Systems 3

Input/Output

Marcel Waldvogel

Department of Computer and Information Science
University of Konstanz

Winter 2019/2020

Chapter Goals

- What are some typical I/O requirements (rates, latencies)?
- How is I/O achieved? How has it improved over time?
- What is the motivation for device drivers?
- What is their interface/interaction with the OS?
- What is their operation?
- Explain the delays related to hard disk I/O.

I/O Devices

Device	Data rate
Keyboard	10 B/s
Mouse	100 B/s
52x CD-ROM	8 MB/s
USB 2.0	60 MB/s
Gigabit Ethernet	125 MB/s
SATA hard disk	100...200 MB/s
SATA 6G bus	480 MB/s
PCI bus	528 MB/s
USB 3.0	625 MB/s
PCIe 4.0 x4	8 GB/s
PCIe 6.0 x16	128 GB/s

Table: Some typical device, network, and bus data rates.

Simple Bus

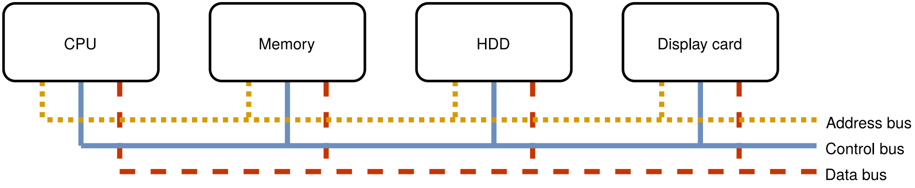


Figure: Simple system bus.

Device Controllers

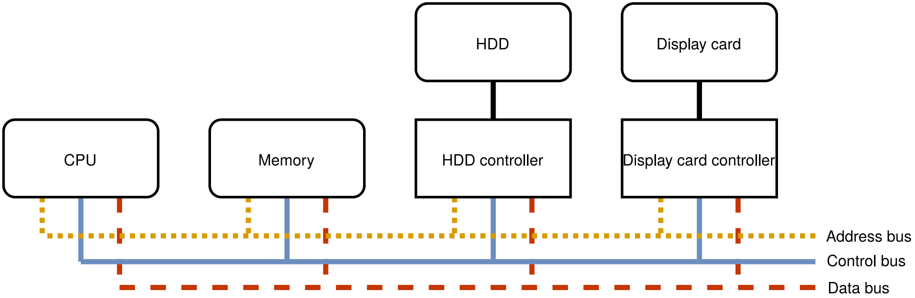
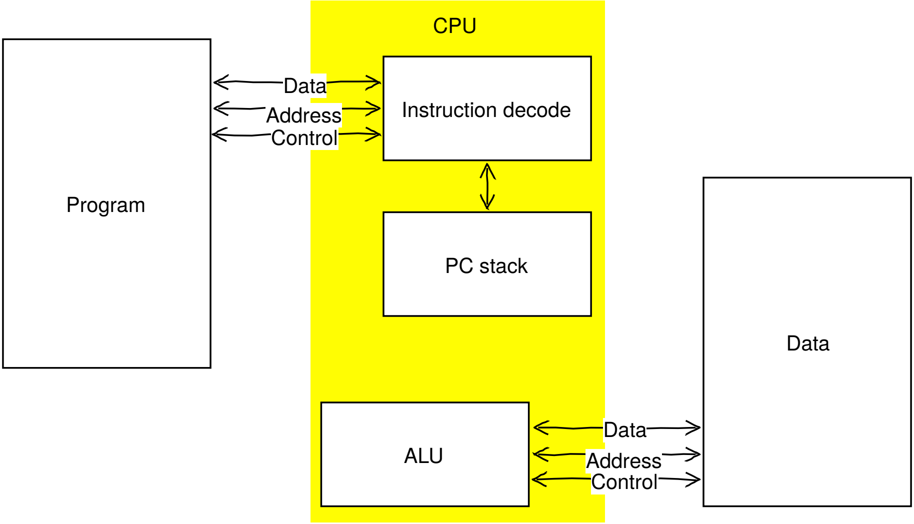


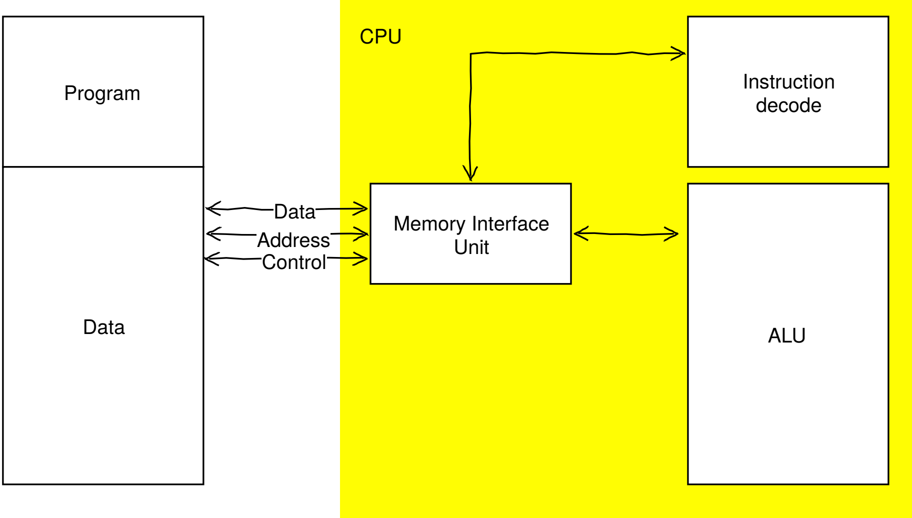
Figure: Simple system bus with device controllers.

Today's computer buses are more hierarchical. See e.g. AMD Zen architecture.

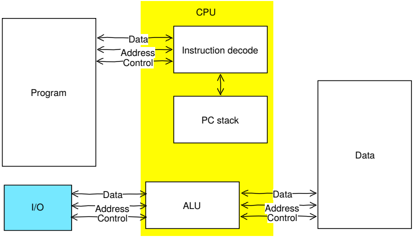
Recapitulation: Harvard architecture



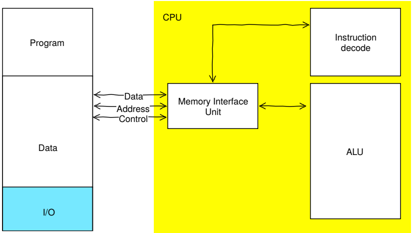
Recapitulation: Von-Neumann architecture



How to access I/O?



Harvard-style I/O: Separate I/O address space (and bus)



Von-Neumann-style I/O: Single I/O address space (and bus)

(Many hybrid forms exist as well)

MIPS architecture

Register	Assembly name	Comment
r0	$zero	Always zero
r1	$at	**A**ssembler **T**emp: Reserved for assembler
r2-r3	$v0-$v1	**V**alue
r4-r7	$a0-$a3	Function call **A**rguments
r8-r15	$t0-$t7	**T**emporary values (not saved)
r16-r23	$s0-$s7	**S**aved values
r24-r25	$t8-$t9	**T**emporary values (not saved)
r26-r27	$k0-$k1	Reserved for OS **K**ernel
r28	$gp	**G**lobal **P**ointer
r29	$sp	**S**tack **P**ointer
r30	$fp	**F**rame **P**ointer
r31	$ra	**R**eturn **A**ddress

MIPS function call register lifetime

Register	Function entry	Function exit	Saved by
$v0-$v1	Undefined	Return value or undefined	caller
$a0-$a3	Arguments or undefined	clobbered	caller
$t0-$t9	Undefined	clobbered	caller
$s0-$s7	Undefined	unmodified	call**e**e

```
float log_base(float x, float b)
{
    /* Save $s0-$s7 if used in this function */
    float lx, lb, lr;
    /* Save $a1 and any of $t0-$t9 which should survive */
    lx = log(x);
    /* Restore $a1, $t0-$t9 */
    /* Save $t0-$t9, if needed */
    lb = log(b);
    /* Restore $t0-$t9 */
    lr = lx / lb;
    /* Restore $s0-$s7 if they were used in this function */
    return lr;  /* Fill $v0 */
}
```

Someone's gotta save[1] (massively simplified MIPS-32)

```
float log_base(float x, float b)
{ float lx, lb, lr;   lx = log(x);   lb = log(b);   lr = lx / lb;   return lr; }
```

```
log_base:
    add     $sp,$sp,-12      /* Reserve space on stack */

    st      $ra,8($sp)       /* Save $ra, $a1 */
    st      $a1,4($sp)
    jal     log              /* Call log() with $a0 to $v0; saves $pc in $ra */

    st      $v0,0($sp)       /* Save $v0 */
    ld      $a0,4($sp)       /* Restore original $a1 as $a0 */
    jal     log              /* Call log with $a0 to $v0; saves $pc in $ra */

    ld      $t0,0($sp)       /* Restore saved $v0 as $t0 */
    div.s   $v0,$t0,$v0      /* $v0 = $t0 / $v0 */

    ld      $ra,8($sp)       /* Restore $ra */
    add     $sp,$sp,12       /* Free space on stack */
    j       $ra              /* Restores $pc from $ra */
```

[1] With more local variables, we would need to save/restore $s0-$s7 at entry/exit

I/O for single-process machines

Programmed I/O (polling or delay loops)

```
/* Disk I/O; diskdevicespace points to I/O address of disk controller */
diskdevicespace->sector  = s;
diskdevicespace->track   = t;
diskdevicespace->head    = h;
diskdevicespace->command = DISK_READ;
while (!(diskdevicespace->status & DISK_OP_COMPLETE)) {
   while (!(diskdevicespace->status & DISK_BYTE_READY)) {
      /* Busy waiting */
   }
   *dst++ = diskdevicespace->data;
}
switch (diskdevicespace->status & DISK_ERROR_MASK) {
case OK:
   ...
}
```

I/O for multi-process machines: Slave device

Interrupt+PIO

```
/* Disk I/O; diskdevicespace points to I/O address of disk controller */
diskdevicespace->sector  = s;
diskdevicespace->track   = t;
diskdevicespace->head    = h;
diskdevicespace->command = DISK_READ;
switch_to_other_process();

void interrupt_handler(void)
{
   ...
   if (diskdevicespace->status & DISK_INTERRUPTED) {
      while (!(diskdevicespace->status & DISK_OP_COMPLETE)) {
         *dst++ = diskdevicespace->data;
      }
      switch (diskdevicespace->status & DISK_ERROR_MASK) {
      case OK:
      ...
      }
   }
   ...
}
```

Intermission: What is an interrupt?

Interrupt handler invocation

'Involuntary' subprogramm call

- Not triggered by an opcode being executed
- Triggered by external hardware (interrupt pin),
  "a `call` opcode inserted between unsuspecting opcodes"
- Executes with privileges

Interrupt handler structure

1. Save registers
2. Check/handle device activity
3. Restore registers
4. Return to (unsuspecting) calling program

I/O for multi-process machines: Slave device

Interrupt+DMA

```
/* Disk I/O; diskdevicespace points to I/O address of disk controller */
dmadevicespace->base_address = dst;
diskdevicespace->sector  = s;
diskdevicespace->track   = t;
diskdevicespace->head    = h;
diskdevicespace->command = DISK_READ;
switch_to_other_process();

void interrupt_handler(void)
{
   ...
   if (diskdevicespace->status & DISK_INTERRUPTED) {
      switch (diskdevicespace->status & DISK_ERROR_MASK) {
      case OK:
      ...
      }
   }
   ...
}
```

I/O for multi-process machines: Master device

Interrupt+Bus Master

```
/* Disk I/O; diskdevicespace points to I/O address of disk controller */
diskdevicespace->transfer_address = dst;
diskdevicespace->sector  = s;
diskdevicespace->track   = t;
diskdevicespace->head    = h;
diskdevicespace->command = DISK_READ;
switch_to_other_process();

void interrupt_handler(void)
{
   ...
   if (diskdevicespace->status & DISK_INTERRUPTED) {
      switch (diskdevicespace->status & DISK_ERROR_MASK) {
      case OK:
      ...
      }
   }
   ...
}
```

I/O evolution

Comparison

Polling+PIO CPU busy-waits for device to be ready and transmits all bytes itself

Interrupt+PIO CPU works on other stuff[2] and is interrupted by the device, when it is ready. Bytes are transferred by the CPU

Interrupt+DMA CPU works on other stuff. When the device is ready, the DMA controller does the byte transfer instead of the CPU. The CPU is only interrupted at the end.

Interrupt+Bus master The device itself can access the memory as needed[3].

[2] or goes to sleep, if no other activity is pending

[3] could also read the next job from a job queue

I/O software in the OS: Goals

1. device independence
2. uniform naming
3. error handling
4. synchronous vs. asynchronous
5. buffering

I/O Software layers

I/O Software is often organized in four layers:

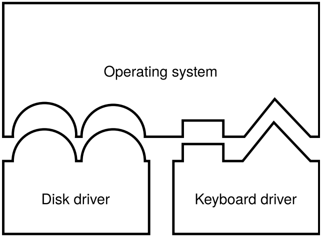
Hardware

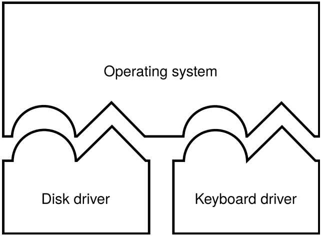
1. Interrupt handlers
2. Device drivers
3. Device-independent OS software (next slide)
4. User-level I/O software

User

Device-Independent I/O Software

1. Uniform interfacing
2. Buffering
3. Error reporting
4. Dedicated devices
5. Block size





User-level I/O software

- library procedures (e.g. I/O calls, formatting)
- spooling

Storage devices

- Tape
- DVD/CD/. . . /WORM
- HDD
- SSD/Flash
- RAM disk

Disk Software

Read/Write timing factors

1. Seek time: Arm onto right track
   1 ms (track-to-track)...10 ms (average random seek)
2. Rotational delay: Sector start under head
   $\frac{1}{2}$ rotation: 1/(rotation speed [RPM]/60)
3. Data transfer time: Sector(s) passing by
   bit density [b/cm] * rotation speed [RPM]/60 * circumference [cm]

Disk Arm Movement (1)

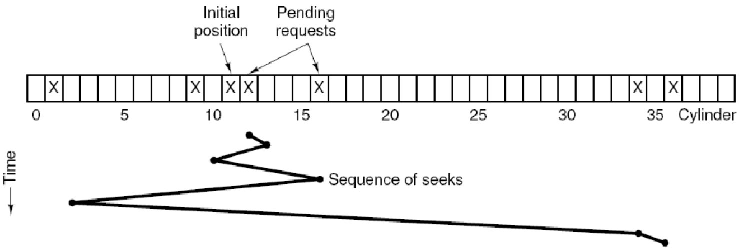


Figure: Shortest Seek First (SSF) disk scheduling algorithm. (Tanenbaum fig. 3.21

Disk Arm Movement (2)

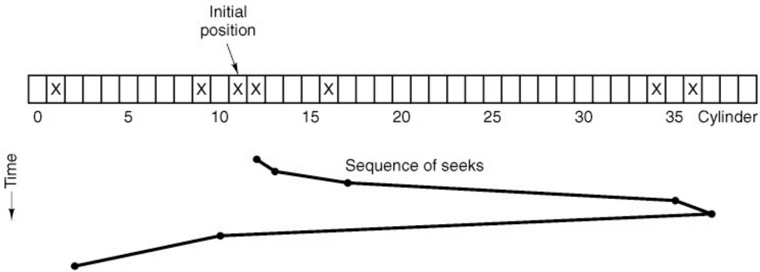


Figure: The elevator algorithm for scheduling disk requests. (Tanenbaum fig. 3.22

Common Hard Drive Errors

1. Programming error
   e.g. request for nonexistent sector
2. Transient checksum error
   e.g. caused by vibration during read
3. Permanent checksum error
   e.g. disk block physically damaged
4. Seek error
   e.g. arm was sent to cylinder 6 but it went to 7
5. Controller error
   e.g. controller refuses to accept commands

Terminals

Terminal Hardware

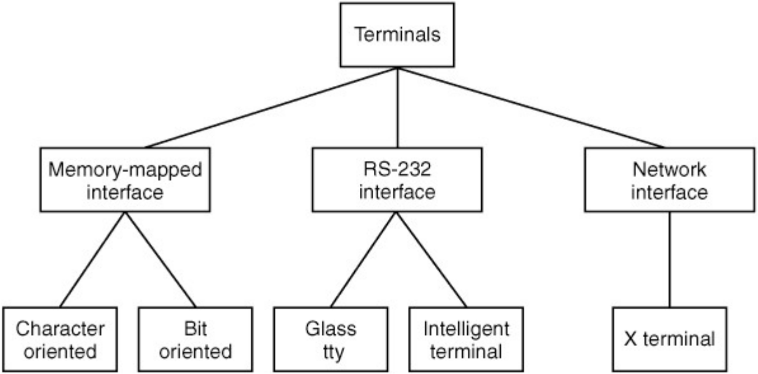


Figure: Different types of terminals. (Tanenbaum fig. 3.24

Memory-mapped interface

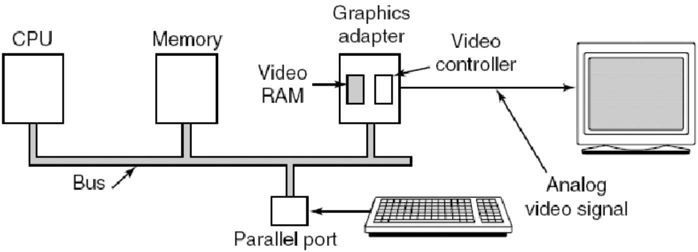


Figure: Memory-mapped terminals write directly into video RAM. (Tanenbaum fig. 3.25

RS-232 interface

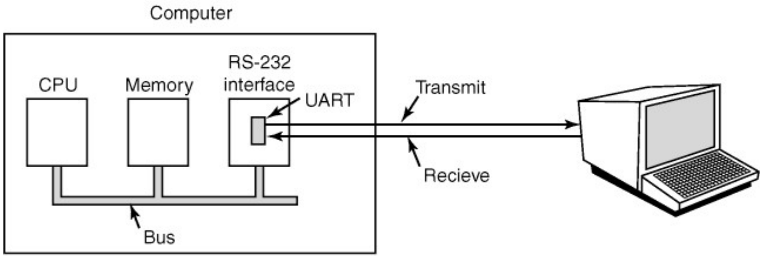


Figure: An RS-232 terminal communicates with a computer over a communication line, one bit at a time. The computer and the terminal are completely independent. (Tanenbaum fig. 3.27

Input software

Character	POSIX name	Comment
CTRL-D	EOF	End of file
	EOL	End of line
CTRL-C	INTR	Interrupt process (SIGINT)
CTRL-U	KILL	Erase entire line beeing typed

Table: Characters that are handled specially in canonical (cooked) mode.

Control/Escape sequences

Code	...0	...1	...2	...3	...4	...5	...6	...7
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
Code	**...8**	**...9**	**...A**	**...B**	**...C**	**...D**	**...E**	**...F**
0...	BS	HT	LF	VT	FF	CR	SO	SI
1...	CAN	EM	SUB	ESC	FS	GS	RS	US
7...	x	y	z	{	|	}	~	DEL

Sequence	Comment
`ESC [31;42m`	red letters on green background
`ESC [0m`	reset all attributes
`ESC [1E`	move cursor to beginning of next line
`ESC [5T`	scroll page down by 5 lines

Table: ANSI escape sequences[5] are used to control the terminal and are not interpreted as text.

[5] Introduced by '`ESC [`' aka '`CSI`' (control sequence introducer); ended by letter

Input codes

ASCII codes Only 7 bits, legacy[6]

USASCII code chart

b_7 b_6 b_5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4	b_3	b_2	b_1	Row ↓ / Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	−	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Unicode 20.1 bits(!), current (files, screen); often as encoded as UTF-8 (backward compatible to ASCII)

Scan codes Position on keyboard; unfortunately current (USB, Bluetooth input devices)

[6] Dozens of mostly incompatible 8 bit extensions exist