On communication among threads

Runtimes for concurrency and distribution

Tullio Vardanega, tullio.vardanega@unipd.it

Academic year 2020/2021

Premise – 1

- Concurrency is eminently collaborative
  - The threads in a concurrent program hardly are fully independent of one another
  - If they were, they would be perfectly parallel
- Stipulating the communication interfaces allowable among them is a crucial aspect of the design of a concurrent language
  - The chosen model of communication has large influence on the quality of the program

Premise – 2

- Communication can be
  - **Direct**, only involving active entities
  - **Indirect**, mediated by reactive entities
- Classic models
  - **Message passing**, direct
    - No sharing: awkward when running on a single processor, but also **very scalable**
  - **Shared variables**, indirect
    - Natural when running on shared memory
    - But also very risky and **not scalable**

Premise – 3

- Synchronizing (waiting for one another) to communicate defeats parallelism
- When data sharing cannot be avoided in a parallel system, **wait-free synchronization** solutions become desirable
  - Spin locking can be afforded sometimes
  - Transactional memories can be useful
    - They use concurrency control mechanisms similar to those required for DBs, except they are in HW
    - Consistency (writes are serialized) and isolation (no partial state leaks) warrant atomicity

Shared variables – 1

- **Bernstein's condition**
  - Atomic execution is guaranteed if shared variables that are read and modified by a **critical section** are not modified by any other concurrently executing section of code
    - IEEE TREC 15(15), 1966
- If that condition does not hold, the risk of **data race** arises
  - R. Netzer and B. Miller have shown that ascertaining the presence of data races in a program is inordinately complex (NP-hard) in the general case
    - ACM LoPLAS 1(1), 1992

Shared variables – 2

- The code fragments that operate on shared variables are termed **critical sections**
  - A very general definition that does not make assumption on the structuredness of the language
- The possibility that program execution may give rise to uncontrolled accesses to a shared variable is termed **race condition**
  - Race conditions cause non-determinism, which is antagonistic to program verification
  - Interestingly, educated forms of non-determinism may be desirable for concurrent programs !

Defeating data races

- The problem has two parts
  - How to ensure that critical sections execute atomically (**P1**)
    - Errors of this type cause **low-level** data races
  - How to single out critical sections correctly (**P2**)
    - Errors of this type cause **high-level** data races
- Two types of P2-type errors exist
  - **Non-atomic protection fault**, when a thread's operation on a shared variable is fragmented in multiple disjoint partial accesses
  - **Lost-update fault**, when a concurrent write of a shared variable occurs between the read and the subsequent functionally-related write of it by one and the same thread

P1-type problem: example – 1

```
// thread A needs to access shared
// variable X
// to this end, it checks whether
// X is free
if (lock == 0) {
   // X is being used
   // try again (busy wait)
}
else {
   // X is free
   // set it to «in use»
   lock = 0;
   <critical section S1(X)>;
   // free X
   lock = 1;
}
```

```
// thread B needs to access shared
// variable X
// to this end, it checks whether
// X is free
if (lock == 0) {
   // X is being used
   // try again (busy wait)
}
else {
   // X is free
   // set it to «in use»
   lock = 0;
   <critical section S2(X)>;
   // free X
   lock = 1;
}
```

Critical sections S1 and S2 are not atomic: why?

P1-type problem: example – 2

```
/* DEPOSIT */

amount = read_amont();
lock(); // this opens
        // a critical section

balance = balance + amount;
interest = interest + rate *
           balance;

unlock(); // this closes
          // a critical sction
```

```
/* WITHDRAW */

amount = read_amount();
if (balance < amount) {
   // notify caller that
   // the operation is denied
}
else {
   balance = balance - amount;
   interest = interest +
              rate * balance;
}
```

`Withdraw` exposes `Deposit`'s critical section to a ***low-level data race***

P2-type problem: example – 1

```
/* Updater Task */

// set status value reading
synchronized (table){
  table[N].value = V;
}

… // do work

// set system status for value N
synchronized (table) {
  table[N].achieved = true;
}
```

In this time span, `table[N]` is not protected

```
/* Monitor Daemon */

synchronized (table){
 if (table[N].achieved &&
   system_state[N] !=
     table[N].value){
    // inconsistent system state
    issueWarning();
   }
}
```

NASA
Remote Agent (1997)
using Java and LISP

A case of ***non-atomic protection fault***

P2-type problem: example – 2

```
/* WITHDRAW */

void withdraw(int amount){
 lock(l);
 int tmp = balance;
 unlock(l);
 if tmp > amount){
  lock(l);
  balance = tmp - amount;
  unlock(l);
 }
}
```

Read access

Write access

```
/* DEPOSIT */

void deposit(int amount){
  lock(l);
  balance = balance + amount;
  unlock(l);
}
```

A case of ***lost-update fault***

Access control fundamentals – 1

- **Exclusion synchronization**
  - At any point in time, no more than one thread may have access to a shared resource
    - Access is exclusive
- **Avoidance synchronization**
  - Certain preconditions must hold before access can be granted
    - Dependent on the program logic
    - Epitomized on the case of the *bounded buffer*

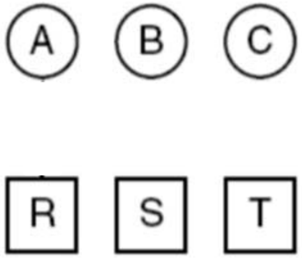
Access control fundamentals – 2

- Synchronization is exposed to risks
  - **Deadlock** or **starvation** (aka lockout)
- Deadlock causes all participants to wait indefinitely
  - When 4 conditions hold simultaneously
    1. Mutual exclusion is in use
    2. Resource access cannot be pre-empted
    3. Resource accumulation is allowed with hold-and-wait
    4. The wait condition is circular

Access control fundamentals – 3

- 4 types of reaction to deadlock
  - Ostrich (don't look and hope for the best)
  - Design-time prevention
    - Condition-4 potential can be detected if the participant set is fully and statically known
    - Condition 3 can be defeated forbidding resource accumulation
  - Run-time prevention
    - To combat condition 4, the runtime must keep current of the status of all shared variables (holding, waiting)
    - Denying access if allowing it may lead to circular wait
    - Or requiring that access is granted only in a fixed order
  - Run-time detection
    - Oh boy, some threads are not touching the ready queue …

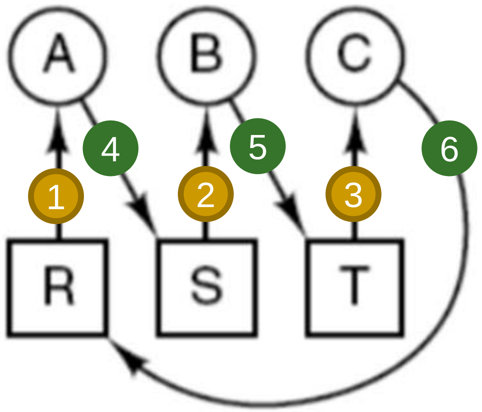
An example of deadlock prevention



A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R

This request causes circular wait



Imagine now that resources could only be accessed in a given order (e.g., R, S, T).
In that case, C should request R *before* requesting T …

Access control fundamentals – 4

- Wait time should be bounded
  - Only FIFO queuing ensures it
    - This policy is (bounded) fair and warrants liveness
  - Any other policy, no matter how common-sense, is exposed to **starvation**
    - Priority ordering
    - LIFO
    - Urgency

Synchronization solutions – 1

- Good solutions satisfy 4 conditions
  1. Exclusive access
  2. Bounded wait
  3. No assumptions on the behaviour of the execution environment
  4. No threads outside of the critical section can influence …

Synchronization solutions – 2

- Regulatory control with a shared variable and strict alternation

```
Thread A ::
while (TRUE) {
  while (turn != 0); // busy wait
  critical_section();
  turn = 1; // alternation
  ...
}
```

```
Thread B ::
while (TRUE) {
  while (turn != 1); // busy wait
  critical_section();
  turn = 0; // alternation
  ...
}
```

- Defects
  - Busy wait
  - The decision on the alternation is taken *outside* of the critical section
  - Risk of data race on the control variable (not severe)

Synchronization solutions – 3

- **Dekker's algorithm**

```
var flag: array [0..1] of boolean;
turn: 0..1; -- i, j are two threads
repeat
      flag [i] := true;
      while flag [j] do
         if turn = j then
            begin
               flag [i] := false;
               while turn /= i  do no-op;
               flag [i] := true;
            end;
         end if;
      end while;
      critical section
      turn := j;
      flag [i] := false;
      remainder of computation
until false;
```

Busy wait!

Conceived by T.J. Dekker (says E.W. Dijkstra) and later improved (1981).
By setting **flag[i] ← true** , thread **i** requests access. Similarly for thread j.
The value of **turn** arbitrates access between the two threads (i and j).
Can be generalized to more than 2 threads

Synchronization solutions – 4

- **Peterson's algorithm**
  - For pairs of threads
  - Access control logic similar to Dekker's
    - A private flag
    - A shared control variable
  - Exposed to data races if control variable is cached
  - Bounded fair
    - Booking request gives priority to the contender

```
set (flag.mine);
coin := other;
loop
 if (flag.other = clear) continue;
 if (coin = mine) continue;
end loop
// CRITICAL SECTION
clear (flag.mine);
```

Synchronization solutions – 5

```
typedef struct {
    int count;
    queue q; /* queue of threads waiting on this semaphore */
} Semaphore;
```

The initialization value set to **`count`** determines the type of semaphore:
- `count=1` → binary semaphore
- `count>1` → counting semaphore
- `count=0` → barrier

```
void P(Semaphore s)
{
    Disable interrupts;
    if (s->count > 0) {
        s->count -= 1;
        Enable interrupts;
        return;
    }
    Add(s->q, current_thread);
    sleep(); /* re-dispatch */
    Enable interrupts;
}
```

Argh!

```
void V(Semaphore s)

{
    Disable interrupts;
    if (isEmpty(s->q)) {
        s->count += 1;
    } else {
        thread = RemoveFirst(s->q);
        wakeup(thread); /* put thread on the ready queue */
    }
    Enable interrupts;
}
```

Who calls these?

Leaving the use of `P(s)` and `V(s)` to the programmer's discipline is risky

Synchronization solutions – 6

- The **monitor**
  - An explicit *syntactic structure* that encapsulates the shared variable and publishes the operations that are allowed to access it
    - Charles A R Hoare, "*Monitors – An Operating System Structuring Concept*", CACM 17(10):549-557 (1974)
  - The shared variable is not visible outside of the monitor
  - Calling monitor operations triggers access control by the runtime
    - Not the programmer!

Monitor details – 1

- What if the shared variable's state is not fit for use by a caller that has gained access to it?
  - Cannot write into a shared buffer that is full
  - Cannot read from a shared buffer that is empty
- The monitor provides condition variables that can be signalled and waited for
  - If `Var` is false, **`Wait`**`(Var)` places the caller in a wait queue until `Var` turns true
  - The lock holder calls **`Signal`**`(Var)` to set `Var` to true

Monitor details – 2

```
monitor Container
   condition not-empty := false;
             not-full := true;
   integer content := 0;

   procedure Insert(prod : integer);
   begin
      if content = N then Wait(not-full);
      <add prod to container>;
      content := content + 1;
      if content = 1 then Signal(not-empty);
   end;

   function Fetch : integer;
   begin
      if content = 0 then Wait(not-empty);
      content :=  content – 1;
      if content = N-1 then Signal(not-full);
      return (<fetch from container>);
   end;

end monitor;
```

```
thread Producer ::
 prod : integer;
begin
   while true do
   begin
      Produce(prod);
      Container.Insert(prod);
   end;
end;
```

```
thread Consumer ::
 prod : integer;
begin
   while true do
   begin
      prod :=
        Container.Fetch;
      Consume(prod);
   end;
end;
```

Monitor details – 3

- Calling **`Wait`** on `Var` blocks the caller when `Var` is false
  - Variable `Var` should describe the resource state
  - The caller is placed in a wait queue
  - What happens to the lock at this point?
- Calling **`Signal`** on `Var` releases the thread at the top of the wait queue for `Var`
  - The program's logic decides when `Signal` should be called
  - Who gets the lock at this point?
- The compiler makes sure that such calls are atomic and therefore exempt from data races

Monitor details – 4

- The monitor concept is vastly better than semaphore-protected critical sections
- But has defects too
  - The monitor does not let the program decide the order of calls to it dynamically
    - The thread that gets there first calls it and then perhaps has to wait on a false condition variable: big waste!
  - The monitor leaves to the programmer the choice of when to call **`Wait`** and **`Signal`**
    - Yes, this is part of the program's logic
    - But the programmer may get it wrong

Java's failed monitor – 1

```
class Monitor{
   private int cont = 0;
   public synchronized void Insert(int prod){
      if (cont == N)
         Block();
      <add prod to container>;
      cont = cont + 1;
      if (cont == 1)
         [this.]notify();
   }
   public synchronized int Fetch(){
      if (cont == 0)
         Block();
      cont = cont - 1;
      if (cont == N-1)
         [this.]notify();
      return(<fetch from container>);
   }
   private void Block(){
      try{[this.]wait();
      } catch(InterruptedException exc) {};}
}
```

```
static final int N = <…>;
static Monitor Container = new Monitor();
// …
Monitor.Insert(prod); // producer
// …
prod = Monitor.Fetch (); // consumer
```

For real?

Java's failed monitor – 2

- In truth, exclusion synchronization and avoidance synchronization are orthogonal problems
  - ES minds access control
  - AS minds that the callers' operation are consistent with the resource logic
- Java collapses them into a single wait queue
  - What blocked caller does **`notify()`** awaken?
    - **`notifyAll()`** was invented to do damage control, yielding worse chaos
  - Who gets the lock after **`wait()`** and **`notify()`**?

Message passing – 1

- Its synchronous variant requires both parties to wait for one another
  - In this way, both parties know about the progress state of the other even *without* exchanging data
- As synchronization does not scale, the asynchronous variant becomes attractive
  - Sending is non-blocking
    - The sender proceeds if there is no receiver
    - The two parties no longer know about each other's progress
  - Receiving blocks until synchronization ends
    - The receiver waits until the sender arrives

Message passing – 2

- Both variants can be played with to inverse their behaviour
  - Synchronous to Asynchronous
    - Placing an intermediary between Sender and Receiver
  - Asynchronous to (almost) Synchronous
    - Having Sender await an ack from Receiver
- How do Sender and Receiver get to know each other?
  - By unique name (of thread, of mailbox)
    - CSP's message passing is synchronous and unidirectional
    - Totally unfit for servers !
  - By type of message / channel at destination

Message passing – 3

- Synchronous communications allow for *bidirectional* data exchange
  - First S to R, then R to S
- Receivers can become servers by exposing multiple bidirectional channels (**entries**)
  - Entries have by-copy `in` and `out` parameters
  - A server exposing multiple entries must specify explicitly which one to service at a given time
- Callers (clients) must name the server and the entry of interest
- Thanks to synchronization, receivers (servers) do **not** need to name their callers
  - This makes the naming relation *asymmetric*

Message passing – 4

- Prefixing specific preconditions (**guards**) to attending to entries, allows servers to implement their service logic
  - Dijkstra's model of non-deterministic guarded select receive
    - E.W. Dijkstra, "*Guarded Commands, Nondeterminacy, and Formal Derivation of Programs*", CACM, 18(8):453-457 (1975)

```
select
 Guard_1 => accept Service_1(…);
or
 …
or
 Guard_K => accept Service_K(…);
end select;
```

- Guards are Boolean expressions
  - When they are true (open) the respective receive command (**`accept`**) is enables on the corresponding channel (entry)
  - When multiple guards are open and calls are pending on the corresponding entry, the choice is non-deterministic