Input/Output, Files and Directories

- refs: [Tanenbaum, sects 5.5.7, 6.2 & 6.4.3]; [O’H&Bryant, ch 11]; [Null&Lobur, sect 7.1–7.4]

- issues

- low-level input/output (I/O)
  - programmed
  - interrupt-driven
  - direct memory access (DMA)

- virtual I/O
  - associated system calls; memory-mapped I/O
  - file, disk and directory organisation

- other issues:
  - review issues around Linux memory management
  - Assignment 2: feedback, solution up
  - Lab 9: challenges anyone?
Review: Memory Layout of Executable Programs

- given the code, assuming that \texttt{GetBits(pte, 4, 1) \gg 4} always = 0 or 1:
  ```
  int IsDirty(int pte) {
    char result[1];
    sprintf(result, "%d", GetBits(pte, 4, 1) \gg 4);
    ... }
  ```

  what is the ‘best’ length that should be given to \texttt{result[]}: (a) 2 (b) 8 (c) 32 (d) 256

  Consider ‘best’ from the point of view of maintainability & space or time efficiency.

- what are the 3 factors required for buffer overflow exploits?
  - why is the SPARC still vulnerable?

- given the PeANUt program:
  
x: block 1
main: ...
  end main

  what is the closest corresponding C program:

  (a) \texttt{int x; int main() \{...\}} (b) \texttt{int main() \{int x;...\}}
  (c) \texttt{static int x; int main() \{...\}} (d) \texttt{int main() \{static int x;...\}}
Issues in I/O

- device differences:
  - speed: from disk drive to keyboard
  - unit of transfer: characters, words, bytes, blocks or records
  - data representation: different encoding
  - allowed operations: rewind or no rewind
  - error conditions: detection and action

- design objectives:
  - efficient
  - character code independent
  - device independent: “any disc drive will do”
  - uniform device treatment: (as far as possible)

⇒ requires suitable abstractions!
Programmed I/O

● simplest form of I/O: one character is in-/output at a time

● to output:
  ■ wait until terminal device is ready
  ■ give character to terminal device

● to input:
  ■ wait until keyboard device is ready
  ■ get character from keyboard device
  ■ set ReadyToReceiveData flag

● these ‘tight’ loops are referred to as busy wait loops

● very wasteful of CPU resources (i.e. doesn’t allow other users to use CPU during I/O)

● PeANUt I/O (trap #2, trap #3) can be thought of as programmed I/O
Interrupt-Driven I/O

- slightly more sophisticated: one character I/O at a time
- to output:
  - request interrupt from terminal device when ready
  - do something ... (until interrupt is received)
  - give character to terminal device
- to input:
  - request interrupt from keyboard device when ready
  - do something ... (until interrupt is received)
  - get character from keyboard device
- good: can productively multiprocess while waiting
- bad: (frequent) process switching is expensive, so some time is still wasted
DMA (Direct Memory Access) I/O

- usually multiple characters (often KBs or MBs) are sent in sequence
- with DMA, a special purpose external processor (DMA controller) does the programmed I/O for us, and then tells us when it is finished
- number of interrupts is reduced dramatically over that of interrupt-driven I/O
- each transfer uses the bus ([O’H&Bryant, fig 1-4]); what if CPU needs to use it?
- example: output 4096 characters from address 1024 onwards to a disk:
  - send start address 1024 and the length 4096 to the DMA controller
  - in turn, sends a code (number) for the disk device (e.g. 4), and a write code
  - the DMA controller transfers the data from memory to the disk using programmed I/O
  - when finished, the DMA controller sends an interrupt signifying completion
- when directly communicating with a device (e.g. DMA controller), the CPU reads/writes to registers on that device:
  - this could be done via special instructions (drawbacks?)
  - or memory-mapped I/O: map these registers to special locations in main memory (thus memory can provide an abstraction to devices)
Responsibility: Virtual I/O

- If direct access to I/O devices was allowed at user-level:
  - inconvenient: requires users to have substantial knowledge of the devices
  - also when reading and writing, many different (correctable) errors may occur
  - arbitration problems: what if two users tried to access the disk simultaneously?
  - this would leave the machine without any data security!

- Instead, O/S level I/O instructions (traps) provide an abstraction to I/O:
  - users usually don’t want to know (just want a successful transfer)
  - hide much of this complexity from the user and provide a level of security

- The parts of the OS which access devices are called device drivers

- OS structure for virtual I/O:
Basic Input/Output Operation

USER

Requesting Process

Issue IO request
(scanf, printf etc)

Wait request complete
Check error condition

OPERATING SYSTEM

I/O Procedure

Identify device
Perform error check
Assemble details of IO request
Place IO request on device queue
Exit

Device Handler

Wait for requests
Initiate IO operation

Wait for complete
Error checks
Signal complete

Device

Begin operation

Operation complete

BEGIN

COMP2300 Lecture 04: OS Concepts: Storage Management 2009
File Organisation and Types

● files provide a means of organising and accessing data in a convenient manner

● a file is a linear sequence of 8 bit bytes indexed from 0 to some maximum (e.g. $2^{32} - 1$ or $2^{64} - 1$)

● associated with every open file is a pointer to the next byte to be read or written

● in UNIX, even devices such as printers and terminals are treated as (special) files, e.g. /dev/lp /dev/tty; also selected kernel data (/proc)
  ■ this enables these to be read/written to as files - a powerful abstraction

● file types:

  ■ sequential files:
    ◆ five basic operations: open, read, write, rewind, close
    ◆ read/write from/to next item in file
    ◆ rewind operation permits return to beginning of file
    ◆ access analogous to that of a magnetic tape

  ■ random access files:
    ◆ file position (pointer) can be explicitly set
Principal Unix File System Calls

- a system call is the HLL interface to the OS
  - thus, the system call function’s code executes a corresponding trap instruction
- supports both sequential and random access file types

<table>
<thead>
<tr>
<th>system call</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>creat(name, mode)</td>
<td>create a file; mode specifies the protection mode</td>
</tr>
<tr>
<td>unlink(name)</td>
<td>delete a file (assuming that there is only 1 link to it)</td>
</tr>
<tr>
<td>open(name, mode)</td>
<td>open or create a file and return a file descriptor</td>
</tr>
<tr>
<td>close(fd)</td>
<td>close a file</td>
</tr>
<tr>
<td>read(fd, buf, count)</td>
<td>read count bytes into buf</td>
</tr>
<tr>
<td>write(fd, buf, count)</td>
<td>write count bytes from buf</td>
</tr>
<tr>
<td>lseek(fd, offset, w)</td>
<td>move the file pointer as required by offset and w</td>
</tr>
<tr>
<td>stat(name, buf)</td>
<td>return information about a file</td>
</tr>
<tr>
<td>chmod(name, mode)</td>
<td>change the protection mode of a file</td>
</tr>
<tr>
<td>fcntl(fd, cmd, ...)</td>
<td>do various control operations such as locking (part of) a file</td>
</tr>
<tr>
<td>mmap(buf, count, mode, flags, fd, offset)</td>
<td>memory map VM pages from buf to buf + count–1 to pages in file fd from offset</td>
</tr>
</tbody>
</table>
Memory Mapping: Files in Virtual Memory

- instead of file access via `read()` / `write()`, we can directly map pages in a file into the virtual memory of a process
  - thus, reading and writing to page in a file can be achieved by normal (to the user program) memory accesses
  - if a page is written to, the corresponding page in file is updated when swapped out of VM system
  - note: in VM, the ‘master’ image of a page is on disk anyway

- in Unix, the link loader in fact does not copy the ELF image file sections into memory; it just sets up page mappings
  - when such a page is touched, a copy is made in physical memory
  - even the BSS section (uninitialized global data) is `mmap`’ed to an anonymous file (of zeroes)
  - if page is writable, a (VM) copy is made upon write (so the ELF file does not get modified)
  - (dynamically linked) pages of shared libraries are also `mmap`’ed

- again, provides a powerful abstraction
File Management: Directories

- Directories are just files with special properties
  - Are files that contain pointers to files
  - Allow information to be organised within the file system
  - Some early O/S did not support directories (e.g. MS-DOS)
- Directories are maintained by the operating system
- Must account for: creation, deletion, renaming, protection (access rights)
## Principal Unix Directory Management Calls

<table>
<thead>
<tr>
<th>system call</th>
<th>meaning:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir(name, mode)</td>
<td>create a new directory</td>
</tr>
<tr>
<td>rmdir(name)</td>
<td>delete an empty directory</td>
</tr>
<tr>
<td>opendir(name)</td>
<td>open a directory for reading</td>
</tr>
<tr>
<td>readdir(dirpointer)</td>
<td>read the next entry in a directory</td>
</tr>
<tr>
<td>closedir(dirpointer)</td>
<td>close a directory</td>
</tr>
<tr>
<td>chdir(dirname)</td>
<td>change working directory to dirname</td>
</tr>
<tr>
<td>link(name1, name2)</td>
<td>create a directory entry name2 pointing to name1</td>
</tr>
<tr>
<td>unlink(name)</td>
<td>remove name from its directory</td>
</tr>
</tbody>
</table>

aside: on Linux, the `strace` command can be used to trace system calls, e.g.

```
me@partch:~> strace touch f
execve("/usr/bin/touch", ["touch", "f"], [/* 72 vars */]) = 0
...
open("f", O_WRONLY|O_NONBLOCK|O_CREAT|O_NOCTTY|O_LARGEFILE, 0666) = 3
close(3) = 0
...
```
File and Directory Implementation (UNIX)

- Each file/directory is associated with a 64 byte block of information called an index-node or i-node. I-nodes:
  - Tell who owns the file, permissions, location of data etc
  - Are located in predefined locations in numerical sequence, i.e. so they can be easily located

- A directory is a file and an i-node number. On the call `open("foo.c", 0);`
  - The OS searches the working directory for file `foo.c` to find its i-node number
  - Reads i-node and accesses file

- Typical i-node contents:
  - File type (ordinary file, directory, special file, blocked or unblocked), the 9 RWX protection bits and a few other bits
  - Number of links to the file (directory entries)
  - Owners ID and group ID
  - File length (bytes)
  - Thirteen disk addresses
  - Last time file read / written / i-node changed
Disk Based I/O Devices

- recall that disks are organised into cylinders, tracks and sectors

- disk storage allocation strategy
  - files are partitioned into fixed-length units called blocks
  - blocks are allocated consecutive sectors where possible
  - fragmentation problems arise from variable file lengths and isolated unused blocks
  - the OS performs reads/writes to disk files in whole blocks

- block addressing scheme:
  - addresses of first 10 blocks are explicitly given in the i-node
  - i-node address 11 points to a disk block containing (block size/address size) further addresses (indirect block)
  - i-node address 12 points to a block containing pointers to blocks containing address (double indirection) which blocks are allocated to a file
  - i-node address 13 gives triple indirection!
  - with a 4KB block size, UNIX i-nodes can reference files up to 4 TB!
UNIX I-node Addressing Scheme
Typical OS Structure

- Scheduling and Resource Allocation
- File Access
- I/O
- Memory Management
- Wait and Signal
- FLIH
- Dispatcher