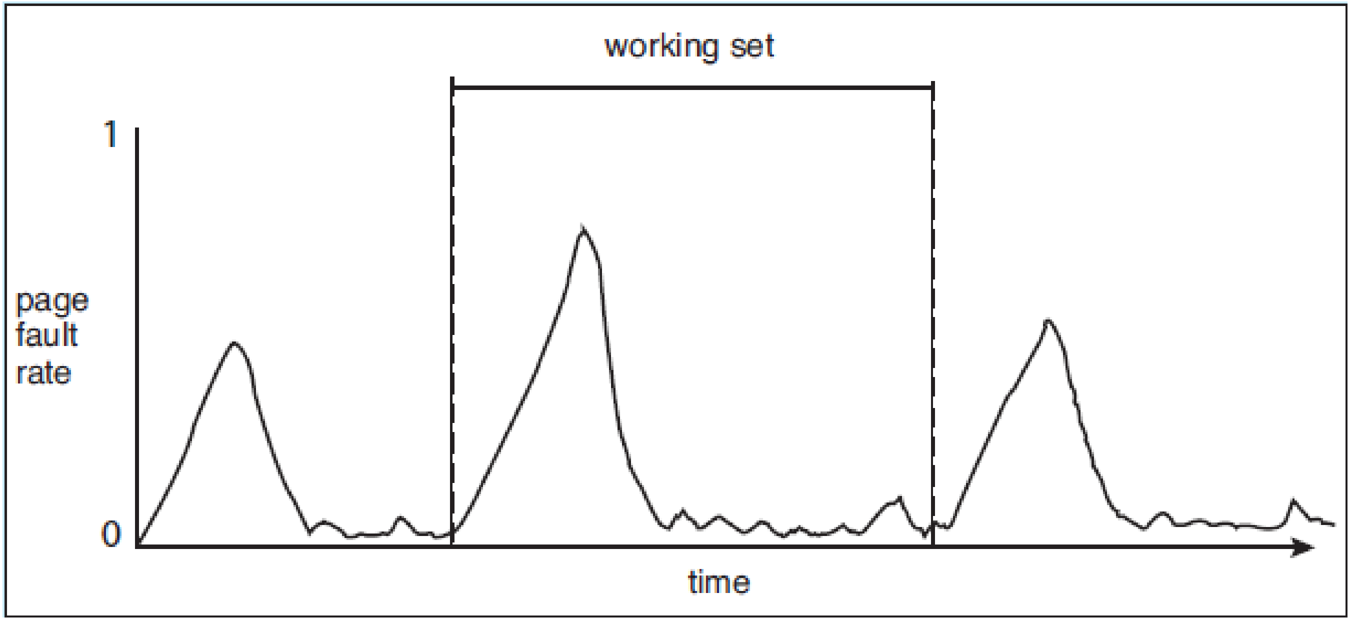

Figure Page-fault rate over time

Memory Compression

- An alternative to paging
  - ✓ Rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage.
- In Figure 10.24, the free-frame list contains six frames.
  - ✓ Assume that this number of free frames falls below a certain threshold that triggers page replacement.
  - ✓ The replacement algorithm (say, an LRU approximation algorithm) selects four frames—15, 3, 35, and 26—to place on the free-frame list. It first places these frames on a modified-frame list.
- In Figure 10.25, frame 7 is removed from the free-frame list.
  - ✓ Frames 15, 3, and 35 are compressed and stored in frame 7, which is then stored in the list of compressed frames. The frames 15, 3, and 35 can now be moved to the free-frame list.
  - ✓ If one of the three compressed frames is later referenced, a page fault occurs, and the compressed frame is decompressed, restoring the three pages 15, 3, and 35 in memory.

Memory Compression

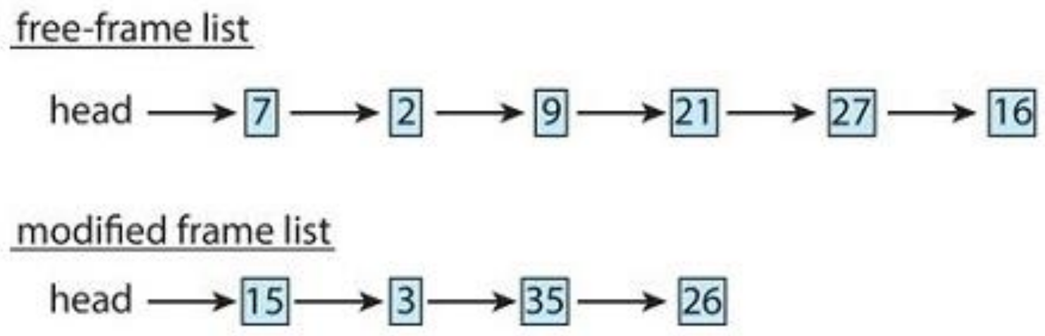

Figure 10.24 Free-frame list before compression.

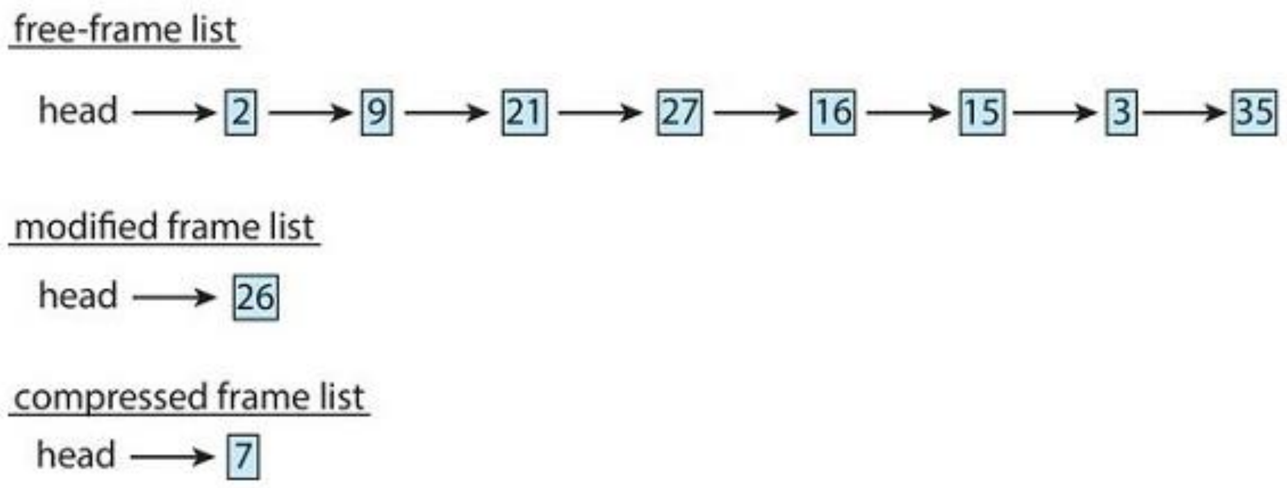

Figure 10.25 Free-frame list after compression

Allocating Kernel Memory

- When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernel.
- If a user process requests a single byte of memory, internal fragmentation will result, as the process will be granted an entire page frame.

– Buddy System

- The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages.
  - ✓ Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, and so forth).
  - ✓ For example, a request for 11 KB is satisfied with a 16-KB segment.

Allocating Kernel Memory

- Buddy System
  - Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory.
    - ✓ The segment is initially divided into two *buddies* - which we will call A_L and A_R - each 128 KB in size.
    - ✓ One of these buddies is further divided into two 64-KB buddies – B_L and B_R.
    - ✓ However, the next-highest power of 2 from 21 KB is 32 KB, so either B_L or B_R is again divided into two 32-KB buddies, C_L and C_R. One of these buddies is used to satisfy the 21-KB request.
  - An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as **coalescing**.
  - The obvious drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments.

Allocating Kernel Memory

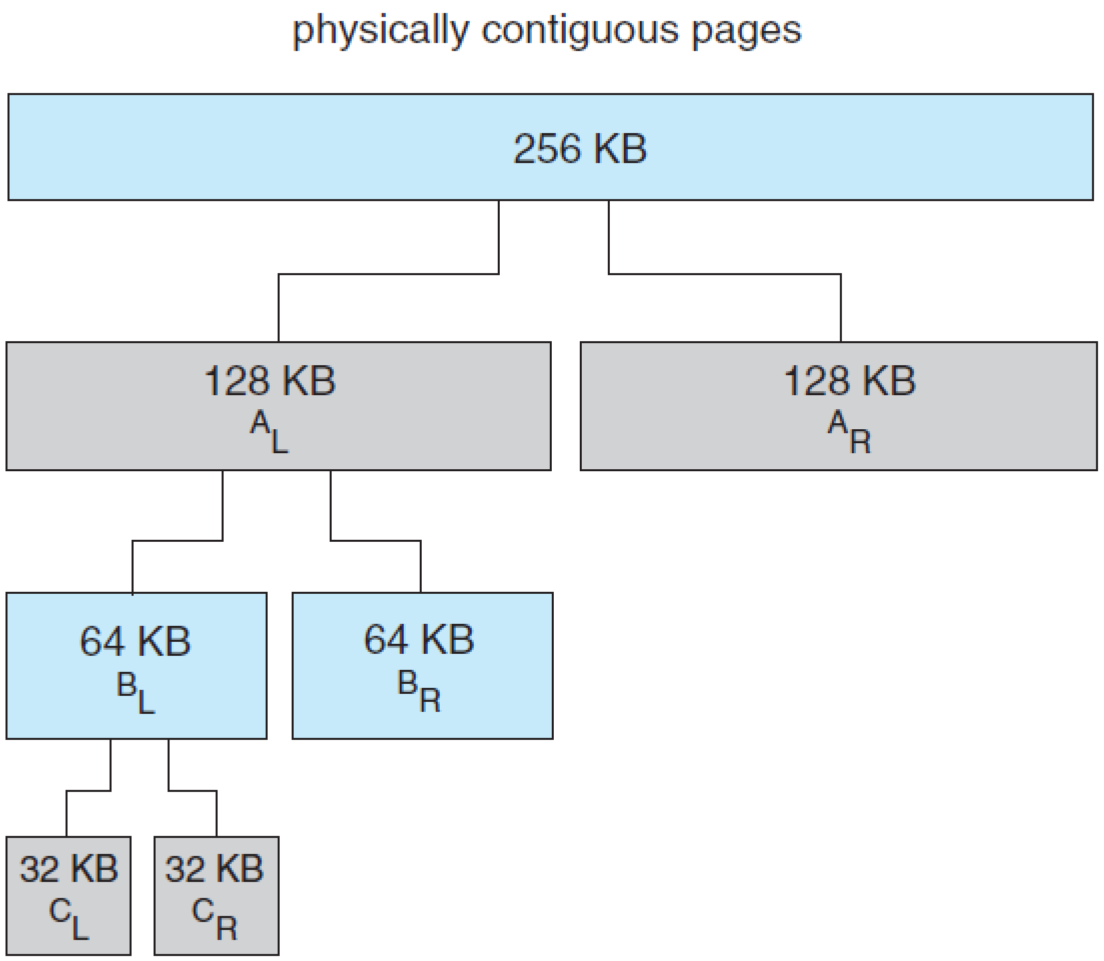

Figure 10.26 Buddy system allocation

Other Considerations for Paging Systems

- Prepaging
  - To prevent this high level of initial paging.
  - The strategy is to bring some – or all – of the pages will be needed into memory at one time.
  - The question is simply whether the cost of using prepaging is less than the cost of servicing the corresponding page faults.
  - It may well be the case that many of the pages brought back into memory by prepaging will not be used.
  - Assume that s pages are prepaged and a fraction α of these s pages is used ($0 \leq \alpha \leq 1$).
    - ✓ The question is whether the cost of the $s * \alpha$ saved page faults is greater or less than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages.
      - ❖ If α is close to 0, prepaging loses
      - ❖ If α is close to 1, prepaging wins.

Other Considerations for Paging Systems

- Page Size
  - One concern is the size of the page table.
    - ✓ For a given virtual memory space, decreasing the page size increases the number of pages and hence the size of the page table.
      - ❖ For a virtual memory of 4 MB (2^{22}), for example, there would be 4,096 pages of 1,024 bytes but only 512 pages of 8,192 bytes.
  - Memory is better utilized with smaller pages.
    - ✓ We can expect that, on the average, half of the final page of each process will be wasted.
      - ❖ This loss is only 256 bytes for a page of 512 bytes but is 4,096 bytes for a page of 8,192 bytes.
      - ❖ To minimize internal fragmentation, then, we need a small page size.

Other Considerations for Paging Systems

- Page Size
  - Another problem is the time required to read or write a page. I/O time is composed of seek, latency, and transfer times.
    - ✓ Latency and seek time normally dwarf transfer time.
      - ❖ At a transfer rate of 50 MB per second, it takes only 0.01ms to transfer 512 bytes. Latency time, though, is perhaps 3ms and seek time 5ms.
    - ✓ A desire to minimize I/O time argues for a larger page size.
  - With a smaller page size, though, total I/O should be reduced, since locality will be improved. A smaller page size allows each page to match program locality more accurately.
    - ✓ A smaller page size should result in less I/O and less total allocated memory.
  - As we have seen, some factors (**internal fragmentation**, **locality**) argue for **a small page size**, whereas others (**table size**, **I/O time**) argue for **a large page size**.

Other Considerations for Paging Systems

- TLB Reach
  - The hit ratio is related to the number of entries in the TLB, and the way to increase the hit ratio is by increasing the number of entries.
  - Related to the hit ratio is a similar metric: TLB reach
    - ✓ The amount of memory accessible from the TLB
    - ✓ the number of entries multiplied by the page size
    - ✓ The working set for a process is stored in the TLB. If it is not, the process will spend a considerable amount of time resolving memory references in the page table rather than the TLB.
  - Another approach for increasing the TLB reach is to either increase the size of the page or provide multiple page sizes.
    - ✓ If we increase the page size – say, from 4 KB to 16 KB – we quadruple the TLB reach. (internal fragmentation problem)
    - ✓ Most architectures provide support for more than one page size.
      - ❖ For example, the default page size on Linux systems is 4 KB; however, Linux also provides huge pages.

Other Considerations for Paging Systems

- Inverted Page Table
  - The purpose of this form of page management is to reduce the amount of physical memory needed to track virtual-to-physical address translations.
  - Accomplish this savings by creating a table that has one entry per page of physical memory, indexed by the pair <process-id, page-number>.
  - Inverted page tables reduce the amount of physical memory needed to store this information.

Other Considerations for Paging Systems

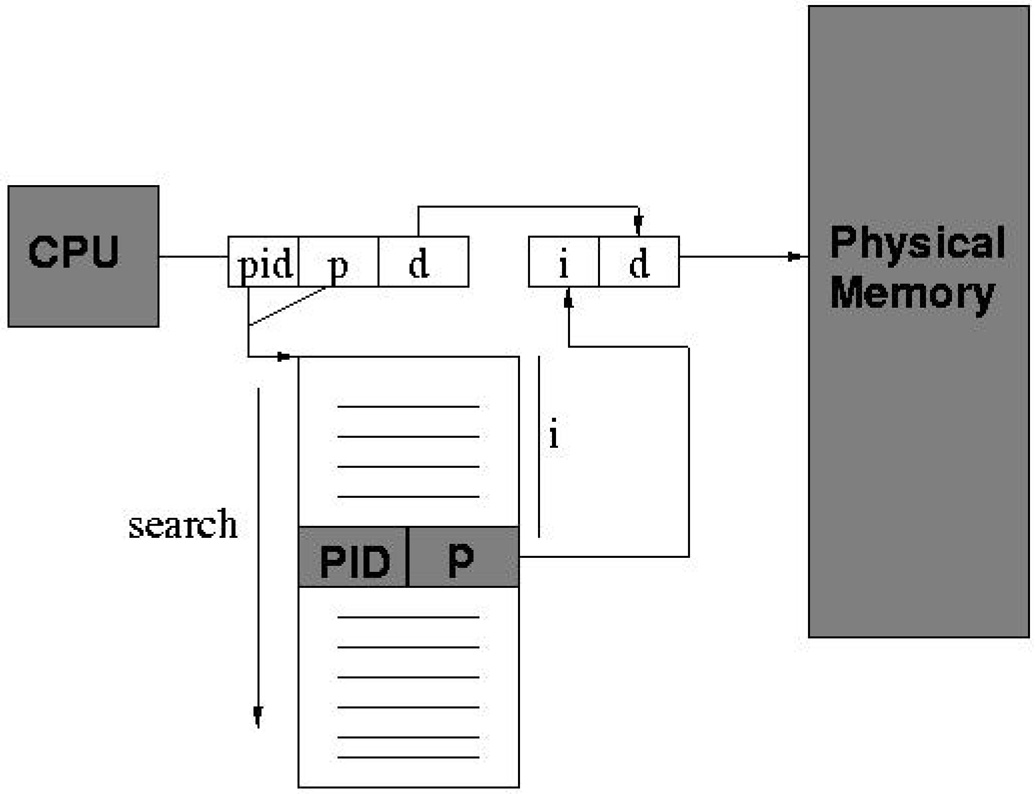

Figure Inverted page table

Other Considerations for Paging Systems

– Program Structure

- Assume that pages are 128 words in size. Consider a C program whose function is to initialize to 0 each element of a 128-by-128 array.
  - ✓ Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i][j] = 0;
```

❖ The array is stored row major.

❖ The array is stored data[0][0], data[0][1], · · ·, data[0][127], data[1][0], data[1][1], · · ·, data[127][127].

❖ The preceding code zeros one word in each page, then another word in each page, and so on.

» 128 x 128 = 16,384 page faults

Other Considerations for Paging Systems

- Program Structure
  - ✓ Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i][j] = 0;
```

❖ This code zeros all the words on one page before starting the next page, reducing the number of page faults to 128.

- Careful selection of data structures and programming structures can increase locality and hence lower the page-fault rate and the number of pages in the working set.

Other Considerations for Paging Systems

- I/O Interlock
  - When demand paging is used, we sometimes need to allow some of the pages to be locked in memory.
  - For example, a controller for a USB storage device is generally given the number of bytes to transfer and a memory address for the buffer (Figure 10.28).
  - We must be sure the following sequence of events does not occur:
    - ✓ A process issues an I/O request and is put in a queue for that I/O device. Meanwhile, the CPU is given to other processes. These processes cause page faults, and one of them, using a global replacement algorithm, replaces the page containing the memory buffer for the waiting process.
    - ✓ The pages are paged out. Some time later, when the I/O request advances to the head of the device queue, the I/O occurs to the specified address.
    - ✓ However, this frame is now being used for a different page belonging to another process.

Other Considerations for Paging Systems

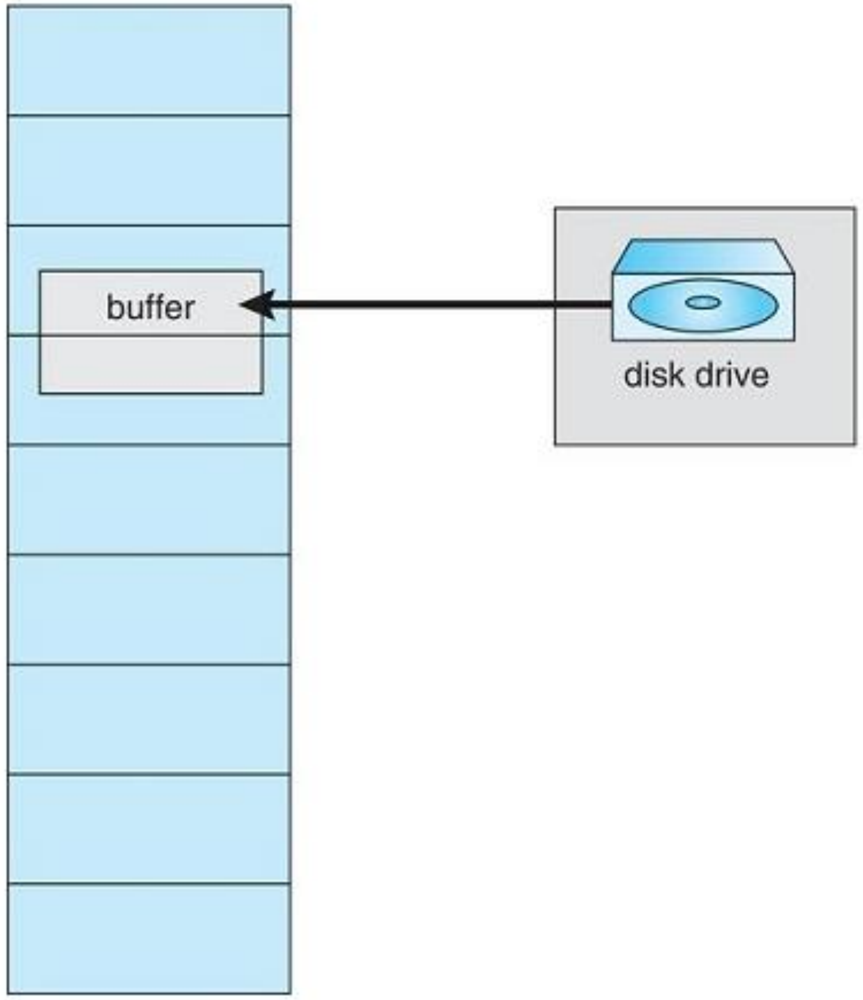

Figure 10.28 The reason why frames used for I/O must be in memory.

Other Considerations for Paging Systems

- I/O Interlock
  - Two common solutions
    - ✓ One solution is never to execute I/O to user memory. Instead, data are always copied between system memory and user memory.
      - ❖ I/O takes place only between system memory and the I/O device. To write a block on tape, we first copy the block to system memory and then write it to tape.
      - ❖ This extra copying may result in unacceptably high overhead.
    - ✓ Another solution is to allow pages to be locked into memory.
      - ❖ A lock bit is associated with every frame. If the frame is locked, it cannot be selected for replacement.
      - ❖ Locked pages cannot be replaced. When the I/O is complete, the pages are unlocked.

Operating-System Example – Linux

- To manage memory, Linux maintains two types of page lists: an `active_list` and an `inactive_list`.
  - ✓ `active_list` contains the pages that are considered in use.
  - ✓ `inactive_list` contains pages that have not recently been referenced and are eligible to be reclaimed.
- Each page has an ***accessed*** bit that is set whenever the page is referenced.
  - ✓ When a page is first allocated, its accessed bit is set, and it is added to the rear of the `active_list`.
  - ✓ Whenever a page in the `active_list` is referenced, its accessed bit is set, and the page moves to the rear of the list.
  - ✓ Periodically, the accessed bits for pages in the `active_list` are reset. Over time, the least recently used page will be at the front of the `active_list`.
  - ✓ From there, it may migrate to the rear of the `inactive_list`. If a page in the `inactive_list` is referenced, it moves back to the rear of the `active_list`.

Operating-System Example – Linux

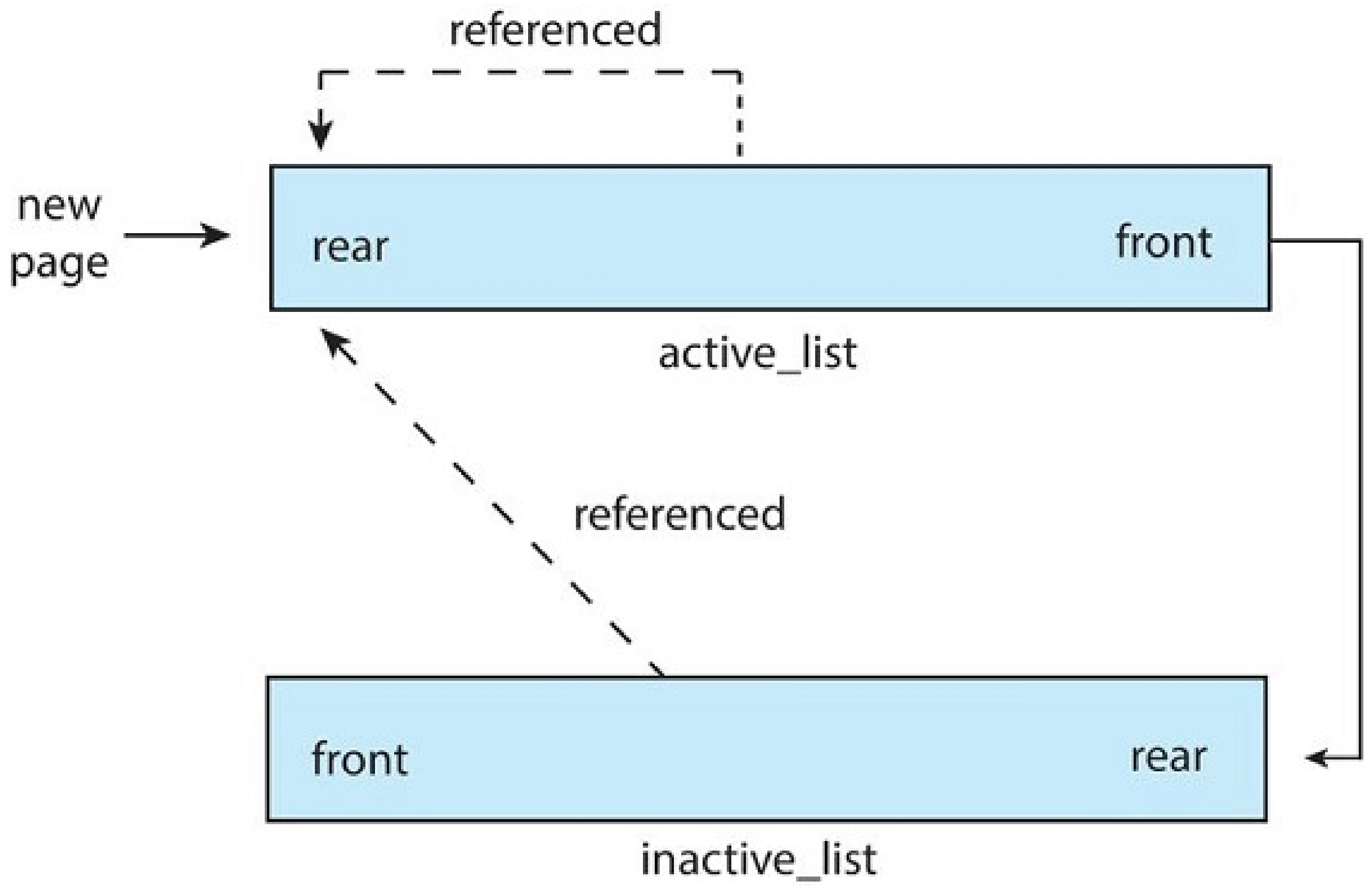

Figure 10.29 The Linux active_list and inactive_list structures.