Operating Systems
Functions

Steven Hand
8 lectures for CST Ib and Diploma

*Lent Term 2000*

Handout 3
I/O Devices

• Typically several external 'devices' which interact with computer via I/O:
  1. Human readable: graphical displays, keyboard, mouse, printers
  3. Communications: line drivers, modems, network interfaces

• They differ significantly from one another with respect to
  1. Data rate – several orders of magnitude between keyboard and network
  2. Application – affects policy
  3. Complexity of control
  4. Unit of transfer
  5. Data representation
  6. Error handling
Devices

How are devices accessed by programs:

- OS deals with processor and devices:
  - I/O instructions v. memory mapped (where?)
  - I/O hardware type (e.g. 10's of serial chips)
  - polled v. interrupt driven
  - processor interrupt mechanism

- programs access virtual devices:
  - terminal streams not terminals
  - windows not frame buffer
  - event stream not raw mouse
  - files not disk blocks
  - printer spooler not parallel port
  - transport protocols not raw Ethernet

- virtual devices implemented:
  - in kernel, e.g. files, terminal devices
  - in demons, e.g. spooler, windowing
  - in libraries, e.g. terminal screen control, sockets
## Processor and Devices

Users must be prevented from accessing physical devices and associated data structures:

- **data protection**: e.g. passwords typed in on terminal
- **bozo programmer**: illegal sequence of actions to an I/O device could lock processor
- **multiplexing**: concurrent use of device must be properly controlled

### How:

- **trust**: all device actions dealt with by kernel.
- **multiplexing**: monitors, locks etc. in kernel
- **typical mechanisms**:
  - make I/O instructions only available in supervisor mode, and/or
  - make I/O devices only available in supervisor memory map (c/f memory management).
Polling

Continuously interrogate devices:

A mechanism of limited use:

- how does the ‘user’ get a look in
  — need to trust them.

- poor worst case response.

- how is priority done?

- but, very simple to program and can know worst case performance
  — in a terminal
  — real time control
  — dally in interrupt handler
Polling Example: Serial Output

#define COM1 0xF8  /* COM1 Port Address */
#define THR 0x0  /* Xmit Holding Reg Offset */
#define LSR 0x5  /* Line Status Reg Offset */
#define COM1_THR (COM1+THR)  /* THR Port Address */
#define COM1_LSR (COM1+LSR)  /* LSR Port Address */

... void serial_out(unsigned char c) {
    while(!(inb(COM1_LSR) & 0x20)); /* Wait 'til THR free */
    outb(c, COM1_THR); /* Put "c" into THR */
    while(!(inb(COM1_LSR) & 0x20)); /* Wait 'til "c" gone */
}

On x86, inb/outb are part of instruction set. On other architectures, need to provide mapping onto 'physical' addresses: e.g. on Alpha machines with alcor chipset:

#define ALCOR_IO 0x8580000000UL

unsigned int inb(unsigned long addr) {
    long result = *(volatile unsigned int *)
        ((addr << 5) + ALCOR_IO + 0x00);
    result >>= (addr & 3) * 8;
    return 0xffUL & result;
}
Interrupts

Polling poor $\Rightarrow$ most OSs use interrupts.

Most modern processors provide at least a basic interrupt mechanism:

- at end of each instruction, check interrupt line(s) for pending interrupt
- save program counter
- save processor status
- change processor mode
- jump to well known address (or its contents)

Some processors provide:

- multiple levels of interrupts
- hardware vectoring of interrupts
- mode dependent registers
Direct Memory Access

Can reduce interrupt overhead with DMA:

- get the device to read and write processor memory directly
- one interrupt at end of data transfer
- a generic DMA “command” might include:
  - source address
  - source increment / decrement / do nothing
  - sink address
  - sink increment / decrement / do nothing
  - transfer size
- DMA channels are often implemented on devices themselves:
  - e.g. a disk controller
  - pass disk address, memory address and size
  - give instruction to read or write
- Also get “stand-alone” programmable DMA controllers (e.g. PC-AT)
Interrupts: Implementation

Interrupt handler maps from h/w interrupts to ISR invocations. Handler may need to:

- save more registers
- demultiplex interrupt in software
- establish a language environment (e.g. a C runtime stack)

Interrupt Service Routines (ISRs):

- device, not processor, specific (unless asm!)
- for programmed I/O device:
  - transfer data
  - clear interrupt (sometimes a side effect of transfer)
- for DMA device:
  - acknowledge transfer
- request another transfer if any more I/O requests pending on device
- signal any waiting threads
- enter scheduler or return

Question: who is scheduling who?
Interrupt Handler Implementation

_do_irq:
  sub   r14, r14, #4  @ Fix up link register
  stmfdr13!, {r0-r6, r12, r14}

  @ first time through loop - pick up current ints.

  mov   r4, #0x32000000  @ IOC Base
  ldrb  r5, [r4, #0x14]  @ IRQ Request A
  ldrb  r6, [r4, #0x24]  @ IRQ Request B

do_irq_loop:
  tst   r6, #0x0a
  blne  _unexpected_hardware_signals
  tst   r5, #0x80
  beq   I_3

  mov   r0, #0x20
  strb  r0, [r4, #0x18]  @ disable this interrupt
  bl    _irq_atm_interface
Interrupt Handler Implementation

I_3:

tst r6, #0x20 @ eXpansion Cards are on Bit 5
blne _xcb_interrupt @ ARM Podule Bus

tst r6, #0xc0 @ kcart ints are SRx & STx bits 6, 7
blne _kbd_irq @ Keyboard IRQ

tst r6, #0x04 @ R6551 on bit 2 SLCI - pin IL2
blne _r6551_irq @ Serial line controller

tst r5, #0x20 @ timer 0 is bit 5
beq I_2
mov r0, #0x20
strb r0, [r4, #0x14] @ Clear timer

bl _inttimer @ Advance the clock
bl _clocksweep @ Any timers gone off?

I_2:

@ loop back in case there are more ints
1dbr r5, [r4, #0x14] @ IRQ Request A
1dbr r6, [r4, #0x24] @ IRQ Request B
orr r0, r5, r6
bne do_irq_loop @ something happening? go round
Interrupt Handler
Implementation

no_more_ints:
  ldr r12, _cur_thread       @ load current tcb
  cmp r12, #0                @ do we have a thread?
  ldmeqfd r13!, {r0-r6, r12, r15}^ @ nope - return

  ldr r0, I_RP                @ address of reschedule flag
  ldr r1, [r0]                @ load it
  mov r2, #0
  str r2, [r0]                @ clear it
  cmp r1, #0                  @ reschedule needed?
  ldmeqfd r13!, {r0-r6, r12, r15}^ @ nope - return

@ Regs are on stack.
  ldmfd r13!, {r0-r6}          @ previous r0 to r6
  stmea r12!, {r0-r11}         @ and save
  ldmfd r13!, {r0,r1}          @ previous r12 and r15
  stmea r12!, {r0, r13, r14}^ @ and save
  stmea r12!, {r1}             @ save prev r15

@ Irqs off not fiqs, Supervisor mode
  teqp r15, #SUPER_MODE|IBIT  @ 26-bit magic
  mov r0, r0                  @ nop
  stmea r12!, {r13, r14}      @ Supervisor r13 and r14
  mov r0, #7                  @ Setup "reason" and
  bl _scheduler               @ invoke scheduler
  b _do_brick_wall1

12
ISR Implementation

#define R6551_DATA ((volatile u_char *)0x33b0000) /* Data R/W */
#define R6551_STATUS ((volatile u_char *)0x33b0004) /* Status R */
#define R6551_CNTRL ((volatile u_char *)0x33b000c) /* Control R/W */
#define R6551_CMD ((volatile u_char *)0x33b0008) /* Command R/W */
#define R6551_RESET ((volatile u_char *)0x33b0004) /* Soft Reset W */

#define CMD_IRQ_INIT 0x0a /* all ints off, RTS_bar low */
#define CMD_IRQ_OFF 0x0b /* TX ints off, DTR_bar low */
#define CMD_IRQ_ON 0x07 /* TX ints on, DTR_bar low */

#define STATUS_IRQ_PEND 0x80 /* interrupt pending */
#define STATUS_TDRE 0x10 /* TDR empty => can send */
#define STATUS_IRQ_TDRE (STATUS_IRQ_PEND | STATUS_TDRE)

static char r6551_buf[R6551_BUFSIZE];
static int r6551_producer = 0;
static int r6551_consumer = 0;
static int r6551_freespace = R6551_BUFSIZE;

void r6551_irq()
{
    u_char c = *R6551_STATUS;
    if(!(c & STATUS_IRQ_TDRE)) return;

    /* need to send next data */
    if(r6551_producer != r6551_consumer) {
        *R6551_DATA = r6551_buf[r6551_consumer++];
        if(r6551_consumer == R6551_BUFSIZE)
            r6551_consumer = 0;
        r6551_freespace++;
    } else { /* no data to tx - disable the interrupt */
        *R6551_CMD = CMD_IRQ_OFF;
    }
}
I/O Buffering

- Important that a process waiting on I/O does not consume excess system resources
- CPU should reschedule and run another process
- To avoid difficulties of page management OS can use some form of *buffering*:
  1. Single buffering — OS assigns a system buffer to the user request
  2. Double buffering — process consumes from one buffer while system fills the next
  3. Circular buffers — most useful for burst-oriented I/O
- Many aspects of buffering dictated by device type:
  - character devices ⇒ line probably sufficient.
  - network devices ⇒ bursty (time & space).
  - block devices ⇒ lots of fixed size transfers.
  - (last usually major user of buffer memory)
I/O: Summary

- Messiest part of OS:
  - huge variety of devices.
  - large variety of device “classes”.

- Key design issues:
  1. Efficiency
     - Key performance issue is that of I/O buffering
     - Also important to schedule I/O to meet performance requirements of system
  2. Stability
     - Need to handle heavy I/O loads.
     - Decoupling ISR and device driver is good.
  3. Generality
     - Want to provide useful abstraction (e.g. Unix files)
     - But need to be careful don’t lose performance/functionality (e.g. direct access, asynchrony).
Disk I/O

- Performance of disk I/O is crucial to swapping/paging and file system operation

- Key parameters:
  1. Wait for controller and disk.
  2. Seek to the appropriate disk cylinder
  3. Rotational delay for the desired block to come under the head
  4. Data transfer

- Performance depends critically on **how the disk is organised**
Disk Scheduling

- In a typical multiprogramming environment have multiple users queueing for access to disk

- Also have VM system requests to load/swap/page processes/pages

- We want to provide best performance to all users — specifically reducing seek time component

- Several policies for scheduling a set of disk requests onto the device, e.g.
  1. FIFO: perform requests in their arrival order
  2. SSTF: if the disk controller knows where the head is (hope so!) then it can schedule the request with the shortest seek from the current position
  3. SCAN ("elevator algorithm"): relieves problem that an unlucky request could receive bad performance due to queue position
  4. C-SCAN: scan in one direction only
  5. N-step-SCAN and FSCAN: ensure that the disk head always moves
Reference String = 55, 58, 39, 18, 90, 160, 150, 38, 184

FIFO

SSTF

SCAN

C-SCAN
Other Disk Scheduling Issues

• Priority: usually beyond disk controller’s control.
  – System decides to prioritise, for example by ensuring that swaps get done before I/O.
  – Alternatively interactive processes might get greater priority over batch processes.
  – Or perhaps short requests given preference over larger ones.

• SRT disk scheduling (e.g. Cello, USD):
  – Per client/process scheduling parameters.
  – Two stage: admission, then queue.
  – Problem: overall performance?

• 2-D Scheduling:
  – Try to reduce rotational latency.
  – Typically require h/w support.

• Bad blocks:
  – Remapping typically transparent ⇒ can undo scheduling benefits.
Logical Volumes

Modern OSs tend to abstract away from physical disk; instead use *logical volume* concept.

- Partitions first step.

- Augment with “soft partitions”:
  - allow v. large number of partitions on one disk.
  - can customize, e.g. “real-time” volume.
  - aggregation: can make use of v. small partitions.

- Overall gives far more flexibility:
  - e.g. dynamic resizing of partitions
  - e.g. *striping* for performance.

- E.g. IRIX xlm, OSF/1 lvm, NT FtDisk.

- Other big opportunity is *reliability* ...
RAID

RAID = Redundant Arrays of Inexpensive Disks:

- Uses various combinations of striping and \textit{mirroring} to increase performance.
- Can implement (some levels) in h/w or s/w
- Many levels exist:
  - RAID0: striping over $n$ disks (so actually !R)
  - RAID1: simple mirroring, i.e. write $n$ copies of data to $n$ disks (where $n$ is 2 ;-).
  - RAID2: hamming ECC (for disks with no built-in error detection)
  - RAID3: stripe data on multiple disks and keep parity on a dedicated disk. Done at byte level $\Rightarrow$ need spindle-synchronised disks.
  - RAID4: same as RAID3, but block level.
  - RAID5: same as RAID4, but no dedicated parity disk (round robin instead).
- AutoRAID trades off RAIDs 3 and 5.
Disk Cacheing

- Cache holds copy of some of disk sectors.

- Can reduce access time by applications if the required data follows the locality principle.

- Design Issues
  - Transfer data by DMA or by shared memory?
  - Replacement strategy: LRU, LFU, etc.
  - Reading ahead: e.g. track based.
  - Write through or write back?
  - Partitioning? (USD ...)

- Typically O/S also provides a cache in s/w:
  - May be done per volume, or overall.
  - Also get *unified* caches — treat VM and FS caching as part of the same thing.

- Software caching issues:
  - Should we treat all filesystems the same?
  - Do applications know better?
4.3 BSD Unix Buffer Cache

- Name? well *buffers* data to/from disk, and *caches* recently used information.

- Modern Unix deals with *logical* blocks, i.e. FS block within a given partition / logical volume.

- “Typically” prevents 85% of implied disk transfers.

- Implemented as a hash table:
  - Hash on (devno, blockno) to see if present.
  - Linked list used for collisions.

- Also have *LRU* list (for replacement).

- Internal interface:
  - bread(): get data & lock buffer.
  - brelse(): unlock buffer (clean).
  - bdwrite(): mark buffer dirty (lazy write).
  - bawrite(): asynchronous write.
  - bwrite(): synchronous write.

- Uses *sync* every 30 secs for consistency.

- Limited prefetching (read-ahead).
NT Cache Manager

- NT Cache Manager caches “virtual blocks”:
  - viz. keeps track of cache “lines” as offsets within a file rather than a volume.
  - disk layout & volume concept abstracted away.

- Completely unified cache:
  - cache “lines” all in virtual address space.
  - decouples physical & virtual cache systems: e.g. virtually cache in 256K blocks, physically cluster up to 64Kb.
  - NT virtual memory manager responsible for actually doing I/O.
  - allows lots of FS cache when VM system lightly loaded, little when system is thrashing.
  - is this good?

- NT also provides some user control:
  - if specify temporary attrib when creating file ⇒ will never be flushed to disk unless necessary.
  - if specify write_through attrib when opening a file ⇒ all writes will synchronously complete.
File systems

What is a filing system:

- Directory service, provides
  - naming mechanism
  - access control
  - existence control
  - concurrency control

- Storage service, provides
  - integrity, data needs to survive:
    * hardware errors
    * OS errors
    * user errors
  - archiving
  - mechanism to implement directory service

What is a file?

- an ordered sequence of bytes (UNIX)
- an ordered sequence of records (ISO FTAM)
File Mapping Algorithms

How is a file mapped to blocks:

1. chaining in the material
2. chaining in a map
3. table of pointers to blocks
4. extents

Aspects to consider:

- integrity checking after crash
- automatic recovery after crash
- efficiency for different access patterns
  - of data structure itself
  - of IO operations to access it
- ability to extend files
- efficiency at high utilization of disk capacity
Chaining in the Media

Each disk block has pointer to next block in file. Can also chain free blocks.

- OK for sequential access – poor for random access
- cost to find disk address of block $n$ in a file:
  - best case: $n$ disk reads
  - worst case: $n$ disk reads
- Some problems:
  - not all of file block is file info
  - integrity check tedious …
Chaining in a map

Maintain the chains of pointers in a map (in memory), mirroring disk structure.

- disk blocks only contain file information
- integrity check easy: only need to check map
- handling of map is critical for
  - performance: must cache bulk of it.
  - reliability: must replicate on disk.
- cost to find disk address of block \( n \) in a file:
  - best case: \( n \) memory reads
  - worst case: \( n \) disk reads
Table of pointers

- access cost to find block $n$ in a file
  - best case: 1 memory read
  - worst case: 1 disk read
- integrity check easy: only need to check tables
- free blocks managed independently (e.g. bitmap)
- table may get large $\Rightarrow$ must chain tables, or build a tree of tables (e.g. UNIX inode)
- access cost for chain of tables? for hierarchy?
Extent lists

Use of contiguous blocks can increase performance …

- list of disk addresses and lengths (extents)

- access cost: [perhaps] a disk read and then a searching problem, $O(\log(\text{number of extents}))$

- can use bitmap to manage free space (e.g. QNX)

- system may have some maximum \#extents
  - could copy file (i.e. compact into one extent)
  - or could chain tables or use a hierarchy as for table of pointers.
File meta-data I

What information is held about a file?

- times: creation, access and change?
- access control: who and how
- location of file data (see above)
- backup or archive information
- concurrency control

What is the name of a file?

- simple system: only name for file is human comprehensible text name
- perhaps want multiple text names for file
  - soft (symbolic) link: text name → text name
  - hard link: text name → file id
  - if we have hard links, must have reference counts on files

Together with the data structure describing the disk blocks, this information is known as the file *meta-data*. 
File meta-data II

Where is file information kept:

- no hard links: keep it in the directory structure.

- hard links, keep file info separate from directory entries
  - file info in a block: OK if blocks small (e.g. TOPS10)
  - or in a table (UNIX i-node / v-node table)

- on OPEN, (after access check) copy info into memory for fast access

- on CLOSE, write updated file data and meta-data to disk

How do we handle caching meta-data?
Directory Name Space

• simplest - flat name space (e.g. Univac Exec 8)

• two level (e.g. CTSS, TOPS10)

• general hierarchy
  – a tree,
  – a directed (acyclic) graph (DAG)

• structure of name space often reflects data structures used to implement it
  – hierarchical name space ↔ hierarchical data structures
  – but, could have hierarchical name space and huge hash table!

General hierarchy:

• reflects structure of organisation, users' files etc.

• name is full path name, but can get rather long:
  e.g.  /usr/groups/X11R5/src/mit/server/os/4.2bsd/utils.c
  – offer relative naming
  – login directory
  – current working directory
Directory Implementation

- directories often don’t get very large (especially if access control is at the directory level rather than on individual files)
  ✔ often quick look up
  ❌ directories may be small compared to allocation unit

- But: assuming small dirs means lookup is naïve ⇒ trouble if get big dirs:
  – optimise for iteration.
  – keep entries sorted (e.g. use a B-Tree).

- Query based access:
  – Split filesystem into system and user.
  – User wishes ‘easy’ retrieval.
  – What about access control?
Immutable files

Do away with concurrency problems — use write once files with atomic close! Implemented by:

- copy on write
- multiple version numbers: foo!11, foo!12
- invent new version number on close (i.e. sequence all close operations)

Problems:

- disk space
  - only keep last \( k \) versions (archive rest?)
  - have a explicit keep call
  - share disk blocks between different versions — complicated file system structures

- name without version usually means ‘latest’ — ambiguous

- and the killer ... directories aren’t immutable!
But:

• concurrency control only required on version number

• could be used (for files) on unusual types of media
  – write once optical disks
  – erasable disks
  – remote servers (e.g. Cedar FS)

• provides an audit trail
  – required by the spooks
  – often implemented on top of normal file system; e.g. UNIX RCS

• coming back into vogue (e.g. Elephant)
**Multi-level stores**

Archiving (c.f. backup); keep frequently used files on fast media, migrate others to slower (cheaper) media. Can be done by:

- user – *encouraged* by accounting penalties
- system – migrate files by periodically looking at time of last use
- can provide transparent naming but not performance!

Integrate multi-level store and ideas from immutable files, e.g. Plan-9:

- file servers with fast disks
- write once optical juke box
- every night, mark all files immutable
- start migration of files which changed the previous day to optical disk
- access to archive explicit
  e.g. `/archive/12Jan2000/users/smh/ ...`
**Integrity: Backups**

Backup; keep (recent) copy of whole file system to allow recovery from:

- CPU software crash
- bad blocks on disk
- disk head crash
- fire, war, famine, pestilence

What is a *recent* copy:

- in real time systems (e.g. airline booking) recent means mirrored disks
- daily usually sufficient for ’normal’ machines

Can use *incremental* technique, e.g.

- full dump performed daily or weekly
  - copy whole file system to another disk or tape
  - could take hours [esp. if copy across a network]
  - best done while file system live (although can give us consistency problems).
• incremental dump performed hourly or daily
  
  – only copy files and directories that have changed since the last time.
  
  – can either mark files explicitly (perhaps at log out), or use last modification time in file meta-data.

• e.g. 3-level scheme

![Size of Dump](image)

- to recover:
  
  – first restore full dump,
  
  – then add in incrementals.
Ruler Function

- Want to minimise \#tapes needed, time to backup
- Want to maximise the time a file is held on backup
  - Number days starting at 1
  - On day $n$ use tape $t$ such that $2^t$ is highest power of 2 which divides $n$
  - whenever we use tape $t$, dump all files modified since we last used that tape, or any tape with a higher number
- If file is created on day $c$ and deleted on day $d$ a dump version will be saved substantially after $d$
- the length of time it is saved depends on $d - c$ and the exact values of $c, d$
**Integrity: Processor crash**

If the processor terminates unexpectedly – OS bug, power failure – the main problem is that modified data structures exist in memory and have not been completely written to disk.

- most failures affect only files being modified

- as disk is still intact, can usually recover a more recent version of file system from its state than from backup

- at start up after a crash run a *disk scavenger*
  - try to recover data structures from memory (bring back core memory!)
  - get current state of data structures from disk
  - identify inconsistencies (may require operator intervention)
  - isolate suspect files and reload from backup
  - correct data structures and update disk

- usually much faster and better (i.e. more recent) than recovery from backup.
• can make scavenger’s job simpler:
  – replicate vital data structures
  – *spread* replicas around the disk
  – provide redundancy in data structures for consistency check

• even better: use *journal* [or *log*] file to assist with recovery.
  – ensure log records always written prior to actual modification.
  – allows very fast recovery after a crash (e.g. a few seconds).
  – e.g. NTFS, XFS.
Log-Structured File Systems

Radically different file system design:

- Premise 1: CPUs getting faster faster than disks.
- Premise 2: memory cheap $\Rightarrow$ large disk caches
- Premise 3: large cache $\Rightarrow$ most disk reads “free”.
  $\Rightarrow$ performance bottleneck is writing & seeking.

Basic idea: solve write/seek problems by using a log:

- log is [logically] an append-only piece of storage comprising a set of records.
- all data & meta-data updates written to log.
- periodically flush entire log to disk in a single contiguous transfer:
  - high bandwidth transfer.
  - can make blocks of a file contiguous on disk.
- have two logs $\Rightarrow$ one in use, one being written.

What are the problems here?
1. How do we find data in the log?
   - can keep basic UNIX structure (inodes, indirect blocks, etc)
   - then just need to find a file’s inode ⇒ use *inode map*
   - inode maps live in fixed region on disk.

2. What do we do when the disk is full?
   - need asynchronous *scavenger* to run over old logs and free up some space.
   - two basic alternatives:
     1. compact live information to free up space.
     2. thread log through free space.
   - Neither great ⇒ use *segmented log*:
     - divide disk into large fixed-size segments.
     - compact within a segment, thread between segments.
     - when writing use only clean segments
     - occasionally clean segments
     - choosing segments to clean is hard …

Log-structured file systems are the subject of ongoing debate in the OS community …