

**CS134a: Filesystems and I/O**

- Filesystems
  - Physical layout
  - Device Management



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

**Physical organization: review**

- A file is a sequence of records  $R_1, R_2, \dots, R_n$ 
  - In general, records may be fixed or variable size
- A file may be open for *sequential* or *direct* access
- We'll assume that there is a pointer that points to the next record that is to be accessed



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

**Physical file organization**

- Space on a physical device is allocated in *physical* records (like disk sectors)
- Physical records may differ from logical records
- Four types of physical allocation:
  - Contiguous
  - Linked
  - Index-sequential
  - B-tree structured

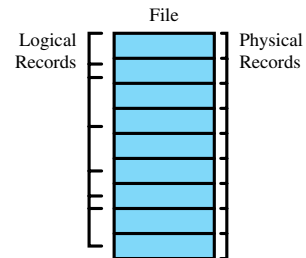


Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

**Contiguous organization**

- Each file is mapped onto a sequence of physical records



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

**Contiguous organization**

- When reading a record, fetch each physical block that is part of the record and copy its data to the buffer
- When writing a record that partially fills a physical record, fetch the physical block, modify the record, write the block
- If logical records are variable size, may want an *index* for their starting addresses
- Contiguous allocation is easy for reading/writing, but
  - maximum file size must be declared in advance
  - deleting and inserting logical records requires physically shifting the data

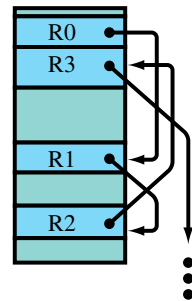


Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

**Linked organization**

- The physical blocks for a file may be scattered throughout secondary storage
- Each logical is linked to the next through a forward pointer
  - May also want a doubly-linked list
- Each read/write must follow the links to find the logical record
- Linked list is typically maintained in main memory and periodically saved to secondary storage (what happens on a system crash?)



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

### Index-sequential organization

- Permits direct access to records, but fixes the problem with insertion/deletion of contiguous allocation
- File is divided into *record-groups* containing  $n$  logical records
- Each record-group uses contiguous allocation (typically a sector on the disk)
- There is an index table; each entry contains
  - the key of the first record
  - a pointer to the first record
  - number of records in the group

Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>
November 2, 2001

### Index-sequential organization

Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>
November 2, 2001

### Unix filesystem

- Each inode contains 13 pointers
  - The first 10 point directly to blocks
  - Block 10 points to a second-level inode
  - Blocks 11, 12 point to a third-level inode
  - Block 13 points to a fourth-level inode
- Advantage: inode size is small for small files

Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>
November 2, 2001

### Unix filesystem

Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>
November 2, 2001

### B-Tree organization

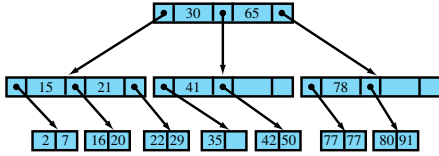
- Index files can quite large, so they are often saved in secondary storage
- B-Trees help minimize accesses to the index
- A B-Tree is a tree
  - Each node contains  $s$  slots and  $s+1$  pointers
  - Invariant: each node is always at least half-full

Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>
November 2, 2001

### Example B-Tree

Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>
November 2, 2001

**Example B-Tree after insertion/deletion**



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

**B-Trees**

- Key is max+1 of nodes to the left
- To find a entry  $i$ , recursively search
  - if  $i < key$ , go left
  - if  $i \geq key$ , go right
- B-Trees are *balanced*: distance to any leaf is a constant
- Insertion/deletion may need to rebalance the tree
- Search takes  $O(\log n)$  time



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

**B-Trees**

- *Sequential* access is hard
- B+-Trees maintain a linked list of the leaves
- What happens if an index block gets corrupted
  - In *index-sequential*?
  - In *B-tree*?



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

**Management of secondary storage**

- How should requests for physical blocks be granted?
- IO efficiency is the main criterion:
  - minimize rotational delays and seek times
  - allocate blocks for the same file "close" together
  - Often, try to use sequential allocation for blocks in the same file
- Free space
  - *Linked list*: difficult to find the right block (what happens if a block get corrupted?)
  - *Bit vector*



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

**The IO subsystem**

- Two remaining parts:
  - *Device IO techniques*
  - *IO scheduling and control*
- Objective: translate read/write commands to IO primitives
- Generate necessary device-specific commands
- Interpret results



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

**Schemes for CPU/device communication**

- Programmed IO
- Direct memory access (DMA)
- Interrupt-driven IO



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

### Programmed IO

- Used for slow devices like printers and terminals
- All data is explicitly transferred to device by CPU
- For example, to print a character the CPU reads the char and writes it to a specific device register
- On completion, the device raises a flag
- Three possible device tests:
  - *As part of the IO instruction*
  - *Explicitly by another IO instruction*
  - *On every instruction (as an interrupt)*



Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

November 2, 2001

### Testing as part of the IO instruction

- IO hardware implementation:
 

```
void read() {
    device_flag = 1;
    Read the next char;
    while(device_flag); // wait until flag is false
}
```
- CPU blocks during instruction execution
- Data is copied word-by-word
- With slow devices, this is extremely inefficient because CPU is stalled during IO operation



Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

November 2, 2001

### Explicit testing (polling)

- Allows computation to be overlapped with IO
- Special test-device instruction
- Hardware implementation
 

```
void read() {
    device_flag = 1;
    Read the next word;
}
```
- The *device\_flag* can be tested for IO completion
- Allows the CPU to overlap computation with IO
- But, program must be carefully structured



Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

November 2, 2001

### Interrupt-driven

- Hardware implementation is the same
 

```
void read() {
    device_flag = 1;
    Read the next word;
}
```
- Hardware *interrupts* CPU when IO is complete, which passes control to an *interrupt handler*
- For example, on a read operation
  - *Execute IO read instruction*
  - *Wait for interrupt*
  - *CPU may be used by other processes while waiting for an interrupt*



Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

November 2, 2001

### DMA interface

- Programmed IO with interrupts is suitable only when devices have low throughput (since all data must pass through the CPU)
- For fast, block-oriented devices, it is better to have hardware that explicitly accesses main memory
- In the general case, the CPU constructs an IO *channel program* to be executed by the device
- The CPU only initiates an IO operation

```
startio(channel, channel_prog);
```



Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

November 2, 2001

### DMA programs

- For read/write operations, the *channel\_prog* contains
  - *main storage address*
  - *amount of data to be copied*
  - *commands to select and activate the device*
- IO channel and CPU compete for memory cycles; normally channel has highest priority
- Simplified commands: *index* is the address of the physical block; *buf* is the location in main memory,

```
Read(ch, index, buf)
Write(ch, index, buf)
```




Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

November 2, 2001

### DMA operation

- The *startio* (*Read*, *Write*) commands immediately free the CPU
- The status of the channel may be interrogated by the CPU, or
- The device interrupts the CPU when IO is complete



Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

November 2, 2001


### DMA with channel polling

- Example: reading and processing sequence of blocks

```

Read(ch, 0, buf[0]);
for(int i = 1; i < len; i++) {
    while(busy(ch)); // wait for last IO
    Read(ch, i, buf[i]); // start next read
    Compute(buf[i - 1]); // process old buf
}
    
```

- Again, the code has to be structured to take advantage of overlaps



Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a


November 2, 2001

### DMA with channel interrupts

- Uses *bounded-buffer* data structure
  - int get\_buf()*: returns the index of a free buffer
  - void release\_buf(int id)*: frees the buffer
- Main program looks like:
 

```

while(true) {
    ...
    int index = get_buf();
    Compute on buf[index];
    release_buf(index);
    ...
}
            
```



Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

November 2, 2001

### DMA channel monitor

```

monitor DMA_channel {
    char buffer[N][BUF_SIZE];
    int dma_in, next_in, empty_count;
    Condition c;

    void init() {
        /* Start initial IO transfer */
        Read(ch, next_record++, buffer[0]);
        empty_count = N;
        dma_in = 0;
        next_get = 0;
    }


    void interrupt_handler() {
        dma_in = (dma_in + 1) mod N;
        empty_count--;
        c.signal;
        if(empty_count)
            Read(ch, next_record++, buffer[dma_in]);
    }

    int get_buf() {
        if(empty_count == N)
            c.wait;
        int index = next_in;
        next_in = (next_in + 1) mod N;
        return index;
    }

    void release_buf() {
        empty_count++;
        if(!busy(ch))
            Read(ch, next_record++, buffer[dma_in]);
    }
}
    
```

**L2**

**L1**




Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

November 2, 2001

### Interrupt-Driven IO

- In interrupt-driven IO, there is a clear distinction between levels L2 (Device IO) and L1 (Device scheduling and control)
- Typically, levels L1 and L2 are implemented as separate processes communicating through a shared buffer pool




Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

November 2, 2001

### Device drivers (level L1)

- receives IO requests, typically through a queue
- invoke appropriate allocators or schedulers for channels and controllers
- initiates IO operations
- handles general transmission errors
- processes interrupts
- sends completion messages back to the requesting process



Computing Systems  
http://www.cs.caltech.edu/cs134/cs134a

November 2, 2001

### Recovery from system failures

- Hardware and software failures can be reduced, but not eliminated, by careful design
- Failures may be *catastrophic*, or they may be localized
- An OS should be designed with tools for analyzing and recovering from errors
- Recovery should ideally be easy and transparent
- **Reinstalling the system is a bad way to recover from errors!**



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

### Hardware and system failures

- Hardware errors may be localized (an instant disk crash), or the failure may propagate to other components
- Systems and user programs usually have hidden errors
  - *Buffer overflows*
  - *Dangling pointers*
  - *Infinite loops*
- The OS can isolate most user programs
- In a protected OS, systems failures are most often caused by systems programs



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

### Effect of errors

- Destruction of systems tables, such as process queues and directories
- Addresses and pointers that do not point to valid items (such as pointers in file directories that do not refer to files)
- Incorrect resource lists (such as the list of free blocks in a filesystem)
- Reading or writing data from the wrong files



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

### Recovering from errors

- Consistency checks
  - *Following pointers in tables to flush out bogus pointers*
  - *Recovery from redundant information*
  - *Application of checksums*
- Systems dumps
  - *Full backups of system at periodic intervals*
  - *Incremental dumps to save more frequently updated files*



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

### Filesystem errors

- The filesystem may be inconsistent after a system crash
  - *Unix/Windows use a buffer cache to save most-recently-used file blocks in main memory, this buffer is periodically sync'ed with the disk*
  - *A system crash will omit updates to the buffer cache*
  - *A filesystem check may be able to recover*
- Possible solutions:
  - *Atomic disk updates*
  - *Log-based filesystems with transactional operations (but fast writes, slow reads)*



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001

### Last resorts

- Restore the system from the most recent dump
- If there are no backups, remember to do them in the future, and reinstall the system



Computing Systems  
<http://www.cs.caltech.edu/cs134/cs134a>

November 2, 2001