Betriebssysteme 12/13
Input and Output

27. November 2013
Overview

- handling I/O one of the main tasks of an operating system
  - controlling I/O
  - handling interrupts
  - handling errors
- all in a device independent way
- providing simple, unified interfaces to the applications
Types of Hardware

- Different types of devices
  - programmers view
  - related to mode of operation of devices
- block devices
- character devices
- other devices
- generic model keeping parts of the operating system device independent
character devices

- handles character streams without block structure
- cannot be read/written independently
- no block structure
- examples: keyboard, mouse, printer, network
block devices

- stores information in blocks of fixed size
- can be read/written independently
- examples: hard disks, SSD, DVD
- can also be handled as block devices
other devices

- clock
- screen (frame buffer)
- ...
Controller

- I/O-Devices consist of
  - „mechanical“ component (the device)
  - electronic component (controller)

- Controller
  - controls device
  - converts data streams
Example: Hard disk

- Disk delivers
  - preamble (cylinder #, sector #, sector size, synchronization information, ...)
  - 4096 bits in a sector
  - error correcting code (ECC)

- Controller
  - gets data bit by bit
  - converts data to block of bytes
  - does error correction (if necessary)
  - copies data into memory
Communication with controller

- Using device registers
- Write to registers:
  - issue command (deliver data, accept data, turn off, ...)
  - send data
- Read from registers
  - check status
  - get data
Two ways: (1) I/O Ports

<table>
<thead>
<tr>
<th>I/O commands using I/O ports</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IN</strong> REG, PORT</td>
</tr>
<tr>
<td><strong>OUT</strong> PORT, REG</td>
</tr>
</tbody>
</table>

separated data and I/O addressspaces

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IN</strong> REG, 4</td>
</tr>
</tbody>
</table>

... is something different than ...

**MOV** REG, 4
Two ways: (2) Shared Memory

- Map Control Registers into Main Memory
- Use standard commands addressing memory
- Memory Mapped I/O
- Addresses in upper memory region
- Some systems support both versions (Ports and memory) like Pentium
Shared Memory

Advantages

- no special I/O commands, no need for Assembly - you can program in e.g., C or C++
- control can be realized by mapping virtual pages into any process
- allows to realize device drivers as separate processes
- can make use of more sophisticated memory-based instructions - shorter and more efficient code possible
Shared Memory

Disadvantages

- selectively disable caching for these addresses
  - more complex hardware necessary
  - more complexity in OS - must manage caching
- devices and memory must check all memory references
  - sometimes a faster bus for memory - addresses never reach devices
  - solvable with special hardware support to filter addresses (on PCs: northbridge / southbridge)
Getting data byte by byte is too much effort

So we make use of DMA: Direct Memory Access

DMA-Controller moves data to memory without CPU support

Separate Controller or part of the device controller
DMA

1. CPU programs the DMA controller

2. DMA requests transfer to memory

3. Data transferred

4. Ack

Interrupt when done

Bus

Drive

Main memory

CPU

DMA controller

Address

Count

Control

Disk controller

Buffer
DMA controller gets data words from device
buffers data
(needs larger buffer since data may continuously arrive while waiting on bus)
needs bus for transfer, wait for bus
gets bus - now processor must wait for bus
to modes:
cycle stealing
  one transfer only
burst mode
  blocks of data transferred, more efficient but blocks bus longer
Goals of I/O Software

device independence

- programs should be able to access any I/O-device without having to specify the device in advance
  - e.g. read file
    - same code for files on hard disk, DVD, network etc.
- also use files and devices as input and output, e.g.
  sort < input > output
Goals of I/O Software

- uniform naming
  - closely related to device independence
  - name independent from device
  - in Unix: everything somewhere under `/`
    - other devices mounted into `/`
    - raw devices accessible at `/dev`
Goals of I/O Software

- also closely related to device independence
- error handling ideal done by controller or the operating system
- but not the application
- transient errors should be transparent
Goals of I/O Software

- **more of operation**
  - synchronous (blocking) I/O versus asynchronous (interrupt driven) I/O
  - physical I/O typically asynchronous
    - CPU starts transfer
    - does something else
    - interrupt arrives
    - CPU takes care of interrupt
  - most applications want sync. I/O
    - read syscall blocks process until data available
  - Task of OS to make this happen
Goals of I/O Software

buffering

- related to synchronous I/O
- data that arrives cannot necessarily be stored where the application needs it
  - read system call did not ask for all the bytes that arrived
  - no read system call has arrived yet
- for certain devices data to be output must be readily available (like audio)
- decoupling of buffer filling and emptying necessary
- lot of copying: may impact performance
How to do I/O

Three methods:

- programmed I/O
- interrupt driven I/O
- DMA
Example: process wants to send characters to printer

- syscall to open printer
- if printer used by other process: syscall fails. Application needs to do error handling
- finally process gets the printer device
- User process now makes syscall sending „ABCDEFGH“ to printer
Programmed I/O
for (i=0; i<count; i++) {
  while (*printer_status_reg != READY);
  *printer_data_register = p[i];
}
return_to_user();
Programmed I/O

- Drawback: CPU is busy until printing done
- fine, if
  - waiting time between two characters very short
  - processor has nothing else to do (e.g. on an embedded device)
- in more complex systems not acceptable
Interrupt driven I/O

- let’s do something meaningful between two characters
- so
  - first character to device
  - then call scheduler
  - printer to send interrupt when ready for next character
  - interrupt service routine responsible for providing that
Interrupt driven I/O

copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY);
*printer_data_register = p[0];
next_character = 1;
scheduler();
interrupt_handler:

if (count == 0) {
    unblock_user();
} else {
    *printer_data_register = p[next_character];
    count--;
    next_character++;
}

acknowledge_interrupt();
return_from_interrupt();
disadvantage of interrupts: cost time

DMA is an alternative

only one interrupt per transfer

be aware: DMA uses physical addresses

so OS must transform virtual addresses before setting up a transfer
I/O Software Layers

- User-level I/O software
- Device-independent operating system software
- Device drivers
- Interrupt handlers
- Hardware
Interrupt Handlers

- Interrupts not easy to handle, but necessary
- Ideally, driver starts I/O-operation and blocks until I/O completed
- Can be managed using semaphores
- Interrupt handler: does what’s necessary and unblocks driver
- Good if driver are kernel processes or threads
- In reality, it’s a little more complex
- Let’s look at the steps involved
  - Attention: these are system dependent things!
Handling an interrupt

1. save any registers that have not been saved by the hardware
Handling an interrupt

1. save any registers that have not been saved by the hardware
2. if necessary setup context for the interrupt service procedure (flush TLB, setup paging etc.)
Handling an interrupt

1. save any registers that have not been saved by the hardware
2. if necessary setup context for the interrupt service procedure (flush TLB, setup paging etc.)
3. set up a stack for handling the interrupt
Handling an interrupt

1. save any registers that have not been saved by the hardware
2. if necessary setup context for the interrupt service procedure (flush TLB, setup paging etc.)
3. set up a stack for handling the interrupt
4. acknowledge interrupt controller and/or reenable interrupts (this may cause an immediate interrupt!)
Handling an interrupt

1. save any registers that have not been saved by the hardware
2. if necessary setup context for the interrupt service procedure (flush TLB, setup paging etc.)
3. set up a stack for handling the interrupt
4. acknowledge interrupt controller and/or reenable interrupts (this may cause an immediate interrupt!)
5. copy the registers from where they were saved to the process table
Handling an interrupt

1. save any registers that have not been saved by the hardware
2. if necessary setup context for the interrupt service procedure (flush TLB, setup paging etc.)
3. set up a stack for handling the interrupt
4. acknowledge interrupt controller and/or reenable interrupts (this may cause an immediate interrupt!)
5. copy the registers from where they were saved to the process table
6. run the interrupt service procedure
Handling an interrupt

1. save any registers that have not been saved by the hardware
2. if necessary setup context for the interrupt service procedure (flush TLB, setup paging etc.)
3. set up a stack for handling the interrupt
4. acknowledge interrupt controller and/or reenable interrupts (this may cause an immediate interrupt!)
5. copy the registers from where they were saved to the process table
6. run the interrupt service procedure
7. run the scheduler - interrupt may have caused a high priority process to get unblocked
Handling an interrupt

1. save any registers that have not been saved by the hardware
2. if necessary setup context for the interrupt service procedure (flush TLB, setup paging etc.)
3. set up a stack for handling the interrupt
4. acknowledge interrupt controller and/or reenable interrupts (this may cause an immediate interrupt!)
5. copy the registers from where they were saved to the process table
6. run the interrupt service procedure
7. run the scheduler - interrupt may have caused a high priority process to get unblocked
8. set up context for selected process
Handling an interrupt

1. save any registers that have not been saved by the hardware
2. if necessary setup context for the interrupt service procedure (flush TLB, setup paging etc.)
3. set up a stack for handling the interrupt
4. acknowledge interrupt controller and/or reenable interrupts (this may cause an immediate interrupt!)
5. copy the registers from where they were saved to the process table
6. run the interrupt service procedure
7. run the scheduler - interrupt may have caused a high priority process to get unblocked
8. set up context for selected process
9. load the new process’s registers
Handling an interrupt

1. save any registers that have not been saved by the hardware
2. if necessary setup context for the interrupt service procedure (flush TLB, setup paging etc.)
3. set up a stack for handling the interrupt
4. acknowledge interrupt controller and/or reenable interrupts (this may cause an immediate interrupt!)
5. copy the registers from where they were saved to the process table
6. run the interrupt service procedure
7. run the scheduler - interrupt may have caused a high priority process to get unblocked
8. set up context for selected process
9. load the new process’s registers
10. start running new process
Handling an interrupt

1. save any registers that have not been saved by the hardware
2. if necessary setup context for the interrupt service procedure (flush TLB, setup paging etc.)
3. set up a stack for handling the interrupt
4. acknowledge interrupt controller and/or reenable interrupts (this may cause an immediate interrupt!)
5. copy the registers from where they were saved to the process table
6. run the interrupt service procedure
7. run the scheduler - interrupt may have caused a high priority process to get unblocked
8. set up context for selected process
9. load the new process’s registers
10. start running new process
Device Drivers

- remember: device controller has device registers that can be used to control device
- every device is different
- we need a special driver for each device or class of related devices
- must be able to access hardware: typically part of the kernel
  - buggy driver: big problem!
- possible to have drivers in user space, needs syscall to read/write device registers (or map them into address space of driver)
So device drivers are part of the kernel...

... but written by „outsiders“

requires well-defined model and interfaces

Often two types of interfaces:

- interface for block device drivers
- interface for character device drivers
Adding devices

- once upon a time...
- when you added a new device you had to recompile the kernel
- not a good idea for mum-and-dad-type users
- so dynamically loadable drivers have been developed
Tasks of a driver

- initialize device
- handle read and write requests
  - checking parameters, availability of device etc.
- energy management
- event handling
  - interrupts, errors, device removal, new device added
while a driver does his task, an interrupt may occur
interrupt may be caused by the device the driver is managing
interrupt handling will involve device driver
so: driver is interrupted by himself, somehow
consequence: driver must be reentrant, i.e. he must expect
that and ensure that this causes no problems.
Device independent I/O software

- some of I/O-software is device specific
- some things can be done device independently:
  - uniform interfacing
  - buffering
  - error reporting
Uniform interfacing

- make all I/O-devices and drivers look more or less the same
- don’t want to change the OS for every new device
- interface driver / OS one aspect
- uniform interfaces allows easy integration of new drivers
Interfacing

(a) Operating system

Disk driver  Printer driver  Keyboard driver

(b) Operating system

Disk driver  Printer driver  Keyboard driver
Device Naming

- device independent software maps symbolic device names to driver (e.g. /dev/sda)
- specifies inode for a special file
- inode contains
  - major device number
    - locates driver
  - minor device number
    - passed as parameter to driver
Hard disks

- Important aspect: fault tolerance
- required on file system level: robust against sudden power failure
- required on hardware level: robust against hardware failure
RAID

- Redundant Array of Independent (Inexpensive) Disks (Patterson 1988)
- combines disks, looks like a big disk to the system
RAID 0

- Goal: Speed
- partition disk into n strips - striping
- good for large requests, parallel processing makes transfers faster
- not good for single block requests
- reliability problem: any disk failing causes problems for all files
RAID 1

- Goal: Reliability
- duplicate disks
- excellent fault tolerance
- write performance as for single drive
- read performance better if both drives used
RAID 4

- Parity strip contains XOR of multiple strips on other disks
- One disk fails: content can be recalculated
- Improves security, no performance improvement
- Parity has to be recalculated at every write
  - Requires reading of old data and old parity
RAID 5

- as RAID 4, but removes the bottleneck „parity disk“
- spreads parity info over all disks
- RAID 6: uses two parity stripes, allows two disks to fail
Other RAIDs

RAID 0+1

RAID 0

A1
A3
A5
A7

A2
A4
A6
A8

RAID 1

A1
A3
A5
A7

A2
A4
A6
A8
Other RAIDs

RAID 30

RAID 0

RAID 3

RAID 3

A1 A3 A5 A7 B1 B3 B5 B7 P1 P3 P5 P7 A2 A4 A6 A8 B2 B4 B6 B8 P2 P4 P6 P8
Other RAIDs

Diagram showing RAID configurations with disks labeled Disk 0 to Disk 8, each with 120GB capacity.
Stable Storage

- Hard disks can break - RAID helps in principle
- Backups help against data loss
- but neither backups nor RAID protect from crashes when writing on DISK - inconsistency
- Sometimes important not to lose any data.
- Stable Storage: writes are performed correctly - or not at all.
Stable Storage - Assumptions

- disk writes a block: either
  - write is correct or
  - write is incorrect and this error can be detected
- correctly written sector may become corrupt spontaneously
  - same sector can only go bad on a second drive after a reasonable interval (1 day)
- CPU may fail at any time (potentially leading to disk errors)
Stable Storage

- Now we can make the storage 100% reliable, using two identical disks
- Two corresponding blocks form an error-free block
- No errors: corresponding blocks are identical
- Three operations:
  - Stable write
  - Stable read
  - Crash recovery
Stable Write

- write block to drive one
- read block to check it was written correctly
  - if not, repeat up to \( n \) times, then block is marked bad and different block used
- write block to drive two
- read block to check it was written correctly
  - if not, repeat up to \( n \) times, then block is marked bad and different block used
Stable Read

- Read data from drive one
- Check ECC - if incorrect, read again
  - try n times, then read from drive two
- following our assumptions, stable read always succeeds
Crash Recovery

- scan both disks
- read corresponding blocks
  - if equivalent, content is fine
  - if one has an ECC error, write other block on this block
  - if both are good, but different: write block from disk one onto disk two
CPU Crash

(a) Disk 1 Old
    Disk 2 Old
    Crash

(b) Disk 1 Old
    Disk 2 Diagonally striped
    Crash

(c) Disk 1 New
    Disk 2 Old
    Crash

(d) Disk 1 New
    Disk 2 Diagonally striped
    Crash

(e) Disk 1 New
    Disk 2 New
    Crash