Real Time Support in Programming Languages

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

Aim of the Lecture

brief overview, not a tutorial

to illustrate:

- how different programming languages realize general concepts
- that each programming languages focuses on different aspects

About (Not Just) Programming ...

- choose the right tool (language) for a given problem
  - lectures can help
  - often it is not your decision
- master the tool
  - practice, practice, practice, ...

Contents

Ada

- designed for United States Department of Defense during 1977-1983
- targeted at embedded and real-time systems
- Ada 95 revision
- used in critical systems (avionics, weapon systems, spacecrafts)
- free compiler: `gnat`

Ada Lovelace (1815-1852)

Main Principles

- structured, statically typed imperative computer programming language
- strong typing
- modularity mechanisms (packages)
- run-time checking
- parallel processing (tasks)
- exception handling
- object-oriented programming (Ada95)

Concurrency: Tasks

- task = the unit of concurrency
- explicitly declared (no fork/join statement, cobegin, ...)
- tasks may be declared at any program level
- created implicitly upon entry to the scope of their declaration or via the action of an allocator

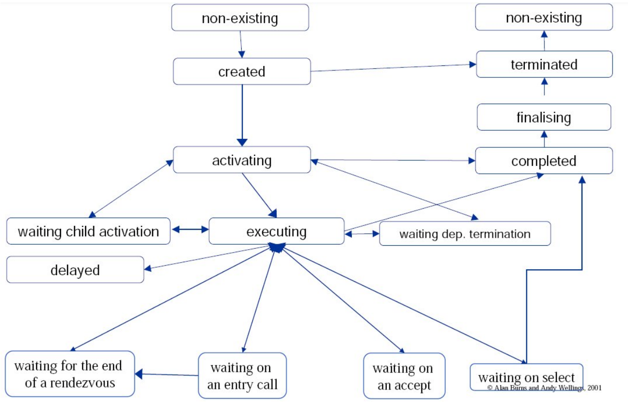
Tasks: interaction

- communicationa and synchronization via a variety of mechanisms:
  - rendezvous (a form of synchronised message passing)
  - protected units (a form of monitor)
  - shared variables
- support for hierarchies, parent-child, guardian-dependent relations

Communication

- remote invocation with direct asymmetric naming
- one task defines an entry and then, within its body, accepts any incoming call (`accept` statement)
- a randezvous occurs when one task calls an entry in another task
- selective waiting allows a process to wait for more than one message

Task States


non-existing
non-existing
created
terminated
finalising
activating
completed
waiting child activation
executing
waiting dep. termination
delayed
waiting for the end of a rendezvous
waiting on an entry call
waiting on an accept
waiting on select
© Alan Burns and Andy Wellings, 2001

Time

- access to clock:
  - package `Calendar`
  - abstract data type `Time`
  - function `Clock` for reading time
  - data type `Duration` predefined fixed point real for time calculations
  - conversion utilities (to human readable units)
- waiting: `delay`, `delay until` statements

Example

```
task Ticket_Agent is
  entry Registration(...);
end Ticket_Agent;

task body Ticket_Agent is
  -- declarations
  Shop_Open : Boolean := True;
begin
  while Shop_Open loop
    select
      accept Registration(...) do
         -- log details
      end Registration;
    or
      delay until Closing_Time;
      Shop_Open := False;
    end select;
    -- process registrations
  end loop;
end Ticket_Agent;
```

Java

- object-oriented programming language
- developed by Sun Microsystems in the early 1990s
- compiled to bytecode (for a *virtual machine*), which is compiled to native machine code at runtime
- syntax of Java is largely derived from C/C++

Concurrency: Threads

- predefined class `java.lang.Thread` – provides the mechanism by which threads (processes) are created
- to avoid all threads having to be child classes of Thread, it also uses a standard interface:

```
public interface Runnable {
   public abstract void run();
}
```

- any class which wishes to express concurrent execution must implement this interface and provide the `run()` method

Threads: Creation

- dynamic thread creation, arbitrary data to be passed as parameters
- thread hierarchies and thread groups can be created
- no master or guardian concept

Threads: Termination

- one thread can wait for another thread (the target) to terminate by issuing the join method call on the target's thread object
- the `isAlive` method allows a thread to determine if the target thread has terminated
- garbage collection cleans up objects which can no longer be accessed
- main program terminates when all its user threads have terminated

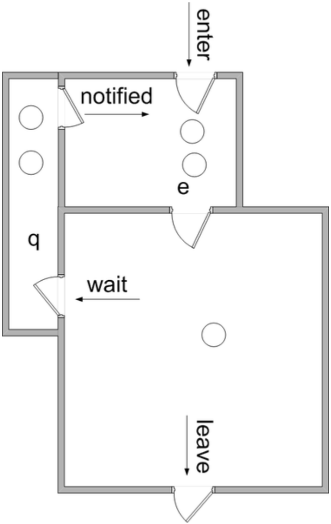
Synchronized Methods

- monitors can be implemented in the context of classes and objects
- lock associated with each object; lock cannot be accessed directly by the application but is affected by
  - the method modifier `synchronized`
  - block synchronization
- `synchronized` method – access to the method can only proceed once the lock associated with the object has been obtained
- non-synchronized methods do not require the lock, can be called at any time

Waiting and Notifying

- `wait()` always blocks the calling thread and releases the lock associated with the object
- `notify()` wakes up one waiting thread; the one woken is not defined by the Java language
- `notifyAll()` wakes up all waiting threads
- if no thread is waiting, then `notify()` and `notifyAll()` have no effect

Illustration



Real Time Java

- Java is not directly suitable for real time systems:
  - no support for priority based scheduling
  - does not prevent priority inversion
  - garbage collection introduces unpredictable delays
- Real-Time Specification for Java (RSTJ), enhanced areas:
  - thread scheduling and dispatching
  - memory management (garbage collection)
  - synchronization and resource sharing
  - asynchronous event handling, transfer of control, thread termination
  - physical memory access

Clocks

- `java.lang.System.currentTimeMilis` returns the number of milliseconds since Jan 1 1970
- Real Time Java adds real time clocks with high resolution time types

More Exotic Languages

- Real Time Euclid
- Occam
- Pearl

Real Time Euclid

- real-time language, restriction to time-bounded constructs
- programmer is forced to specify time bounds and timeouts in all loops, waits and device accessing statements
- restrictions:
  - absence of dynamic data structures
  - absence of recursion
  - time bounded loops — maximum number of iterations must be specified
- only academic proposal, never widely used

Occam

- concurrent programming language that builds on the Communicating Sequential Processes (CSP) formalism
- concurrency: cobegin (PAR)
- mainly of pedegogical interest, not widely used

```
ALT
   count1 < 100 & c1 ? data
     SEQ
       count1 := count1 + 1
       merged ! data
   count2 < 100 & c2 ? data
     SEQ
       count2 := count2 + 1
       merged ! data
   status ? request
     SEQ
       out ! count1
       out ! count2
```

Pearl

- Process and Experiment Automation Realtime Language
- language designed for multitasking and real-time programming
- developed since 1977
- used mainly in Germany

Pearl: Scheduling support

Scheduling on events and time instants, examples:

- `ALL 0.00005 SEC ACTIVATE Highspeedcontroller;`
  cyclical activation of a controller with a frequency of 20 kHz
- `AT 12:00 ALL 4 SEC UNTIL 12:30 ACTIVATE lunchhour PRIO 1;`
  cyclical scheduling, every 4 seconds between 12:00 and 13:00 hrs with high priority
- `WHEN fire ACTIVATE extinguish;`
  activation of the task 'extinguish', when interrupt 'fire' occurs.

POSIX

- **P**ortable **O**perating **S**ystem **I**nterface for uni**X**
- standardised operating system interface and environment, including:
  - system calls
  - standard C libraries
  - a command shell
- based on various flavors of Unix, but vendor-independent
- original release in 1988, formally designated as IEEE 1003

POSIX Versions

Modularized set of standards:

- POSIX.1, Core Services
    - standard C
    - process creation, control
    - signals, segmentation violations, illegal instructions, bus errors
    - floating point exceptions
- POSIX.1b, Real-time extensions
    - priority scheduling
    - real-time signals, clocks and timers
    - semaphores, message passing, shared memory
- POSIX.1c, Threads extensions
    - thread creation, thread scheduling
    - thread synchronization, signal handling

Outline

- threads (`pthread.h`)
- time (`time.h, sys/time.h`)
- signals (`signal.h`)

Concurrency in POSIX

- provides two mechanisms: fork and pthreads
- fork creates a new process
- pthreads are an extension to POSIX to allow threads
- flat structure

Pthreads

- pthread = posix thread
- specified by the IEEE POSIX 1003.1c standard (1995)
- set of C language programming types and procedure calls, implemented with a `pthread.h` header/include file and a thread library; compilation: `gcc -pthread`
- Pthreads API:
  - thread management (creation, termination, joining, ...)
  - mutexes (lock, unlock, ...)
  - condition variables (not covered in lecture)

Example I

```
#include <pthread.h>

pthread_t id;
void *fun(void *arg) {
  // Some code sequence
}

main() {
  pthread_create(&id, NULL, fun, NULL);
  // Some other code sequence
}
```

Example II

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   printf("\n%d: Hello World!\n", threadid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0; t<NUM_THREADS; t++){
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Semaphors = Mutexes

- `pthread_mutex_init (mutex,attr)`
- `pthread_mutex_lock (mutex)` — attempt to lock a mutex, if the mutex is already locked, this call blocks the thread
- `pthread_mutex_trylock (mutex)` — if the mutex is locked, returns immediately with "busy" error code
- `pthread_mutex_unlock (mutex)`

Communication

- signals
- message passing

Signals: Motivation

classical interrupts:

- external interrupt ⇒ (short-lived) execution of a pre-installed interrupt-handler
- normal execution temporarily suspended during the run of an interrupt-handler
- even if a handler has been installed by a certain process, its execution will interrupt any process that happens to be active when the corresponding interrupt signal is received

Virtual Interrupts

- process with threads $\sim$ virtual computer
- can we use virtual interrupts within the process?
- $\Rightarrow$ signals

Signals

- signal is sent towards a particular process, and handlers can be installed that are guaranteed to interrupt that process only
- signal can be sent to a process by executing `kill(pid, sig)` where `pid` is process number(0 means self)
- signals are also generated by dividing by zero, addressing outside your address space, etc.
- each thread can block incoming signals on a per-signal basis, define signal handlers for each signal it might receive, and queue signals
- no data transfer
- can be used also for exception handling

List of Signals

SIGABRT	Abnormal termination signal caused by the abort() function.
SIGALRM	The timer has timed-out.
SIGFPE	Arithmetic exception, such as overflow or division by zero.
SIGHUP	Hangup detected on controlling terminal or death of a controlling process.
SIGILL	Illegal instruction indicating a program error.
SIGINT	Interrupt special character typed on controlling keyboard (Ctrl-C).
SIGKILL	Termination signal. This signal cannot be caught or ignored.
SIGPIPE	Write to a pipe with no readers.

List of Signals (cont.)

SIGQUIT	Quit special character typed on controlling keyboard.
SIGSEGV	Invalid memory reference. Like SIGILL, portable programs should not intentionally generate invalid memory references.
SIGTERM	Termination signal.
SIGUSR1	Application-defined signal 1.
SIGUSR2	Application-defined signal 2.
SIGCHLD	Child process terminated or stopped.
SIGCONT	Continue the process if it is currently stopped; otherwise, ignore the signal.

Signal Handling

- same basic idea as for real interrupt-handling; a handler for a signal gets called "spontaneously", just as if the interrupted code had made the call itself
- like an interrupt handler ignores what process is running, a signal handler ignores what thread is running
- difference: signals are not delivered until the receiving process is actually running
- internally generated signals – the receiving process is already running per definition

Messages

- support for interprocess communication
- see `sys/ipc.h`, `sys/msg.h`, `mqueue.h`, ...
- also note *Messsage Passing Interface* (MPI)
  - not directly related to POSIX
  - used mainly for distributed computation

Getting Time

- POSIX requires at least one clock of minimum resolution 50Hz (20ms)
- `time()` — seconds since Jan 1 1970
- `gettimeofday()` — seconds + nanoseconds since Jan 1 1970
- `tm` — structure for holding human readable time

Timers

- simple waiting: `sleep`, `nanosleep`
- timers: `timer_t`, can be set:
  - relative/absolute time
  - single alarm time and an optional repetition period
- timer “rings” by sending a signal

Specifics of RT OS

- support for real time operations (timers), concurrency (task scheduling), ...
- deterministic timing behaviour, predictability

Obstacles of Predictability

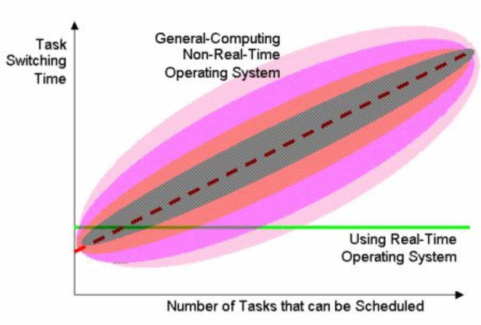
- direct memory access (DMA)
  - DMA takes control of I/O
  - I/O shares bus with CPU, DMA can block CPU (cycle stealing)
- caches, memory management (page faults, page replacements)
- interrupts
- system calls (what is the worst case execution time?)

Functionality

- basic services:
  - task management
  - interprocess communication and synchronization
  - timers
  - memory allocation
  - device I/O supervision
- trade-off:
  - more features, more complex, performance degradation, more difficult to analyze
  - less features, better performance, easier to analyze

Task Scheduling

- typically based on priority based preemtive scheduling
- equal priority processes: FIFO, round-robin (time slicing)
- switch time should be load-independent



Standards

RT-POSIX

OSEK
: Offene Systeme und deren Schnittstellen fr die Elektronik in Kraftfahrzeugen ("Open Systems and their interfaces for the Electronics in Motor vehicles"), founded in 1993 by a german automotive companies consortium

APEX
: avionics standard

ITRON
: Industrial TRON (The Real-time Operating System Nucleus), started 1984 in Japan, about 50 kernel products

Implementations

Examples of POSIX-compliant implementations:

- commercial:
  - VxWorks
  - QNX
  - OSE
- Linux-related:
  - RTLINUX
  - RTAI

Summary

- Ada, Java
- C/C++ and POSIX
- specifics of real time operating systems