File Management

COMP 229 (Section PP) Week 9

Prof. Richard Zanibbi
Concordia University
March 13, 2006

Last Week...Process Management

Managing Classic and Modern Processes (pp. 199-206)

Resources

Process Address Space

Collection of addresses (bytes) a thread can reference

OS Families

Share a System Call Interface (e.g. “UNIX”, “Windows”)

The Hardware Process (pp. 206-208)

Sequence of instructions physically executed by a system

Abstract Machine Interface & Implementation (pp. 208-225)

Process and Thread Abstractions

including Process and Thread Descriptors; traps for system calls

Execution States

Resource Management

Generic “Mechanisms” and Resource-Specific “Policies”

Generalizing Process Management Policies (pp. 226-228)

Processes, The Address Space, and Tracing the Hardware Process

Figures 6.1, 6.3

- Comparison of classic, modern processes
- Simulations of multiprogramming by the hardware process (actual machine instructions)

Address Spaces

- New elements in the address space (add files, other memory-mapped resources)
- Figure 6.4: binding resources into the address space

Tracing the Hardware Process

- Fig. 6.5: note that the Process Manager nearly alternates with the execution of all other processes in the system
  - allows proper management of the processes (e.g. enforcing resource isolation, sharing, and scheduling policies)

Example: Compiling a C Program to Use an Abstract Machine Interface

C Program

```
...
a = b + c;
pid = fork();
...
```

Compiled machine user instructions:

```
...
// a = b + c
load            R1,b
load            R2, c
add             R1, R2
store           R1, a
// now do the system call
trap            sys_fork
...
```

User mode machine instructions

System call (OS executes privileged instructions)

Process and Thread Descriptors, Process and Thread States, Resource Descriptors

Tables 6.3 and 6.4

- Recall that for modern processes, threads are represented using separate descriptors.
- For classic processes, there is only one base thread, represented within the process descriptor.

Process/Thread State Diagrams

- Simple model (Fig. 6.10)
- Unix model (Fig. 6.11)
- Generalization: parent processes can suspend child processes (Fig. 6.14)

Resource Descriptors

- Table 6.5 (resource descriptor)
- Reusable (fixed number of units, e.g. disk) vs. Consumable (unbounded number of units, e.g. messages produced/consumed by processes) resource types

This week...File Management

File System Types (pp. 514-528)

Low-Level File System *(for Byte Stream Files)*

Structured File System *(e.g. for records, .mp3 files)*

Low-Level File System Implementations (pp. 529-544)

Low-level file system architecture

Byte stream file *Open* and *Close* operations

Block Management

Reading and writing byte stream files

Overview, and File System Types

Files and the File Manager

Files As Resource Abstraction

Files are used to represent data on storage devices (e.g. disk)

Files As a Central Abstraction of Computation

- Most programs read one or more files, and eventually write results to one or more files
- Some view programs as filters that read data, transform the data, and write the result to a new file (is this an "extreme view"?)
  - e.g. UNIX shell applications using the pipe operator (|)
- In C: by default, stdin, stdout, stderr are defined

File Manager (OS Service Responsible for Files)

- Implements file abstraction
- Implements directory structure for organizing files
- Implements file systems for organizing collections of directories (e.g. on separate disks)
- File management involves operations on external devices & memory

Hard Disk Structure (Quickly)

A Multiple-Surface Disk

- Has a set of circular surfaces
- Has a group of read/write heads that move together, one per surface

Block

The smallest (fixed size) area that can be read or written to on a disk

Track

A set of blocks on a single disk surface, arranged in a circle

Cylinder

A set of tracks that a hard disk may access from a given position for the read/write heads

File Manager Types

External View of File Manager

Part of the System Call Interface implemented by the file manager (see Fig. 13.2)

Low vs. High-Level (Structured) File Systems

See Fig. 13.3

- Low Level File System implements only Stream-Block Translation and Byte-Stream Files (e.g. WIndows, Unix)
  - Applications must translate byte streams to/from abstract data types used in programs
- Structured (High-Level) File Systems also implement Record-Stream translation and Structured Record files (e.g. MacOS, systems for commercial applications, e.g. some IBM systems)
  - Have a language for defining record types, keys for searches

Marshalling: producing blocks from records ("flattening")
Unmarshalling: producing records from blocks

Multimedia Data

Media Types

– Different media types may require different access and modification strategies for efficient I/O (e.g. image vs. floating point number)

Low-level File Systems

– Not designed to accommodate multimedia data

– Less efficient than using built-in high-performance access methods in High-Level File Systems

File Descriptors

See Page 529 in Part II of text

- Make note of the "sharable" field, which defines whether processes may open the file simultaneously, and for which operations (read/write/execute)
- Storage Device Detail field: which blocks in secondary storage (e.g. on a disk) are used to store the file data (more on this later in lecture)

File System Types:
Low-Level File Systems

pp. 514-521

"Low Level Files" = Byte Stream Files

File Pointer

On file open:
(default) pointer set to 0

Read/Write K bytes:
Advance pointer by K

Setting File Pointer:
`lseek()` **(POSIX)**
`SetFilePointer()` **(Win)**

No "Type" for Bytes:
Treated as "raw" bytes

Name: Test

Byte	Value	(ASCII)
0	0100 0001	A
1	0100 0010	B
2	0100 0011	C
3	0100 0100	D
4	0100 0101	E
5	0100 0110	F
6	0100 0111	G

Example: POSIX File System Calls

System Call	Effect
`open()`	Open file for read or write. OS creates internal representation, optionally **locks** the file. Returns a file descriptor (integer), or -1 (error)
`close()`	Close file, releasing associated locks and resources (e.g. internal representation)
`read()`	Read bytes into a buffer. Normally blocks (suspends) a process until completion. Returns # bytes read, or -1 (error)
`write()`	Write bytes from a buffer. Normally blocks (suspends) a process until completion. Returns # bytes written, or -1 (error)
`lseek()`	Set the file pointer location
`fcntl()`	("File Control"): various options to set file attributes (locks, thread blocking, ...)

Stream-Block Translation (for Low-Level (Byte Stream) Files)

See Fig. 13.4

– Note API for low-level files, shown to the left of the “Stream-Block Translation” oval.

Structured File Types

pp. 522-528

Structured Files

Common applications

Business/Personnel Data

Multimedia data formats (e.g. images, audio)

Provide Data Structure Support

...within the file manager

Support for indexing records within a file, direct access of records, efficient update, etc.

See Figures 13.5, 13.6

Note that Record-Block Translation is achieved by combining Block-Stream translation with Stream-Record translation (see Fig. 13.3)

Supporting Structured File Types

Prespecified Record Types

Access Functions provided by File Manager (e.g. read/write for images)

A More General Approach

Programmer-defined abstract data types

Programmer-defined record read/write methods (e.g. using standard, predefined access function names)

File Manager invokes programmer routines

Structured Sequential File for Email Data

See Fig. 13.7: user-defined methods passed to file manager, which then uses them to read email folder files

message: abstract data type for an email message

getRecord: gets the next record under the file pointer (current file position)

putRecord: appends a message to the end of the file

Common Structured File Types

(Record-Oriented) Structured Sequential Files

Records organized as a list

Record attributes encoded in file **Header**

Indexed Sequential Files

- Records have an integer index in their header
- Records contain one or more fields used to index records in the file (e.g. student #)
- Either applications or the file manager **define tables associating record attributes with index values**
- Representation: just index values in records, linked lists (one per *key*), or stored index table (used by file manager)
- Popular in business, human resources applications

Inverted Files

- Generalized external (system) index tables used by file manager: allow entries to point to groups of records or fields
- New record fields are extracted and placed in the index table, with pointer in the table to the new file where appropriate
- Records accessed using the index table rather than location in file

Examples

Sequential Files

API operations on page 523

Indexed Sequential Files

See Fig. 13.8, API operations on page 526

Inverted Files

pages 526-527

Additional Storage Methods, Notes

Databases

p. 527

– data schemas used to define complex data types

Multimedia Data

p. 528

– variable sizes of data / performance issues require sophisticated storage, retrieval, and updating techniques for acceptable performance (e.g. for streaming or searching audio/video)

Low-level File System Architecture

Low-Level File Implementation

Disk Organization

Volume directory (defines location of files)

External file descriptor, one per file

File Data (the “files themselves”)

Disk Operations

Include reading, writing fixed size blocks

Low-Level File System Architecture

See Fig. 13.9

- Tapes, other sequential access media store files as contiguous blocks
- Disks provide random access: blocks in a file are often not contiguous (adjacent) on the disk surface.

Opening a File

See Fig. 13.10

- Buffers and other resources must be initialize in order to process the file
- File permissions compared against the process requesting the file, and the owner of that process to insure that the file should be accessible for the desired operation
- External file descriptor: on disk
- Internal file descriptor: created in memory

Opening a File in Unix (see Fig. 13.11)

- Note that in Unix, the “process-specific” file session information is stored in two data structures: the Open File (“descriptor”) Table, and a File Structure Table. Both of these are process-specific (i.e. each process has an open file table and file structure table)
- An internal file descriptor is called an “inode”

Closing a File (e.g. using `close()`)

When Issued:

- All pending operations completed (reads, writes)
- Releases I/O buffers
- Release locks a process holds on a file
- Update external file descriptor
- Deallocate file status table entry

Block Management for Low-level Files (Byte Streams)

Block Management

Purpose

Organizing the blocks in secondary storage (e.g. disk) that hold file data

(blocks containing file data are referenced from the "Storage Device Detail" field in a file descriptor; see p.530)

Computing number of blocks for a file

See page 534 in text

Block Management Strategies:

1. Contiguous allocation
2. Linked lists
3. Indexed allocation

Examples of Block Management

Contiguous Allocation

See Fig. 13.12

All blocks are adjacent (contiguous) on storage device: fast write/read

Problems with external fragmentation (space between files)

- must be space for the whole file when file is expanded; if not, the whole file must be relocated to another part of storage (e.g. the disk)

NOTE: “Head Position” is the location of

Linked Lists (Single and Doubly-Linked)

See Figs. 13.13 and 13.14

Blocks have fields defining the number of bytes in a block and link(s) to the next (also previous, if doubly-linked) block in the file

Blocks do not have to be contiguous, allowing us to avoid the fragmentation problems with contiguous allocation

Indexed Allocation

See Fig. 13.15

Size/link headers for blocks separated from the blocks and placed in an index

Index is stored with the file, and loaded into memory when a file is opened

Speeds searching, as all blocks are stored within the index table (no link following to locate blocks in the file)=

Block Representation in Unix File Descriptors

UNIX File Descriptors

See Table 13.1

UNIX File Block Representation

See Fig. 13.16

- 12 direct links to data blocks
- 3 index block references, which are singly, doubly, and triply indirect, respectively

Block Types: Data or Index

Index Block: list of other data or index blocks

The Unix block representation can represent more locations than most machines can store (example numbers given in text)

Next Week...

File Management, Cont'd

pp. 544-559 (Part II, Ch. 13)

Device Management (introduction)

pp. 152-163 (Part II, Ch. 5)