

Informatik 3

Mitschrift von www.kuertz.name

Hinweis: Dies ist **kein offizielles Script**, sondern nur eine private Mitschrift. Die Mitschriften sind teilweise **unvollständig, falsch oder inaktuell**, da sie aus dem Zeitraum 2001–2005 stammen. Falls jemand einen Fehler entdeckt, so freue ich mich dennoch über einen kurzen Hinweis per E-Mail – vielen Dank!

Klaas Ole Kürtz (klaasole@kuertz.net)

Inhaltsverzeichnis

0	Introduction: TANENBAUM'S „Modern Operating Systems“	2
0.1	Two views of an Operating System	2
0.2	History of Operating Systems	4
0.3	PETERSON'S Mutual Exclusion Algorithm	7
1	The XINU-Approach	8
1.1	Operating Systems	8
1.2	Our Approach	8
1.3	What an Operating System is <i>not</i>	8
1.4	An Operating System viewed from the Outside	8
1.4.1	The XINU Small Machine Environment	8
1.4.2	XINU Services	9
1.4.3	Concurrent Processing	9
1.4.4	The Distinction between Programs and Processes	9
1.4.5	Process Exit	10
1.4.6	Shared Memory	10
1.4.7	Synchronization	10
1.4.8	Mutual Exclusion	10
1.5	An Operating System viewed from the Inside	10
1.6	Summary	11
2	Overview of the Machine and its Run-Time Environment	11
2.1	The Machine	11
2.1.1	Physical Organization of the LSI 11/2	11
2.1.2	Logical Organization of the LSI 11/2	12
2.1.3	Registers in the LSI 11/2	12
2.1.4	Address space	12
2.1.5	Processor Status Word	13
2.1.6	Vectored Interrupts	13
2.1.7	Exceptional Conditions	14
2.1.8	Asynchronous Communication	14
2.1.9	LSI 11 Asynchronous Serial Line Hardware	14
2.1.10	Addressing a Serial Line Unit	14
2.1.11	Polled vs. Interrupt-Driven I/O	14
2.2	Disk Storage Organization	14
2.3	The C-Run-Time Environment	14
2.3.1	Conventions for translating procedures in the C-Compiler	15
2.4	Summary	17

3	List and Queue Manipulation	17
3.1	Linked Lists Of Processes	17
3.2	Implementation of the Q-Structure	18
3.2.1	In-Line Q Functions	18
3.2.2	FIFO Queue Manipulation	18
3.3	Priority Queue Manipulation	18
3.4	List Initialization	18
3.5	Summary	18
4	Scheduling and Context Switching	19
4.1	The Process Table	19
4.2	Process States	19
4.3	Selecting a Ready Process	19
4.4	The Null Process	21
4.5	Making a Process Ready	22
4.6	Summary	22
5	More Process Management	22
5.1	Process Suspension and Resumption	22
5.1.1	Implementation of Resume	22
5.1.2	The Return Values <code>SYSERR</code> and <code>OK</code>	23
5.2	System Calls	23
5.2.1	Implementation of Suspend	24
5.2.2	Suspending the current Process	24
5.3	Process Termination	24
5.4	Kernel Declarations	25
5.5	Process Creation	25
6	Process Coordination	25
6.1	Low-Level Coordination Techniques	26
6.2	Implementation of High-Level Coordination Primitives	26
6.3	Semaphore Creation and Deletion	26
7	Message Passing	26
8	Memory Management	27
9	Interrupt Processing	27
9.1	Dispatching Interrupts	27
9.2	Input and Output Interrupt Dispatchers	27
9.3	The Rules for Interrupt Processing	28

10 Real-Time Clock Management	29
10.1 The Real-Time Clock Mechanism	29
10.2 Optimization of Clock Interrupt Processing	29
10.3 The use of the real-time clock	29
10.4 Delta List Processing	30
10.5 Putting a Process to Sleep	30
10.6 Delays measured in Seconds	30
10.7 Awaking Sleeping Processes	30
10.8 Deferred Clock Processing	31
10.8.1 Procedures for Changing to and from Deferred Mode	31
10.9 Clock Interrupt Processing	31
10.10 Clock Initialization	31
10.11 Summary	31
11 Device independent Input and Output	31
11.1 Properties of the input and output interface	32
11.2 abstract Operations	32
11.3 Binding abstract Operations to Real Devices	33
11.4 Binding I/O calls to Device Drivers at Run-Time	33
11.5 Implementation of high-level I/O-operations	34
11.6 Opening and Closing Devices	34
11.7 Null and Error Entries in Device Table	34
11.8 Initialization of the I/O System	34
11.9 Interrupt vector Initialization	35
12 An Example Device Driver	35
12.1 The device type <code>tty</code>	35
12.2 Upper and Lower Halves of the Device Driver	35
12.3 Synchronization of the Upper and Lower Halves	36
12.4 Control Block and Buffer Declarations	36
12.5 Upper-Half <code>tty</code> Input Routines	36
12.6 Upper-Half <code>tty</code> Output Routines	36
12.7 Lower-Half <code>tty</code> Driver Routines	36
12.7.1 Watermarks and Delayed Signals	36
12.7.2 Lower-Half Input Processing	36
12.7.3 <code>cooked</code> Mode and <code>cbreak</code> -Mode Processing	37
12.8 <code>tty</code> Control Block Initialization	37
12.9 Device Driver Control	37
12.10 Summary	37

13 System Initialization	37
13.1 Starting from Scratch	37
13.2 Booting XINU	38
13.3 System Startup	39
13.4 Finding the size of Memory	39
13.5 Initializing System Data Structures	39
13.6 Transforming the Program into a Process	39
13.7 The Map of Low Core	39
13.8 Summary	39
17 A Disk Driver	39
17.14Summary	40
17.1 Operations supplied by the Disk Driver	40
17.2 Controller Request and Interface Register Descriptors	41
17.3 The List of pending Disk Requests	41
17.4 Enqueuing Disk Requests	42
17.5 Optimizing the Request Queue	42
17.6 Starting a Disk Operation	42
17.7 The upper-half read Routine	42
17.8 Driver Initialization	42
17.9 The upper-half output Routine	43
17.10The upper-half output Routine	43
17.11The upper-half seek Routine	43
17.12The lower-half of the Disk Driver	43
17.13Flushing Pending Requests	43

Organisatorisches

- In der zweiten Woche nach Semesterende wird ein fakultatives „Hands-on“-Praktikum (eine Woche) angeboten.
- relevant für die Note:
 - Mittsemester-Test am Dienstag, den 03.12.2002 (ca. 30%)
 - Endsemester-Test am Samstag, den 08.02.2003 (ca. 40%)
 - Hausaufgaben (ca. 30%)
- Literatur, die als Kopie ausgeteilt wird:
 - Kapitel 1 bis 2 aus: ANDREW S. TANENBAUM: Modern Operating Systems
 - Kapitel 1 bis 13, 17 bis 18 aus: DOUGLAS COMER: Operating System Design - The XINU Approach
 - Abschnitt „Operating Systems“ aus der C.S.Encyclopedia

Dafür bitte **5 €** mitbringen.

- Literaturempfehlung: William Stallings: „Operating Systems“ (4th Edition) oder auf deutsch „Betriebssysteme“ (4. Auflage, ISBN 3-8273-7030-2) - siehe www.pearson-studium.de
- Accounts: Von der Rechnerbetriebsgruppe kann sich jeder einen individuellen dauerhaften Account bereitstellen lassen:
 1. Formular online ausfüllen: www.informatik.uni-kiel.de ⇒ Rechnerbetriebsgruppe ⇒ Anmeldung
 2. Paßwort abholen bei Frau Dort, Verwaltungshochhaus Zimmer 807

0 Introduction: TANENBAUM'S „Modern Operating Systems“

0.1 Two views of an Operating System

- Without software a computer is a **useless lump of metal**. Computer-software is divided into:
 1. **Systems Programs**, managing the operation of the computer itself
 2. **Application Programs**, solving the problems of its users
- Most fundamental system program: **Operating Systems**
 - controlling a computer's resources
 - providing the basis upon which *application programs* can be written
- Modern computer systems consist of ≥ 1 processors, memories, clocks, I/O devices, terminals, disks, network interfaces - writing programs coordinating these resources is **extremely difficult!** Even restricting oneself to all details involving disk drives is almost impossible: If every programmer had to write these, writing application programs would be impossible. Solution: write layers on top of hardware shielding users from that complexity.
- „The most basic commands are READ and WRITE, each of which requires 13 parameters, packed into 9 bytes. These parameters specify such items as the address of the disk block to be read, the number of sectors per track, the recording mode used on the physical medium, the intersector gap spacing, and what to do with a deleted-data-address-mark. If you do not understand this mumbo jumbo, do not worry, that is precisely the point - it is rather esoteric. When the operation is completed, the controller chip returns 23 status and error fields packed into 7 bytes. As if this were not enough, the floppy disk programmer must also be constantly aware of whether the motor is on or off. If the motor is off, it must be turned on (with a long start-up delay) before data can be read or written. The motor cannot be left on too long, however, or the floppy disk will wear out. The programmer is thus forced to deal with the trade-off between long start-up delays versus wearing out floppy disks (and losing data on them).“ (from „modern operating systems“ by ANDREW S. TANENBAUM, 1.1.1)

- A computer system consists of hardware, system programs and application programs:

Banking system	Airline reservation	Adventure games	} Application programs
Compilers	Editors	Command interpreter	
Operating system			} System programs
Machine language			
Microprogramming			
Physical devices			

- *Devices*: chips, wires, clocks, power supply; *programming*: microprograms control these devices (in read-only memory, basically interprets executing instructions as ADD, MOVE, JUMP, ...). In this language I/O-devices are controlled by loading values into *device registers* (i.e. for disk: disk address, memory address, byte count, read/write). The Operating System hides all this **complexity**: „write block X to file Y“.
- Operating System runs in **kernel** or **supervisor mode**: protected from user tampering by hardware, compilers and editors run in **user mode** - which user is free to change¹.
- There exist two views of an operating system²:

1. Bottom Up: As a **Resource Manager**: This view holds that the primary task of an Operating System is to keep track of who is using which resource, to grant resource requests, to mediate conflicting requests between different users.

This view provides a bottom-up view: The Function of an Operating System is to manage all resources. Multiple usage/users competing for multiple resources ⇒ need for management & protection (memory, I/O devices, ...)

This need arises because users must *share* these resources, and because users want to *share information*. So, in this view, the Operating System ...

- keeps track of who uses which resource
- it grants resource requests
- it accounts for usage (time used ...)
- mediates in conflicting requests

¹„Und für das Script brauche ich fünf Öro!“

²„Wenn man kein C kennt: Kein Problem, ich kann's auch nicht!“

2. Top Down: As **providing a simple high-level abstraction** that hides the truth about horrible hardware from the programmer by providing a simple interface for a *virtual machine* (= the real machine and its system programs)

0.2 History of Operating Systems

The history of Operating Systems is intimately linked with the history of computers.

0. CHARLES BABBAGE'S (1792 - 1871) **Analytical Engine** (only in theory)
1. 1945 - 1955: based on **Vacuum Tubes & Plugboards**³
 - No separation between builders and users because vacuum tubes broke down too often (approx. 20.000 of them were used) and programming was only possible by wiring plugs up to control machine's basic functions
 - The first **BUG** was found by GRACE MURRAY HOPPER on the Mark II computer⁴, is now in the National Museum of American History at the Smithsonian Institute.
 - In the early 1950's punched cards were used instead of wired-up plugboards.
2. 1955 - 1965: based on **Transistors & Batch Systems**
 - gain in reliability, possibility to sell computers (split between builders/designers and users/maintenance personnel)
 - introduction of high-level machine languages (COBOL, FORTRAN, ALGOL, LISP)
 - mode of operation⁵
 - there was a split between:
 - word-oriented large scale *scientific* computers for numerical calculations

³see Tanenbaum 1.2.1

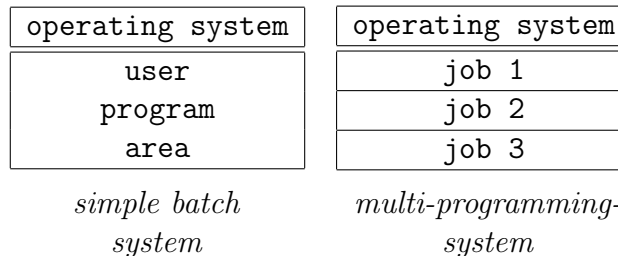
⁴legt Folie vom Logbuch des Mark II mit dem eingeklebten ersten Bug (einer Motte) auf: „Das wird Euer Leben sein! (...) Das ist Sauron! Sauron kann nicht bekämpft werden, Sauron steht immer wieder auf!“

⁵see Tanenbaum page 7

- character-oriented *commercial* computers for tape sorting and printing

3. 1965 - 1980: that of **Integrated Circuits**, based on **IC & Multiprogramming**:

- request for *single family* satisfying needs of all customers: IBM 360 concept: in principle, same program could be run on all computers of the 360 line
- range: from less powerful, decently priced computers to highly expensive ones, resulting in a much better *price/performance ratio*⁶
- difficulties with programming their Operating System led in 1967 to coining the term „*software crisis*“, everyone’s wave of corrections in millions of lines of code triggered a *new wave of errors* ⇒ debugging difficult because of lack of structures⁷
- advantages⁸:
 - *multiprogramming* to optimize processor usage. Why?
 - * with commercial data processing I/O wait time 80 - 90% of total computer time, so CPU idled too much
 - * memory partitioned with different job in each partition, special protection needed for each job to prevent mischief by other processes (built into 360):



- *spooling*: enables Operating System to load a new job from disk in a now-empty partition
- *timesharing*: multiprogramming & fast turn around time & interaction, each user has its own on-line terminal and protection of hardware segment in memory, created the illusion that each user had a machine all on its own (see text pg. 10); first

⁶„Die Idee war phantastisch, genau wie alle Lügen von Schröder in der Wahlperiode!“


⁷„Ingenieure sind konservativ: Eine Brücke, die heute noch steht, steht auch in ein paar Jahren noch!“

⁸„puff puff puff, tschuka tschuka tschuk“

time-sharing system CTSS (1962) implemented on a modified IBM 7094

- MULTICS (MIT, BELL LABS, GENERAL ELECTRIC): This *Multiplexed Information and Computing Service* should run on one such machine and provide computer power for everyone in Boston! ⇒ GENERAL ELECTRICS stopped producing computers, BELL LABS quit project
- Growth of Minicomputers: PDP1-11, VAX, based on LSI (large scale integrated circuits) for single client use; Ken Thomson, who worked on the MULTICS-project, found a not used PDP-7 minicomputer and wrote *a stripped down, single user version of MULTICS: UNIX!*

4. 1980 - now: that of **personal computers**

- based on vlsi circuits: better price/performance ration
- the micro-processor-chip made it possible for a single individual to have her/his own personal computer: laptop; most powerful versions: workstations, linked together by a network: distributed computing/web
- main-factor for Operating Systems: ***user friendliness: These systems were intended for users, not knowing anything about computers, and having no intention of learning their structure***⁹. 
- MS-DOS was dominant on Intel-processors (no protection of hardware, virus sensitive) and UNIX was dominant on non-Intel-processors for workstations (with hardware protection)
- in a *Network Operating System* users login on remote machines, copy files from one machine to another; Network Operating System consists of a single processor Operating System & network interface control & remote login
- *Distributed Operating System*: composed and running on multiple processors, but appears to users as a traditional processor; different from single-processor-systems because of the complex scheduling and problem of communication delays

⁹„Das erklärt, warum wir in InformatikI 230 Studenten und hier noch 80 Studenten haben!“

0.3 PETERSON'S Mutual Exclusion Algorithm

1. Listing:

$P_1 \equiv l_0$: loop forever do l_1 : noncritical section l_2 : $(y_1, s) := (1, 1)$; l_3 : $\text{wait}(y_2 = 0) \vee (s \neq 1)$; l_4 : critical section l_5 : $y_1 := 0$; od	$P_2 \equiv m_0$: loop forever do m_1 : noncritical section m_2 : $(y_2, s) := (1, 2)$; m_3 : $\text{wait}(y_1 = 0) \vee (s \neq 2)$; m_4 : critical section m_5 : $y_2 := 0$; od
--	--

- critical section and noncritical section do not refer to y_i, s
- wait b : traffic light, passable when b holds
- y_i : Set to 1 by P_i to signal to P_j that P_i wants to enter critical section, $i \neq j$. set to 0 by P_i to signal to P_j that P_i has exited critical section.
- s : Serves as a logbook to resolve tie between 2 processes wishing both to enter critical section; contains identity of the latest processes willing to enter.
- **problem:** $(y_1, s) := (1, 1)$ is a simultaneous assignment! !

2. Listing:

$P_1 \equiv l_0$: loop forever do l_1 : noncritical section l_2 : $s = 1$; l_3 : $y_1 = 1$ l_4 : $\text{wait}(y_2 = 0) \vee (s \neq 1)$; l_5 : critical section l_6 : $y_1 := 0$; od	$P_2 \equiv m_0$: loop forever do m_1 : noncritical section m_2 : $s = 2$; m_3 : $y_2 = 1$ m_4 : $\text{wait}(y_1 = 0) \vee (s \neq 2)$; m_5 : critical section m_6 : $y_2 := 0$; od
--	--

- **problem:** $l_0, l_1, l_2, m_0, m_1, m_2, m_3, m_4, m_5, l_3, l_4, l_5$!

3. Listing:

$P_1 \equiv l_0$: loop forever do l_1 : noncritical section l_2 : $y_1 = 1$ l_3 : $s = 1$; l_4 : $\text{wait}(y_2 = 0) \vee (s \neq 1)$; l_5 : critical section l_6 : $y_1 := 0$; od	$P_2 \equiv m_0$: loop forever do m_1 : noncritical section m_2 : $y_2 = 1$ m_3 : $s = 2$; m_4 : $\text{wait}(y_1 = 0) \vee (s \neq 2)$; m_5 : critical section m_6 : $y_2 := 0$; od
--	--

- this solution works!

1 The XINU-Approach

1.1 Operating Systems

1.2 Our Approach

The XINU-approach is a **practical approach**, showing the details of a real system; starting with a micro computer and proceeding step-by-step through the construction of a layered system.

Advantages: see how an entire system fits together, no mystery about any part of the implementation, possibility to experiment with the system.

The programs and code form an **integral part** of the book and the lectures, beginning with a layering scheme and following it consistently.

1.3 What an Operating System is *not*

1. *a language or a compiler:* one needs no special language or compilers to write Operating Systems
2. *command interpreter:* in modern systems the user can choose its own command interpreter
3. *library of commands:* programs that edit files, send mails, compile programs ar just utility programs

1.4 An Operating System viewed from the Outside

An Operating System provides services, programs access these services by making **system calls**. These system calls establish a boundary between the running program and the Operating System, an Operating System can even be described by its services.

With XINU as illustrating example we explain how to read characters from a keyboard, display characters on a terminal, manage multiple processes and so on.

1.4.1 The XINU Small Machine Environment

Sometimes computers are too small to compile operating systems. How to get an Operating System on the computer?

- minimum configuration on client

- entire system on a larger machine (host)
- user compiles on host with cross-compiler
- downloader copies the memory image to the client (serial connection)
- execution proceeds on the client

Otherwise, one version of the Operating System can be used to create the next version (*Bootstrapping*).

1.4.2 XINU Services

1.4.3 Concurrent Processing

Concurrent processing means that many computations proceed „at the same time“. In contrast, conventional programs are called sequential because the programmer imagines a machine executing the code statement-by-statement.

To create the illusion of concurrent processing the Operating System switches a single processor among multiple programs, the Operating System itself is a good example of a concurrent program. The lowest level of Operating System is the scheduler handling all these processes.

1.4.4 The Distinction between Programs and Processes

There are some major differences between programs and processes:

- A procedure call does not return until the called procedure completes. Create and resume return to the caller after starting the process, allowing execution of both the calling procedure and the named procedure to proceed concurrently.
- A program consists of code executed by a single process. In sharp contrast, processes are not uniquely associated with a piece of code: multiple processes can execute the same code simultaneously.
- Storage for local variables and procedure arguments is associated with the process executing the procedure, not with the code in which they appear.

Just like a sequential program, each process has its own stack of procedure calls. Whenever it executes a call, the called procedure is pushed onto the stack; whenever it returns from that procedure, it is popped off the stack.

1.4.5 Process Exit

The system call `kill(P)` terminates the process with process-ID `P`.

To kill a process with name `process` just call `kill(getPID("process"))` .

1.4.6 Shared Memory

1.4.7 Synchronization

Look at the *consumer/producer-problem*: How can the programmer synchronize producer and consumer, so that the consumer receives every datum produced? The mechanism for synchronization of producer and consumer must be designed carefully, because:

- In a single processor system, no process should use the CPU while waiting for another process. XINU avoids *busy waiting* by supplying coordination primitives called *semaphores*, and system calls `wait(s)` and `signal(s)` operating on semaphore `s`
- `wait(s)` decrements semaphore `s` and causes the process to wait if the result is negative; `signal(s)` increments `s`, allowing some waiting process to continue

1.4.8 Mutual Exclusion

Mutual exclusion can be realized with semaphores, it is used to avoid race conditions - for example in a print spooler. No two processes are in their critical section at the same time, no process running outside the critical section may block other processes.

1.5 An Operating System viewed from the Inside

The **layers** of XINU:

1. Hardware
2. Memory Manager
3. Process Manager
4. Process Coordination
5. Interprocess Communication
6. Real-Time Clock Manager

7. Device Manger and Device Drivers
8. Intermachine Network Communication
9. File System
10. User Programs

1.6 Summary

The operating system manages to provide **reasonable high-level services** with **unreasonably** low-level hardware, hiding the details of the low-level machine. Since 1967 each **layer** of the operating system provides an abstract service, implemented in terms of the abstract machine provided by the lower-level layers. User access the operating system by **system calls**; possibility of concurrent programming (**a grand illusion!**).

The **Distinction between Programs and Processes**¹⁰: A procedure call does not return until the called procedure completes. Create and resume¹¹ return to the caller after starting the process, allowing execution of both the calling procedure and the named procedure to proceed concurrently¹².

2 Overview of the Machine and its Run-Time Environment

2.1 The Machine

The DIGITAL EQUIPMENT CORPORATION **LSI 11/2** 16-BIT-MICROPROCESSOR - a microcomputer version of the PDP11. Discussed here: describing pertinent features of the processor, memory, and communication devices. It explains the architecture, asynchronous communication, disk storage devices and mechanisms like the stack, vectored interrupts and device addressing.

2.1.1 Physical Organization of the LSI 11/2

The LSI 11/2 is constructed from **printed circuit boards**, slotted into the sockets of a backplane. These sockets are wired together to form a **bus** (the

¹⁰Zu seiner eigenen Folie: „Ich hoffe, daß das lesbar ist - meines Erachtens ist es das nicht!“

¹¹„resume ist wie ein Ehepaar, es erzeugt ein Kind und setzt sein Leben fort!“

¹²„Frauen können hervorragend nebenläufig denken, das war früher in den Höhlen so! Concurrent processing findet hauptsächlich statt im Kopf einer Frau!“

Q-bus) consisting of *power lines* (attached to all board in parallel) an *lines linking up the sockets for interboard communication* (signals travel to the board on one contact and away on another). One board contains the 11/2 processor itself, other boards contain memory or device interfaces and so on.

A board communicates with another board by passing signals across the bus. E.g., when the processor board needs to write unto memory, it places *address & data* (16 + 16 Bit) on the bus for the memory board to retrieve & store.

Memory is **logically contiguous**, each board contains switches (hardwired jumpers) that can be changed: it is possible to configure two identical memory boards so that one responds to low memory addresses and the other to high addresses. The physical order of boards along the Q-bus determines their *priority* (used later).

Signals enter a board on one connector and leave it on another one, so the board can decide whether to **intercept the signal or pass it on down the bus**; example: two boards waiting for service.

2.1.2 Logical Organization of the LSI 11/2

2.1.3 Registers in the LSI 11/2

Register R_4 points to *calling* procedures frame, while R_5 points to frame of currently active *called* procedure.

Notation	Register	Use
R_0	0	general purpose
R_1	1	general purpose
R_2	2	general purpose
R_3	3	general purpose
R_4	4	previous display
R_5	5	current display
SP	6	stack pointer
PC	7	programm counter
PS	status	processor status

2.1.4 Address space

The memory is divided into 8-bit-quantities called *bytes* (also called *character*) being the smallest addressable unit. Most instructions operate on two bytes

(a *word*), they affect addressed byte & the next higher byte. All addresses are formulated in *octal*! The LSI 11/2 hat 64K Bytes of Memory (2^{16} bytes)¹³:

from	to	use of memory
0000000	0000777	for interrupts and exception vectors
0001000	0157777	real available memory, the stack grows in direction from 0157777 to 01000
0160000	0177777	address space for devices

2.1.5 Processor Status Word

Processor Priority: To disable interrupts set processor/interrupt mask to $011100_2 = 340_8$.

15 through 8	7 through 5	4	3 through 0
...	Processor Priority	Trace Mode	Condition Codes

2.1.6 Vectored Interrupts

The LSI 11/2 uses the **vectored interrupt scheme**¹⁴ for handling exceptions and interrupts from external devices. Whenever an external device must communicate with the processor, the device places a signal on the interrupt bus line. If the processor runs with interrupts *enabled*, it checks the interrupt line after executing *every* instruction.

To handle an interrupt, the processor sends an **acknowledgement** over the bus, requesting the interrupting device to return an **interrupt vector address (IVA)**. Each device is assigned a unique interrupt vector address, enabling the system software to identify/distinguish among them. The first device with a pending request receives the acknowledgement and responds by returning its interrupt vector address (an interrupt vector consists of two bytes).

When CPU receives the interrupt vector address v from the Q-bus, the processor **pushes** current value of PC and PS on stack, and **loads** a new PC and PS from 2 words in memory starting at location v , and continues execution beginning at the new location addressed by PC .

The interrupt „acts“ like a procedure call, inserted (invisibly) by the hardware in between two instructions in the user’s code. The processor **executes the**

¹³„Dort schläft schon ein Herr, laßt ihn schlafen, er hat’s verdient, er hat heute Nacht viel Spaß gehabt - hoffen wir!“

¹⁴„Das ist nur etwas Blödes, das ist nicht schwer!“

code in the interrupt routine and **returns** to the place where the user's program was interrupted.

2.1.7 Exceptional Conditions

2.1.8 Asynchronous Communication

The serial line unit sends and receives characters asynchronously on three wires (data in two directions and electrical ground). The Transmitter sends series of pulses, 8 bit together with optional start, stop and parity bit; the Conversion is done by *Universal Asynchronous Receiver and Transmitter (UART)*.

Sender and Receiver have own clocks and try to sample the series of pulses, each bit gets sampled several times, inconsistency results in *framing error* and too much characters means *character overrun error*. A line remaining *idle* in the wrong state for an extended period gives a break (e.g. needed for downloading operating system into memory, see (1.4.1)).

2.1.9 LSI 11 Asynchronous Serial Line Hardware

2.1.10 Addressing a Serial Line Unit

The CPU has to read and write to **special addresses beyond the real memory**, e.g. 0177560_8 , 0177562_8 , 0177564_8 , 0177566_8 .

2.1.11 Polled vs. Interrupt-Driven I/O

Polling requires the CPU to check the device repeatedly until it finds that a character is waiting. To use **interrupt-driven** processing instead of polling, each subsequent character will cause an interrupt.

2.2 Disk Storage Organization

Wird erst in den letzten zwei Wochen der Vorlesung behandelt!

2.3 The C-Run-Time Environment

Operating Systems are written in *high-level languages* to make them easier to write, understand, debug and move to other machines. However, sometimes machine *assembly language* procedures are introduced because machine quantities must be directly manipulated, e.g. for saving and restoring the machine's registers, writing context-switching code, writing interrupt-handling,

implement semaphores and so on. The Storage Layout for a C-Program looks like:

0	<code>_etext</code>	<code>_edata</code>	<code>_end</code>	
text	data	bss	free space	← ... stack

Because the whole XINU-system is one C-program, the storage layout when XINU runs is modified:

0	<code>_etext</code>	<code>_edata</code>	<code>_end</code>			
text	data	bss	free space	← stack #3	← stack #2	← stack #1

The symbols `_etext`, `_edata` and `_end` refer to global variables inserted into object program by the loader, they are initialized to the first address beyond text, data and bss segments. Thus a running program can find out how much memory remains between the end of the loaded part and the current top of stack by taking the address of `_end`.

2.3.1 Conventions for translating procedures in the C-Compiler

```

proc A {
    some_code_1;
    call B(arg1, ..., argn);
    some_code_2; }

```

In this example¹⁵, *A* is the calling, *B* the called procedure. How is the code of calling procedure *B* within the procedure body of *A* in C's compiler?

1. The values of **actual parameters** of *B* (in reverse order) are pushed on the stack.
2. The address of **return address** in *A* (i.e. of the instruction following to the call of *B*, *some actions* in picture above) are pushed on the stack.
3. Then the flow of control branches to *B*.

Calling procedure (*A*) is also responsible for popping *B*'s arguments from the stack after the called procedure (*B*) returns.

- code of calling procedure *A*:

```

...          // B's arguments put on stack
jsr pc, addB // pushes return address on stack
              // and branches to subroutine B
...          // arguments of B popped from stack
rts pc       // pops an address from the stack
              // and returns to that address

```

¹⁵ „Mein Gehalt ist so etwa 7 bis 10 € pro Minute, wenn ich vorlese - also nutzt das!“

- code of called procedure *B*:

```

    move r5, r0    // pushing R5's old value onto stack
    move pc+2, r5  // assigning to R5 the return address in B
    jsr r0, csv    // after calling csv, and calling csv

pc+2: ...         // assumption: after executing these fragment
                // of B's code, the stack has the same contents,
                // and the values of SP and R5 are the same
                // as before executing this fragment.

    rts pc        // pops address of cret from stack
                // and returns to cret

```

- *csv* - *C registers save routine*: To save the machine registers, the compiler inserts a call to *csv*, an assembly language routine that saves the registers and jumps back to called routine *B*

```

csv:  move r5, r0    // by assumption R5 contains return address
                // in B, and stack has old value of R5,
                // return address in A and B's arguments on it
    move sp, r5     // R5 stores value of SP, i.e. address of B's frame
    ...            // R4, R3, R2 pushed on stack in that order
    jsr pc, (r0)    // old value of PC, i.e., address of \verb"cret",
                // pushed on stack and control branches to address
                // stored into, i.e., return address in B

```

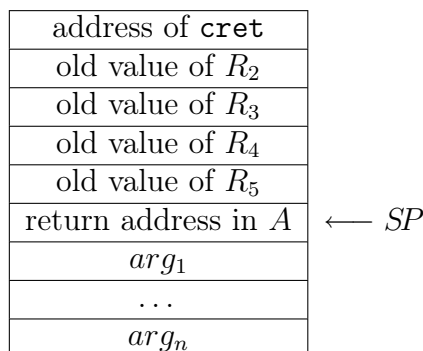
- *cret* - *C registers restore routine*: When code for *B* is finished, compiler inserts call to assembly language routine *cret* to restore the old values of the machine's registers and the stackpointer *SP* (i.e. *R6*) and to return control to the original caller.

```

cret:  move r5, r2    // R2 contains address of B's frame now
    move -(r2), r4    // old values of R4, R3, R2
    ...            // restored in that order!
    move r5, sp     // SP points to B's frame again
    move (sp)+, r5   // old value of R5 restored and
                // afterwards SP := SP + 2
    rts pc         // pops return address in A's code
                // and jumps to that address

```

- Stack:



2.4 Summary

3 List and Queue Manipulation

3.1 Linked Lists Of Processes

List processing is **fundamental** for operating systems since one needs lists of processes to be *scheduled* (*ready* for scheduling), processes *waiting on a semaphore*, processes *ordered by priority (time)* in order to implement *time-sharing (sleep list)*.

Two types are needed here: **FIFO queues** are ordered by *time of insertion*, **priority queues** are ordered by *time to wake up* (in view of time-sharing); necessary operations:

- **inserting** an item *at the tail* of a list or within an *ordered list*
- **removing** an item at the *lead* of a list
- allocating a new list

Programming these operations is simple because only *one process* executes these operations *at a time*. This is due to the fact that they occur only in *critical sections* (programmed by forbidding interrupts¹⁶ in the fields of the process status register). Elements are extracted from FIFO queue by removing them from the head, hence they are inserted at its tail.

If the list is a priority queue, *getfirst(head)* removes the item with the *smallest key*, and *getlast(tail)* removes the item with the *biggest key*. Hence priority queue insertion starts at head of the priority queue.

Items to be stored in lists are **process identifiers** (there are *NPROC* processes) with $MININT \leq \textit{process priority} \leq MAXINT$. All lists are *doubly linked*, i.e. each node points to its successor as well as predecessor, and each node contains a key; each list has both a head and a tail, predecessor of head and successor of tail point to empty list (= -1); head node contains *MININT* as key, tailnode *MAXINT*.

Optimization is possible because a process appears on *at most one* list at any time; queue-array has nodes for each process in its $[0 \dots NPROC - 1]$ -part, and head and tail nodes for every list needed in it $[NPROC \dots NQENT]$ -part.

¹⁶„...interrüpts ...“

	key	next	previous
0			
1			
2	14	33	4
3			
4	25	2	32
...			
<i>NRPOC</i> -1			
<i>NPROC</i>			
...			
32	<i>MININT</i>	4	-1
33	<i>MAXINT</i>	-1	2
...			

Array¹⁷ queue is external, and an array¹⁸ of queue structures, as declared in file `q.h`.

3.2 Implementation of the Q-Structure

3.2.1 In-Line Q Functions

3.2.2 FIFO Queue Manipulation

3.3 Priority Queue Manipulation

3.4 List Initialization

3.5 Summary

Linked lists are kept in a single data structure, the *q* array. Primitive operations for manipulating the lists of processes can produce FIFO queues or priority queues. All lists have the same format: they are doubly-linked, each has both a head and tail, and each node has an integer key field. Keys are used when the list is a priority queue: they are ignored if the list is a FIFO queue.¹⁹

¹⁷„Dies ist eine Vorlesung über Ziegenfortpflanzung, und jetzt geht’s darum, wie Ziegen sich fortpflanzen!“

¹⁸„Kannst Du das beantworten?“ - „Äääähm, ... ich habe die Frage nicht ganz verstanden!“ - „Das ist natürlich das Einfachste!“

¹⁹„Wo ich gelehrt habe im California, da wird man *gefired*, wenn ein Drittel der Studenten wegbleibt!“

4 Scheduling and Context Switching

Why is scheduling and context switching difficult? Because the CPU cannot be stopped at all!

4.1 The Process Table

The `proctab` is an array of structures `pentry`, there is one entry for each process in this table. Because only one process running at the time one of these entries is out-of-date, since corresponding with the currently active process. The other entries correspond to processes which are temporarily halted.

Exactly what info must be saved in `proctab`? All values that will be destroyed when another process runs; e.g. no copy of the stack because there a separate stack areas for different processes. In addition to data that must be reloaded when it resumes a process, the system also keeps information in the process table that it uses to control processes and account for their resources.

Processes are referenced by their process id, which is the index of the saved state information in `proctab`.

4.2 Process States

The system uses the `pstate`-field of the process table to help it keep track of what the process is doing and the validity and semantics of operations performed on it. In XINU there are the following six states (and one to signal that a slot in the process table is free):

Status	Kapitel
PRCURR	(4)
PRREADY	(4)
PRRECV	(7)
PRSLEEP	(10)
PRSUSP	(5)
PRWAIT	(6)
PRFREE	(4)

4.3 Selecting a Ready Process

A process is classified as `PRREADY` when it is eligible for CPU service, but not currently running. The single process served by the CPU is classified as `PRCURR`. Switching context involves selecting a process from those that are

ready or current, giving control to the selected process.

Often it remains eligible to use CPU, even when control is temporarily passed to another process (e.g. as a result of *round-robin scheduling*). Then it's current process state changes to `PRREADY` and it is moved to ready list for later CPU service.

How does the Rescheduler (`resched`) decide whether to move current process to ready list? If current process is not eligible to use the CPU, the system routines assign a desired next state to its `pstate`-field **before** calling `resched`.

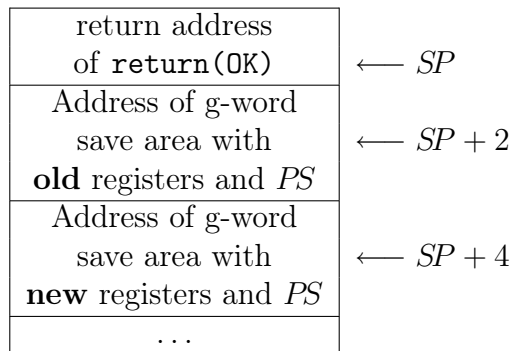
Which processes call `resched`? The process which is executed under XINU on this exact moment! The Processes that are calling `resched` are:

procedure	page
<code>sleep</code>	129
<code>wakeup</code>	133
<code>receive</code>	97
<code>send</code>	96
<code>wait</code>	85
<code>signal</code>	86
<code>sdelete</code>	89
<code>suspend</code>	69
<code>resume</code>	67
<code>kill</code>	71

So `resched()` is a normal procedure, and calling it results in executing it **like a normal procedure**. The code of context-switching procedure `ctxsw` is machine dependent because machine registers should be saved by it:

- The *PC* must be changed **last** to give the CPU the opportunity to continue executing the new process (and not earlier) once info about old process stored into `proctab` and stack.
- On the 11/2 the `rtt` instruction pops both *PS* and *PC* from stack and reloads them **in one step**, so **after** saving the registers associated with the old process in the registers save area of that process.

The stack look like this:



```

move r0,*2(sp) // save old R0 in old register area
move 2(sp),r0 // get address of old register area in R0
add $2,r0 // increment to get position where R1 will be saved
move r1,(r0)+ // save R1 to R5 in successive
move r2,(r0)+ // locations of the old process register
move r3,(r0)+ // save area.
move r4,(r0)+ //
move r5,(r0)+ //
move $2,sp // move SP beyond the return address,
// as if a return had occurred
move sp,(r0)+ // save stack pointer
move -(sp),(r0)+ // save caller's return address as PC
mfps (r0) // save processor status beyond registers
move 4(sp),r0 // pick up address of new registers in R0
// ready to load registers for the new
// process and abandon the old stack
move 2(r0),r1 // load R1 to R5 and SP from
move 4(r0),r2 // the saved area
move 6.(r0),r3 //
move 8.(r0),r4 // the period . makes it decimal
move 10.(r0),r5 //
move 12.(r0),sp // switching stacks
move 16.(r0),-(sp) // push new process PS on new process stack
move 14.(r0),-(sp) // push new process PC on new process stack
move (r0),r0 // finally, load R0 from new area
rtt // load PC, PS and reset SP all at once

```

After return from hibernation²⁰ stack pointer should point to stack segment with parameters of `ctxsw` since these are popped in calling procedure's translation (argument of `ctxsw` must be popped from stack).

4.4 The Null Process

The scheduler assumes that *at least one process* is available on the ready list queue, it does not bother to verify whether the ready lists is empty, so: `resched()` can only switch context from one process to another, so *at least one process must always remain on ready queue*. To ensure that a ready process always exists XINU creates an extra process, the **null process**, when it initializes the system. It has `pid` zero and `pprio` zero; its code consists

²⁰„Wer hat so ungefähr eine blasse Ahnung, wovon ich rede?“

of an infinite loop (see page 196). Because user processes all have priority greater than zero, the scheduler switches to the null process only when no user process is ready to run.

4.5 Making a Process Ready

Making a process eligible for CPU service occurs so often that a *special procedure* has been designed to do so: `ready(pid, resched)`. In principle putting a process on the ready lists should result in a call to `resched` to make sure that the process with the highest priority is running. However, sometimes this results in a too heavy overhead in execution time when many processes are put on the ready queue (example: process wakeup in realtime interrupt processing, see page 133). Then all these processes are put on the ready list without rescheduling after each one, only after the last one `resched()` is called once. This construction is made possible by providing `ready(pid, resched)` with a boolean argument `resched`.

4.6 Summary

5 More Process Management

5.1 Process Suspension and Resumption

Transparentes Einfrieren und Entfrieren eines Prozesses: Wie beim Kontextwechsel zwischen `CURRENT` und `READY`: Retten aller potentiell wichtigen Daten, auf Benutzerwunsch Systemaufruf (*system call*) und Einfrieren „bis auf expliziten Widerruf“ (neuer Zustand `SUSPEND`)

Jeder Prozess hat jeweils genau einen der (in XINU 6) möglichen Zustände. Bisher (die fundamentalen für Timesharing/Multiprogramming): `CURRENT` (laufend) und `READY` (lauffähig, aber gerade nicht dran); neu: `SUSPENDED` (d.h. temporäres, benutzergesteuertes Gestopptsein, also nicht gleich `READY`), es existiert aber keine `SUSPENDED`-Queue!

Der `suspended`-Status wird z.B. benutzt, um *pages* vom Datenträger zu laden, die in Benutzerprogrammen referenziert wurden und im Hauptspeicher erwartet werden.

5.1.1 Implementation of Resume

Wie bei allen Systemaufrufen, werden am Anfang die Interrupts deaktiviert (`disable(ps)`) und am Ende wieder aktiviert (`enable(ps)`). `resume` setzt

den Status eines Prozesses auf **READY** und setzt ihn entsprechend auf die *ready-list*.

Resume calls `resched`, and hence changes the process table and the q-structure. Reason for the **need of turning interrupts off** when accessing the central data structures of XINU: What happens when another process would be simultaneously changing the process table? This situation is, in general, called *race condition*. The final state of the process table is not clear!

$$\textit{simultaneously}\{x = 1 \mid x = 2 \mid x = 3\} \Rightarrow x = ?$$

To prevent such situations from happening, this uncertainty is prohibited by disabling interrupts while accessing the process table.

How can `resched()` be called? By calling `sleep/wakeup`, `receive/send`, `wait/signal`, `suspension/resume` (see page 128). However this kind of call presupposes that `resched()` is called in parallel to the current process. Interleaving the instructions of this concurrent (parallel) processing in between the instructions of the current process implies that, e.g., the real-time clock procedure has interrupted the current process. However, this is impossible when the current process is deaf to interrupts. For then the real-time clock interrupt is temporarily postponed.

Question: Does this necessarily imply that the process reacts to interrupts again? No, the interrupts may have been turned off before calling `resched()`!

5.1.2 The Return Values **SYSERR** and **OK**

Es werden Fehlerbedingungen abgefragt und ggf. wird ein **SYSERR** ausgegeben.

5.2 System Calls

Systemaufrufe sind vom Benutzer aufrufbar, in der Regel sind sie mit besonderen Privilegien ausgestattet; siehe dazu auch Anhang 2.2. Aufgaben sind zum einen, eine Schnittstelle zwischen Anwenderprogrammen und Systemkern zu bieten, also die Dienste des Betriebssystems bereitzustellen, zum anderen, das System zu schützen.

Grob gesagt: Systemaufrufe sind alles, was der Benutzer vom System verlangt. Klassifizierung:

- Prozeßmanagement und Signale, Interprozesskommunikation
- Filemanagement, Verzeichnis- und Filesystemmanagement

- Schutzfunktionen
- Zeitmanagement oder allgemeines Informationsmanagement
- Ein- und Ausgabe (z.B. Öffnen, Schliessen, Terminalattribute, Geschwindigkeiten, ...)
- (Speichermanagement)

There is only *one process table* in the system, shared by all its processes. So how can a process be *sure* that no other process is trying to change the process table at the same time?

1. It should not call `resched()`, because rescheduling switches control to another process and this changes the process table
2. The system should not react to interrupts, since interrupt routines can call `resched()` as well.

5.2.1 Implementation of Suspend

`suspend` löscht einen Prozess mit `READY`-Status von der *ready-list* bzw. stoppt ggf. den aktuell ausgeführten Prozess.

5.2.2 Suspending the current Process

Ein Prozess kann sich selbst suspendieren mit `suspend(getpid())`.

5.3 Process Termination

Suspend freezes processes, but leaves them in the system so they can be resumed later. Another system call, `kill`²¹, stops a process immediately and removes it from the system completely. It checks the `pid` (so the null process can't be killed), decrements the number of processes that currently exists and frees the stack used by the process. The further action taken by `kill` depend on the process' state (handle waiting semaphores, dequeue the process), finally the process state is set to `PRFREE`.

²¹„Wie sagte man früher im Wilden Westen? Der Prozess ist **vogelfrei!**“

5.4 Kernel Declarations

In der Datei `kernel.h` werden zentrale, Betriebssystem-weite Systemkonstanten (Beispiele: Abkürzungen für Register, Typen, Standardwerte, ...) und Routinen (Inline-Prozeduren, u.a. `asm`-Befehle) definiert. Dabei werden die Inline-Funktionen nicht als normale Funktionen deklariert, um damit das Erstellen und Löschen von Stackeinträgen etc. zu sparen, diese Lösung ist effizienter.

5.5 Process Creation

Ein neuer Prozess wird mit dem Systemaufruf `create` erschaffen, es erfolgt die Zuweisung aller benötigten Ressourcen und Daten (entsprechend den Parametern des Systemaufrufs). Es wird eine neue `pid` vergeben, es wird der Eintrag in der Prozesstabelle angelegt, der Stackbereich wird reserviert und Variablen wie Priorität, Name, Anzahl der Argumente werden festgelegt. Der neue Prozess kann nicht sofort `CURRENT` werden, sondern muß mit `resume` gestartet werden.

Nicht alle Prozesse werden vom Benutzer einzeln getötet, *Problem*: wie bereitet man den **natürlichen Tod** eines Prozesses vor? *Antwort*: indem der Prozeß beim letzten `return` nicht „irgendwohin“ zurückkehrt, sondern *anständig weggeräumt wird*. Mit der vorgestellten Lösung werden die C-Konventionen beachtet, es wird ein Prozeduraufruf simuliert (*Pseudocall*), dessen initialer Stack bereits den Konventionen gehorchen muß und entsprechend gefälschte Rücksprungadresse (zu `userret()`) und Argumente enthält.

Der Code von `create` ist in drei Phasen gegliedert: Stackbereich reklamieren, Daten u.a. in die Prozesstabelle eintragen und `pid` zurückgeben.

6 Process Coordination

Prozeßkoordination ist notwendig um Aktionen von Prozessen zu synchronisieren und um den Zugriff auf gemeinsame Ressourcen zu regeln, beispielsweise beim Producer-Consumer-Problem oder bei Mutual Exclusion. Die einfachste Möglichkeit zur Prozeßkoordination bieten Semaphore.

6.1 Low-Level Coordination Techniques

6.2 Implementation of High-Level Coordination Primitives

Jede Semaphore s ist im Prinzip ein Integer-Wert. Der Systemaufruf `wait(s)` reduziert s um 1, `signal(s)` erhöht s um 1. Wird s beim Aufruf von `wait(s)` negativ, wird der Prozeß angehalten; beim Aufruf von `signal(s)` wird ein Prozeß, der auf s wartet, aufgeweckt. Die XINU-Implementierung vermeidet *busy waiting* durch Einführung eines neuen Prozeßzustandes `PRWAIT`. Zu jeder Semaphore gehört eine eigene Warteschlange.

A nonnegative semaphore count means that the queue is empty, a semaphore count of negative n means that the queue contains n waiting processes.

6.3 Semaphore Creation and Deletion

7 Message Passing

Interprozesskommunikation ist ein wichtiges Mittel in Betriebssystemen, z.B. shell-pipeline in Unix (IPC zwischen Benutzerprozessen). Verschiedene Mechanismen zur Implementierung sind beispielsweise Semaphore zur Koordination/Synchronisation; Monitore; gemeinsame Variablen oder *message passing*. Anders als Synchronisation durch Semaphore kann message passing unsynchronisiert sein; es wird implementiert durch die Systemaufrufe `send(...)` und `receive(...)`.

Senden und Empfangen können blockierend (beide blockierend: *rendez-vous*) oder nicht-blockierend sein; die Kapazität der Verbindung (Pufferlänge) spielt eine Rolle (was passiert, wenn der Puffer voll ist?); sollen Nachrichten fester oder variabler Länge gesendet sein; soll es möglich sein, mehrere Prozesse als Empfänger anzugeben?

In XINU existieren zwei Formen des Nachrichtenaustauschs: 1. direkt von Prozeß zu Prozeß, 2. über Rendezvous-Punkte. Dabei sind `send` und `recv1r` asynchron, während `receive` synchron arbeitet. In XINU existiert nur ein Empfangspuffer der Länge eins, d.h. nur die erste Nachricht wird übertragen. Zudem wird ein neuer Prozeßzustand `PRRECV` eingeführt.

Das Senden signalisiert einen Fehler, falls Prozeß-ID nicht paßt oder der Empfangspuffer bereits voll ist. Das Asynchrone Empfangen ist ähnlich dem synchronen Empfangen, falls die Nachricht vorhanden ist: Rückgabe der

Nachricht, sonst OK. Beim synchronen Empfangen wird der Prozess ggf. auf den Prozeßzustand `PRRECV` gesetzt und wartet zunächst, bis er wieder von `send` aufgeweckt wird.

8 Memory Management

9 Interrupt Processing

9.1 Dispatching Interrupts

Two-Level-strategy to handle interrupts:

1. interrupts branch to low-level **interrupts dispatch routine** written in assembly language, they
 - save and restore R_0 and R_1
 - identify the interrupting device
 - handle return from interrupt
 - and call the second level:
2. **high-level interrupt routines** (for r/t, i/o passing, disc processing), written in C

In XINU there are three interrupt dispatchers²²: clock interrupts, input- and output-interrupts.

How is the interrupting device identified? At the interrupt vector address the PC (address of interrupt dispatch routine) and the PS (0341 for device 1, 034*i* for device i ($i = 0, \dots, 15$)) is stored, the PS is used to identify the device.

9.2 Input and Output Interrupt Dispatchers

The interrupt dispatchers assume that the PC and the PS are on top of the stack upon entry. Low order 4 bits of the current PS contain the device descriptor. Interrupts are disabled.

²²Man sollte immer auch den Text lesen, der Text ist immer besser als mein Vortrag - das ist der Nachteil, wenn man ein gutes Buch liest.


```

_outint:
    mfps -(sp)          // get new PS to identify interrupting device
    mov  r0,-(sp)       // save R0 (because csv does not)
    mov  $_intmap+4,r0  // point R0 to output in intmap
    br   ioint         // go to common part of code
_inint:
    mfps -(sp)          // get new PS to identify interrupting device
    mov  r0,-(sp)       // save R0 (because csv does not)
    mov  $_intmap,r0    // point R0 to output in intmap

ioint:
    mov  r1,-(sp)       // save R1 (csv does not)
    mov  4(sp),r1       // get saved PS in R1
    bic  $177760,r1     // translate 034i to 000i
    ash  $3,r1          // shift left 3 times to get (i*8) to correctly
                        // jump in the interrupt table, which uses 8 bytes
                        // per interrupt
    add  r1,r0          // shift pointer in interrupt table i*8 bytes
    mov  2(r0),-(sp)    // push icode or ocode from intmap as argument
    jsr  pc,*(r0)       // call high-level interrupt routine

// here: just returned from high-level interrupt routine
    mov  2(sp),r1       // restore R1 and R0 from stack
    mov  4(sp),r0       //
    add  $8,sp          // delete stack frame: arg, saved R0, R0 and PS
    rtt                 // return from interrupt

```

9.3 The Rules for Interrupt Processing

Interrupt routines examine and modify global data structures, so **interference from other processes must be prevented**. This is essentially done in two ways:

- disabling interrupts, with interrupts still disabled after interrupt routine returns; so only after `ioint` returns interrupts enabled again.
- disabling interrupts, but: high-level interrupt routines may **enable interrupts** by indirectly calling `resched`, in case the CPU switches to a process with interrupts enabled!

Interrupt routines should leave global data in a valid state before calling `resched()`. No procedure enables interrupts unless it previously disabled them (when calling `resched()`).

10 Real-Time Clock Management

10.1 The Real-Time Clock Mechanism

Three kinds of clocks associated with a computer²³:

1. *central system clock*: controlling the rate the CPU executes instructions (belongs to MHz or FSB or so)
2. *real-time clock*: pulsing regularly an integral number of times each second, signalling the CPU each time a pulse occurs by posting an interrupt
3. *time-of-day clock*: a chronometer like a watch, the CPU controls this clock

The *real-time clock* does not contain a counter (con to the *time-of-day clock*), does not accumulate interrupts (left to the system) and controls the CPU. Responsibility for counting interrupts falls upon the system:

If the CPU takes too long to service a real-time clock interrupts, of if it operates with interrupts disabled for more than one clock cycle, it will miss the interrupt.

So systems must be designed to **service clock interrupts quickly**.

10.2 Optimization of Clock Interrupt Processing

Effective clock ratio has to be adjusted to match the system - how is this done in XINU? The clock interrupt handler/dispatcher `clkint` simulates a slower-rate clock by dividing the clock rate: The LSI 11/2 *real-time clock* generates 60 pulses per second and `clkint` ignores 5 clock interrupts in a row by serving them not at all (very quickly!) **before processing the 6th one**. This reduces the *effective* clock rate, called the *tick rate*.

10.3 The use of the real-time clock

Operating Systems use real-time clocks to compute the time-slices allotted for the execution of each process by scheduling a pre-emption event. This event is used to prevent processes from running forever. It is set in `resched()` by `preempt = QUANTUM`; (see page 58) The clock interrupt dispatcher `effcint`

²³zu Hause angucken: Befehle `mov ri, -(sp)` und `mov (sp)+, ri` für $i = 0,1$ in `clkint.s`, möglicherweise wichtig für Klausur!

decrements preempt on each tick, calling `resched()` when `preempt = 0`.

The Operating System also provides processes with timed delays: The system maintains a list of processes, ordered by the time they should be awakened. When the real-time clock interrupts, it examines this list and wakes up processes for which the delay has expired. A preemption

10.4 Delta List Processing

Because it cannot afford to search through long lists of sleeping processes to find those that should awaken on each clock tick, the system keeps sleeping processes in a data structure called a *delta list*:

Processes on `clockq` are ordered by the time at which they will awaken: each key tells the number of clock ticks that the process must delay beyond the preceding one on the list

Example waiting times: $P_1 = 5$, $P_2 = 27$, $P_3 = 28$, $P_4 = 28$, $P_5 = 35$. The Q-Structure used to save this data:

head \longrightarrow $P_1 : 5 \longrightarrow P_2 : 22 \longrightarrow P_3 : 1 \longrightarrow P_4 : 0 \longrightarrow P_5 : 7 \longrightarrow$ tail

10.5 Putting a Process to Sleep

System call `sleep10(n)` delays the calling (current) process for n tenth-of-a-second, by moving the current process to the delta list `clockq`. This requires introducing a new process state for that moved process: `SLEEPING` (see figure 10.1).

10.6 Delays measured in Seconds

The size of integer (16 bits) limits the delay time of `sleep10(n)` to approx. 55 minutes. System call `sleep(n)`, which puts a process to sleep for n seconds, provides a way to delay up to 9 hours.

10.7 Awaking Sleeping Processes

that `clkint` decrements the count of the first key on `clockq` at each time, until this key equals 0 at which time the high-level interrupt procedure `wake-up()` is called, to put processes with `key = 0` on to ready list.

10.8 Deferred Clock Processing

The *deferred mode* allows the system to **accumulate clock-ticks** in var `clktick` without initiating events. I.e., it **postpones the reaction** upon realtime-clock interrupts. The clock handler can schedule events that *should have occurred* as soon as it leaves deferred mode and returns to original mode.

10.8.1 Procedures for Changing to and from Deferred Mode

A process can place the clock in *deferred mode* by calling `STOPCLK` and return the clock to real-time mode by calling `STRCLK`. `STOPCLK` counts deferral requests by incrementing `defclk`; `STRCLK` counts restart requests by decrementing `defclk`. As long as `defclk` remains positive, the interrupts handler counts clock ticks in `clktick` without processing them.

10.9 Clock Interrupt Processing

10.10 Clock Initialization

To determine if the system has a *real-time clock*, `setclkr()`; runs through a loop 30.000 times. If there's a *real-time clock*, it should at least once **interrupts this loop**²⁴. So the interrupt handler is temporarily overwritten with a procedure which **enables the clock**.²⁵

10.11 Summary

11 Device independent Input and Output

Operating Systems control the I/O-devices for three reasons:

1. The hardware interface to such devices is crude, requiring complex software packets for their control, called *device drivers*.
2. Device drivers are **shared resources**, which need to be protected and allocated in a *fair* and *safe* way.
3. A uniform, flexible interface should be provided, a **high-level interface**, allowing users to write programs without knowing the machine configuration.

²⁴„Das ist das Schöne an System Programmierung: Die richtige Lösung ist immer kurz.“

²⁵„Also ein bißchen studieren zu Hause und Kai Baukus lieb angucken. Er guckt schon dunkel, er kann's auch nicht lesen.“

Focus of this chapter: The selection of a set of **machine-independent high-level I/O primitives** and the **data structure** required to implement these primitives to *specific devices*.

How are these primitives selected? By generating a list of desirable properties, deriving a set of high-level primitives, and giving their meaning with respect to certain abstract (classes of) devices, for example terminals, discs etc. The last step is to build software mapping the abstract devices to particular instances of that device.

11.1 Properties of the input and output interface

Question: Should processes **block** while performing I/O operations (*synchronous operations*) or should they continue executing and be notified when the operation completes (*asynchronous operations*)? Asynchronous operations are useful for controlling overlap, i.e. more parallelism of comp and I/O operations. Synchronous operations delay *input* operations until data arrives and delays *output* operations until data has been consumed. Their advantage is that users can depend on data immediately after an input operations and change data immediately after an output operation.

Also a question: which format does the data have? *Single-byte transfer* (teletype terminals, e.g. console) or *block mode* (a block of many bytes like on a disk)?

11.2 abstract Operations

- `getc` and `putc` deal with single-character transfer (read or display *one* character)
- `read` and `write` deal with transfer to/from contiguous blocks of memory
- `control` allows control of the device (driver), e.g. whether the system *echoes* each character as it is typed on the keyboard
- `seek` applies only to randomly accessible memory and then searches for a particular position
- `open` and `close` inform device (driver) that data transfer will begin or has ended (applies to disk and file access)
- `init` initializes the device and device driver at system's startup

11.3 Binding abstract Operations to Real Devices

The system maps these high-level I/O operations to specific device drivers: it **hides** the details of the hardware and the device drivers and it makes the programs **independent** of the particular hardware configuration.

The high-level calls of these operations constitute the environment which the system presents to running programs - i.e. the programs only see the peripheral devices through these abstract calls.

The system also maps *abstract names* to real devices.²⁶

Coded into the system is a description of each abstract device, e.g. the device driver routines which it uses, the address of the real device to which it corresponds. When a new device is added to the system, or, e.g., the device addresses are modified, the system must be **altered and recompiled**.

11.4 Binding I/O calls to Device Drivers at Run-Time

Rotines like read in the compiled code should map abstract device descriptors, s.a. console, to device driver routines and real device addresses. In XINU, each abstract device is assigned an integer device descriptor (0, ..., 8) at system configuration. E.g., console has the **same device descriptor** in all XINU systems.

In the **device switch table**, each entry corresponds to a *single device*, containing:

- *dvnum*: the corresponding entry into the interrupt dispatch table *intmap*
- *address of the device driver routines* for that device (*dvgetc*, *dvputc*, *dvread*, *dvwrite*, *dvcontrl*, *dvseek*, *dvinit*)
- *device address* and *other info*, since more than one device can use the same device driver

Device switch table also contains²⁷:

- hardware device addresses (*dvcsr*)
- interrupt vector addresses (*dvivec*, *dvovec*)

²⁶Zur seitenlangen Liste in *conf.h*: „Ich werde keinem den Kopf abhacken, wenn er diese Liste nicht kann in der Prüfung!“

²⁷Zu einem, der gesappelt hat: „Dann geh mal zu einem anderen Professor, so Kluge zum Beispiel, da bekommst Du gleich 'ne Fünf!“

- the interrupt routines for input (`dviint`) and output (`dvioblk`)
- buffer pointer `dvioblk`
- an integer `dvminor` distinguishing among multiple copies of a device²⁸

11.5 Implementation of high-level I/O-operations

There is a procedure for each of the abstract operations `getc`, `putc`, `read`, and so on. They call low-level device drivers indirectly through the device switch table. For example, the C code in file `read.c` implements the `read` operation:

```
read(int descrp, char *buff, int count) {
    ...
    devptr = &devtab[descrp];
    return ((*devptr->dvread)(devptr, buff, count));
}
```

11.6 Opening and Closing Devices

Some disk devices require the programs to start them before performing a transfer operations, and to stop them when the transfer completes.

11.7 Null and Error Entries in Device Table

Not all combinations of operations and device descriptors are meaningful:

- `ionull()` signifies an unnecessary byte (e.g., open a console), returns `OK`
- `ioerr()` signifies an illegal operation (e.g., seek on a console), returns `SYSERR`

11.8 Initialization of the I/O System

The device table `devtab` is generated when the system is configured, so it is completely filled in by the time the system is compiled. Generating the file `conf.h` is discussed in chapter 20.

²⁸„Ich habe am Wochenende „Superstar“ geguckt. Ich habe gemerkt, daß ich eine laute Stimme habe - schade, daß ich, als ich jung war, nicht mitgemacht habe. Also: Ich wäre ein guter Sänger gewesen.“

11.9 Interrupt vector Initialization

Interrupt vectors and the interrupt dispatch table²⁹ `intmap` are initialized at runtime, using the information in `devtab`. The system calls `init(k)` for each device k at startup, before it starts executing the user's program (done in chapter (13)).

The device table forms the general framework for linking interrupts, devices and device driver routines.

12 An Example Device Driver

This chapter discusses the *device driver routines* for a **standard computer terminal with a keyboard**, called teletype or `tty`.

12.1 The device type `tty`

To minimize the interference between I/O devices and running processes, the driver uses *interrupts-driven processing* to

- transmit characters when serial line unit (SLU) is idle
- read characters when these are received by the unit
- handle details like receiver errors

The `tty` driver operates using parameters, so it can be used for a variety of terminals in a variety of system configurations. E.g., several parameters control the *echo* of characters typed on the keyboard onto the screen (*full-duplex mode*: do not display characters as the user types them; *half-duplex mode*: directly echo keystrokes). There are other parameters concerning unprintable chars as printable combinations (to move to a new line, terminal must receive *both* `newline` and `return`).

12.2 Upper and Lower Halves of the Device Driver

A device driver is a set of procedures controlling a peripheral hardware device. Its routines are partitioned into

- *upper-half device driver*: called from user programs

²⁹„Pro Minute für Studenten: 10 EUR. Mein Gehalt staffelt sich nach den Minuten, die ich vorlese. Ein bißchen dumm, aber so ist das deutsche Gesetz.“

- *lower-half device driver*: handling device interrupts

These two halves communicated via a shared data structure, the device control block.

Upper-half routines enqueue requests for data transfer or device control; they do not interact with devices directly. Lower-half routines transfer data from buffers or control devices; they do not interact with user programs directly.

12.3 Synchronization of the Upper and Lower Halves

Output operations issued by the calling user process are deposited as to be written characters in the output buffer, and return to the caller.

Whenever the SLU receiver interrupts after it has received a character, the interrupt dispatcher calls the lower-half input interrupt routine. The lower-level interrupt input handler reads the waiting character and deposits it in the circular buffer. A process waiting for input from the (empty) circular buffer is started as soon as the next character arrives.³⁰

12.4 Control Block and Buffer Declarations

Both lower-half of the device driver and the ... use the minor device number as an index in the array of `tty`-structures.

12.5 Upper-Half `tty` Input Routines

12.6 Upper-Half `tty` Output Routines

12.7 Lower-Half `tty` Driver Routines

12.7.1 Watermarks and Delayed Signals

12.7.2 Lower-Half Input Processing

Input interrupt processing is more complex because it cares for character echo, line editing, processing of input errors and buffer overflow. It operates in one of the following three modes:

- **raw**: accumulates chars in input buffer without further processing

³⁰ „... Signatur „Rembrandt Kai Baukus“ ...“

- **cooked**: does character echo, honors suspend or restart output, accumulates full lines before passing them on to upper-half routines, honors input editing by erasing previous chars or killing entire lines
- **cbreak**: honors control chars but does no line editing

12.7.3 cooked Mode and cbreak-Mode Processing

12.8 tty Control Block Initialization

12.9 Device Driver Control

12.10 Summary

13 System Initialization

Many microcomputers require no more than what's been discussed in chapters 1 to 12.

How does CPU know when a character is received in RBUF-field of CSR, or when a character has been sent after having been placed in XBUF-field of CSR? It uses one of two techniques: *Polled I/O* or *interrupts-driven I/O*. Polling is used for initialization or debugging because based on busy-waiting: It requires CPU to check device repeatedly until it finds a character is waiting or has been sent.

- **RCSR**: Bit 7 whether character received along Serial Line Unit, Bit 6 Serial Line Unit will post interrupt when character received in RBUF
- **XCSR**: Bit 7 when character has been transmitted, Bit 6 Serial Line Unit will post interrupt when character has been sent, Bit 0 transmitter is forced in BREAK condition; must be set to 0 to clear break condition

13.1 Starting from Scratch

A crash occurs when hardware executes an ivali operation, caused because code or data in the OS has been destroyed. A crash means the contents of memory have been corrupted or lost!

*How can a machine, devoid of valid programs, spring into action and begin executing? **It cannot!***

Somehow a program must be deposited in memory before the machine can start. On the oldest cptrs this happend **by hand**, using switches. Later

standard keyboards built to that purpose, i.e. special terminals, now micro- and mini-computers are used to load the initial program from disk or tape storage attached to the system.

Once the initial program has been loaded, the CPU can execute the startup program which reads a larger program (usually from a specific location on a specific disk). Then the CPU branches to a larger program which reads the entire OS into memory and branches to the later's beginning. This process is called **rebooting the system**.

The main goal of the chapter is to explain the steps necessary to transform the single, sequential program into an operating system.

13.2 Booting XINU

XINU is downloaded from another computer in the following steps:

1. Host computer generates a break condition to halt the 11/2 processor.
2. 11/2 responds in Octal Debugging Technique (ODT) mode; it sends a prompt and recognizes commands to display and change memory locations and registers.
3. Hosts loads *initial program* in 11/2 memory starting at *location 0*, and starts 11/2 executing it.
4. initial boot program reads characters, using polled I/O, and deposits them in memory starting at *highest location*. Host sends *second boot program* to 11/2.
5. When finished, host sends a break, forcing 11/2 into ODT mode.
6. When ODT responds, hosts starts executing on 11/2 of second boot program.
7. Host and second boot program communicate, with the host sending „packtes“ of bytes (one-at-a-time), and boot program acknowledging receipt or requesting transmission
8. Host (either) tells second bootstrap *to branch to the start of XINU* (or to halt and await ODT commands)
9. Host directs second bootstrap to branch to XINU, and CPU begins executing *start* program.³¹

³¹Zu einem, der zu spät kommt: „Noch so ein Kraftprotz, der alles schon kennt“

13.3 System Startup

The startup program³²

13.4 Finding the size of Memory

At label `start`, XINU is creating a valid runtime environment for *C* by setting Stack Pointer to the highest available Memory Address. This is done by try-and-error (with interrupts).³³

13.5 Initializing System Data Structures

13.6 Transforming the Program into a Process

13.7 The Map of Low Core

13.8 Summary

17 A Disk Driver

At the level of device driver routines, a disk is viewed as a *randomly accessible array* of blocks (here, of size 512 bytes). The interaction between driver software and the disk device itself is more complicated than for a serial line unit. Therefore, an extra controller (disk-controller) is needed with its own format of *controller request records* (see page 34, figure 2.11).

A disk is a random access device. Therefore, a disk must *position the disk arm* as well as *transfer data*. Disk hardware uses *direct-memory-access mode* (DMA), allowing for the direct transfer to/from memory blocks without help from the CPU. So it does not interrupt the CPU for the transfer of every single character. Hence CPU can continue operation while block transfer takes place.

A disk consists of

- a disk drive (i.e., platters (spindle) which rotates at a high speed, and a physical arm)
- an electronic controller (contains microprocessor(s), positions disk arm, controls transfer of data, can operate multiple disk drives (but not able to transfer data to/from one diskplatter simultaneously))

³²„Was ist das für eine Zahl? Das ist keine Macht von Zwo!“

³³Machst Du auch so einen Kurs, wo man lernt, wie man Männern in die Eier tritt? Das muß man nämlich lernen!

- a host interface (connects controller to system bus, passes requests for an I/O operation from processor to controller)

Disk devices are slow and awkward, compared with main memory operating at speeds measured in nanoseconds. Disk host interface starts at address 0177460 beyond „real“ memory. It has six word format, incl. four essentially used words:

Name	Contents
CCSR	Completion status Register
CSR	Control & Status Register
DAR	DMA Address Register
CAR	Control Request Address Register
XDAR	Extension for DAR (not used)
XCAR	Extension for CAR (not used)

The Control and Status Register (CSR) contains bits for: *P* (Parity error, bit 13), *D* (Operation completed, bit 7), *I* (Interface will interrupt when operation completed, bit 6), *G* („go“, bit 0) and other ones.

17.14 Summary

1. Lower-Level disk driver implements four operations. Upper-half routines merely enqueue requests for service. Whenever an operation (data transfer) completes, lower-half routines takes next pending request from queue, starts up hardware performing that operation, and *wake up* process waiting for previous operation to end.
2. Driver reduces time to honor requests be *reordering* them to minimize arm movement.
3. Because copying data for each transfer takes to much CPU time, the driver accepts *output* requests, and returns asynchronously to caller - *without having copied the data in the output butter into the system buffers* - a **dangerous** strategy!

17.1 Operations supplied by the Disk Driver

At the device driver level, a disk is nothing more than a large array of data blocks that can be accessed. Randomly using three basic operations: Select a block (**seek**), copy the contents of the selected block from disk to memory

(**read**) and copy the contents of memory to the selected block on disk (**write**).

Format of instructions: `read(DISKDEV, buff, blocknr)` means „read disk block with nr. `blocknr` into memory starting at address `buff`“. So, why is operation `seek` needed? To optimize disk access!

17.2 Controller Request and Interface Register Descriptors

The *Xebec Disk Controller Layout* is described by structure `xbdc` on page 284 and 285, corresponding to figure 2.11. Notice the many possible operations listed on page 285! For us, only `XOREAD`, `XOWRITE` and `XOSEEK` are interesting. The *disk controller host interface* on page 286 parallels figure 2.12. Notice that the bits in figure 2.13 correspond to operations `DTGO` (set bit nr. 0), `DTRESET` (bit 1), `DTINTR` (bit 6), `DTDONE` (bit 7) and `DTERROR` (bit 15).

17.3 The List of pending Disk Requests

For disks, the required info for transfer between upper- and lower-level routines resides in control blocks. Disk driver control blocks: `dstab[]`, consisting of structures `dsblk`, containing:

- `dvioblk`: should contain address of control block
- `dcsr`: should contain address of host interface
- `dreqlst`: `DRNULL`
- `dibsem`: semaphore dealing with access to the *index block list* in file system
- `dflsem`: semaphore dealing with access to the *free list of data blocks* in file system
- `ddirsem`: semaphore dealing for *directory access* in the file system
- `dnfiles`: number of open files, initially 0
- `ddir`: address for incore memory for the data directory: `getbuf(dskdbp)` (disk data block buffer pool)

17.4 Enqueueing Disk Requests

The disk driver maintains the following invariant:

The first request on the pending request list is always the one the hardware is performing, if the list is empty, the hardware is idle.

The lower-half records info about errors occurred, the upper-half extracts that info and passes it to the caller.

*The disk driver processes I/O requests in fht order in which they occur on the request list. When one completes, it is removed from the list and the next one is started. When procedure `dskeng` adds a request for block *B* to the existing list of requests, it schedules it to be performed **between requests *i* and *i* + 1**, if the disk arm will pass over block *B* **on its way from *i* to *i* + 1**.*

17.5 Optimizing the Request Queue

In case of a read operation to be enqueued for which there is a write for the same block in the list of pending requests, the read operation. . .

17.6 Starting a Disk Operation

Procedure `dskstrt` on page 294 starts a disk operation by building a controller request structure in the `ddcb` field of the disk control block, feeding that request to the controller through the interface.

17.7 The upper-half read Routine

17.8 Driver Initialization

Procedure `dsinit` fills in the disk control block, fills in the corresponding *interrupts vectors* and the corresponding entries into the interrupt dispatch table by calling `iosetvec`, sets up a read command for the desk directory to be read. Because the system executes the initialization routine before interrupts are enabled, disk initialization runs in polled mode.

- 17.9 The upper-half output Routine
- 17.10 The upper-half output Routine
- 17.11 The upper-half seek Routine
- 17.12 The lower-half of the Disk Driver
- 17.13 Flushing Pending Requests