File Attributes

A file's attributes vary from one operating system to another but typically consist of these:

• Name. The symbolic file name is the only information kept in human-readable form.

• Identifier. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.

• Type. This information is needed for systems that support different types of files.

• Location. This information is a pointer to a device and to the location of the file on that device.

• Size. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.

• Protection. Access-control information determines who can do reading, writing, executing, and so on.

• Time, date, and user identification. This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring

File Operations

The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

- Creating a file.
  Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
- Writing a file.
  To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- Reading a file.
  To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated.
- Repositioning within a file.
  The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file seek.
- Deleting a file.
  To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
- Truncating a file.

The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length.

Basic Page Replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 9.10). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
1. Find a free frame:
   a. If there is a free frame, use it.
   b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
   c. Write the victim frame to the disk; change the page and frame tables accordingly.
2. Read the desired page into the newly freed frame; change the page and frame tables.
3. Continue the user process from where the page fault occurred.

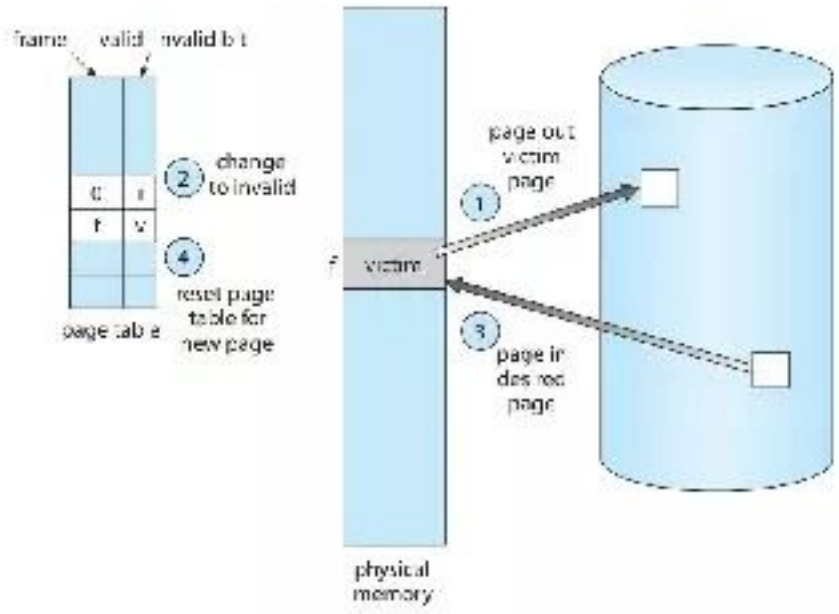

Figure 9.10 Page replacement.

Describe Banker's Algorithm:

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may
not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave

the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

- **Available**: A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Rj are available.
- **Max**: An n × m matrix defines the maximum demand of each process. If Max[i][j] equals k, then process Pi may request at most k instances of resource type Rj .
- **Allocation**: An n × m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals k, then process Pi is currently allocated k instances of resource type Rj .
- **Need**: An n × m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task. Note that Need[i][j] equals Max[i][j] − Allocation[i][j].

These data structures vary over time in both size and value.
We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let Work and Finish be vectors of length m and n, respectively.
    a. Initialize Work = Available and Finish[i] = false for i = 0, 1, ..., n − 1.
2. Find an index i such that both
    a. Finish[i] == false
    b. b. Needi ≤ Work
    If no such i exists, go to step 4.
3. Work = Work + Allocationi
    Finish[i] = true
    Go to step 2.
4. If Finish[i] == true for all i, then the system is in a safe state.

The algorithm for determining whether requests can be safely granted is described as follows
Let Requesti be the request vector for process Pi.
If Requesti [j] == k, then process Pi wants k instances of resource type Rj . When a request for resources is made by process Pi, the following actions are taken:

1. If Requesti ≤ Needi , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Requesti ≤ Available, go to step 3. Otherwise, Pi must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available = Available–Requesti ;

Allocationi = Allocationi + Requesti ;
Needi = Needi –Requesti ;

If the resulting resource-allocation state is safe, the transaction is completed, and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for Requesti , and the old resource-allocation state is restored.

Explain the basic Hardware of Memory Management System

Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

Registers that are built into the CPU are generally accessible within one cycle of the CPU clock while the main memory is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock.