Real Time Programming: Concepts

Radek Pelánek

Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky.

EVROPSKÁ UNIE

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

OP Vzdělávání pro konkurenceschopnost

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Plan

- at first we will study basic concepts related to real time programming
- then we will have a look at specific programming languages and study how they realize these concepts

Real Time and Concurrency

- typical architecture of embedded real time system:
  - several input units
  - computation
  - output unit
  - data logging/storing
- i.e., handling several concurrent activities
- concurrency is natural for real time systems
- motivation: Johan Nordlander's slides

Concurrent Programming

- programming notation and techniques
- expressing potential parallelism and solving the resulting synchronization and communication problems
- implementation of parallelism is essentially independent of concurrent programming
- concurrent programming provides an abstract setting in which to study parallelism without getting bogged down in the implementation details

Automatic Interleaving

- interleaving of processes (threads) is automatic
- programmer doesn't have to execute specific instructions to make switching processes happen, or take specific action to save the local context when switching occurs
- programmer must be prepared that switching might occur at any time
- who does the switching?

Support for Concurrent Programming

- support by the programming language
  - examples: Ada, Java
  - advantages: readability, OS independence, checking of interactions by compiler
- support by libraries and the operating system
  - examples: C/C++ with POSIX
  - advantages: multi-language composition, possibly more efficient, OS standards

More about these issues in the next lecture.

Implementation of Concurrent Programming

multiprogramming processes multiplex their execution on a single processor

multiprocessing processes multiplex their execution on a multiprocessor system with access to shared memory

distributed processing processes multiplex their execution on several processors which do not share memory

Variation

Concurrent programming languages differ in:

- structure:
  - static: the number of processes fixed and known at compile time
  - dynamic: processes created at run-time
- level:
  - flat: processes are defined only at the outermost level of the program
  - nested: processes are allowed to be defined within another processes
- granularity:
  - coarse: few long-lived processes
  - fine: many short-lived processes

About Processes...

- what is process
- process vs thread
- lifecycle of a process – creation, termination
- interprocess relations

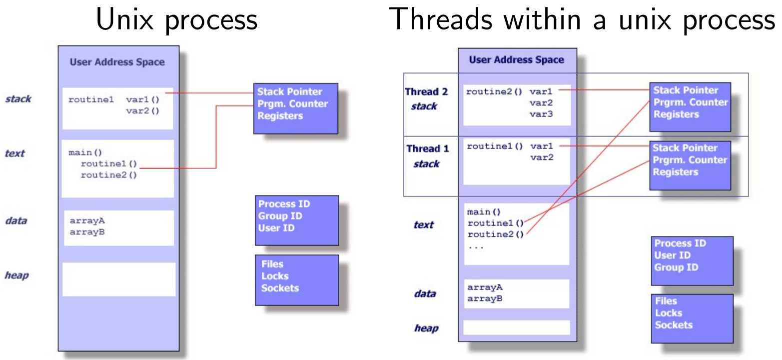
Process

- process is a running instance of a program
- processes execute their own virtual machine to avoid interference from other processes
- it contains information about program resources and execution state, e.g.:
    - environment, working directory,...
    - program instructions
    - registers, heap, stack
    - file descriptors
    - signal actions, inter-process communication tools (pipes, messages)

Thread

- exists within a process, uses process resources
- unique execution of machine instructions, can can be scheduled by OS and run as independent entities
- keeps it own: execution stack, local data, etc.
- share global process data and resources
- “lightweight” (compared to processes)

Processes and Threads



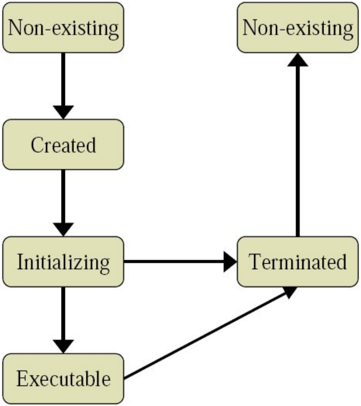
Threads: Resource Sharing

- changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads
- two pointers having the same value point to the same data
- reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer

Processes and Threads

- in most of the following we will not strictly distinguish between processes and threads
- we use ‘process’ as a general term

Process States



Process Representation

- explicit process declaration
- cobegin, coend
- fork and join

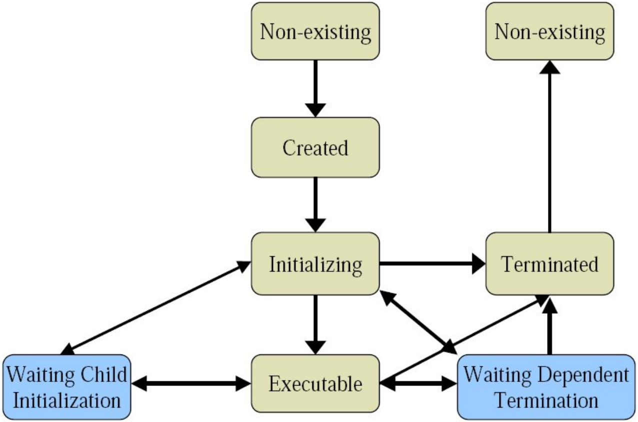
Process Termination

- completion of execution of the process body
- ‘suicide’ by execution of a self-terminate statement
- abortion, through the explicit action of another process
- occurrence of an error condition
- never (process is a non-terminating loop)

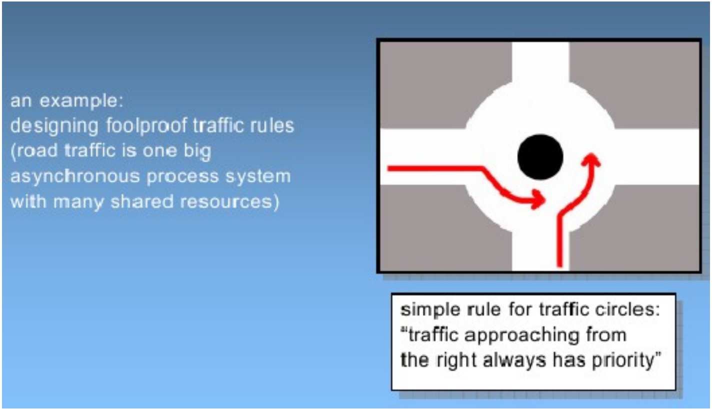
Interprocess Relations

- parent-child: a parent is a process that created a child; parent may be delayed while child is being created and initialized
- guardian-dependent: a guardian is affected by termination of a dependent

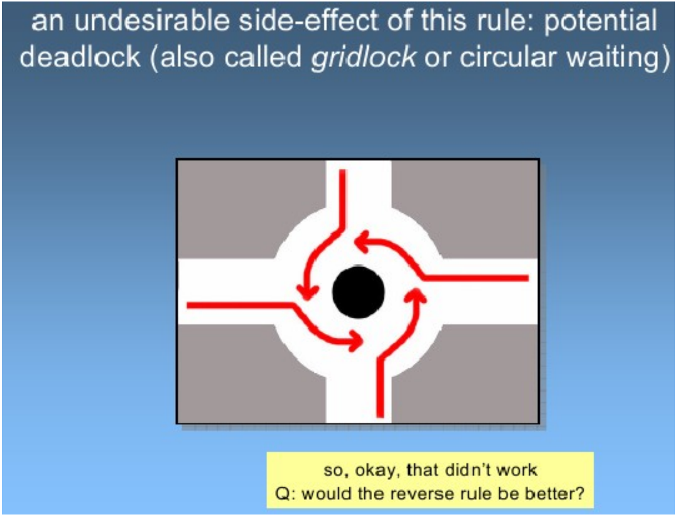
Process States II



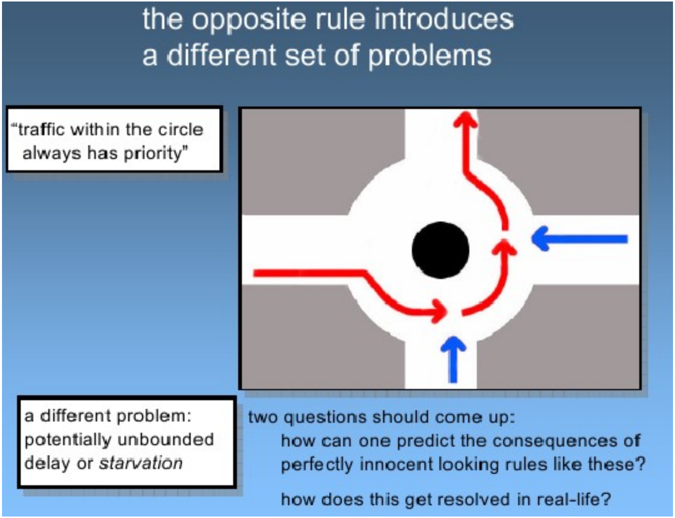
Concurrency is Complicated ...



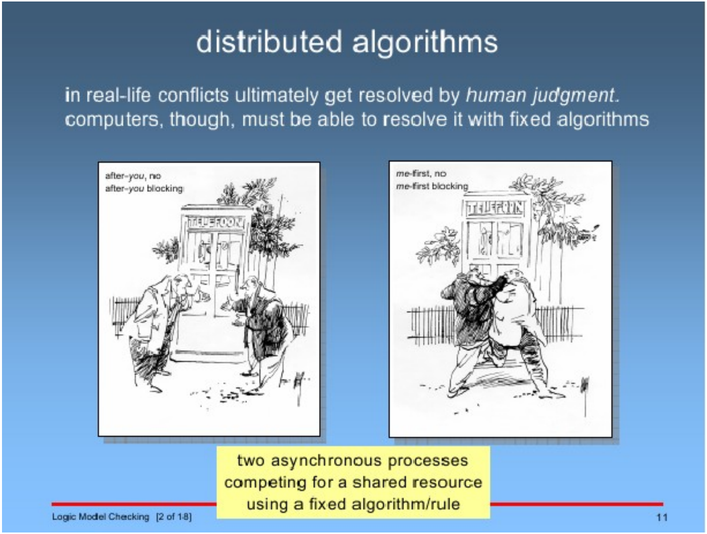
Source: G. Holzmann



Source: G. Holzmann



Source: G. Holzmann



Source: G. Holzmann

Puzzle

(puzzle illustrating that concurrency is complicated)

$$c := 1, x_1 := 0, x_2 := 0$$

$$\begin{array}{l} x_1 := c \\ x_1 := x_1 + c \\ c := x_1 \end{array} \quad \| \quad \begin{array}{l} x_2 := c \\ x_2 := x_2 + c \\ c := x_2 \end{array}$$

- Both processes repeat the given block of 3 commands.
- Can c attain value 5?
- Can c attain any natural value?

Communication and Synchronization

synchronization

- satisfaction of constraints on the interleaving of actions of processes
- e.g., action by one process occurring after an action by another

communication

- the passing of information from one process to another

Communication and Synchronization

Linked concepts:

- communication requires synchronization
- synchronization $\sim$ contentless communication

Data Communication

- shared variables
- message passing

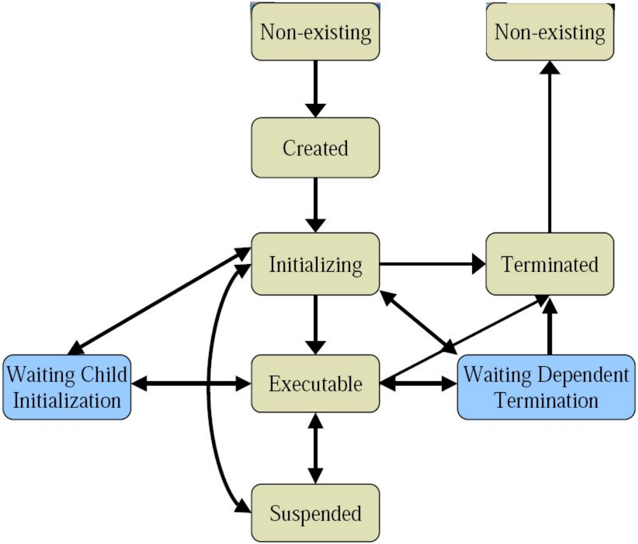
Shared Variables Communication

- unrestricted use of shared variables is unreliable
- multiple update problem
- example: shared variable X, assignment $X := X + 1$
  - load value of X into a register
  - increment value of the register
  - store the value in the register back to X
- two processes executing these instructions $\Rightarrow$ certain interleavings can produce incorrect results

Avoiding Interference

- parts of process that access shared variables must be executed indivisibly with respect to each other
- these parts are called critical section
- required protection is called mutual exclusion

Process States III



Mutual Exclusion

- specialized protocols (Peterson, Fischer, ...)
- semaphores
- monitors

Semaphores

- semaphore may be initialized to non-negative value (typically 1)
- wait operation: decrements the semaphore value, if the value becomes negative, the caller becomes blocked
- signal operation: increments the semaphore value, if the value is not positive, then one process blocked by the semaphore is unblocked (usually in FIFO order)
- both operations are atomic

Criticism of Semaphores

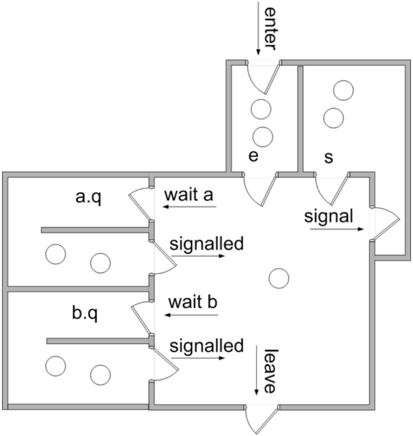
- elegant low-level primitive
- usage is error-prone
- hard to debug
- more structured synchronization primitive is useful

Monitores

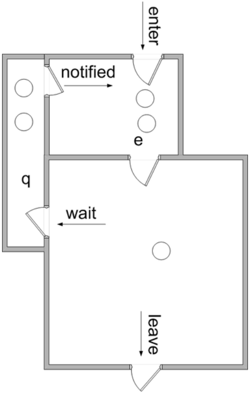
- encapsulation and efficient condition synchronization
- critical regions are written as procedures
- all encapsulated in a single module
- all variables that must be accessed under mutual exclusion are hidden
- procedure calls into the module are guaranteed to be mutually exclusive

Monitors

Hoare style



Java style



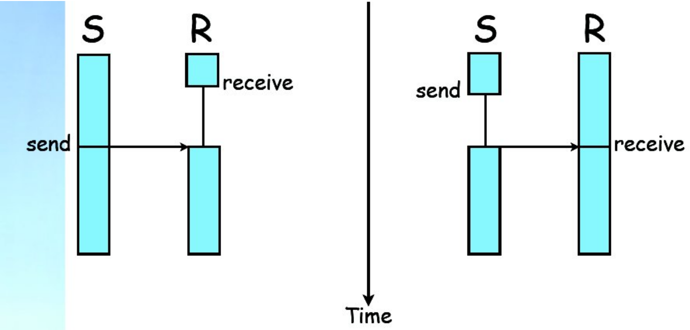
Messages Passing: Synchronization Models

asynchronous (no-wait) send operation is not blocking, requires buffer space

synchronous (rendezvous) send operation is blocking, no buffer required

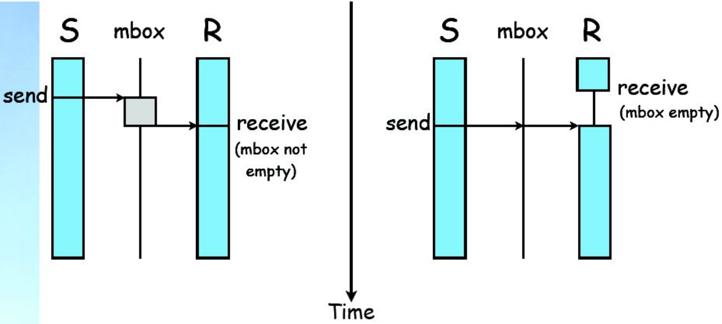
remote invocation (extended rendezvous) sender is blocked until reply is received

Synchronous Messages



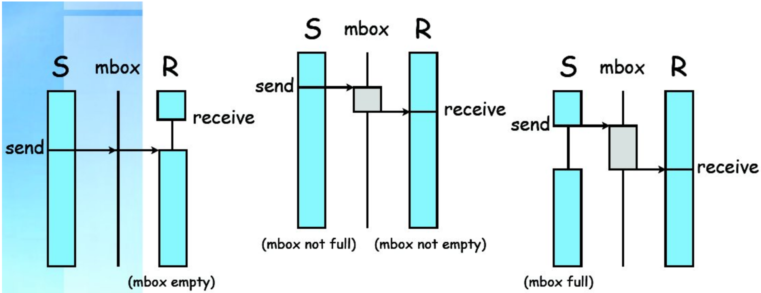
Both send and receive may block indefinitely!

Asynchronous Messages



Only the receiver might block indefinitely!

Asynchronous with Bounded Buffer



Receiver blocks if mbox is empty,
sender blocks if mbox is full

Note: size = 0 gives synchronous communication!

Message Passing: Naming

- (in)direction
  - direct naming: `send msg to process-name`
  - indirect naming: `send msg to mailbox`
- symmetry
  - symmetric: both sender and receiver name each other
  - asymmetric: receiver names no specific source

Aspects of Real Time

- An external process to sample
  - a program can read a real-time clock just as it samples any external process value (e.g. the temperature)
- An external process to react to
  - a program can let certain points in time denote events (e.g. by means of interrupts by a clock)
- An external process to be constrained by
  - a program might be required to "hurry" enough so that some externally visible action can be performed before a certain point in time

What Time?

- units?
  seconds, milliseconds, cpu cycles, system “ticks”, ...
- since when?
  Christ’s birth, Jan 1 1970, system boot, program start, explicit request, ...
- real time, cpu time, ...
- resolution

Importance of Units

- Mars Climate Orbiter, $125 million project
- failure
- mix up between metric and imperial units

Requirements for Interaction with ‘time’

For RT programming, it is desirable to have:

- access to clocks
- delays
- timeouts
- deadline specification and scheduling

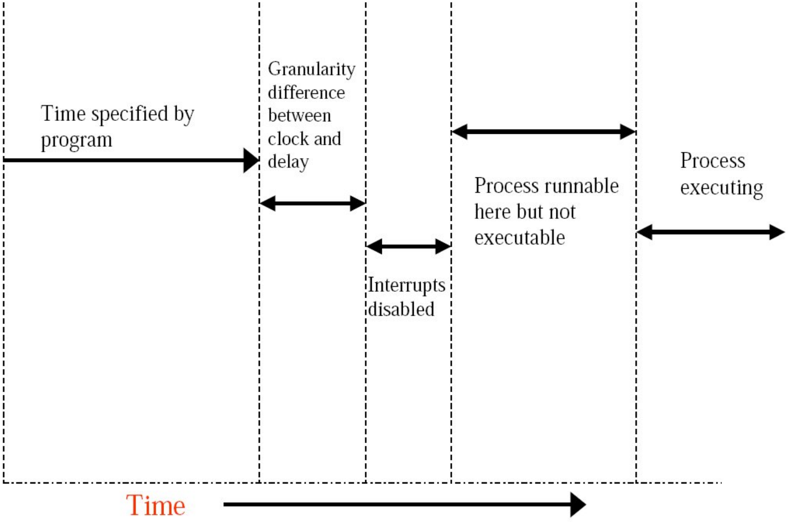
Access to Clock

- requires a hardware clock that can be read like a regular external device
- mostly offered as an OS service, if direct interfacing to the hardware is not allowed

Delays

- absolute delay (wake me at 2 hours)
- relative delay (wake me in 2 hours)
- delaying (sleeping) amounts to defining a point in time before which execution will not continue — a lower real-time constraint

Delays


Time specified by program
Granularity difference between clock and delay
Interrupts disabled
Process runnable here but not executable
Process executing
Time

A Cyclic Task (An Attempt)

```
while (1) {
  delay(100);
  do_work();
}
```

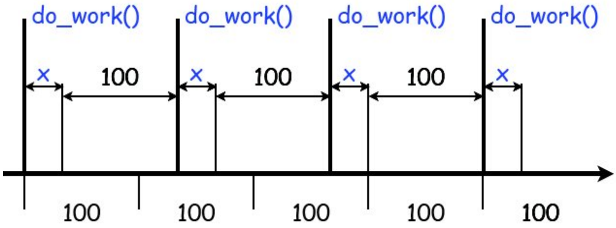
What is wrong with this piece of code?

A Cyclic Task (An Attempt)

```
while (1) {
  delay(100);
  do_work();
}
```

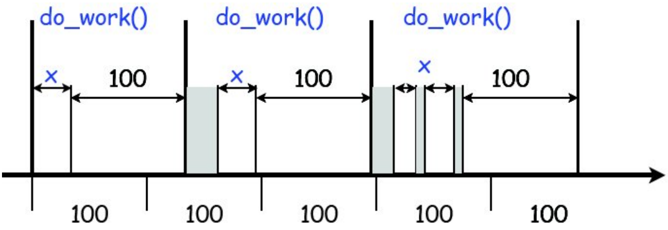
- What is wrong with this piece of code? Nothing, but ...
- if the intent is to have `do_work()` run every 100 miliseconds
- the effect will not be the expected one
- accumulating drift

Accumulating Drift



Each turn in the loop will take at least $100 + x$ milliseconds, where x is the time taken to perform `do_work()`

Accumulating Drift II



Delay is just lower bound, a delaying process is not guaranteed access to the processor (the delay does not compensate for this)

Eliminating Drift: Timers

- set an alarm clock, do some work, and then wait for whatever time is left before the alarm rings
- this is done with timers
- a timer could be set to ring at regular intervals
- thread is told to wait until the next ring — accumulating drift is eliminated
- even with timers, drift may still occur, but it does not accumulate (local drift)

Timeouts

- timeouts useful for communication and synchronization
- implemented by timers

Summary

- real time programming is closely connected with concurrent programming
- processes (threads), process states, initialization, termination, relations
- communication, synchronization: shared variables (mutual exclusion), message passing
- real time requirements: access to clocks, delaying, timeouts