Betriebssysteme

Operating Systems

and some other things

Peter B. Ladkin

Administrative Details

`ladkin@rvs.uni-bielefeld.de`
`http://www.rvs.uni-bielefeld.de/~ladkin`
Sprechstunden: Mi 1100-1200
D6-131

Tutors: Michael Blume, Dirk Henkel,
Michael Höhl, Karsten Loer:
`mblume@rvs.uni-bielefeld.de`
`dhenkel@rvs.uni-bielefeld.de`
`mhoehl@rvs.uni-bielefeld.de`
`karlo@rvs.uni-bielefeld.de`
Times and Places:
Mo 16-18; Do 16-18; Fr 8-10; 13-15.
We take the first twenty per session.
Anmeldungsformular: See the course WWW page

Course Page and Notes: On the WWW,
accessible from my home page,
and the `rvs/lehre` page
in `dvi` and `ps` form.

What will you have to learn?

Use of the WWW. This is very easy

Simple use of the document processing system LaTeX. This is easy, but not *very* easy.

Simple use of the specification language TLA+, a machine-independent way of describing distributed and concurrent algorithms for operating systems and other uses

Both LaTeXand TLA+ are designed by Leslie Lamport.

What is a *distributed* algorithm?

It runs on more than one physical machine, and the machines are usually some distance away from each other

What is a *concurrent* algorithm?

One which is divided into parts which run *simultaneously* (virtually or actually)

Operating systems are full of concurrent algorithms and programs

Using the WWW

Log in, use command `Mosaic`

Much better is `netscape`
but it might not work for you. If not,
use `/vol/netscape/bin/netscape`
on the `techfak` net

Click on `Open`
An address window will appear

Fill in the address window with
`http://www.rvs.uni-bielefeld.de`
(as above) to get our home page.
Click on a *sensitive word* to
follow a link.
Follow `courses and seminars` then
Betriebssysteme to get the notes

After a while, you'll want to write your own WWW pages. This is also easy.

The language is called `html`

The easiest way to write it is:
look at and copy someone else's WWW page (this is trivial through `netscape`'s `View` menu.)
Then put in your own text.

But you don't **need** to do this for this course, although you will need to do so for any future work with computers

You **will** need to know a little LaTeX.

LaTeX is a *markup language*, that is, the formatting instructions are included in the source itself along with the text

Here's the text for this page:

```
\begin{slide}
Here's the text for this page:

\begin{verbatim}

.......

\end{ verbatim}

(Actually, I cheated)

\end{slide}
```

(Actually, I cheated)

Here's how a document looks in LaTeX

```
\documentclass{article}
\usepackage{tla,rawfonts}
\author{Not Me}
\title{A Short History of the World}

\begin{document}
\maketitle

\section{The Parable}
Once upon a time, there were no computers.
Life was
\begin{itemize}
\item Simple
\item Brutish
\item Short
\end{itemize}
rather like the people

\end{document}
```

Here's (roughly) how it looks compiled:

A Short History of the World

Not Me

1 The Parable

Once upon a time, there were no computers. Life was

- Simple
- Brutish
- Short

rather like the people

So, to get you using LaTeX :

Exercise 1.
Find and List all the **Keywords** (Schlüsselwörter) for important concepts in operating systems that occur in Chapters 1-9 of Galvin/Silberschatz or Tannenbaum or equivalent, and arrange them in a hierarchy.

Classify them into

- Data Structures (e.g. Files)
- Objects (e.g. Processes)
- Methods (e.g. Remote-Procedure Call)
- Features (e.g. Concurrency)
- Any other class you can justify
  (with your justification)

Organise them into a tree of *subclasses*
(e.g. is RPC a form of *communication*?)

Write your answer in LaTeX

Warning: When you do a Prüfung,
I'll ask you about the concepts that
weren't on your list .....

You'll find **exercises** by looking at the Course page. The other exercises will lead you to specify a *round-robin time-slice scheduler* in TLA+. This will involve understanding and using semaphores as the synchronisation primitive, and structuring your specifications in modules, just like a program. In fact, using TLA+ is rather like programming - you write descriptions of state machines - but in logic.

Hint for Exercise 1:
You can use `itemize` environments
inside other `itemize` environments, like:

```
\begin{itemize}
\item One thing
\begin{itemize}
\item Lots of little things
\item More little things
\end{itemize}
\item One more big thing
\end{itemize}
```

This gives

- One thing
  - Lots of little things
  - More little things
- One more big thing

Now for some stuff about
operating systems

What is an operating system?
What does it do?

An operating system is a program or a collection of programs that

- allows the computer to look as if it's a much more sophisticated virtual machine than the 'raw' processor (CPU)

- allows all the various hardware bits and pieces to work together to get their respective jobs done

- organises the data and program structures on the hardware

In the beginning

Switch the PC on. What happens?

A program runs. It checks the *hardware*.

Hardware: Floppy disk, hard disk, memory.
Then you get an MS-DOS prompt.

The program that checks the hardware
and gives you the prompt
and arranges for commands to be
executed is the *operating system*.
It also does other things.

Another story

Switch the Sparcstation on.
What happens?

A program runs. It checks memory, file system, network services, ...

File system? Network Services?

What you see and what's available when you turn the computer on depends upon what **operating system** the computer is running. Here, *Solaris*, which is a version of *Unix*, a very popular system for scientific and technical computing, and the system on which the Internet grew up (mostly).

Processor

Central Processing Unit, CPU.
A piece of hardware that
executes an instruction.

Instruction operates on *data*.
Data may reside locally in *registers*,
or in a *cache* elsewhere in the chip,
or further away in *memory*,
or even further away on *disk*.

It *adds*, *multiplies* (unless it's a Pentium)
puts a value in memory,
gets a value from memory.

To learn something about operating
systems, we must learn a little about
how a processor works

What's the difference between an operating system and other user software?

You, the user

save **files**,
send an **email message** to a colleague
    on another machine
compile a program
    while reading your mail,
call the Web pages
    from a machine in California
(say, to learn about *TLA*)

Some of these actions are system actions, some are applications software actions, and whether, say, a window system is OS or application really depends on *convention*, that is, how the structure of the system was conceived by its designers - although there is a lot of agreement.

How is all this done?

The fundamental concept is that of a *virtual machine*.

The processor is a (real) virtual machine, with limited capabilities.

An operating system such as Unix gives *interprocess* and *intermachine* communication capabilities. That's a more sophisticated virtual machine.

The window system, which organises the user interface, lies 'above' that.

One can view the various virtual machines as arranged in a *hierarchy* of levels of increasing functional sophistication, in which a virtual machine at Level n provides functions which are used as 'primitives' by Level $(n+1)$, and which itself uses the functions provided by Level $(n-1)$ as primitive.

The bottom of the hierarchy:–

The processor '*knows*':
`get`, `add`, `multiply`, `store`
bytes from *memory* and *registers*.

Data structures: bytes.

Near the top of the hierarchy:–

The networked machine '*knows*':
`verify` password, `store` files
(on another machine),
`call procedure` (from another machine),
`start` applications (editor, mailserver),
`save state` of application,
`swap` one running program for another,
`kill` application.

Data structures: files, memory locations, byte streams, messages, control blocks, Ethernet addresses, address tables, ports, Internet addresses, passwords, login names,.......

Many structures in computer engineering are based on this idea of a hierarchy of virtual machines

The ISO Open Systems Inteconnection standard for inter-machine communication is based on seven layers of protocols. The TCP/IP packet-switching protocols on which the Internet is based have fewer.

The *PSOS* (Provably Secure Operating System) and the SIFT OS for digital flight control (both SRI, 1970's-80's) were based on hierarchical design so that they could be *verified* (proved correct) by SRI's *EHDM* (Extended Hierarchical Development Method) verification system

Hierarchical decomposition is still the most fruitful way of proving algorithms and designs correct, and systems such as PVS (from SRI) and TLA depend on it

The lower-level Virtual Machine
implements
the higher-level Virtual Machine

How?

The LLVM *simulates* the HLVM.

- Define higher-level data structures 'in terms of' lower-level data structures:
  words in terms of *bytes*,
  arrays in terms of *words*
  sets in terms of *arrays*

- Define higher-level actions
  'in terms of' lower-level actions:
  send-message in terms of
  a *C program*

(This technology is also used
when building compilers.)

The definition of higher-level functions
in terms of lower-level functions is more
accurate if it is done **rigorously**.
We use mathematics, especially logic
and set theory, or algebra.

I like to use the *Temporal
Logic of Actions* (TLA) of Lamport

TLA is a logic which includes
set theory, to describe mathematical
properties of program variables,
and operators to describe preconditions
and postconditions of actions, as well as
properties that hold for all possible
executions of the program

A TLA program specification

- desribes the possible actions in Formal Logic

- describes properties of the machine:

  - Initial (Starting) Condition

  - Safety (the only possible actions are those described)

  - Liveness (if certain actions *can* happen, eventually they *will* happen)

Program specification and verification in TLA proceeds as follows:

We specify both the HLVM and LLVM, then **prove** using logic that

LLVM.Spec $\Rightarrow$ HLVM.Spec

There is a *formal logic* for doing that supported by

- a proof system (a set of rules)
- a specification-writing $\LaTeX$ style file
- a proof-writing $\LaTeX$ style file

We won't be using the proof system, but we *will* be using the language

In order to make full use of the computing power of the hardware on which they run, operating systems nowadays are (complicated or extremely complicated) concurrent programs.

A concurrent program is one which does (has the capability of doing) multiple tasks ‘at once’ (that is, a task may be started before other tasks already running have finished

We consider now how to write a concurrent program in TLA. Such a program describes a state machine

A TLA ‘program’ is a mixture of imperative commands (expressed as TLA *actions*)and assertions about the state of the machine.

We first write a simple concurrent program in a *procedural* language which allows concurrent programming and non-deterministic actions

It's a variant of a language due to E. W. Dijkstra, a pioneer in the logical and mathematical design of operating systems

Dijkstra also invented the *semaphore*, a construct used to make programming concurrent systems much easier

An example of TLA from Lamport, *The Temporal Logic of Actions*, ACM Trans. Prog. Lang. and Sys., 16(3), 872–923, May 1994.

Here is a program written in a simple Dijkstra-like procedural language

$$
\begin{array}{l}
\textbf{var} \quad \textbf{natural} \;\; x, y = 0\ ; \\
\textbf{do} \;\; \langle\ \text{true} \rightarrow x := x + 1\ \rangle \\
\quad\;\; \Box \\
\qquad \langle\ \text{true} \rightarrow y := y + 1\ \rangle \qquad \textbf{od}
\end{array}
$$

It means: x and y are natural numbers. In the initial state, they're both 0. A step of the program either increments x or increments y (it is not determined which) and this step is iterated forever. (The increment statements are conditional, but the condition is trivial.)

Let's now write a TLA description of this program. The goal is to ensure that anything that satisfies the TLA will do what the 'Dijsktra' program is supposed to do.

The TLA description uses two special symbols: $\Box$ and $'$

$\Box$ says '*for every state in the future*'
It's like a universal quantifier over future states.

If z is a program variable, then
z' is the value of this variable
at the next state.

A program *action* is specified in TLA by giving conditions on the current values of the program variables (the *preconditions*), and conditions on their values in the next state in which *some* program variable values have changed (the *postconditions*).

Note that the *next-change* state is not necessarily the *next* state!.

Since x and y are the two program variables, a program action will state logical conditions on x, y, x' and y'.

So here is (almost) the same program written in TLA

$$\begin{aligned}
Init_{\Phi} &\triangleq (x = 0) \land (y = 0) \\
\mathcal{M}_1 &\triangleq (x' = x + 1) \land (y' = y) \\
\mathcal{M}_2 &\triangleq (y' = y + 1) \land (x' = x) \\
\mathcal{M} &\triangleq \mathcal{M}_1 \lor \mathcal{M}_2 \\
\Phi &\triangleq Init_{\Phi} \land \Box\mathcal{M}
\end{aligned}$$

TLA is a logic, so these are logical formulas. There is a major difference distinguishing the TLA program from the 'Dijkstra' program. Can you see it?

There is another not so obvious difference. The 'Dijkstra' program is expected to run. It's in the meaning of the statements. But nothing in the TLA logic says that $\mathcal{M}$ has to do anything at all!

Φ asserts that the $Init_\Phi$ial condition holds, and that '*at all future states*' $\mathcal{M}$ holds, which is defined to mean that $\mathcal{M}_1$ or $\mathcal{M}_2$ holds, which means that x is incremented and y remains unchanged or that y is incremented and x remains unchanged.

But I said that a TLA specification says how x and y change at the next program change, rather than at the next system state change.

Incrementing x might take 4 processor steps. This specification could not describe such an implementation, because x does not change during the *next* step, but rather after three more (micro-)steps.

I introduce some more notation.
Firstly, the assertion
'$\mathcal{A}$ holds or f remains unchanged':

$$[\mathcal{A}]_f \triangleq \mathcal{A} \vee (f' = f)$$

So we can say
'$\mathcal{M}_1$ or $\mathcal{M}_2$ hold or the program variables x and y remain unchanged':

$$\begin{aligned} [\mathcal{M}]_{\langle x,\, y \rangle} &\equiv \mathcal{M} \vee (\langle x,\, y \rangle' = \langle x,\, y \rangle) \\ &\equiv \mathcal{M} \vee ((x' = x) \wedge (y' = y)) \end{aligned}$$

The (unlive) specification of the program Φ is now written thus

$$\Phi \triangleq Init_{\Phi} \wedge \Box[\mathcal{M}]_{\langle x,\, y \rangle}$$

But how do we ensure that actions
$\mathcal{M}_1 \triangleq (x' = x + 1) \wedge (y' = y)$
and
$\mathcal{M}_2 \triangleq (y' = y + 1) \wedge (x' = x)$
are actually carried out?????????

Each action is a combination of
program statement with **condition**

It's the purpose of specification to describe *what* a program shall do. The programmer must find a way of implementing the specification. But when it's found, it may be described in TLA also, and then *proven* to satisfy the specification

A short introduction to computer architecture

I describe here the structure of a *von Neumann* architecture, in which programs and data are stored, and programs are executed one instruction after another

Such a machine is called a *SISD* (Single Instruction stream, Single Data stream) machine

A computer has a Central Processing Unit (CPU) which does all the calculations.

It has local memory (called *registers*) in which all the arguments and values of calculations are held.

All the data that a program needs to use lies in *main memory*, which is divided into memory that can only be read (*Read-Only Memory*, ROM) and memory that can be read and written at any particular location at any time (*Random-Access Memory*, RAM)

In addition, a computer has much slower *secondary memory*, such as a hard disk or a (DAT streamer) tape, which stores much moredata than main memory, much cheaper, but is also much slower.

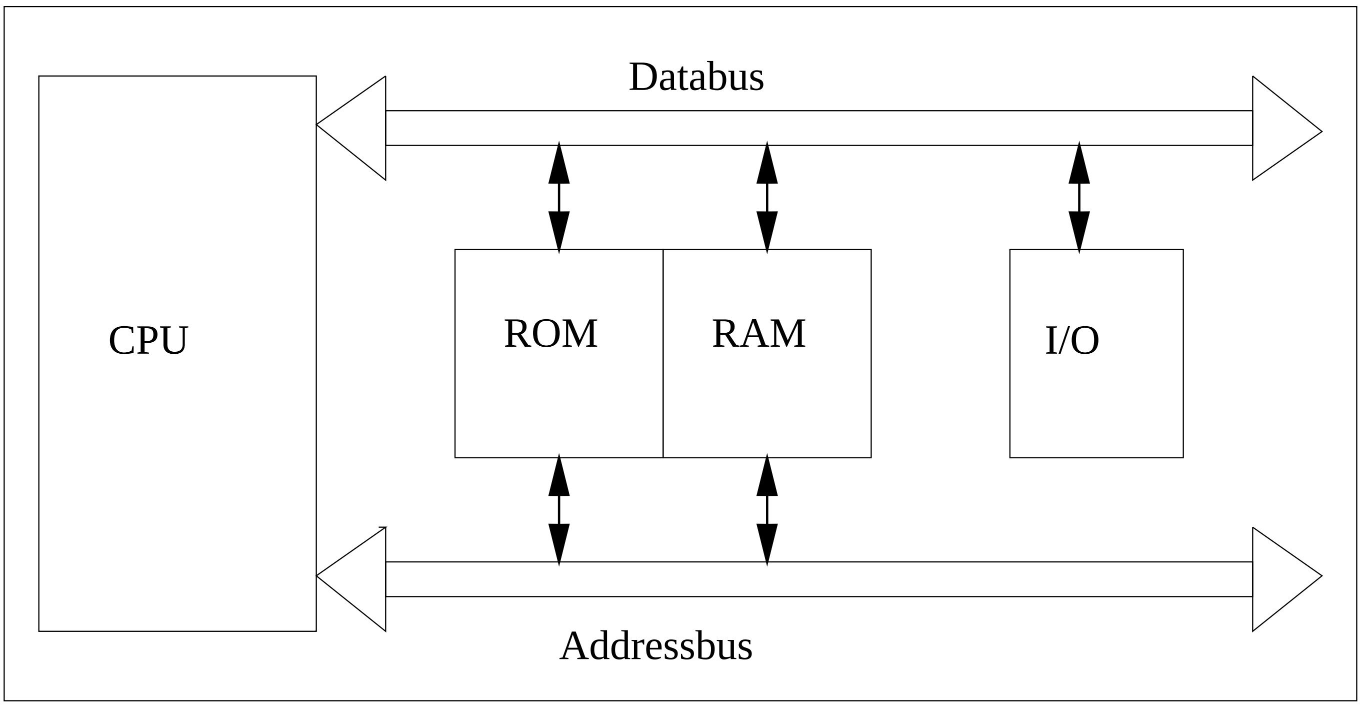
Data has to be transferred between secondary memory and main memory and then from main memory to CPU before it can be manipulated.

A long cable with many parallel tracks, called a *databus*, or simply *bus*, carries the data between the units of a computer.

The picture on the next page is Encapsulated Postscript. If your dvi viewer doesn't show it, view the Postscript version of this file.

If you keep a local copy, copy file `vNeum2.eps` to the same directory as this file.

The same comment holds for the CPU diagram to follow shortly. Name `2cpu-struct.eps`


Databus
CPU
ROM
RAM
I/O
Addressbus

An instruction includes an operation and (an) address(es)

For example

```
ADD R1 R2
```

is an instruction that adds the contents of register 1 to the contents of register 2 and places the result in register 2

```
FETCH <Addr> R1
```

would copy the contents of memory at memory address `Addr` into register 1.

```
STORE R2 <Addr>
```

would copy the contents of register 1 into memory address `Addr`.

I write `<Addr>` here to stand for an arbitrary memory address. Such an instruction would actually look like (in octal notation)

```
STORE R2 326551611142
```

How big is an instruction?

Suppose there are 64 possible instructions. If we give each instruction a particular code then we need 6 bits to code $2^6 = 64$ instructions.

With 16 registers, one needs only 4 bits to identify a register, so instructions such as *ADD R1 R2* can be written in $6 + 4 + 4 = 14$ bits, which is one word.

However, suppose there are $2^{32} = 4KMb$ possible memory addresses. One needs a single 32-bit word per memory address. So

STORE R2 326551611142

needs two words: $6 + 4 = 10$ bits for `STORE` and `R2`, and the next word of 32 bits for the memory address.

How does the processor know whether to read one word or two? The first six bits of an instruction are read and these say what instruction it is. Then, the processor knows what the arguments must be and thus how much further to read (rest of word or more words).

There is a register which contains the address of the next operation to be performed. At the end of execution of this instruction, the next instruction is fetched from this address.
This register is called the *program counter*, PC.

In the *Fetch phase* of execution, an instruction is fetched from the location specified in PC and placed in MAR.

The instruction code is 'chopped off' and the arguments read. The arguments are usually addresses, so these refer to registers, or memory. If memory, the address is sent to MBR and a request to fetch/store sent on the *address bus*
to main memory, to fetch or store the data which passes on the *databus* to the CPU.

The PC is set to the address of the next instruction to be executed,
usually $PC' = PC + 1$
but $PC' =$ <new-address> if the instruction is GO TO <new-address>

There is an *Arithmetic Logic Unit*, ALU, which actually performs the mathematical operations on the data.

There is also a piece of the CPU, IR, which is used for indirect addressing (when the address to which something is to be stored/fetched is itself held in the memory address specified in the instruction, so two addresses (indirect and direct) must be stored.

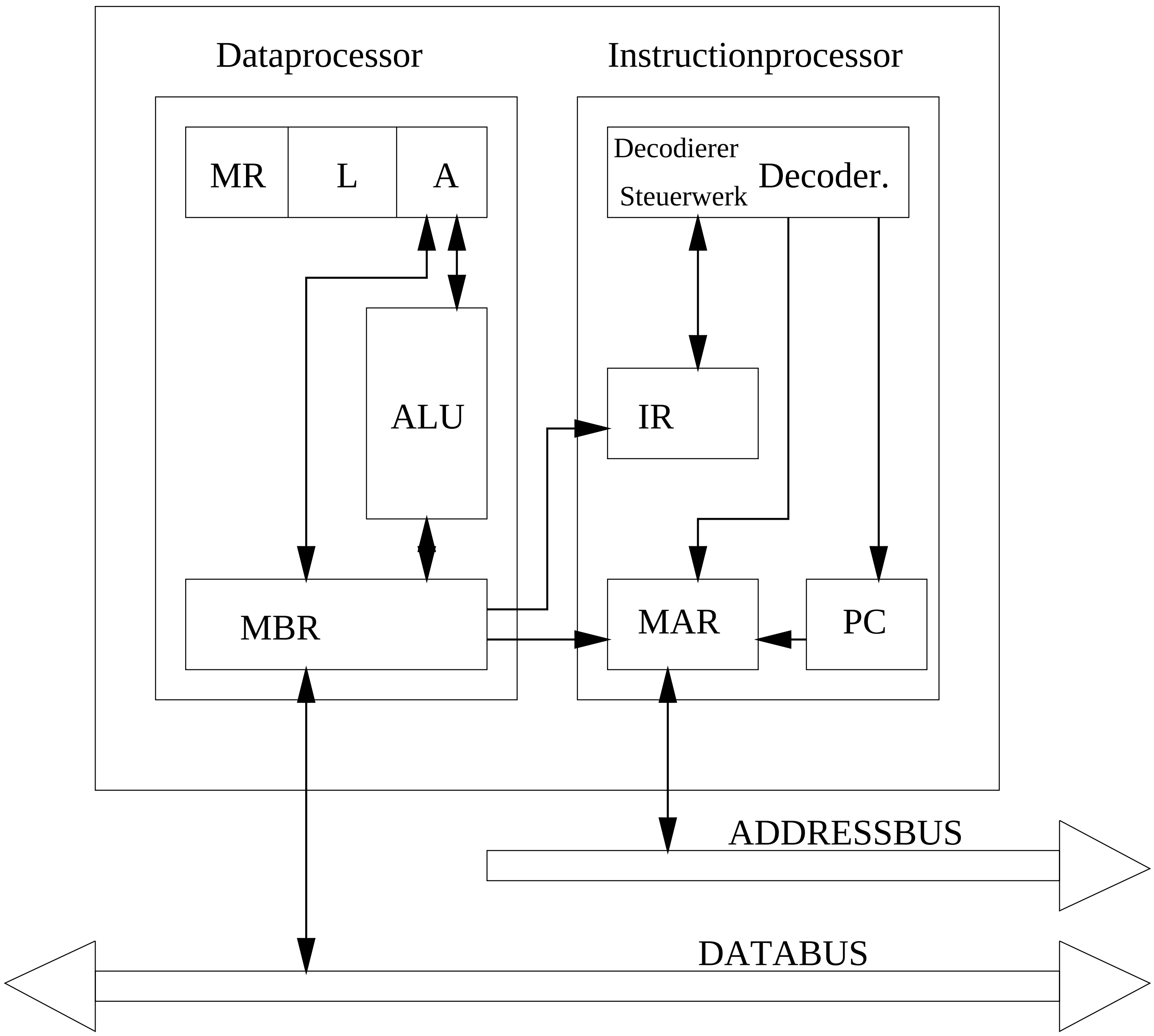
In the *execution phase*, the ALU performs the operations on the data it sees before it.

In summary, a von Neumann CPU must

```
loop
```

- fetch instruction from address given by PC
- decode an instruction word into
  - instruction code
  - arguments
- fetch (data) arguments
- perform the operation specified
- set PC to next instruction

```
endloop
```



Dataprocessor
Instructionprocessor
MR
L
A
Decodierer
Steuerwerk
Decoder.
ALU
IR
MBR
MAR
PC
ADDRESSBUS
DATABUS

Process Synchronisation

Coordination (or non-interference) of multiple processes which share resources is a source of difficult problems

Consider two concurrent processes reading and writing shared memory

```
Process 1:  (x :  integer)
begin
x ← 0;
x ← x+1
stop

Process 2:  (x :  integer)
begin
read x
stop
```

What is the value of `x` read by the second process if they run concurrently?

Another shared-memory puzzle:

```
Process 1:  (x :  integer)
begin
x ← 0;
x ← x+1
stop

Process 2:  (x,y :  integer)
begin
y ← 0;
y ← x+1
stop
```

If the memory location of x is the same as the memory location of y, what is the value after these processes have finished?

A third shared-memory puzzle:

The value of the variable `z` is
`1` if there are 20 blocks or more
of available memory; and
`0` if there are less than 20 blocks
of available memory.

Suppose the value of `z` is `1`.
Suppose `Process 1` needs 15 blocks
and `Process 2` needs 15 blocks.

Suppose they both read `z` at
the same time. What happens?

Unless great care is taken, the following kind of behavior can happen

Suppose *Program 1* and *Program 2* read variable $turn$, which can be written by *Program 3*. *Program 1*, *Program 2* and *Program 3* thus share *turn*.

For example, *Program 3* is the operating system, which allows just one process at a time to send something to the printer. Is it to be *Process 1*? Or *Process 2*? The value of *turn* will tell.

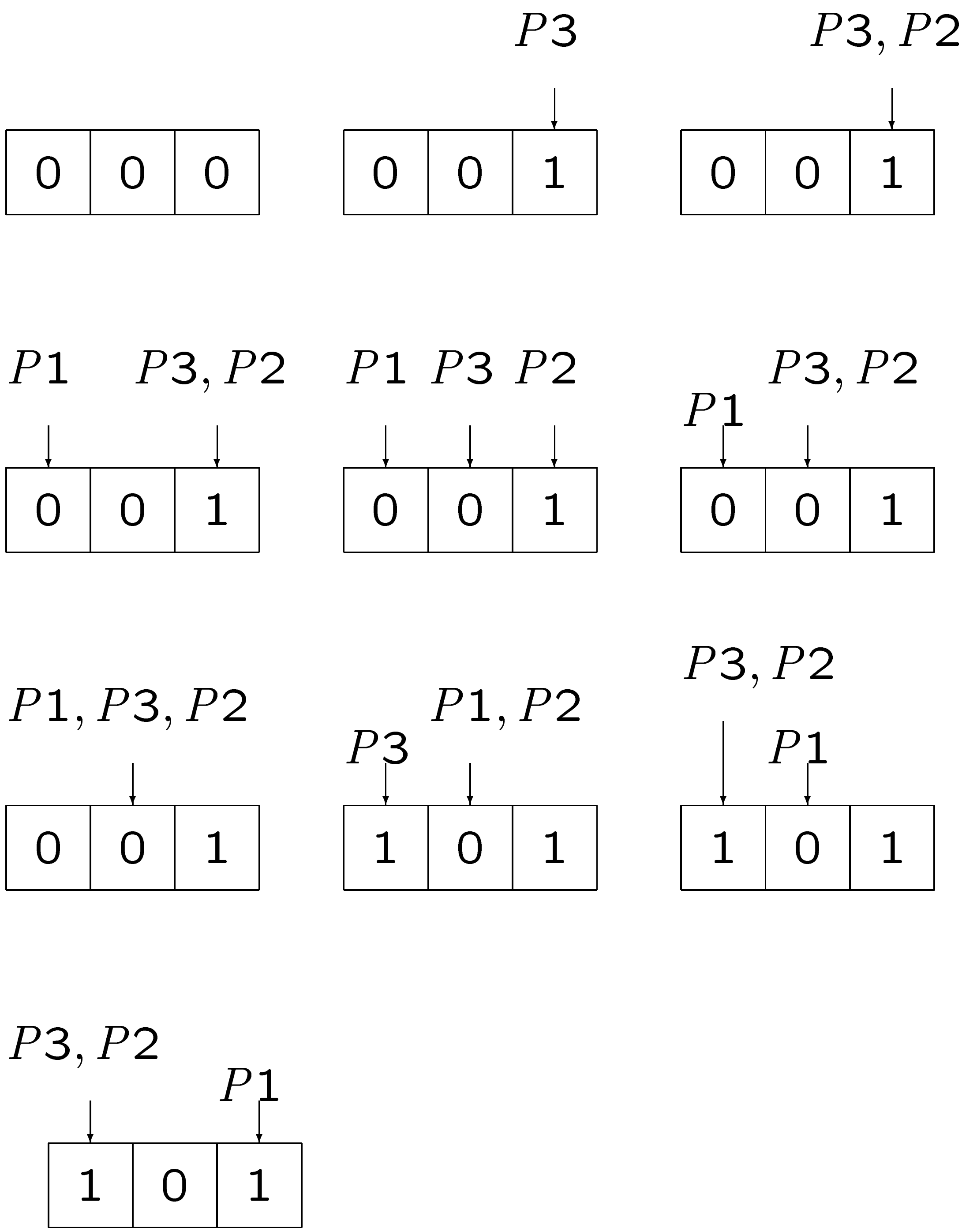
Suppose *turn* has 3 bits.
(We'll see later why I chose this name.)

Initial value of *turn* is 000
Value 001 corresponds to *Process 1*
Value 101 corresponds to *Process 2*

Process 3 writes value 101 from right to left, slightly before *Process 2* reads from right to left, and *Process 1* from left to right, as follows:

P3 writes bit3
P2 reads bit3
P1 reads bit1
P3 writes bit2
P2 reads bit2
P1 reads bit2
P3 writes bit1
P2 reads bit1
P1 reads bit3

This looks as follows



0 0 0
P3
0 0 1
P3, P2
0 0 1
P1
P3, P2
0 0 1
P1 P3 P2
0 0 1
P3, P2
P1
0 0 1
P1, P3, P2
0 0 1
P1, P2
P3
1 0 1
P3, P2
P1
1 0 1
P3, P2
P1
1 0 1

The result is that
P1 reads 001, P2 reads 101

Both processes send their files to the printer at the same time.

An *artistic* outcome, maybe
but not what is wanted

Imagine if this happened in an airplane flight control system!

So, one solution is to make sure that only one process has access to the variable at one time

This is called *mutual exclusion*

The construct used by Dijkstra is called a *semaphore*

It's like a token that only one process *has* at one time

Processes try to grab the token. Only at most one can obtain it. The others have to wait until it's free again.

Specifically, a semaphore is a
shared variable whose access is limited

Only two operations may be performed:
it may be *set* by any process
and *unset* by a process that set it.

It's an *interlock* that prevents another process from **entering its critical section** while the semaphore is set.

A semaphore is (for our purposes) a single bit, that can only be set by operation P ('*passeren*') and released by operation V ('*vrijgeven*'). These operations can only be successfully executed by at most one process at a time. The others must wait.

That is, the P and V operations are *atomic*.

A semaphore can thus be used to construct complex *atomic actions* as follows

Here's how two concurrent program parts can use a semaphore to protect their critical sections

var **integer** x, y $= 0$;
semaphore *sem* $= 1$;

cobegin **loop** α_1: $\langle\ P(sem)\ \rangle$;
β_1: $\langle\ x\ := x + 1\ \rangle$;
γ_1: $\langle\ V(sem)\ \rangle$ **endloop**

□

loop α_2: $\langle\ P(sem)\ \rangle$;
β_2: $\langle\ y\ := y + 1\ \rangle$;
γ_2: $\langle\ V(sem)\ \rangle$ **endloop**

coend

Here is a similar program written now in TLA.

First, the initial condition:

$$
\begin{aligned}
Init_{\Psi} \;\triangleq\;\; & \wedge\ (pc_1 = \text{“a”}) \\
& \wedge\ (pc_2 = \text{“a”}) \\
& \wedge\ (x = 0) \wedge (y = 0) \\
& \wedge\ sem = 1
\end{aligned}
$$

Next, the three operations of the first coroutine:

$$
\begin{array}{lcl}
\alpha_1 & \triangleq & \land\ (pc_1 = \text{“a”}) \\
 & & \land\ (0 < sem) \\
 & & \land\ pc'_1 = \text{“b”} \\
 & & \land\ sem' = sem - 1 \\
 & & \land\ \mathit{Unchanged}\ \langle x,\ y,\ pc_2 \rangle \\
\beta_1 & \triangleq & \land\ pc_1 = \text{“b”} \\
 & & \land\ pc'_1 = \text{“g”} \\
 & & \land\ x' = x + 1 \\
 & & \land\ \mathit{Unchanged}\ \langle y,\ sem,\ pc_2 \rangle \\
\gamma_1 & \triangleq & \land\ pc_1 = \text{“g”} \\
 & & \land\ pc'_1 = \text{“a”} \\
 & & \land\ sem' = sem + 1 \\
 & & \land\ \mathit{Unchanged}\ \langle x,\ y,\ pc_2 \rangle
\end{array}
$$

The three operations of the second coroutine are similar:

$$
\begin{array}{lcl}
\alpha_2 & \triangleq & \land\ (pc_2 = \text{“a”}) \\
 & & \land\ (0 < sem) \\
 & & \land\ pc_2' = \text{“b”} \\
 & & \land\ sem' = sem - 1 \\
 & & \land\ Unchanged\ \langle x,\ y,\ pc_1 \rangle \\
\beta_2 & \triangleq & \land\ pc_2 = \text{“b”} \\
 & & \land\ pc_2' = \text{“g”} \\
 & & \land\ y' = y + 1 \\
 & & \land\ Unchanged\ \langle x,\ sem,\ pc_1 \rangle \\
\gamma_2 & \triangleq & \land\ pc_2' = \text{“a”} \\
 & & \land\ pc_2 = \text{“g”} \\
 & & \land\ sem' = sem + 1 \\
 & & \land\ Unchanged\ \langle x,\ y,\ pc_1 \rangle
\end{array}
$$

The coroutines are defined as $\mathcal{N}_1$ and $\mathcal{N}_2$ and the program as $\mathcal{N}$. The (live) specification of the program is Ψ.
Ψ includes two assertions SF$w\mathcal{N}_1$ and SF$w\mathcal{N}_2$ of *fairness* of the coroutines

$$
\begin{aligned}
\mathcal{N}_1 &\triangleq \alpha_1 \vee \beta_1 \vee \gamma_1 \\
\mathcal{N}_2 &\triangleq \alpha_2 \vee \beta_2 \vee \gamma_2 \\
\mathcal{N} &\triangleq \mathcal{N}_1 \vee \mathcal{N}_2 \\
w &\triangleq \langle x, y, sem, pc_1, pc_2 \rangle
\end{aligned}
$$

$$
\Psi \triangleq \begin{array}{l} \wedge\ Init_{\Psi} \\ \wedge\ \Box[\mathcal{N}]_w \\ \wedge\ \mathsf{SF}_w(\mathcal{N}_1) \\ \wedge\ \mathsf{SF}_w(\mathcal{N}_2) \end{array}
$$

In TLA, when we want program variables to keep the same value during an action, we have to say so explicitly

For example, consider the imperative program ‘command’ $x := x + 1$

This says ‘increment x’.
This means: the value of x in the next state shall be 1 greater than the value of x in the current state

But it says nothing about the value of another program variable y.
Is y allowed to change, or not?

Why should we bother about this?
Isn't this a little peculiar?

In serial programs, it's not expected that other program variables will change when x is incremented.

A *serial program* is one which has only one 'thread of control'

A *thread of control* is a linear sequence of executed or executing program statements

When a serial program executes an *if P then A else B* statement, it executes either *A* or *B*, but not both. There is a single thread of control

When a program executes a **cobegin** *A* — *B* **coend** statement, the program executes both *A* and *B* at the same time. There are two threads of control.

Concerning programs with multiple threads of control (like most modern operating systems), when we specify how a machine behaves, we must say not only that x *increments*, in TLA $x' = x + 1$ in procedural non-mathematical languages $x := x + 1$, but that other variables don't change

Suppose the other variables are y, z

In TLA we say
$$\begin{aligned} &\land\ x' = x + 1 \\ &\land\ y' = y \\ &\land\ z' = z \end{aligned}$$

If we don't specify that they don't change, then they may do so.

That *may* mean that the program doesn't do what we want it to do

For example, let's see what happens with the semaphore example when we don't specify $\mathit{Unchanged}$.

Let λ_1 be the action α_1 *without* the $\mathit{Unchanged}$ assertion. Similarly for λ_2 and α_2.

$$\lambda_1 \triangleq \begin{array}{l} \land\ (pc_1 = \text{“a”}) \\ \land\ (0 < sem) \\ \land\ pc_1' = \text{“b”} \\ \land\ sem' = sem - 1 \end{array}$$

$$\lambda_2 \triangleq \begin{array}{l} \land\ (pc_2 = \text{“a”}) \\ \land\ (0 < sem) \\ \land\ pc_2' = \text{“b”} \\ \land\ sem' = sem - 1 \end{array}$$

Can λ_1 and λ_2 happen together?

To see whether they can, we consider the action $\lambda_1 \wedge \lambda_2$. This is the action that is an λ_1 action *and* an λ_2 action. If it's consistent, they can happen together. If it's contradictory, then not.

$$
\begin{aligned}
\lambda_1 \wedge \lambda_2 \equiv\ & \wedge\ (pc_1 = \text{“a”}) \\
& \wedge\ (pc_2 = \text{“a”}) \\
& \wedge\ (0 < sem) \\
& \wedge\ pc'_1 = \text{“b”} \\
& \wedge\ pc'_2 = \text{“b”} \\
& \wedge\ sem' = sem - 1
\end{aligned}
$$

It's possible (it's not contradictory). Therefore, a joint execution of λ_1 and λ_2 is possible. Afterwards, both $\mathcal{N}_1$ and $\mathcal{N}_2$ execute instructions in their *critical sections* β_1 and β_2.

That's what a semaphore is supposed to prevent.

So in this case, as in general, the assertion $Unchanged$ is necessary to ensure that the behavior of the programs is correctly stated (correctly specified).

When writing program code in most procedural languages, one cannot write $Unchanged$.

So one must be sure to write a program so that it can be proved that the program leaves unmentioned variables $Unchanged$ if they have to be so

How?

We consider now some algorithms for *mutual exclusion*. These algorithms may be used directly if there is no semaphore facility available in the OS. Or they may be used in the OS itself to implement semaphores. Then, these semaphores could be used by other OS programs or by user programs.

This is known as *bootstrapping* - solving a problem in one special case so that other cases can use the special-case solution

One solution: a *flag* variable that lets one process at a time into the critical section.

Initialise $flag = 0$

Program 0

loop

while $flag \neq 0$ **do** *no-op*;

critical section;

$flag = 1$

endloop

Program 1

loop

while $flag \neq 1$ **do** *no-op*;

critical section;

$flag = 0$

endloop

Problem: Process 0 and Process 1 must alternate in their critical sections.

If Process 1 never wants to enter its critical section, Process 0 can never enter it again.

Problem: We must know how many processes are competing before we program this. For a printer queue, this is no good.

Problem: While a process is waiting, it's executing code.
If many processors are sharing one **CPU**, that is a waste of CPU time.

Another solution: make *flag* an array

flag: **array** [0..1] of 0..1

Program 0

```
loop
    flag[0] = 1;
    while flag[1] = 1 do no-op;
    critical section;
    flag[0] = 0
endloop
```

Program 1

```
loop
    flag[1] = 1;
    while flag[0] = 1 do no-op;
    critical section;
    flag[1] = 0
endloop
```

This solves the alternation problem.
But.....

Problem: Suppose Process 0 sets
flag$[0] = 1$, and then
before executing the **while**,
Process 1 sets *flag*$[1] = 1$.

Both processes wait forever.

Problem: We must know how many
processes are competing before we
program this.

flag: **array** [0..1] of 0..1
turn: $\{a, b\}$

Program 0

loop

1. *flag*[0] = 1;
2. *turn* = b;
3. **while** *flag*[1] = 1 **and** *turn* = b
   do *wait*;
4. critical section;
5. *flag*[0] = 0

endloop

Program 1

loop

1. *flag*$[1] = 1$;
2. *turn* $= a$;
3. **while** *flag*$[0] = 1$ **and** *turn* $= a$
   do *wait*;
4. critical section;
5. *flag*$[1] = 0$

endloop

turn is a shared variable.

What happens when both Process 0 and Process 1 set *turn* at the same time?

We must be ensured that
either a **or** b results!

If *turn* is one bit, $a \triangleq 0$ and $b \triangleq 1$,
we must be ensured that the bit
has final value either 0 or 1

We hope this is ensured by the hardware.
Such hardware is called an *arbiter*

The *arbitration problem*: Lamport has shown that under reasonable physical assumptions a perfect arbiter does not exist. But in practice it doesn't seem to be a problem.

But if there are many processes, one bit does not suffice. *turn* must then have many bits, and when two processes change *turn*, who knows what will result?

Suppose *turn* has 3 bits, as before. Remember that two processes reading *turn* simultaneously may read different values if *turn* is being changed at the time

One might read a value that's not valid! This happens if not all bit combinations correspond to a valid value. (Suppose 000 and 101 are valid, 001 not, in our example. $P1$ reads an invalid value.)

This cannot happen if *every possible bit combination corresponds to a valid value*. This can be ensured – maybe some processes must have multiple corresponding values.

There are also provably good algorithms for simultaneous reading and writing of bits. For example, our earlier example would not work if *all* reading/writing is from right to left.

The algorithm again:

flag: **array** [0..1] of 0..1

turn: $\{a, b\}$

Program 0

loop

1. *flag*[0] $= 1$;
2. *turn* $= b$;
3. **while** *flag*[1] $= 1$ **and** *turn* $= b$
   do *wait*;
4. critical section;
5. *flag*[0] $= 0$

endloop

Program 1

loop

1. *flag*$[1] = 1$;
2. *turn* $= a$;
3. **while** *flag*$[0] = 1$ **and** *turn* $= a$
   do *wait*;
4. critical section;
5. *flag*$[1] = 0$

endloop

Informal Reasoning About the Algorithm:

Preliminaries

turn is the only shared variable.
Although *flag* is shared by $P0$ and $P1$,
the individual elements of *flag* are
written by one process only. Since the
elements of *flag* are 1 bit,
we may assume they're written and read
atomically. Likewise for *turn*,
but *turn* is written and read
by both processes.

The reasoning proceeds by considering
states. Processes progress from
state to state by actions. An action is
'*between*' states. Conversely,
a state is '*between*' actions.
(Remember, in TLA, an action is a
relation between states.)

The notation 0.N denotes statement N in P0, similarly for P1. We introduce the Boolean variables (‘*state predicates*’) *at*–0.N, *after*–1.N:

at–0.N is true if and only if P0 is in a state in which it is about to execute statement N (informally, after action $(N-1)$ and before N).
Similarly *after*–1.N.

The state predicates *at*–$x.N$, *after*–$x.N$ (where x is 0 or 1) are program variables. think of them as 1-bit program variables:

at-$x.N$ abbreviates the statement
(*at*–$x.N = 1$)

¬*at*–$x.N$ abbreviates the statement
(*at*–$x.N = 0$)

Safety (mutual exclusion)

Mutual exclusion means that both processes cannot be simultaneously in their critical sections. We show they cannot both be at location *4*.
This means that there is no state in which *at*−0.4 and *at*−1.4 are both true.

*******Unfinished****
Assume there is a state in which
at−0.4 and *at*−1.4 are both true.
In this state, *flag*[0] = *flag*[1] = 1.
This is easy to see, because only $P0$ sets *flag*[0] and only $P1$ sets *flag*[1] and $P0$ and $P1$ are serial processes.
These values are both set before this point and not changed until afterwards.

To arrive at this state in which (at–0.4 $\wedge$ at–1.4) is true, 0.1, 0.2, 0.3 and 1.1, 1.2, 1.3 have been executed. The postconditions of 0.3 and 1.3 (respectively, $turn = a$ and $turn = b$) cannot both have been true simultaneously since *turn* has a unique value (we assume $a \neq b$). This follows from the *arbiter assumption*

Assume that $turn$ is set atomically in 0.2, 1.2. That means that the statements 0.2, 1.2 have been executed either in the order 0.2; 1.2 or in the order 1.2; 0.2. Let's assume the first. So $turn = b$ in the current state (no set of $turn$ occurs between 0.2 and now).

Now consider the tests in 0.3, 1.3. Whether they happen one after the other or simultaneously, the test in 0.3 fails and that in 1.3 succeeds.

Therefore Process 1 waits and Process 0 proceeds into the critical section, and exits. *after*−0.4 = *at*−0.5 is true. When 0.5 is executed, *after*−0.5 is true, and $\mathit{flag}[0] \leftarrow 0$. Process 0 has left its critical section, the condition in 1.3 becomes false and Process 1 can continue into 1.4.

Liveness (progress):

If P1 gets stuck,
it sticks in the **while**

flag[0] = 1 and *turn* = 0

If so, P0 is *after 0.2*
It's *at 0.3* or *at 0.4*
(since *at 0.5*, *flag*[0] is reset to 0)

If *turn* remains 0 (hypothesis),
P0 exits 0.3, does 0.4 and 0.5 *after 0.5*,
turn is reset to 1
so P1 is no longer stuck

Processes $P0$ and $P1$ have been written in a procedural language. Actions are written. In TLA, state predicates and state relations are written.

Procedural languages have *sequential composition*.

$(a; b)$ means action b is to follow action a in sequential order. How do we convert that into a TLA statement about state predicates?

Two actions yield three states

The state predicates are
at-a, *after*-a, *at*-b, *after*-b

In every execution of $(a; b)$,
after$-a \equiv$ *at*$-b$

So the three states are described by
at-a, (*after*-a $\equiv$) *at*-b, *after*-b

These state predicates serve the same function as the *program counter* in hardware or *statement labels* in software

In TLA, actions are binary relations between states. There is no way of talking about a relation between a state and the next-after-next state.

In TLA, we could say

$a \triangleq \textit{at}-a \land (\textit{at}-b)'$

$b \triangleq \textit{at}-b \land (\textit{after}-b)'$

$\textit{perform}-\textit{compose}(a, b) \triangleq a \lor b$

The sequential composition is now a single action *perform-compose* with two 'sub'-actions a and b. A *perform-compose* action is *either* an a-action *or* a b-action. The sequential composition $(a; b)$ is thus two successive *perform-compose* actions.

Exercise 3: Write a TLA specification in a similar way to the previous ones which defines a program consisting of the two processes $P0$ and $P1$ executing the *Peterson* algorithm for two-process mutual exclusion.

It is not only systems with many processors and shared memory which need *concurrency control*. Another sort of system in which it is needed is a *multiprogrammed* system

Multiprogramming:
one processor, many processes
For example, this Sparcbook

Processes you can see:
shell, X Window System, editor (emacs), xdvi, clock (?), console
(to see all, try `ps -al`)

A uniprocessor still needs concurrency control if there is DMA (*direct memory access*), in which disk transfers and other data transfers happen in parallel with processing. The memory (or other resource) is being shared between processor and other hardware simultaneously

Concurrency control is hard to avoid completely

We have been concerned mainly about *safety* (for example, mutual exclusion) but what about *liveness*?

On a uniprocessor, each process must get a chance to progress. That means that each must get regular opportunity to use the processor

This is controlled by a process called the *scheduler*

Different scheduling policies

- each *ready* process is started and run until done

- Each process
  loop runs for a while in a *time slice*
  of the processor and then waits while
  other processes execute their timeslices
  endloop

How do these two policies compare?

Process Liveness

The first policy does not satisfy reasonable liveness properties. If a program goes into an infinite loop, or waits for a data event that doesn't happen, then it does not reach its end and thus no other process can proceed.

The second policy satisfies reasonable liveness properties. A process which loops infinitely or waits prevents only itself from continuing. All other processes proceed as usual in their time slices. (However, maybe the CPU could be more effectively used in this case.)

Concurrency Control

The first policy brings no problems with critical sections. An entire program is run uninterrupted and therefore no concurrency control is needed.

The second policy requires some concurrency control. A critical section may have many operations. The time slice might run out in the middle, interrupting the critical section. Other processes must be hindered from accessing the shared resource.

Setting variables such as *turn* and *flag* may be accomplished atomically within the time slice of some process.

Sophisticated control algorithms are not really needed. Simpler ones suffice.

Applicability

In fact, the first policy is only possible in an environment in which all processes terminate. But most processes in a modern OS *do not terminate*.

When does your shell terminate?
When you send it a signal (Ctrl-D) as input from the keyboard.
No signal, no termination.

Similarly with your editor, dvi viewer, clock display, console,.....

Hence the first policy is not practical for a modern multi-purpose interactive computer.

The first policy is appropriate for real-time process control systems, in which all the processes are precisely known and their running times are also precisely known

Real-time scheduling is often *static*, that is, it's precisely planned beforehand and programmed in to the OS.

We have seen that the second policy could make more effective use of the CPU by not giving it to a process that's waiting or running an infinite loop

There's nothing an OS should do actually to *prevent* processes from waiting on input (for example, from the keyboard) or from looping indefinitely. That's up to the user-programmer.

But observe that if a process is waiting on input, then the OS *knows about it*, because the input must come from another process or data structure (for example, a keyboard input buffer) and the OS must manage this transfer

When the OS can detect that a process is waiting for an event that hasn't yet happened, it can plan to avoid running that process until the event happens.

Process Run-Status

How can the OS maintain this knowledge?

The OS can assign a *run-status attribute* to a process

A process may be

- *running* now
- *ready* but not running
- *waiting* on some event

Since processes have (at least) this attribute, the OS needs to organise data structures to keep this information.

Consider keeping the run-status info in a relational database

Processes must have *ID*'s so that the pair $\langle$*PID*, *run–status*$\rangle$ may be entered in the database and updated.

Consider how the OS decides who shall run next

In the pure database D it must calculate
$\textsc{choose}\ PID : PID \in \{ID \mid \langle ID, \textit{ready}\rangle \in D\}$
which selects a PID from amongst the set of PIDs of ready processes

How does it do so fairly?

Maybe always the same two or three are selected, and the others 'starved out'

So it's *fairer* to maintain, say, a queue of *ready* processes

This can be, for example, a linked list with the links an extra attribute of the relation: $\langle PID, \textit{ready}, \textit{next-ready-process} \rangle$

Ready processes may be inserted into the queue in different ways, to reflect different policies.

For example, processes may have different *priorities*. A *Priority queue* maintains separate queues for each priority level and only runs ready processes of priority $(n+1)$ when there are no more ready processes of priority n.

Thus the relational database becomes larger:
$\langle PID, \textit{ready}, \textit{priority}, \textit{next–ready–process} \rangle$

Priority scheduling is not guaranteed to be fair *between* priority levels, but is fair *within* priority levels

Higher priority processes are *trusted* to complete on time and not hog resources.

Scheduling uses algorithms based on statistical properties of the processes

However, there is a mathematical form to preemptive scheduling of processes on a single processor (as in multiprogramming)

For example, consider the three processes

```
 Process P.1
a:  [......]  ;
b:  [......]  ;
c:  [......]  .

 Process P.2
i:  [......]  ;
ii:  [......]  ;
iii:  [......]  .

 Process P.3
x:  [......]  ;
y:  [......]  ;
z:  [......]  .
```

Here are two execution sequences:

`a; i; ii; x; y; b; c; z; iii`

or,
`i; x; a; b; c; ii; iii; y; z`

In fact, any *interleaving* of the individual
atomic operations of the processes
is a possible execution.

What is an interleaving?

Any sequence of actions E in which

1. every action is an action either of `P.1`, or of `P.2`, or of `P.3`

2. the actions of `P.1` occur in E exactly in the order in which they occur in `P.1`, and similarly for `P.2` and `P.3`

This last condition is also expressed by
saying the *projection* of E on `P.1`
is equal to `P.1` (there is a corresponding
equation), and similarly for `P.2`, `P.3`

In the semantics of TLA, processes change state in discrete time steps. Two actions of a machine may be interleaved, or they may be simultaneous (occurring at different time steps or on the same time step).

But the time steps for different machines do not have to be the same.

In a verification that one virtual machine implements another, one proves mathematically that the steps of one machine are steps (or no-ops) of the other.

So in a verification, time steps are shown to be comparable.

Here's a TLA+ description of a scheduler

module *FCFSScheduler*

parameters

pc : variable

---

include *Process* **as** *P1*
include *Process* **as** *P2*
include *Process* **as** *P3*

module *FCFSScheduler(contd)*

predicates

$$Init \triangleq pc = 0$$

actions

$$
\begin{aligned}
Fetch(P) &\triangleq \land\ RestoreState(P) \\
&\ \land\ pc' = pc + 1 \\
Save(P) &\triangleq \land\ SaveState(P) \\
&\ \land\ pc' = pc + 1 \\
Run(POp) &\triangleq \land\ POp \\
&\ \land\ pc' = pc + 1
\end{aligned}
$$

module *FCFSScheduler(continued)*

O

$$
\begin{aligned}
p \triangleq\ & \lor\ pc = 0 \land Fetch(P1) \\
& \lor\ 1 \leq pc \leq 2 \land Run(P1.Op) \\
& \lor\ pc = 3 \land Save(P1) \\
& \lor\ pc = 4 \land Fetch(P2) \\
& \lor\ 5 \leq pc \leq 6 \land Run(P2.Op) \\
& \lor\ pc = 7 \land Save(P2) \\
& \lor\ pc = 8 \land Fetch(P3) \\
& \lor\ 9 \leq pc \leq 10 \land Run(P3.Op) \\
& \lor\ pc = 11 \land Save(P3) \\
& \lor\ pc = 12 \land pc' = 0
\end{aligned}
$$

module *FCFSScheduler(continued)*

temporal

$$Spec \triangleq \begin{array}{l} \land\ Init \\ \land\ \Box[Op]_{pc} \\ \land\ WF_{pc}(Op) \end{array}$$

But, how does this work? Is the scheduler the most important process?

There is a *clock* that *raises* an *interrupt*

The interrupt is signalled by a *bit* that is *set* asynchronously by the clock

This bit is 'shared' although it's not memory. The clock can set to 1, the processor can read and set to 0.

On the execution cycle of every instruction, the processor looks at the clock interrupt bit. If it's 1, the next value of the program counter is the value A in a particular location L which is designed into the chip. The value A is set by the operating system designer, and is the address of the scheduler.

In practice, the scheduler doesn't execute the actions of the processes, it merely loads and saves process state, and figures which is the next process to have access.

The processes waiting to run are kept on a queue, called the *ready list*.

When a process is `Restore`d, it is removed from the ready list, and its `pc` value is loaded into the `pc` register of the processor, and the *status* of the process becomes `running`.

When a clock interrupt is generated, the first thing that the scheduler does is save the state of the `running` process. The process itself becomes `ready` and joins the tail of the ready list. The process at the head of the ready list is restored.

If a `running` process must wait for something, for example it must obtain some data which is not in main memory but on disk, then at the next clock interrupt, the process's state will be saved and the process (name) put on the `waiting` list/heap. When the wait condition is no longer valid (the data has arrived in main memory), a notification will be put in a special location which is looked at by the scheduler when it is running, and if a `waiting` process has completed its external activity, the scheduler will restore the process (name) to the ready list and remove it from the waiting heap.

But perhaps running the processes in this *First-Come, First-Served* manner is not the best way of *scheduling*.

Other forms of scheduling are

- shortest job first

- priority

- multilevel queue

We now show how process space protection (usually ensured by the architecture, i.e., the hardware) is specified in TLA$^+$.

We specify

- a semaphore
- a process template
- a three-process system

module *BinarySemaphore*

parameters

$BinSemVar$: variable

predicate

$init \triangleq$ ....

actions

$P(BinSemVar) \triangleq$ ....
$V(BinSemVar) \triangleq$ ....

module *BinarySemaphore (cont'd)*

temporal

$$spec \triangleq \ldots\ldots$$

module *ProcessTemplate*

parameters

d, pc : variable

predicate

$init \triangleq$ ....

actions

$TheActions \triangleq$ ....

temporal

$$spec \triangleq \ldots\ldots\ldots\ldots$$

theorem

$$spec \Rightarrow \Box pc \geq 0$$

module *ThreeProcessSystem*

parameters

$x, y, z, Sem,$
$pc_1, pc_2, pc_3, PC_1, PC_2, PC_3$: variable
$ProcessBlockSize,$
$ProcessAddressSpaceStart$: constant

$$vars \triangleq \langle x, y, z, Sem, pc_1, pc_2, pc_3, PC_1, PC_2, PC_3 \rangle$$

include *ProcessTemplate* **as** $P1$
    with $d \leftarrow x, pc \leftarrow pc_1$
include *ProcessTemplate* **as** $P2$
    with $d \leftarrow y, pc \leftarrow pc_2$
include *ProcessTemplate* **as** $P3$
    with $d \leftarrow z, pc \leftarrow pc_3$

$\textbf{include}\ BinarySemaphore\ \textbf{as}\ Semaphore$
$\quad \textbf{with}\ BinSemVar \longleftarrow Sem$

module *ThreeProcessSystem (cont'd)*

predicate

$$
\begin{aligned}
init \triangleq\ & \land\ PC_1 = ProcessAddressSpaceStart + pc_1 \\
& \land\ PC_2 = ProcessAddressSpaceStart \\
& \qquad\quad + ProcessBlockSize + pc_2 \\
& \land\ PC_3 = ProcessAddressSpaceStart \\
& \qquad\quad + (2 \times ProcessBlockSize) + pc_3
\end{aligned}
$$

actions

$$
\begin{aligned}
ProcOp \triangleq\ & \lor \land\ P1.TheActions \\
& \quad \land\ (PC_1)' = (pc_1)' - pc_1 + PC_1 \\
& \lor \land\ P2.TheActions \\
& \quad \land\ (PC_2)' = (pc_2)' - pc_2 + PC_2 \\
& \lor \land\ P3.TheActions \\
& \quad \land\ (PC_3)' = (pc_3)' - pc_3 + PC_3
\end{aligned}
$$

module *ThreeProcessSystem (cont'd)*

temporal

$$Spec \triangleq \begin{array}{l} \land\ init \\ \land\ \Box[ProcOp]_{vars} \\ \land\ \mathit{Liveness} \end{array}$$

module *ThreeProcessSystem (cont'd)*

theorem

$$
\begin{aligned}
Spec \Rightarrow \Box(&\land\ PC_1 = ProcessAddressSpaceStart + pc_1 \\
&\land\ PC_2 = ProcessAddressSpaceStart \\
&\qquad\qquad + ProcessBlockSize + pc_2 \\
&\land\ PC_3 = ProcessAddressSpaceStart \\
&\qquad\qquad + 2 \times ProcessBlockSize + pc_3 \\
&\land\ PC_1 < \\
&\quad ProcessAddressSpaceStart + ProcessBlockSize \\
&\land\ PC_2 < \\
&\quad ProcessAddressSpaceStart + \\
&\quad\ + (2 \times ProcessBlockSize)) \\
&\land\ PC_3 < \\
&\quad ProcessAddressSpaceStart + \\
&\quad\ + (3 \times ProcessBlockSize))
\end{aligned}
$$

Specifications and Proofs in TLA+

A short introduction to the use
of TLA in verification

I specify an abstract buffer (a sequence)
and a concrete buffer (an array) and
prove formally that the concrete buffer
is a correct implementation of the
abstract buffer

This is similar to some early lectures
in my course on verification

First, an abstract buffer with two operations

`push(something)` and `pop`

The figure shows the state of the buffer before each operation and the state after

$$\begin{array}{ccc} \text{buffer} & & \text{buffer}' \\ <\ \text{b,x,y,d}\ > & \xrightarrow{\text{pop}} & <\ \text{x,y,d}\ > \end{array}$$

$$<\ \text{b,x,y,d}\ > \xrightarrow{\text{push(a)}} <\ \text{b,x,y,d,a}\ >$$

Here is a more concrete version of the buffer

It's an array

$\perp$ is the symbol for '**`nothing here`**'

Buffer $\qquad\qquad\qquad\qquad$ Buffer$'$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & \mathsf{b} & \mathsf{x} & \mathsf{y} & \perp & \mathsf{d} & \perp \\ \hline\end{array} \xrightarrow{\text{push(a)}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & \mathsf{b} & \mathsf{x} & \mathsf{y} & \perp & \mathsf{d} & \mathsf{a} \\ \hline\end{array}$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & \mathsf{b} & \mathsf{x} & \mathsf{y} & \perp & \mathsf{d} & \mathsf{a} \\ \hline\end{array} \xrightarrow{\text{move(6)}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & \mathsf{b} & \mathsf{x} & \mathsf{y} & \mathsf{d} & \perp & \mathsf{a} \\ \hline\end{array}$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & \mathsf{b} & \mathsf{x} & \mathsf{y} & \mathsf{d} & \perp & \mathsf{a} \\ \hline\end{array} \xrightarrow{\text{move(2)}} \begin{array}{|c|c|c|c|c|c|c|}\hline \mathsf{b} & \perp & \mathsf{x} & \mathsf{y} & \mathsf{d} & \perp & \mathsf{a} \\ \hline\end{array}$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \mathsf{b} & \perp & \mathsf{x} & \mathsf{y} & \mathsf{d} & \perp & \mathsf{a} \\ \hline\end{array} \xrightarrow{\text{pop}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & \perp & \mathsf{x} & \mathsf{y} & \mathsf{d} & \perp & \mathsf{a} \\ \hline\end{array}$$

Here's how the concrete buffer is supposed
to correspond to the abstract buffer
as a data structure

ConcreteBuffer AbstractBuffer

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & \perp \\ \hline \end{array} \longrightarrow < b,x,y,d >$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & a \\ \hline \end{array} \longrightarrow < b,x,y,d,a >$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & d & \perp & a \\ \hline \end{array} \longrightarrow < b,x,y,d,a >$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline b & \perp & x & y & d & \perp & a \\ \hline \end{array} \longrightarrow < b,x,y,d,a >$$

And here's how the operations correspond

$$\text{Buffer} \qquad\qquad\qquad\qquad \text{Buffer}'$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & \perp \\ \hline \end{array} \xrightarrow{\text{push(a)}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & a \\ \hline \end{array}$$

$$< \text{b,x,y,d} > \xrightarrow{\text{push(a)}} < \text{b,x,y,d,a} >$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & a \\ \hline \end{array} \xrightarrow{\text{move(6)}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & d & \perp & a \\ \hline \end{array}$$

$$< \text{b,x,y,d,a} > \xrightarrow{\texttt{no-op}} < \text{b,x,y,d,a} >$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & d & \perp & a \\ \hline \end{array} \xrightarrow{\text{move(2)}} \begin{array}{|c|c|c|c|c|c|c|}\hline b & \perp & x & y & d & \perp & a \\ \hline \end{array}$$

$$< \text{b,x,y,d,a} > \xrightarrow{\texttt{no-op}} < \text{b,x,y,d,a} >$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline b & \perp & x & y & d & \perp & a \\ \hline \end{array} \xrightarrow{\text{pop}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & \perp & x & y & d & \perp & a \\ \hline \end{array}$$

$$< \text{b,x,y,d,a} > \xrightarrow{\text{pop}} < \text{x,y,d,a} >$$

How does this fit together?

The concrete buffer *simulates* the abstract buffer

- they start in ‘equivalent’ states

- every action of the concrete buffer
  corresponds either to an action
  or to a non-action of the abstract buffer

- when the concrete buffer is sufficiently ‘live’,
  then the abstract buffer actually does some
  desired action

This method of *state machine simulation* is common to many methods, for example

- TLA of Lamport

- Input/Output machines of Tuttle, Lynch, Vaandrager

- the method of Lam and Shankar (also TL-based)

An alternative is to have actions only—then the operation of the system is an abstract machine simulation, but not a *state* machine simulation, since one doesn't have *state*

How does one specify the actions?

Here's one from the abstract buffer

module *AbstractBuffer*

actions

$$push(a) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Len(buffer) < N \\ \land\ buffer' = buffer \circ \langle a \rangle \end{array}$$

Why is it written like this?

Let's compare two ways of writing this action

module *AbstractBuffer*

actions

$$push(a) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Len(buffer) < N \\ \land\ buffer' = buffer \circ \langle a \rangle \end{array}$$

badly-written actions

$$push(a) \triangleq$$
$$a \in Data \land Len(buffer) < N \land buffer' = buffer \circ \langle a \rangle$$

If that didn't persuade you, try this

module *AbstractBuffer*

actions

$$
push(a|b) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Len(buffer) < N \\ \land\ \lor\ buffer' = buffer \circ \langle\, a \,\rangle \\ \ \lor\ buffer' = buffer \circ \langle\, b \,\rangle \end{array}
$$

badly-written actions

$$
\begin{array}{l} push(a|b) \triangleq \\ a \in Data \land Len(buffer) < N \land \\ (buffer' = buffer \circ \langle\, a \,\rangle \lor buffer' = buffer \circ \langle\, b \,\rangle) \end{array}
$$

Now we shall see how to write the two specifications. First, we define the starting states of the variables.

module *AbstractBuffer*

predicates

$$Init \;\triangleq\; buffer = \langle\, \rangle$$

module *ConcreteBuffer*

predicates

$$Init \;\triangleq\; \forall\, n \in 1..N \;:\; Buffer[n] = \perp$$

Next, we define the *push* actions

module *Buffer Actions*

abstract actions

$$push(a) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Len(buffer) < N \\ \land\ buffer' = buffer \circ \langle a \rangle \end{array}$$

concrete actions

$$push(a) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Buffer[N] = \bot \\ \land\ Buffer'[N] = a \\ \land\ \forall\, i \in 1..(N-1)\ : \\ \qquad \textsf{unchanged}\ Buffer[i] \end{array}$$

And now the *pop* actions

module *Buffer Actions*

abstract actions

$$pop \triangleq \begin{array}{l} \land\ Len(buffer) > 0 \\ \land\ buffer' = tail(buffer) \end{array}$$

concrete actions

$$pop \triangleq \begin{array}{l} \land\ Buffer[1] \neq \bot \\ \land\ Buffer'[1] = \bot \\ \land\ \forall\, i \in 2..N\ : \\ \qquad \text{unchanged } Buffer[i] \end{array}$$

And there's one concrete action left

module *Buffer Actions*

abstract actions

$$no - op \triangleq \text{???}$$

concrete actions

$$
\begin{aligned}
move(k) \triangleq\ & \land\ k \in 2..N \\
& \land\ Buffer[k] \neq \bot \\
& \land\ Buffer[k-1] = \bot \\
& \land\ Buffer'[k] = \bot \\
& \land\ Buffer'[k-1] = Buffer[k] \\
& \land\ \forall\, i \in 1..(k-2) : \\
& \qquad \text{unchanged } Buffer[i] \\
& \land\ \forall\, i \in (k+1)..N : \\
& \qquad \text{unchanged } Buffer[i]
\end{aligned}
$$

We have the *initial conditions* and the *actions*. But we don't yet have a *specification*

A specification defines the initial conditions and the actions—

and also the sentence that says

- the system starts in the initial condition

- if variables change values, it must be because of a defined action (safety)

- it's always true that
  some desired action eventually happens
  if it can (liveness)

First some notation.

$[A]_x$ means $A \vee (x' = x)$

$[A]_{\langle x,y \rangle}$ means $A \vee (x' = x \wedge y' = y)$

Intuitively, $[A]_x$ means

Either A or x *doesn't change value*

module *AbstractBuffer*

imports

$Sequences$

parameters

$buffer$: variable
$Data, N$: constant

predicates

$$Init \triangleq buffer = \langle\, \rangle$$

module *AbstractBuffer (continued)*

actions

$$
\begin{aligned}
push(a) \;\triangleq\; & \land\ a \in Data \\
& \land\ Len(buffer) < N \\
& \land\ buffer' = buffer \circ \langle a \rangle \\
pop \;\triangleq\; & \land\ Len(buffer) > 0 \\
& \land\ buffer' = tail(buffer)
\end{aligned}
$$

temporal

$$
\begin{aligned}
Spec \;\triangleq\; & \land\ Init \\
& \land\ \Box[pop \lor \exists\, b\ :\ push(b)]_{buffer} \\
& \land\ WF_{buffer}(pop)
\end{aligned}
$$

module *ConcreteBuffer*

parameters

$Buffer$: variable
$Data, N$: constant

assertions

$\perp \notin Data$

predicates

$$Init \triangleq \land \forall n \in 1..N : Buffer[n] = \perp$$

module *ConcreteBuffer (continued)*

actions

$$
\begin{array}{lll}
push(a) & \triangleq & \land\ a \in Data \\
 & & \land\ Buffer[N] = \bot \\
 & & \land\ Buffer'[N] = a \\
 & & \land\ \forall\, i \in 1..(N-1)\ :\ \textsf{unchanged}\ Buffer[i] \\
pop & \triangleq & \land\ Buffer[1] \neq \bot \\
 & & \land\ Buffer'[1] = \bot \\
 & & \land\ \forall\, i \in 2..N\ :\ \textsf{unchanged}\ Buffer[i] \\
move(k) & \triangleq & \land\ k \in 2..N \\
 & & \land\ Buffer[k] \neq \bot \\
 & & \land\ Buffer[k-1] = \bot \\
 & & \land\ Buffer'[k] = \bot \\
 & & \land\ Buffer'[k-1] = Buffer[k] \\
 & & \land\ \forall\, i \in 1..(k-2)\ :\ \textsf{unchanged}\ Buffer[i] \\
 & & \land\ \forall\, i \in (k+1)..N\ :\ \textsf{unchanged}\ Buffer[i]
\end{array}
$$

module *ConcreteBuffer (will this never end?)*

temporal

$$
\begin{aligned}
Spec \;\triangleq\; & \land\; Init \\
& \land\; \Box[pop \lor \exists\, b \,:\, push(b) \lor \exists\, k \,:\, move(k)]_{Buffer} \\
& \land\; WF_{Buffer}(pop) \\
& \land\; WF_{Buffer}(\exists\, k \,:\, move(k))
\end{aligned}
$$

Verification means that one has

- A description of an implementation

- A specification

One must prove that the description of the implementation fulfils the specification.

For logical methods, ‘*fulfils*’ = ‘*implies*’

For real systems, one must use *hierarchical* methods in order to control the complexity.

Hierarchical methods: one describes an implementation I, then a more abstract view A_1 and proves that

$$I \Rightarrow A_1$$

One describes then an even more abstract view A_2 and proves that

$$A_1 \Rightarrow A_2$$

But if that's true, of course, then simply

$$I \Rightarrow A_2$$

and there's little need for the intermediate step.

It's when there are many separate parts that need to be brought together that hierarchical decomposition pays off.

A simple hierarchical decomposition:

Level 1 a database distributed over 5 different sites with query points over many more

Level 2

- a specification of a serial database
- a specification of a database split into 5 pieces
- a specification of a reliable protocol for queries

Level 3

- a specification of an implementation of a serial database
- a specification of an implementation of a database split into 5 pieces
- a specification of an implementation of a reliable protocol for queries

We start at the beginning. We have two specifications.

module *Specifications*

temporal

$$
\begin{array}{lll}
Conc - Buffer - Spec & \triangleq & \\
& & \land\ Init \\
& & \land\ \Box[\lor\ pop \\
& & \quad\ \lor\ \exists\, b\ :\ push(b) \\
& & \quad\ \lor\ \exists\, k\ :\ move(k)]_{Buffer} \\
& & \land\ WF_{Buffer}(pop) \\
& & \land\ WF_{Buffer}(\exists\, k\ :\ move(k)) \\
Abs - Buffer - Spec & \triangleq & \\
& & \land\ Init \\
& & \land\ \Box[\lor\ pop \\
& & \quad\ \lor\ \exists\, b\ :\ push(b)]_{buffer} \\
& & \land\ WF_{buffer}(pop)
\end{array}
$$

We want to show that

Conc-Buffer-Spec $\Rightarrow$ *Abs-Buffer-Spec*

But it doesn't!

Conc-Buffer-Spec has a variable *Buffer* which doesn't occur in *Abs-Buffer-Spec*

Abs-Buffer-Spec has a variable *buffer* which doesn't occur in *Conc-Buffer-Spec*

But are these variables essential? We merely want to *specify* a buffer, without really caring what specify object is a buffer.

Maybe we want to *hide* the buffer itself. We do this in logic by existential quantification.

We prove

$$
\begin{aligned}
\exists\, Buffer : \ & Conc{-}Buffer{-}Spec \\
& \Rightarrow \exists\, buffer : Abs{-}Buffer{-}Spec
\end{aligned}
$$

Or, more formally, ...

module *Theorems*

include *ConcreteBuffer* **as** *CB(N)*
include *AbstractBuffer* **as** *AB(N)*

theorems

$$\exists\, Buffer\ :\ CB(N).Spec \Rightarrow \exists\, buffer\ :\ AB(N).Spec$$

What does **include** mean?

It means that the variables, definitions, operations, from $ConcreteBuffer$ are visible in this buffer, with the same names, except that every operation or temporal definition is prefixed with the given name of the **include**-ed module.

Similarly for $AbstractBuffer$.

One can **include** variables and constants with names other than the original ones – see *Michael Blume*'s Einführung or the *TLA+ Manual*.

Now to the proof.
The proof is formal, formally laid out
in a hierarchical style.

There are two numbering schemes in *pf.sty*

One is absolute—

the.full.path.number

One is relative—

$\langle$TheLevelNumber$\rangle$TheStepNumber

Here are examples of the same proof
with both schemes.

1. $A \wedge (P \wedge Q \wedge R)$

   Proof:

   1.1. A

      Proof: I guess A just is true. $\square$

   1.2. $P \wedge Q \wedge R$

      Proof:

      1.2.1. $P \wedge Q$

         Proof:

         1.2.1.1. P

            Proof: I guess P just is true. $\square$

         1.2.1.2. Q

            Proof: I guess Q just is true. $\square$

         1.2.1.3. Q.E.D.

            Proof: Conjoin 1.2.1.1 and 1.2.1.2. $\square$

      1.2.2. R

         Proof: I guess R just is true. $\square$

      1.2.3. Q.E.D.

         Proof: Conjoin 1.2.1 and 1.2.2. $\square$

   1.3. Q.E.D.

      Proof: Conjoin 1.1 and 1.2. $\square$

$\langle 1\rangle$1. $A \wedge (P \wedge Q \wedge R)$

Proof:

$\langle 2\rangle$1. A

Proof: I guess A just is true. $\square$

$\langle 2\rangle$2. $P \wedge Q \wedge R$

Proof:

$\langle 3\rangle$1. $P \wedge Q$

Proof:

$\langle 4\rangle$1. P

Proof: I guess P just is true. $\square$

$\langle 4\rangle$2. Q

Proof: I guess Q just is true. $\square$

$\langle 4\rangle$3. Q.E.D.

Proof: Conjoin $\langle 4\rangle$1 and $\langle 4\rangle$2. $\square$

$\langle 3\rangle$2. R

Proof: I guess R just is true. $\square$

$\langle 3\rangle$3. Q.E.D.

Proof: Conjoin $\langle 3\rangle$1 and $\langle 3\rangle$2. $\square$

$\langle 2\rangle$3. Q.E.D.

Proof: Conjoin $\langle 2\rangle$1 and $\langle 2\rangle$2. $\square$

Proof step numbers in the buffer example are relative.

See if you can assign *path-numbers* to the proof steps as we go

So, how do we prove

$\langle 0 \rangle 1.$ $\mathbf{\exists}\, \mathit{Buffer} : CB(N).\mathit{Spec} \Rightarrow \mathbf{\exists}\, \mathit{buffer} : AB(N).\mathit{Spec}$

??

We can prove it if we treat Buffer like a variable (which it is):

$\langle 1 \rangle 1.$ $CB(N).\mathit{Spec} \Rightarrow \mathbf{\exists}\, \mathit{buffer} : AB(N).\mathit{Spec}$

and we try to find some *state function* $f(\mathit{Buffer})$ that can interpret buffer.

By convention, we let

$$\overline{\mathit{buffer}} \;\triangleq\; f(\mathit{Buffer})$$

(this is the *Refinement Mapping*) and

$\overline{AB(N).\mathit{Spec}}$ be $AB(N).\mathit{Spec}$, with every occurrence of buffer replaced by $\overline{\mathit{buffer}}$.

Then, we prove

$\langle 2 \rangle 1.$ $CB(N).\mathit{Spec} \Rightarrow \overline{AB(N).\mathit{Spec}}$

Predicate logic allows us to conclude what we want from this.

Each specification is a conjunction

$$Init \wedge \Box[action \vee action \vee ...]_{variables} \wedge Liveness$$

It seems most reasonable to prove the conjunction bit by bit:

$\langle 3 \rangle 1.$ $CB(N).Spec \Rightarrow \overline{AB(N).Init}$

$\langle 3 \rangle 2.$ $CB(N).Spec \Rightarrow \Box[\vee\ \overline{AB(N).pop}$
$\qquad\qquad\qquad\qquad\vee\ \overline{\exists\, a : AB(N)push(a)}]_{\overline{buffer}}$

$\langle 3 \rangle 3.$ $CB(N).Spec \Rightarrow \overline{WF_{buffer}(AB(N).pop)}$

$\langle 3 \rangle 4.$ Q.E.D.

Proof: Follows by propositional logic from the conjunction of steps $\langle 3 \rangle 1$, $\langle 3 \rangle 2$ and $\langle 3 \rangle 3$. $\square$

In fact, it turns out that we may prove

$\langle 4 \rangle 1.$ $CB(N).Init \Rightarrow \overline{AB(N).Init}$

$\langle 4 \rangle 2.$ Q.E.D.

Proof: Follows directly by propositional logic. $\Box$

$$\begin{array}{ll} \langle 4 \rangle 1. & \wedge\ CB(N).Init \\ & \wedge\ \Box[\vee\ CB(N).pop \\ & \qquad \vee\ \exists\, a\ :\ CB(N).push(a) \\ & \qquad \vee\ \exists\, k\ :\ CB(N).move(k)]_{Buffer} \\ & \Rightarrow \Box[\vee\ \overline{AB(N).pop} \\ & \qquad \vee\ \overline{\exists\, a\ :\ AB(N)push(a)}]_{\overline{buffer}} \end{array}$$

$\langle 4 \rangle 2.$ Q.E.D.

Proof: Follows directly by propositional logic. $\Box$

$\langle 4 \rangle 1.$ $CB(N).Spec \Rightarrow \overline{WF_{buffer}(AB(N).pop)}$

$\langle 4 \rangle 2.$ Q.E.D.

Proof: Follows directly by propositional logic. $\Box$

Consider now the step

$\langle 4 \rangle 1.$ $CB(N).Init \Rightarrow \overline{AB(N).Init}$

$\langle 4 \rangle 2.$ Q.E.D.

Proof: .... is simply math. $\square$

On the other hand, we may prove

$$\langle 4\rangle 1.\ \Box[\vee\ CB(N).pop \\ \vee\ \exists\, a\ :\ CB(N).push(a) \\ \vee\ \exists\, k\ :\ CB(N).move(k)]_{Buffer} \\ \Rightarrow \Box[\vee\ \overline{AB(N).pop} \\ \vee\ \overline{\exists\, a\ :\ AB(N)push(a)}]_{\overline{buffer}}$$

by proving

$$\langle 5\rangle 1.\ (\vee\ CB(N).pop \\ \vee\ \exists\, a\ :\ CB(N).push(a) \\ \vee\ \exists\, k\ :\ CB(N).move(k) \\ \vee\ Buffer' = Buffer) \\ \Rightarrow (\vee\ \overline{AB(N).pop} \\ \vee\ \overline{\exists\, a\ :\ AB(N)push(a)} \\ \vee\ \overline{buffer}' = \overline{buffer})$$

$\langle 5\rangle 2.$ Q.E.D.

Proof: directly by the TLA rule

STL.4 $\dfrac{F \Rightarrow G}{\Box F\ \Rightarrow\ \Box G}$

Let's do it......

$\langle 5 \rangle 1.$ $(\vee\ CB(N).pop$
$\vee\ \exists\, a\ :\ CB(N).push(a)$
$\vee\ \exists\, k\ :\ CB(N).move(k)$
$\vee\ \mathit{Buffer}' = \mathit{Buffer})$
$\Rightarrow (\vee\ \overline{AB(N).pop}$
$\vee\ \overline{\exists\, a\ :\ AB(N)push(a)}$
$\vee\ \overline{\mathit{buffer}}' = \overline{\mathit{buffer}})$

Proof:

$\langle 6 \rangle 1.$ $CB(N).pop \Rightarrow \overline{AB(N).pop}$

$\langle 6 \rangle 2.$ $\exists\, a\ :\ CB(N).push(a) \Rightarrow \exists\, a\ :\ \overline{AB(N).push(a)}$

$\langle 6 \rangle 3.$ $\exists\, k\ :\ CB(N).move(k) \Rightarrow \overline{\mathit{buffer}}' = \overline{\mathit{buffer}}$

$\langle 6 \rangle 4.$ $\mathit{Buffer}' = \mathit{Buffer} \Rightarrow \overline{\mathit{buffer}}' = \overline{\mathit{buffer}}$

$\langle 6 \rangle 5.$ Q.E.D.

Proof: Follows by propositional logic from $\langle 6 \rangle 1$, $\langle 6 \rangle 2$, $\langle 6 \rangle 3$, and $\langle 6 \rangle 4$:

$$(A \Rightarrow X) \wedge (B \Rightarrow Y) \wedge (C \Rightarrow Z) \wedge (D \Rightarrow Z)$$
$$\Rightarrow (A \vee B \vee C \vee D \Rightarrow X \vee Y \vee Z) \; \Box$$

Now, part of this says:
a concrete $push$ is an abstract $push$.

$\langle 7 \rangle$1. $\exists\, a : CB.push(a)$

$$\Rightarrow$$

$$\overline{\exists\, a : AB.push(a)}$$

Proof:

Let: a : constant

$\langle 8 \rangle$1. $CB.push(a)$

$$\Rightarrow$$

$$\overline{AB.push}(a)$$

$\langle 8 \rangle$2. Q.E.D.

Follows from $\langle 8 \rangle$1 by predicate logic. $\square$

$\langle 8 \rangle 1.$ $CB.push(a)$

$$\Rightarrow$$

$$\overline{AB.push}(a)$$

Proof:

$\langle 9 \rangle 1.$ $CB.push(a) \Rightarrow a \in Data$

Proof: Immediate from the definition of $CB.push$. $\square$

$\langle 9 \rangle 2.$ $CB.push(a) \Rightarrow Len(\overline{buffer}) < N$

$\langle 9 \rangle 3.$ $CB.push(a) \Rightarrow \overline{buffer}' = \overline{buffer} \circ \langle a \rangle$

$\langle 9 \rangle 4.$ Q.E.D.

Proof: Follows immediately from $\langle 9 \rangle 1$, $\langle 9 \rangle 2$ and $\langle 9 \rangle 3$ using propositional logic. $\square$

$\langle 10 \rangle 1.$ $CB.push(a) \Rightarrow Len(\overline{buffer}) < N$

Proof:

$\langle 11 \rangle 1.$ $CB.push(a) \Rightarrow Buffer[N] = \bot$

Proof: Immediate from the definition of $CB.push(a)$. $\square$

$\langle 11 \rangle 2.$ $Buffer[N] = \bot \Rightarrow$
$Len(SelectSeq(Buffer, NonVoid)) < N$

Proof: Follows from the definition of $SelectSeq$ and Len, along with a certain amount of data structure manipulation, which is omitted. $\square$

$\langle 11 \rangle 3.$ Q.E.D.

Proof: Follows from $\langle 11 \rangle 1$, $\langle 11 \rangle 2$ and the definition of $\overline{buffer}$ by propositional logic. $\square$

What do all these symbols mean?

Let:
$$\begin{aligned}
&NonVoid(k) &\triangleq\;& k \neq \bot \\
&\overline{buffer} &\triangleq\;& SelectSeq(Buffer, NonVoid) \\
&FirstFull &\triangleq\;& Buffer[1] \neq \bot \\
&NotEmpty &\triangleq\;& \exists\, i \in 1N \;:\; Buffer[i] \neq \bot
\end{aligned}$$

module *Sequences*

import *Naturals*

$mn \triangleq \{i \in Nat : (m \leq i) \wedge (i \leq n)\}$

$Len(s) \triangleq$

$\quad \text{choose } n : (n \in Nat) \wedge ((\text{Domain } s) = (1n))$

$Head(s) \triangleq s[1]$

$Tail(s) \triangleq [i \in 1(Len(s) - 1) \mapsto s[i + 1]]$

$s \circ t \triangleq$

$\quad [i \in 1(Len(s) + Len(t)) \mapsto$

$\qquad \textbf{if } i \leq Len(s) \textbf{ then } s[i]$

$\qquad\qquad \textbf{else } t[i - Len(s)]]$

$Seq(S) \triangleq \text{union } \{[(1n) \rightarrow S] : n \in Nat\}$

$SubSeq(s, m, n) \triangleq$

$\quad [i \in (1(1 + n - m)) \mapsto s[i + m - 1]]$

module *Sequences (continued)*

$$
\begin{array}{l}
\mathit{SelectSeq}(s, \mathit{test}()) \;\triangleq \\
\textbf{let}\;\; F[\,t : \mathit{Seq}(\{s[i] : i \in (1 \mathit{Len}(s))\})\,] \;\triangleq \\
\qquad\qquad \textbf{if}\;\; t = \langle\,\rangle \;\; \textbf{then}\;\; \langle\,\rangle \\
\qquad\qquad\qquad\qquad\qquad \textbf{else}\;\; \textbf{if}\;\; \mathit{test}(\mathit{Head}(t)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle\,\mathit{Head}(t)\,\rangle \circ \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad F[\,\mathit{Tail}(t)] \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\;\; F[\,\mathit{Tail}(t)] \\
\textbf{in}\;\; F[s]
\end{array}
$$

$\langle 10 \rangle 1.$ $CB.push(a) \Rightarrow \overline{buffer}' = \overline{buffer} \circ \langle\, a \,\rangle$

Proof:

$\langle 11 \rangle 1.$ $CB.push(a) \Rightarrow$

$$SelectSeq(Buffer, NonVoid) = SelectSeq([i \in 1(N-1) \mapsto Buffer[i]], NonVoid)$$

Proof:

$\langle 12 \rangle 1.$ $CB.push(a) \Rightarrow Buffer[N] = \perp$

Proof: Immediate from the definition of $CB.push(a)$. $\square$

$\langle 12 \rangle 2.$ $Buffer[N] = \perp$

$$\Rightarrow SelectSeq(Buffer, NonVoid) = SelectSeq([i \in 1(N-1) \mapsto Buffer[i]], NonVoid)$$

Proof: Follows immediately from the definition of $SelectSeq$ using manipulations of the data structure. $\square$

$\langle 12 \rangle 3.$ Q.E.D.

Proof: Follows immediately from $\langle 12 \rangle 1$ and $\langle 12 \rangle 2$ by propositional logic. $\square$

$\langle 11 \rangle 1.$ (was on last slide)

$\langle 11 \rangle 2.$ $CB.push(a)$

$$\Rightarrow$$

$$Buffer' = [i \in 1(N-1) \mapsto Buffer[i]] \circ \langle a \rangle$$

Proof: Follows from the definition of $CB.push$ and the sequence operations. $\square$

$\langle 11 \rangle 3.$ $SelectSeq([i \in 1(N-1) \mapsto Buffer[i]] \circ \langle a \rangle, NonVoid) = SelectSeq([i \in 1(N-1) \mapsto Buffer[i]], NonVoid) \circ \langle a \rangle$

Proof: Follows from the definition of $SelectSeq$ and $NonVoid$. $\square$

$\langle 11 \rangle 4.$ Q.E.D.

Proof: Follows immediately from $\langle 11 \rangle 1$, $\langle 11 \rangle 2$, $\langle 11 \rangle 3$ by propositional logic, substitution, and the definition of $\overline{buffer}$. $\square$