V. The Operating System

The Role of the Operating System (1)

- program that manages the computer
- links the hardware to the applications
- provides various services to the applications
- supervises the way applications work
  - can take control when errors occur

The Role of the Operating System (2)

- needs hardware support to carry out its duties
  - the most important - the interrupt system
- main components
  - kernel
  - drivers

V.1. The Kernel

The Kernel

- mostly independent from the hardware structure it works with
- "the brain" of the operating system
- manages the computing resources - hardware and software
  - proper working
  - fair allocation between the applications

Processor's Operating Modes

- user mode
  - restricted
  - access to memory - only certain zones
  - access to peripheral devices - denied
- kernel mode
  - no restrictions

How Programs Run

- operating system - kernel mode
  - can perform any operation
- applications - user mode
  - cannot perform certain actions
  - must ask the kernel to perform those actions for them

Switching between the Two Modes

- through the interrupt system
- user $\rightarrow$ kernel
  - a software interrupt is called
  - an exception (error) occurs
- kernel $\rightarrow$ user
  - return from the interrupt handling routine

Consequences

- no application code can run in kernel mode
- advantage: the errors of an application do not affect other programs
  - other applications
  - the operating system itself
- drawback: loss of performance

The Structure of the Kernel

- not a unitary program
- more like a collection of cooperating routines
- main functions
  - process management
  - memory management
  - file system management
  - inter-process communication

V.2. System Calls

System Calls

- requests made by applications to the kernel
- actions that applications may not execute themselves
  - can only be performed in processor's kernel mode
  - reason - system safety
- achieved through software interrupts
- similar to function calls

The Phases of a System Call (1)

1. the program lays the parameters of the system call into a certain zone in memory
2. a software interrupt is generated
   - processor switches to kernel mode
3. the requested service is identified and the corresponding routine is called

The Phases of a System Call (2)

4. the routine takes the parameters and checks them
   – if there are errors - the call fails
5. if there are no errors - the requested action is performed
6. the routine ends - results are laid into a memory zone that is accessible to the requesting application

The Phases of a System Call (3)

7. processor switches back to user mode
8. program execution is resumed from the point is had been interrupted
   - using the information memorized when interrupt was generated
9. the program can now take the results of the call

System Calls - Conclusions

- communication between the application and the kernel
  - lay the parameters
  - take the results
- critical actions are performed safely
- high time consuming
- make calls seldom - work with buffers

How We Use Buffers

Example - the *printf* function

- formats the text, then sends it to screen
- no direct access to hardware
  - uses a system call
  - *write* (the Linux case)
- in fact, *printf* keeps the formatted text into a local memory zone (buffer)
  - a system call is made when the buffer is full

V.3. Drivers

What Are Drivers?

- program modules which manage communication to peripheral devices
  - specialized - a specific driver for each peripheral device
- part of the operating system
  - direct access to hardware
  - run in processor's kernel mode

Usage

- drivers are not part of the kernel
  - but are controlled by the kernel
- used by the hardware interrupt handling routines
- change a peripheral $\rightarrow$ change the driver
  - modular structure
  - no need to re-install the entire operating system

V.4. Process Management

Processes (1)

- one can launch more programs at the same time (multitasking)
- parallelism is not real
  - unless the system has more processors
  - otherwise - concurrency
- a program may be split into more sequences of instructions - *processes*
  - can be executed in parallel or concurrently

Processes (2)

- the operating system works with processes
  - not with programs
- when launched, a program consists in a single process
  - it can create other processes
  - which can create other processes, and so on
- at each moment, a processor can execute the instructions of a single process

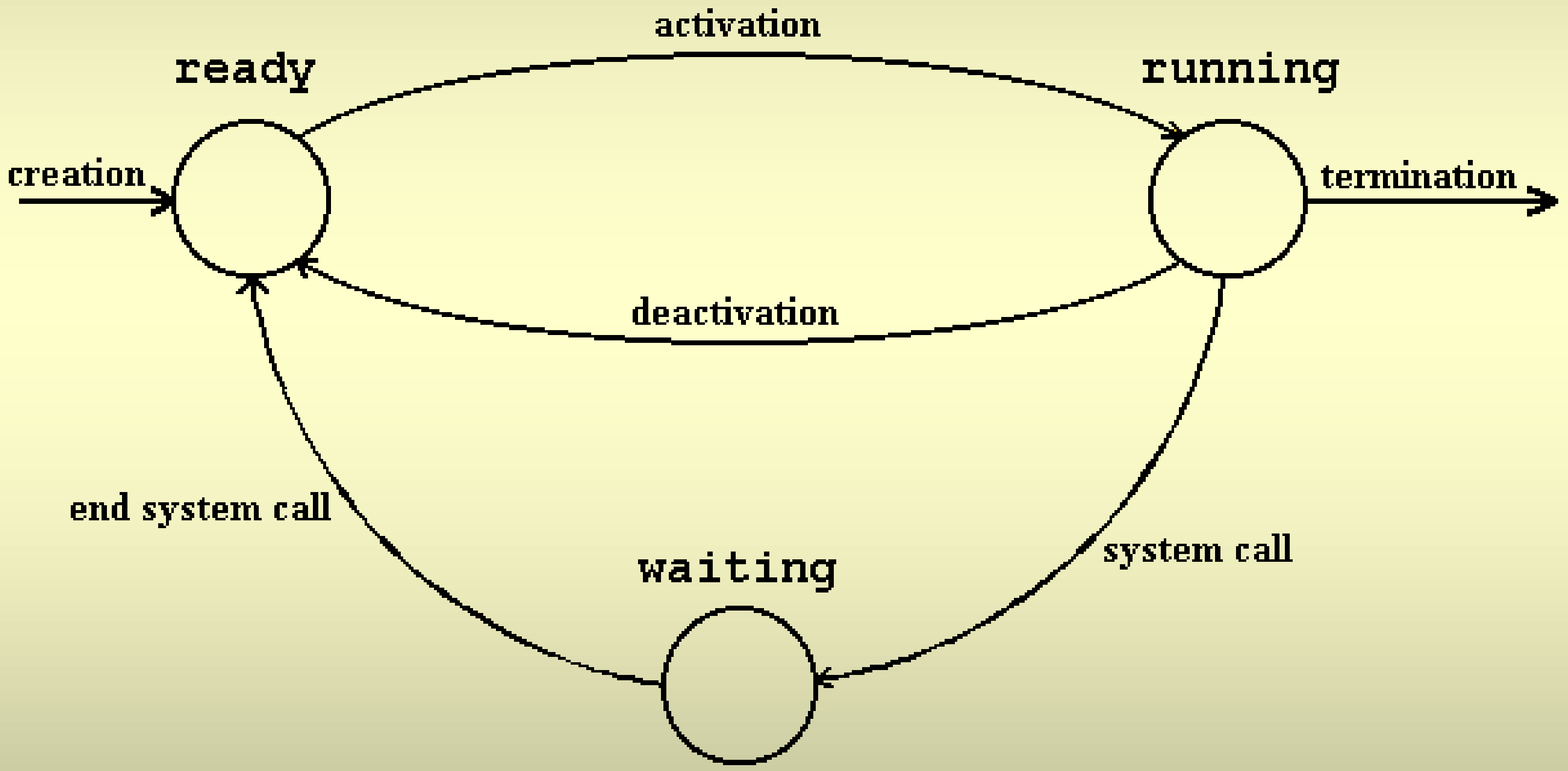
Processes (3)

- each process has its own memory zones (code, data, stack, ...)
  - separated from the other processes' zones
- when a new process is created, proper memory space is allocated to it
- when a process is finished, the memory occupied by that process is released
  - even if the program as a whole continues

States of a Process (1)

- running
  - its instructions are executed by the processor
- ready
  - waits to be executed by the processor
- waiting
  - waits for the completion of a system call
  - currently, it does not compete for being assigned to the processor

States of a Process (2)



States of a Process (3)

- the running process leaves this state
  - when it ends
    - normally or because of an error
  - when it makes a system call ($\rightarrow$ waiting)
  - when its instructions have been executed long enough and it is time for another process to be executed (deactivation)

Multitasking

- non-preemptive
  - does not allow the deactivation of a process
  - a process can stop executing only in one of the other situations described before
  - drawback - programming errors may block the other processes (e.g., an infinite loop)
- preemptive

Deactivation

- how does the operating system know how long has a process been executed?
- a way of measuring the time is required
- real-time clock
  – peripheral device
  – generates interrupt requests periodically

Threads (1)

- a process can be split into more execution threads
  - usually, a thread consists in the execution of a function (part of the process' code)
- threads share the resources of the process (memory, open files, etc.)
  - simpler communication - global variables
  - higher risk of (undesired) interference

Threads (2)

- when a process is scheduled to the processor, one of its threads will execute
  - so some kind of scheduling is needed here too
- who performs the scheduling?
- alternatives
  - the operating system (rather seldom)
  - the application - through dedicated library functions

V.5. Memory Management

Memory Management

Functions

- allocates memory zones to the applications
- prevents the interferences between applications from occuring
- detects and stops illegal accesses

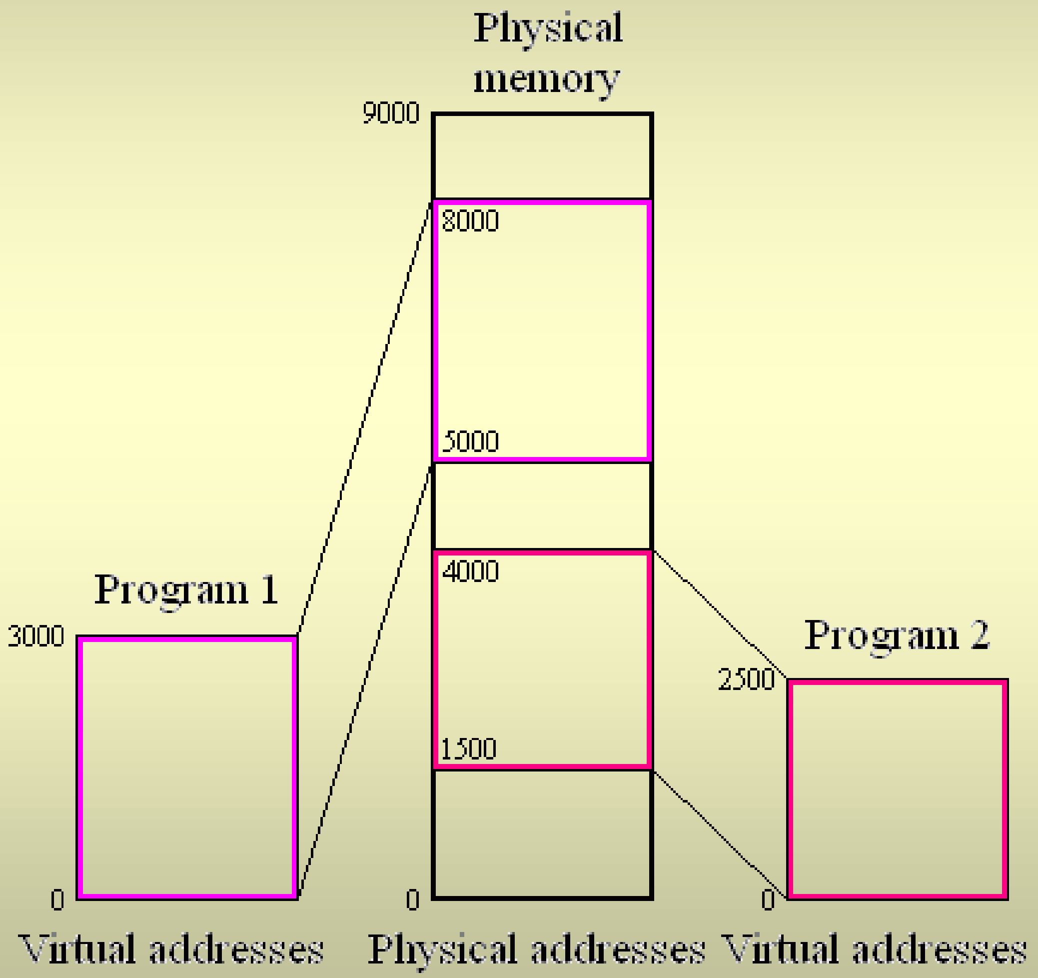
Fundamental Problem

- multiple applications running $\rightarrow$ separate memory zones
- each application $\rightarrow$ certain memory zones; which are these zones?
  - depend on memory configuration at that moment
  - cannot be known at compiling time

Solution

- two kinds of addresses
  - virtual - which the application believes to access
  - physical - really accessed by the processor
- the correspondence between virtual and physical addresses - managed by the operating system

Virtual and Physical Addresses (1)



Virtual and Physical Addresses (2)

Managing virtual and physical addresses

- 2 different methods
  - segmentation
  - pagination
- can also be used together
- dedicated component of the processor - MMU (Memory Management Unit)

V.5.1. Memory Segmentation

Basic Principle (1)

- segment - contiguous memory zone
- contains information of the same kind (code, data, etc.)
- visible to the programmer
- the address of a location - made of 2 parts
  - the start address of the segment
  - the offset within the segment

Basic Principle (2)

- upon different executions of the program, a segment starts at different addresses
- the effect on location addresses
  - the start address of the segment must be updated
  - the offset - not modified
- the problem is only partially solved
  - we want the address not to be modified at all

Descriptors (1)

- segment descriptor - data structure for managing a segment
- the information it keeps
  - start address
  - size
  - access rights
  - etc.

Descriptors (2)

- descriptors - grouped into a table
- access to a segment - based on the index in the descriptor table (selector)
- virtual address - 2 components
  - the index in the descriptor table
  - the offset within the segment
- physical address = start address of the segment + offset

Descriptors (3)

- upon different executions of the program, a segment starts at different addresses
- the effect on location addresses
  - none
  - must only modify the start address of the segment in the descriptor
  - only once (when the segment is loaded into memory)
  - performed by the operating system

Memory Access (1)

- the program requests the virtual address
- the segment descriptor is identified
- the access rights are checked
  – insufficient rights - a trap is generated
- the offset is checked
  – if the offset exceeds the segment size - a trap is generated

Memory Access (2)

- if an error has occurred on previous steps
  - the service routine of the trap ends the program
- if no error has occurred
  - compute the physical address (segment start address + offset)
  - access the location at the computed address

Illustration (1)

Descriptor table (simplified)

Index	Start address	Size
0	65000	43000
1	211000	15500
2	20000	30000
3	155000	49000
4	250000	35000

Illustration (2)

Example 1:

```
mov byte ptr ds:[eax],25
```

- ds = 3 → segment start address = 155000
- eax = 27348 < 49000
  - valid offset (does not exceed the segment size)
- physical address: 155000 + 27348 = 182348

Illustration (3)

Example 2:

```
add dword ptr ss:[ebp],4
```

- ss = 1 → segment start address = 211000
- ebp = 19370 > 15500
  - invalid offset (exceeds the segment size)
  - error → generate trap

Intel Case (1)

3 descriptor tables

- global (GDT - Global Descriptor Table)
  – accessible to all processes
- local (LDT - Local Descriptor Table)
  – specific to each process
- interrupts (IDT - Interrupt Descriptor Table)
  – not directly accessible to the applications

Intel Case (2)

Segments - accessed through the selectors

Selector structure (16 bits)

- first 13 bits - the index in the descriptor table
  - at most 8192 descriptors/table
- 1 bit - the table used (global/local)
- last 2 bits - privilege level
  - 0 - highest, 3 - lowest

Intel Case (3)

31 24					19 16	15	14 13	12 8	7 0
BASE 31-24	G	D / B	0	A V L	LIMIT 19-16	P	DPL	TYPE	BASE 23-16
BASE 15-0						LIMIT 15-0			

Intel Case (4)

- privilege levels of 3 entities are considered

1. CPL (*Current Privilege Level*)
   – level of the process - kept by the processor
2. RPL (*Requested Privilege Level*)
   – the requested level - taken from the selector
3. DPL (*Descriptor Privilege Level*)
   – the accessed segment's level - from the descriptor

Intel Case (5)

- relations between these privilege levels decide if access is allowed
- the condition for allowing the access: CPL<=DPL and RPL<=DPL (simultaneously)
- any other situation indicates an attempted access to a level too high
  - a trap is generated