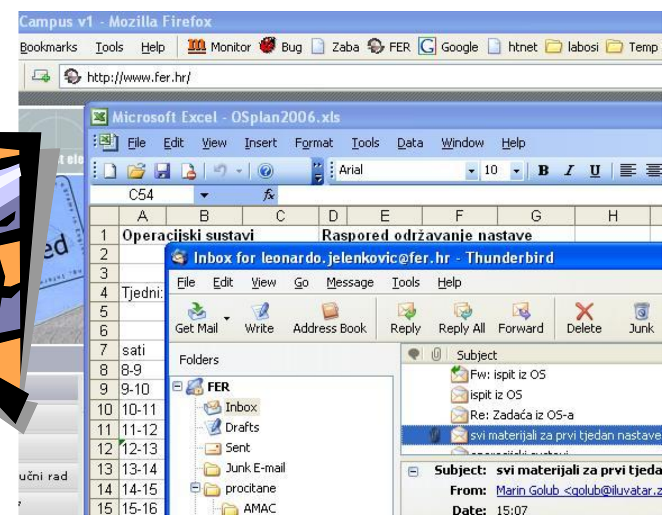
Operating system concepts

Introduction

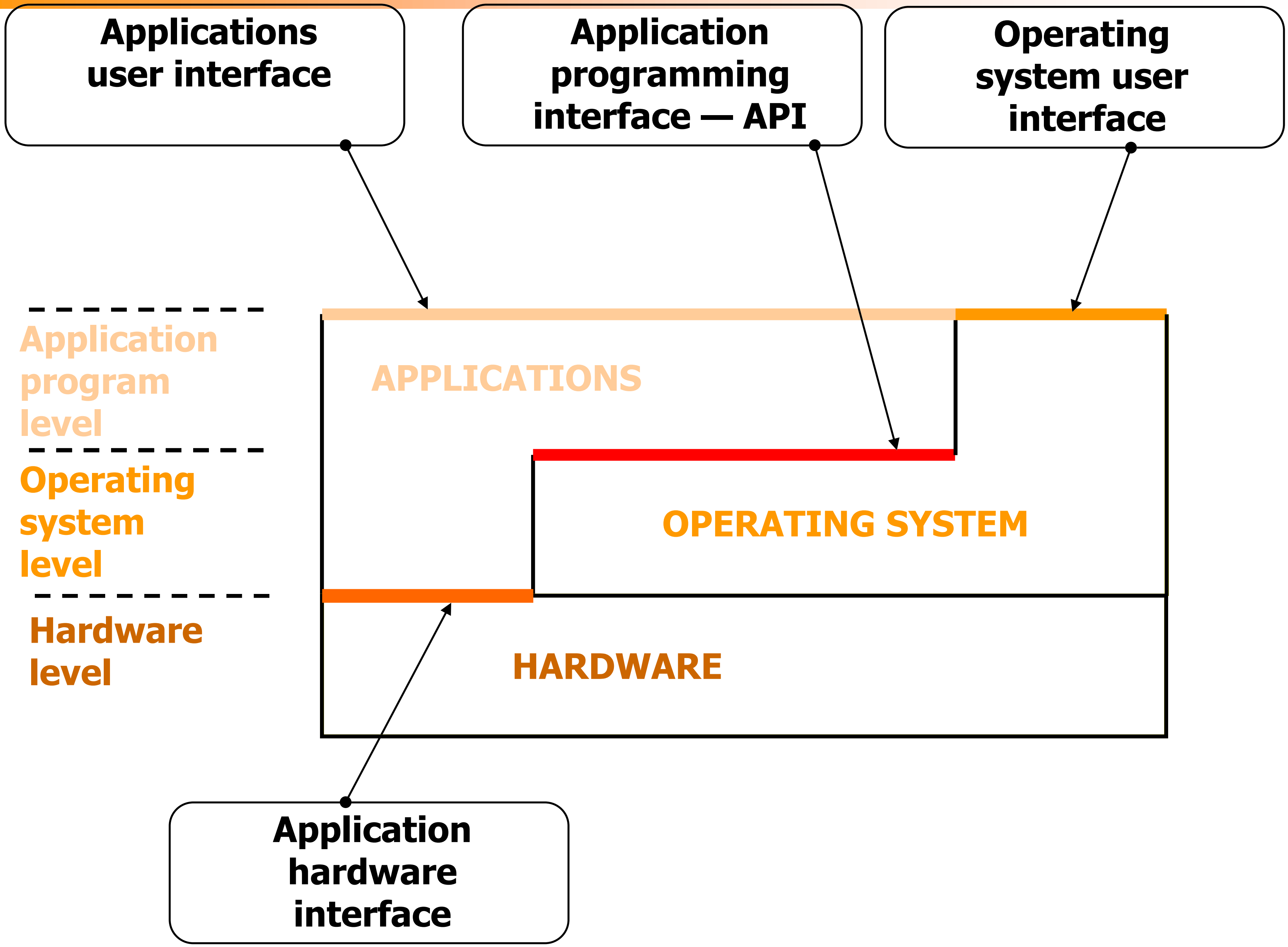
1. Introduction

- Operating system:
  - support for various *application programs*
  - *a collection of programs* which facilitate user operations on a computer
- name: *operating the system…*

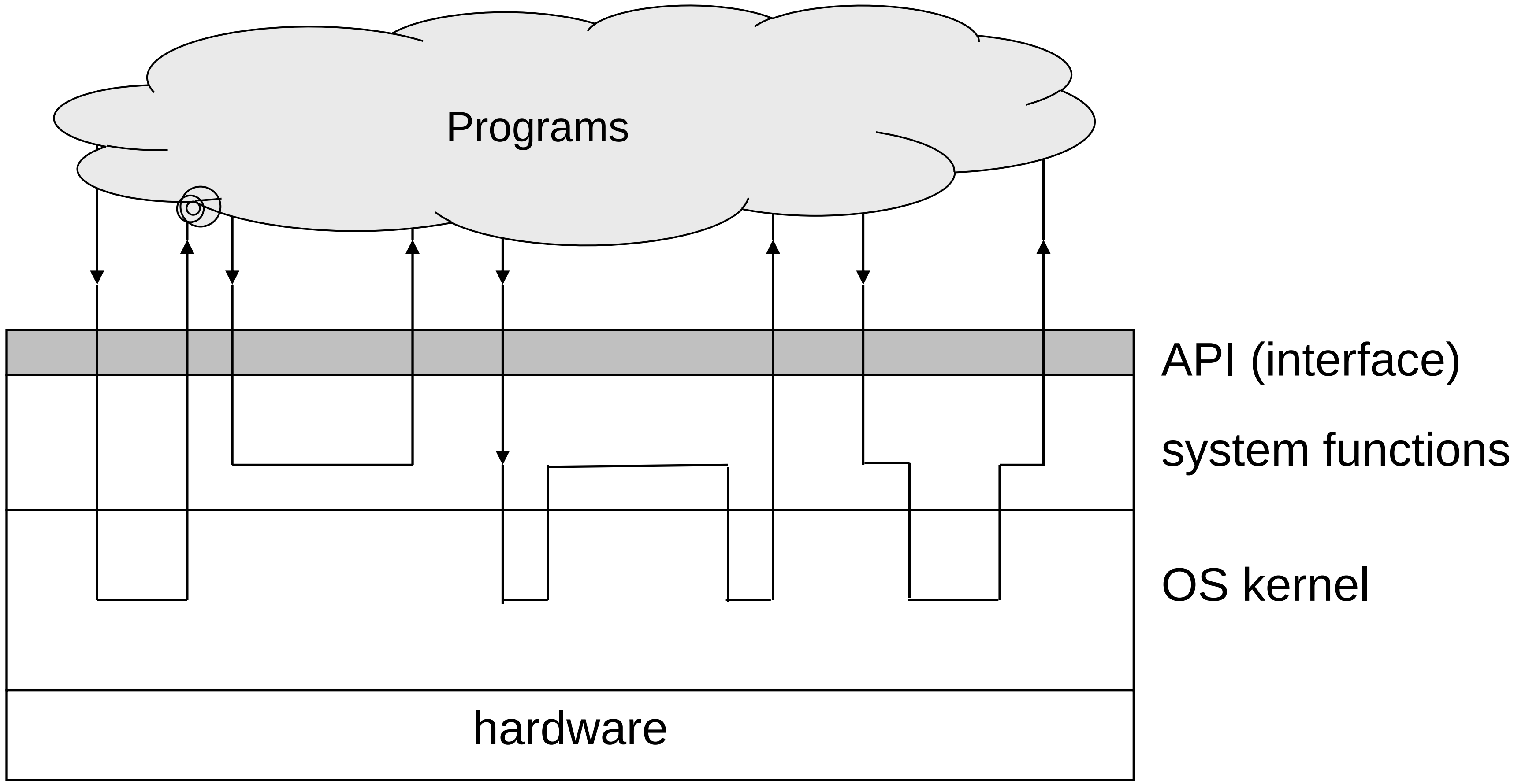


Connecting the components: interfaces

- all OS operations must have a defined interface
- interface determines
  - a way to initiate operations
  - the form of results

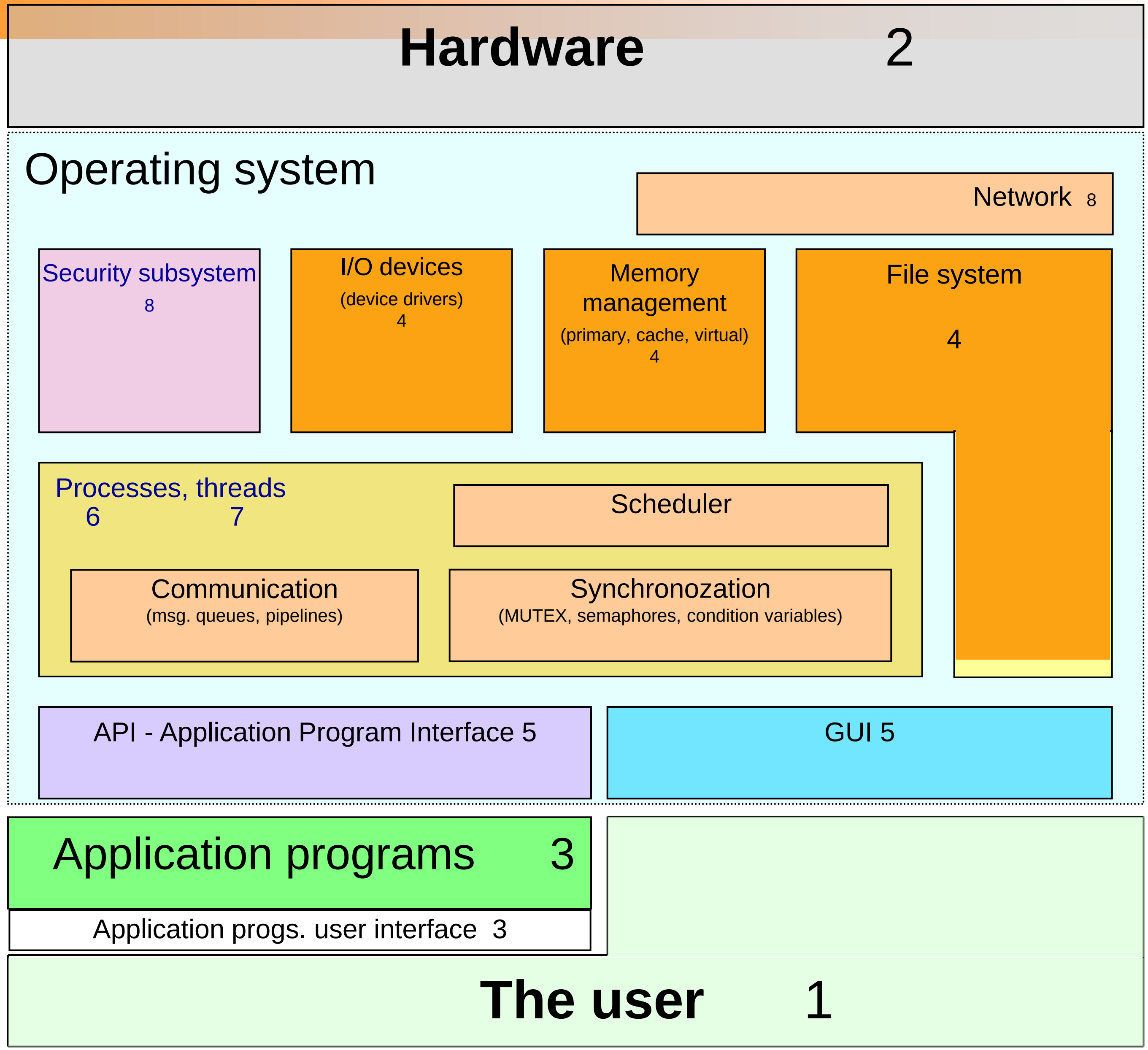

Applications user interface
Application programming interface — API
Operating system user interface
Application program level
Operating system level
Hardware level
APPLICATIONS
OPERATING SYSTEM
HARDWARE
Application hardware interface

Another layered view of the operating system



- **OS kernel**
  - contains minimum data and code (what is considered a *minimum*?)
  - performs basic OS tasks (e.g. handling interrupt requests, manipulating threads/processes/ memory)
- **system functions**
  - use kernel (functions) to implement OS functionality
  - exposed to programmer/user through API/GUI

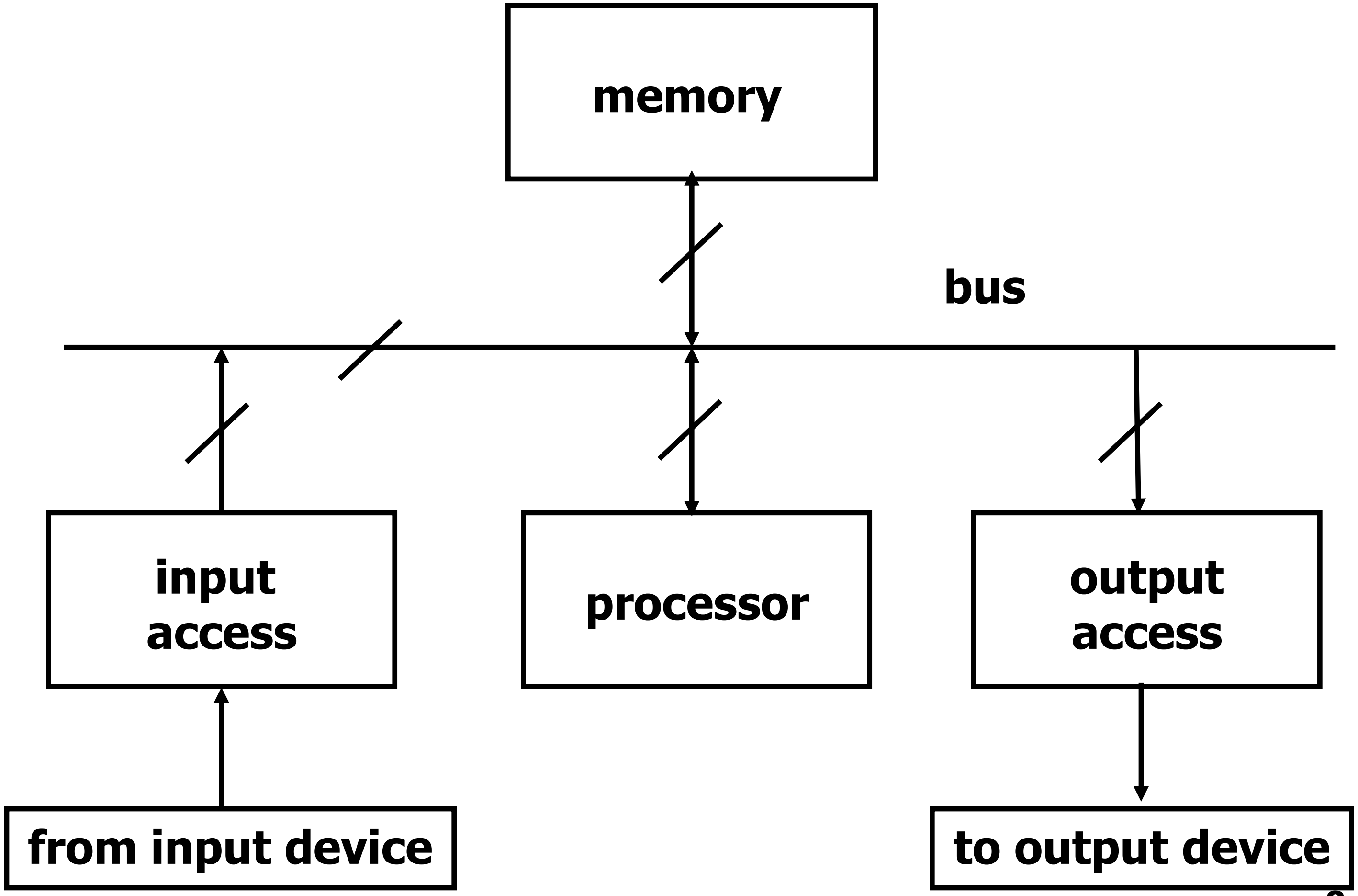
- *How exactly a programmer uses this functionality?*
- *What happens when he does?*
  - will try to give those answers throughout the lectures

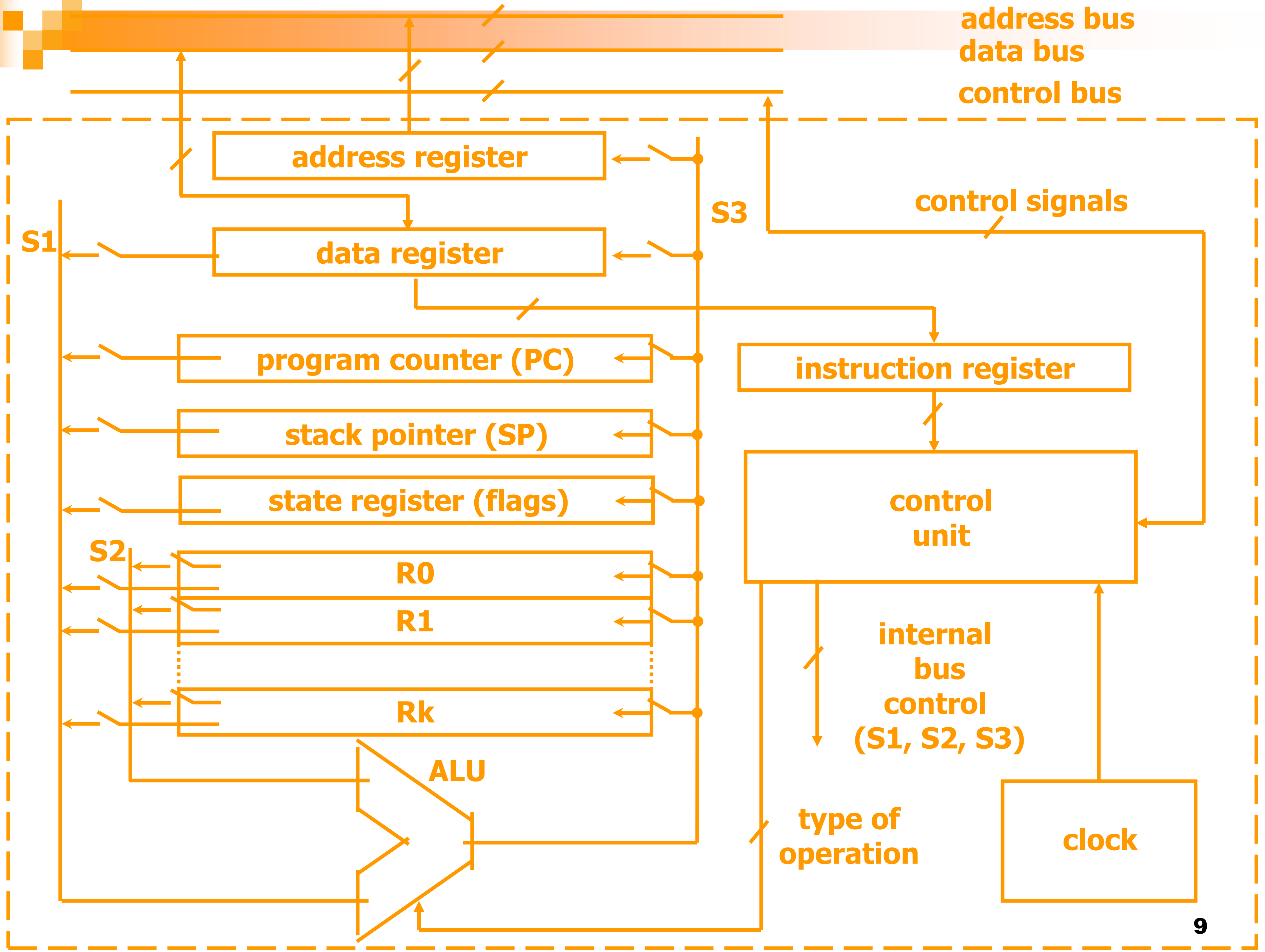


Computer system

Bus-based computer architecture

- all types of data transfer, control signals and instructions are relayed through a (common) *bus*
- bus transfer: divided in *bus cycles*





address bus
data bus
control bus
address register
S3
control signals
S1
data register
program counter (PC)
instruction register
stack pointer (SP)
state register (flags)
control
unit
S2
R0
R1
internal
bus
control
(S1, S2, S3)
Rk
ALU
type of
operation
clock

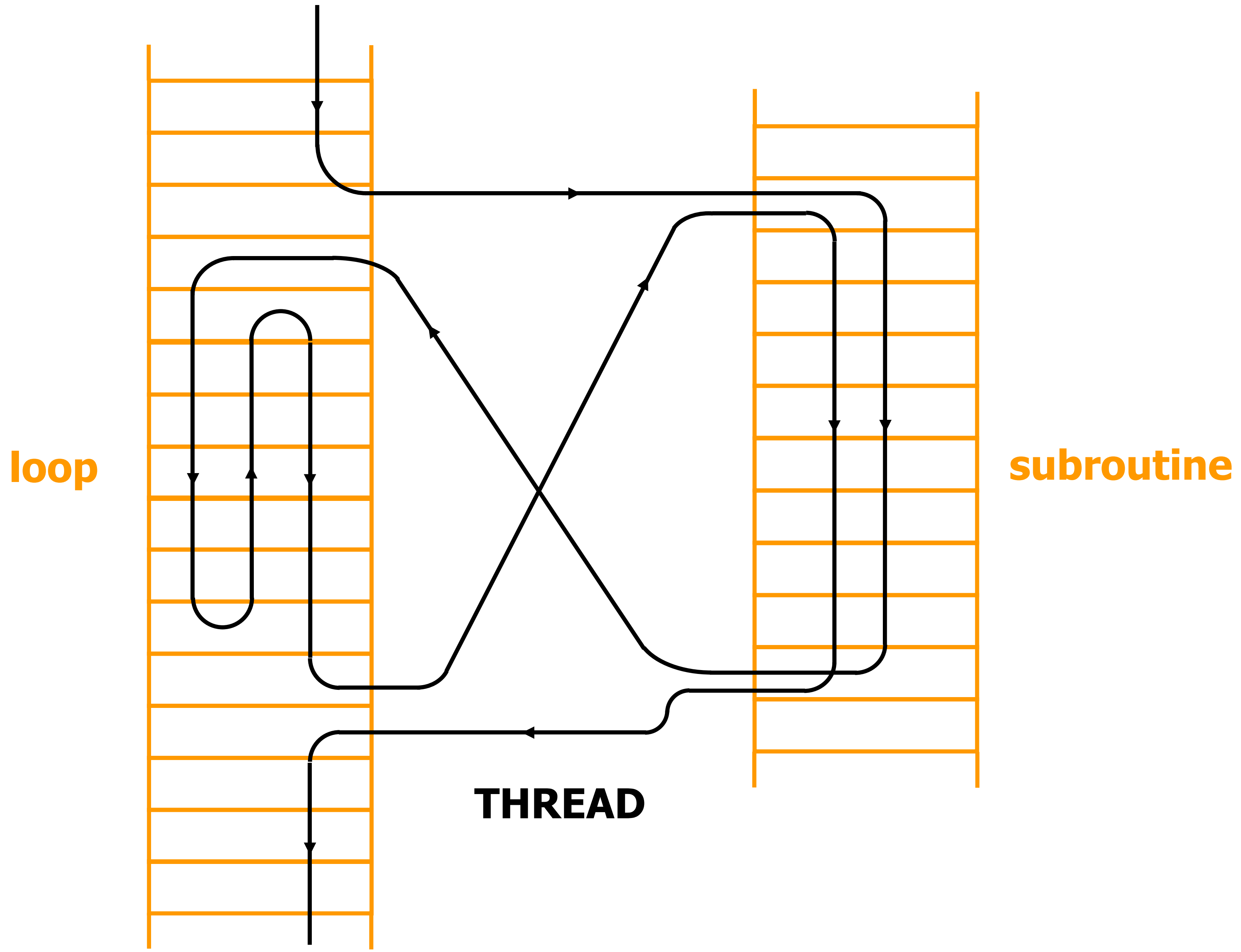
- the processor: an automaton which repeatedly executes a micro-program

```
repeat{
    fetch instruction from memory location stored in program
    counter (PC);
    decode instruction, determine operation;
    increase program counter;
    determine operands’ and result locations;
    relay operands to ALU, perform operation;
    store result;
}
while(power on);
```

Instruction thread

- memory contents may be divided in
  - instruction segment (series of machine instructions)
  - stack segment
  - data segment

- when executing a series of machine instructions, the processor executes an *instruction thread* (short: *thread*)
- basic distinction of a thread: a series of instructions *in time*
  - (as opposed to a series of instructions in memory space)


loop
subroutine
THREAD

Computer process

- *program*: a static object (on some media)
- a program, loaded in memory, given appropriate computer resources and started, becomes a dynamic object
- *process*: an environment in which a program is executed
- operating system provides appropriate resources (e.g. memory space, CPU time, I/O access) for processes

- each process is executed in *at least one* instruction thread

- a process may contain multiple threads which may execute in *parallel*
  - threads may execute *concurrently* if more processors are available
  - threads may interchange if only one processor is available (virtually parallel)
- each thread should be able to use a part of primary memory
- performing multiple threads: *multithreading*

Thread context

- the contents of processor registers during thread execution is *thread context*
- switching threads in multithreading systems:
  1. interrupting the current thread
  2. saving its context
  3. restoring (some other) thread’s context
  4. restarting the thread
- exchange of context makes the apparent parallel thread execution possible (as if each thread runs on its own processor)
- *context switch* is a basic mechanism on which the OS is built upon

- when does context switch occur?
  - how do we ‘exchange’ threads?
- only after an *interrupt*

2. Interrupts

- interrupt subsystem – evolved to facilitate I/O device control
  - instead of CPU checking whether a device is *ready*, the device signals the CPU (requests an interrupt)
- CPU checks for interrupt request at the end of each instruction cycle
  - interrupt service includes a context switch

- who handles interrupt requests?
  - not the user thread!

Some interrupt scenarios:

- a device is ready → service the device
- a time interval has elapsed → schedule some other thread/process (and user) on the processor!
- an illegal instruction is encountered → stop the current process
- …
  - all these scenarios require OS intervention

Interrupt support in CPU hardware:

- OS must execute with the highest available priority
  - i.e. access to all hardware resources
- user thread should not have that access!

Interrupt processing – processor operating modes

- most (non-trivial) processors may operate in several modes:
  - user mode, “normal tasks” (unprivileged)
  - interrupt processing mode
  - system mode, “system tasks” (may be the same as interrupt processing mode)
- in different modes, alternate functionality and resources are available (registers, memory, stack, IO, instructions…)

Upon interrupt request (IRQ), the processor:

- disables (additional) interrupts
- adresses the system memory space and system stack
- saves current program counter (PC) value (and thread context) to system stack
- loads the IRQ subroutine address in program counter

- IRQ subroutine address:
  - hardwired in a specific CPU architecture
  - stored in a specific register
- several IRQ types and addresses usually exist
- hardware device interrupts usually handled by *device drivers* – programs written to handle specific device (*understand* device and its operations)
- device drivers must be *registered* for interrupts of controlling device

- servicing an IRQ is *expensive*
  - the thread context must be preserved – which may include many instructions
- after the context switch, the IRQ *processing* may begin
  - includes a vast variety of data transfer/control
  - duration unknown in advance
- when processing is over, (interrupted) user thread may continue (context restoration)
- problem for many (real time) systems:
  - a low priority interrupt processing may significantly delay a high priority event (signaled by an interrupt)
  - typical solutions include:
    - associate *priority* with interrupts and process them accordingly – interrupt service routine is interruptible
    - divide interrupt processing into two parts:
      - short interrupt processing (non-interruptible)
      - long interrupt processing (interruptible)

Interrupt servicing subsystem:

- disables new interrupts during context switch and internal logic
- *enables* interrupts during IRQ processing
- new interrupt with a higher *priority* will stop a lower priority processing
- new interrupt with a lower priority is put on hold until priority level is lowered

Software interrupts

- How can a user thread:
  - access an I/O device?
  - check if a resource (memory, device) is available?
  - wait for another thread to ‘finish’?
  - exchange message with another thread/process?
  - …

- these operations not available to user threads, because:
  - simultaneous access may invalidate data
  - a thread in error could compromise the system
  - waiting thread should be removed from processor

- need for a unified mechanism

- thread makes a *system call* – invokes a system function and/or OS kernel

- OS kernel is invoked via *software interrupt*
  - reserved instruction in CPU architecture
  - types of interrupts may be conveyed with register values

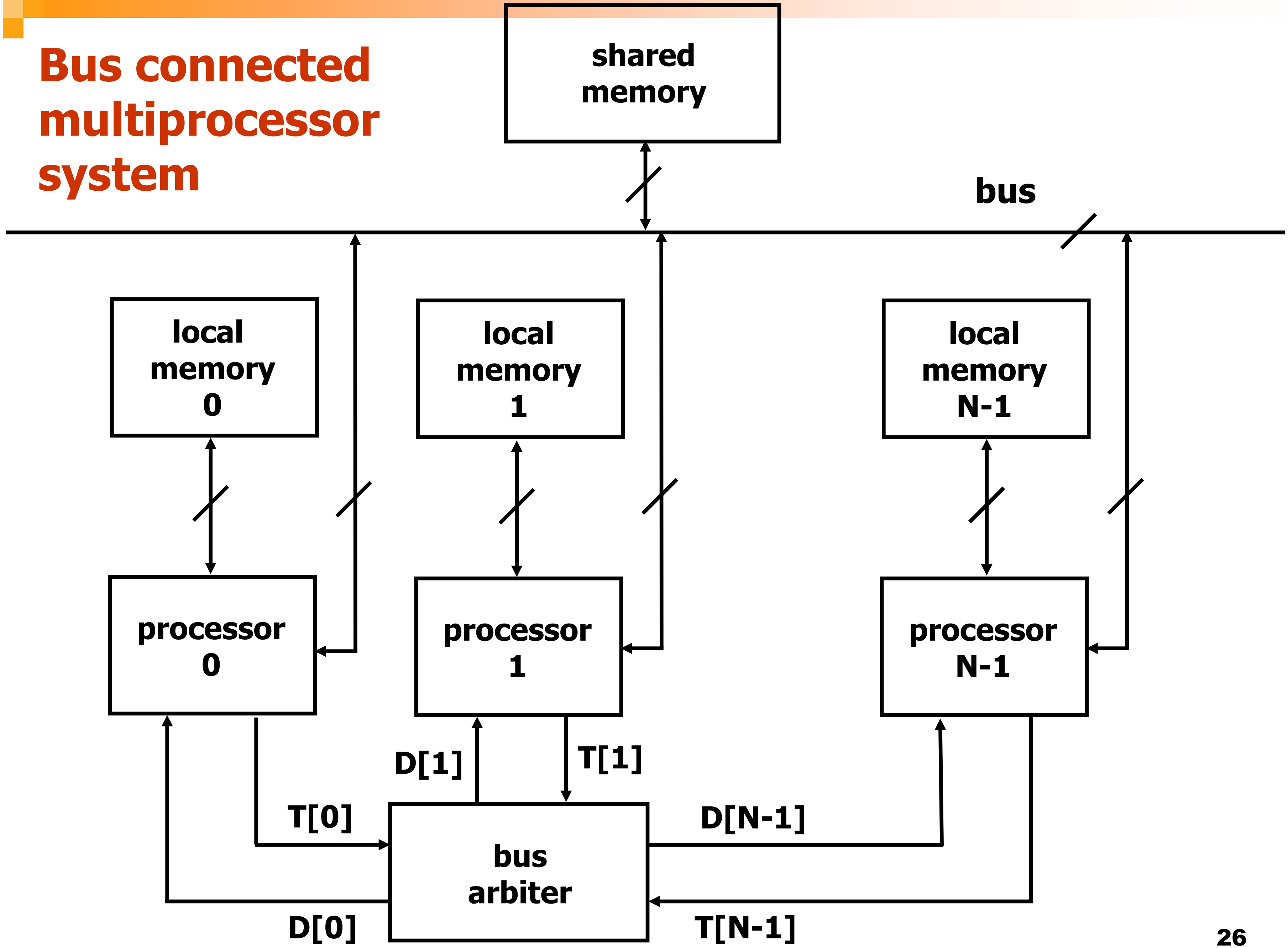
3. Multiprocessor systems, OS kernel

- OS must function the same way regardless of the number of processors

- assumption: there exists a single *shared memory storage* where all kernel data is located
  - users/processes data
  - threads data (contexts)
  - memory management data, I/O data …

- What is a *multiprocessor system*?

Bus connected multiprocessor system



- every processor may access the shared memory
  - only one processor in a single bus cycle!
- bus arbiter: decides who gets the bus

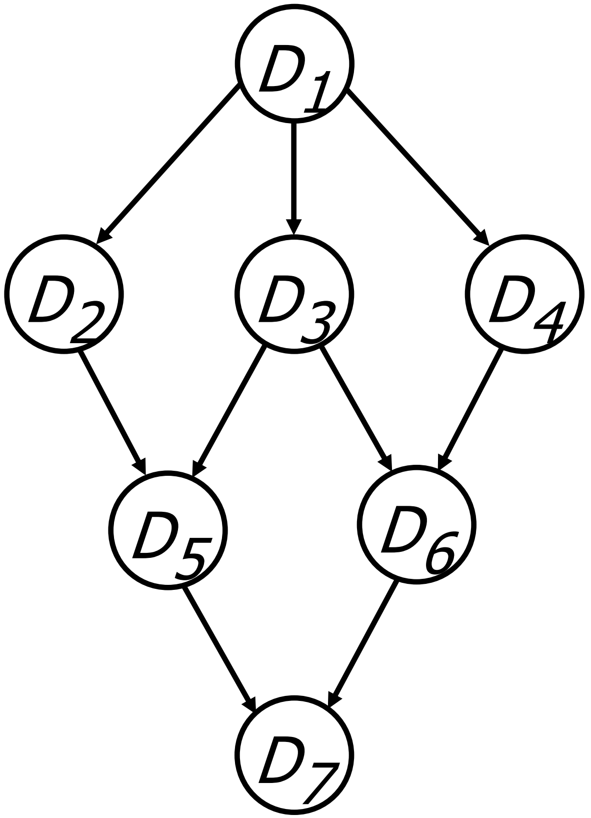
- an SMP system: identical processors/processor cores
- consequence: any thread may execute on any processor

Multitasking

- multitasking (or parallel computing) goals:
  - taking advantage of multiple processing elements
  - alleviating the design and implementation of complex applications
- multitasking – an OS feature
- programming techniques:
  - multithreading (within a single process)
  - multiprocessing (several processes)
  - a combination of both
- multiple processes:
  - execute independently
  - share no data or resources
  - must use OS mechanisms to cooperate!
- multiple threads:
  - execute in a single memory space
  - share all process resources

Multithreaded programming model

- a set of (mutually dependant) tasks
- taskst must use *synchronization primitives* to execute in predefined order and use shared resources
  - do so by calling system/kernel functions
- OS must provide mechanisms for:
  - thread creation
  - thread termination
  - synchronization
  - data exchange (messaging)



Mutual exclusion

- a form of thread synchronization
- *examples*: acces to shared device, global data structure
- a part of thread code using a shared resource: *critical section*
  - critical sections must be executed exclusively – only one thread at any given moment (and processor!)
- part of code not using the resource: *non-critical section*
- mutual exclusion mechanism: provides exclusive use of shared resources

- programming a mutual exclusion synchronization:
  - user thread does not test the condition itself
  - rather, a call to a *kernel function* must be made
- the kernel function:
  - either allows the calling thread to continue
  - or stores the thread’s context and restarts another thread

Operating system kernel

- a set of functions and data structures
- OS kernel primary tasks:
  - thread manipulation (allow multithreaded execution)
  - thread synchronization
  - I/O devices control