

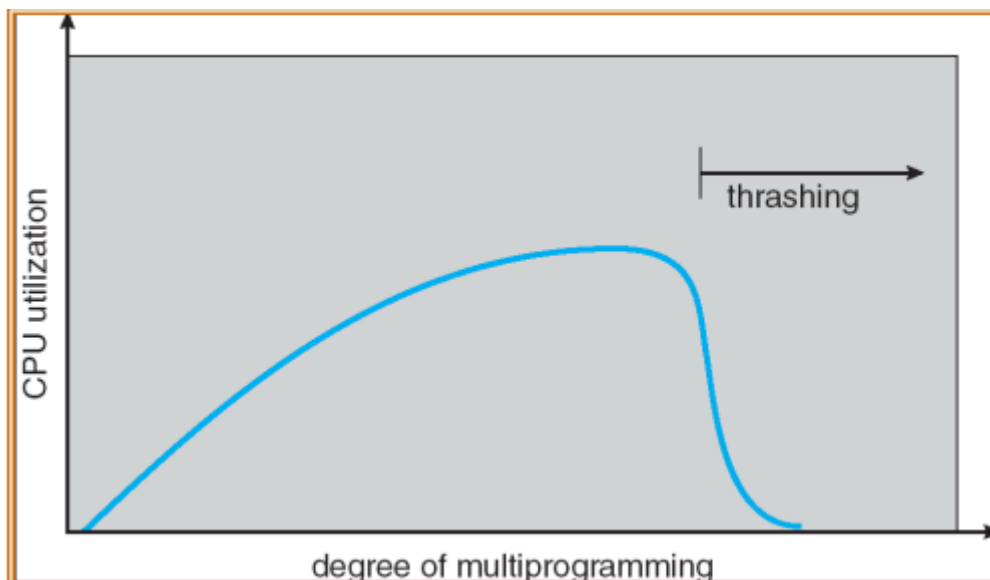
## Thrashing

In fact, look at any process that does not have “enough” frames. If the process does not have the number of frames it needs to support pages in active use, it 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.

We can limit the effects of thrashing by using a **local replacement algorithm** (or **priority 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.
- 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 program is generally composed of several different localities, which may overlap.



For example,

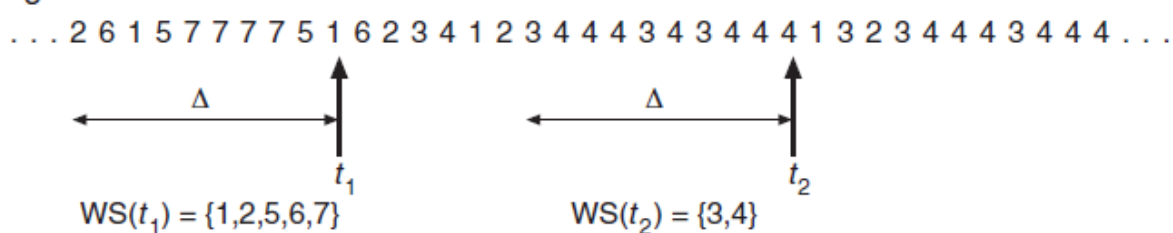
- ❖ When a function is called, it defines a new locality. In this locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables.
- ❖ When we exit the function, the process leaves this locality, since the local variables and instructions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

**Working-Set Model**

As mentioned, the **working-set model** is based on the assumption of locality. This model uses a parameter,  $\Delta$ , to define the **working-set window**. The idea is to examine the most recent  $\Delta$  page references. The set of pages in the most recent  $\Delta$  page references is the **working set**. 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  $\Delta$  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 below, 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  $\{3, 4\}$ .

page reference table



Working-set model.

The accuracy of the working set depends on the selection of  $\Delta$ .

- If  $\Delta$  is too small, it will not cover the entire locality;
- if  $\Delta$  is too large, it may overlap several localities. In the extreme,
- if  $\Delta$  is infinite, the working set is the set of pages affected 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.

## FILES IN OS

For most users, the file system is the most visible aspect of an operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of *files*, each storing related data, and a *directory structure*, which organizes and provides information about all the files in the system.

### File Concept

- ❖ Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the *file*.
- ❖ Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.
- ❖ A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.
- ❖ The information in a file is defined by its creator. Many different types of information may be stored in a file—source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.
- ❖ A file has a certain defined **structure**, which depends on its type. A *text* file is a sequence of characters organized into lines (and possibly pages). A *source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An *object* file is a sequence of bytes organized into blocks understandable by the system's linker. An *executable* file is a series of code sections that the loader can bring into memory and execute.

### File Attributes

- ✓ **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

A file is an **abstract data type**. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let's examine what the operating system must do to perform each of these six basic file operations. It should then be easy to see how other similar operations, such as renaming a file, can be implemented.

- **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. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **currentfile- position pointer**. Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **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—but lets the file be reset to length zero and its file space released.

**File Types**

Common file types.

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information