Lecture 2:
The Critical Section (CS) Problem

K. V. S. Prasad
Dept of Computer Science
Chalmers University
6 Sep 2012

Questions?

- Anything you did not get
- Was I too fast/slow?
- Have you joined the google group? Found a lab partner?
- Haven’t yet heard from all course reps

Plan for today

- Chap 2 (recap + repeat + complete)
- Chap 3
  - define critical section problem
  - attempt with load and store
  - find new primitives
- Chap 6
  - Introduce semaphores (so you can start on lab)

Addenda

- Forgot to say: to pass the course
  - all labs + at least 24 p (/68) from the exam
  - Don't have to pass labs and exam the same term
- My dates refer to concurrency research
    - Hat-tip: Michal
  - Strong sequential mindset in CS (as Java shows)
    - so take up lags by decades (C++ now)
    - Ignorance of concurrency work
    - assumption that it's easy (Therac)

Recap – state diagrams

- (Discrete) computation = states + transitions
  - Both sequential and concurrent
    - Can two frogs move at the same time?
  - We use labelled or unlabelled transitions
    - According to what we are modelling
    - Chess games are recorded by transitions alone (moves)
      - States used occasionally for illustration or as checks

Recap - Radical Concurrency

- Don't start from sequential computation
- Handshake (kids meeting one-on-one)
  - Or like telephone, rendezvous
  - Can only happen when both parties present
    - Either waits for the other
    - With no data, symmetry between sender/receiver
- Broadcast
  - Speaker autonomous
  - Others must hear whatever spoken, whenever
- Our examples – concurrent, parallel, non-deterministic
- Define abstract machines and programming notation

Sadly, back to reality

- Start from sequential programs
  - How to get them to cooperate and synchronise
  - Recap: Shared bank account and counter
    - Don't interleave between load and store
  - Most examples we will see have no parallelism
    - i.e., actual parallelism to save time
    - Only potential parallelism = concurrency
- Concurrent programs can be run on one CPU
  - By switching between the processes

Recap: 60's style structure

- Each I/O device can be a process
- What about the CPU?
  - Each device at least has a "virtual process" in the CPU
- Context switching
  - move next process data into CPU
  - When? On time signal or "interrupt"
  - How? CPU checks before each instruction
- What does each process need to know?
- What does the system need to know about each process?

Operating Systems (60's thru 70's)

- Divided into kernel and other services
  - which run as processes
- The kernel provides
  - Handles the actual hardware
  - Implements abstractions
    - Processes, with priorities and communication
  - Schedules the processes (using time-slicing or other interrupts)
- A 90's terminology footnote
  - When a single OS process structures itself as several processes, these are called "threads"

Terminology

- A ”process” is a sequential component that may interact or communicate with other processes.
- A (concurrent) ”program” is built out of component processes
- The components can potentially run in parallel, or may be interleaved on a single processor. Multiple processors may allow actual parallelism.

Interleaving

- Each process executes a sequence of atomic commands (usually called ”statements”, though I don’t like that term).
- Each process has its own control pointer, see 2.1 of Ben-Ari
- For 2.2, see what interleavings are impossible

State diagrams and scenarios

- Ben-Ari 5 -11, 16 -20, 22 – 24, 28 & 35-36
- In slides 2.4 and 2.5, note that the state describes variable values before the current command is executed.
- Not all thinkable states are reachable from the start state

Why arbitrary interleaving?

- Even with one CPU
  - multiple processes good idea
    - For separate tasks
    - Or even to structure one program (Cremona example)
- Multitasking (2.8 is a picture of a context switch)
  - Context switches are quite expensive
  - Take place on time slice or I/O interrupt
  - Thousands of process instructions between switches
  - But where the cut falls depends on the run
- Runs of concurrent programs
  - Depend on exact timing of external events
  - Non-deterministic! Can't debug the usual way!
  - Does different things each time!

Arbitrary interleaving (contd.)

- Multiprocessors (see 2.9)
  - If no contention between CPU's
    - True parallelism (looks like arbitrary interleaving)
  - Contention resolved arbitrarily
    - Again, arbitrary interleaving is the safest assumption

The counting example

- See algorithm 2.9 on slide 2.24
  - What are the min and max possible values of n?
- How to say it in C-BACI, Ada and Java
  - 2.27 to 2.32

But what is being interleaved?

- Unit of interleaving can be
  - Whole function calls?
  - High level statements?
  - Machine instructions?
- Larger units lead to easier proofs but make other processes wait unnecessarily
- We might want to change the units as we maintain the program
- Hence best to leave things unspecified

Why not rely on speed throughout?

- Don’t get into the train crash scenario
  - use speed and time throughout to design
  - everyday planning is often like this
    - Particularly in older, simpler machines without sensors
    - For people, we also add explicit synchronisation
- For our programs, the input can come from the keyboard or broadband
  - And the broadband gets faster every few months
- So allow arbitrary speeds

Atomic statements

- The thing that happens without interruption
  - Can be implemented as high priority
- Compare algorithms 2.3 and 2.4
    - Slides 2.12 to 2.17
  - 2.3 can guarantee n=2 at the end
  - 2.4 cannot
    - hardware folk say there is a ”race condition”
- We must say what the atomic statements are
  - In the book, assignments and boolean conditions
  - How to implement these as atomic?

What are hardware atomic actions?

- Setting a register
- Testing a register
- Is that enough?
- Think about it (or cheat, and read Chap. 3)

The standard Concurrency model

1. What world are we living in, or choose to?
   a. Synchronous or asynchronous?
   b. Real-time?
   c. Distributed?
2. We choose an abstraction that
   a. Mimics enough of the real world to be useful
   b. Has nice properties (can build useful and good programs)
   c. Can be implemented correctly, preferably easily

Obey the rules you make!

1. For almost all of this course, we assume single processor without real-time (so parallelism is only potential).
2. Real life example where it is dangerous to make time assumptions when the system is designed on explicit synchronisation – the train
3. And at least know the rules! (Therac).

Goals of the course

- covers parallel programming too – but it will not be the focus of this course
- Understanding of a range of programming language constructs for concurrent programming
- Ability to apply these in practice to synchronisation problems in concurrent programming
- Practical knowledge of the programming techniques of modern concurrent programming languages

Theoretical component

- Introduction to the problems common to many computing disciplines:
  - Operating systems
  - Distributed systems
  - Real-time systems
- Appreciation of the problems of concurrent programming
  - Classic synchronisation problems

Semantics

- What do you want the system to do?
- How do you know it does it?
- How do you even say these things?
  - Various kinds of logic
- Build the right system (Validate the spec)
- Build it right (verify that system meets spec)

Chap 3: The Critical Section Problem

- Attempts to solve
  - without special hardware instructions
    - Assuming load and store are atomic
  - Designing suitable hardware instructions

Requirements and Assumptions

- Correctness
  - Both p and q cannot be in their CS at once (mutex)
  - If p and q both wish to enter their CS, one must succeed eventually (no deadlock)
  - If p tries to enter its CS, it will succeed eventually (no starvation)
- Assumptions
  - A process in its CS will leave eventually (progress)
  - Progress in non-CS optional

Comments

- Pre- and post-protocols
  - These don’t share local or global vars with the rest of the program
- The CS models access to data shared between p and q

First try (alg 3.2, slide 3.3)

- The full state diagram shows only 16 states are reachable, out of 32
- These exclude states (p3,q3,*) so mutex is OK.
- The abbreviated program reduces state space
- if p1 is stuck in NCS with turn=1, q starves
- Deadlock free in the sense that p can enter CS
- Error: p and q both set and test ”turn”; if one dies the other is stuck

Second try: alg 3.6, slide 3.12

- Wantp iff p is in CS or wants to get in
  - So wantp is false if p is in NCS, and q is free
- Sadly, no mutex
  - by running in parallel, p and q can both be in CS at the same time

Third try: alg 3.8, slide 3.16

- Flip p2 and p3 of second try; book your place before trying to enter CS
- Similar problem: both can starve.
  - Deadlock by definition
    - (both want CS, neither gets it)
  - Actually, worth calling it ”livelock”
    - If await is a busy wait
- Maybe p should declare intention but not insist on entering CS
  - Instead, try and back off

Fourth try: alg 3.9, slide 3.19

- Again, running in parallel gets p and q into trouble
  - Mutex is fine (show by state diagram)
  - No deadlock : p or q *can* enter CS
  - But they can starve in parallel
- Just when it is beginning to look like a bad joke …

Dekker's alg (3.10, slide 3.21)

- Modify try 4 by adding the turn from try 1
  - To arbitrate away from the parallel starvation
- Prove correctness by state diagram
  - Deductive proof in Sec 4.5
    - Using temporal logic

Rethink

- P checks wantq
  - Finds it false, enters CS,
    - but q enters before p can set wantp
- Could we prevent that?
  - When I find the book free, I take it
    - Before anyone else even sees it free
- ”Test-and-set” instruction
  - See Wikipedia article, also Herlihy 1991

Exchange and other atomics

- Slides 3.22 and 3.23
- Other atomic instructions
  - Compare and swap
  - Fetch-and-add
- All use busy waits
  - OK in multiprocessors
    - Particularly if low contention