CptS 464/564
Lecture 10

13th February 2003

Homework assignment available on home page

Review what we're doing: Ch. 2 is a collection of paradigms for DS: common vocabulary, concepts, techniques, that frequently arise in dealing with DS.

Logical clocks

Assigned 2.7.6 to grad students, but realized I have to cover one part: Lamport's logical clock algorithm which allows us to implement a delivery order consistent with causal delivery quite simply.

- Each process keeps a single integer called its *logical clock* or *lclock*
- Each time a process sends a message it includes its current lclock value in the message header, then increments the clock
- Assumption: message channels are FIFO
- When a message is received it is placed in a waiting queue ordered by its *lclock* field; break ties using the sender ID. Hold the message in the queue until a message arrives with equal or greater *lclock* from every sender in the system. (Note assumption of a closed system!). Algorithms ensures that when this is true, all messages with lower *lclock* have been received. When this is true, the message becomes *deliverable.*

- Deliver the deliverable messages in lclock order; as each message is delivered set the local lclock to 1+(max of its current value and that of the delivered message).

The logical clock delivery algorithm ensures that messages are delivered in causal order – that is if $m \rightarrow n$ then $lclock(m) < lclock(n)$, hence messages are delivered in the correct order; however, $lclock(m) < lclock(n)$ does not imply that $m \rightarrow n$, so the algorithm will delay messages unnecessarily.

Notice that the logical clock algorithm as defined here, also provides a *totally ordered* delivery mechanism: any two messages both received at two sites will be delivered in the same order (total order requirement).

Without the "hold messages" constraint, the above algorithm also provides a way to compute logical time in an execution of a system such that time in each process advances monotonically, and the delivery time of each message is greater than its send time.

Temporally ordered delivery

Problem: causally ordered delivery may be incorrect in applications that have communications outside of the ordered-delivery system, for example, systems that control and sense the state of physical systems.

Valve example 2.17b.

Solution - return to physical ordering as a basis and observe that non-zero minimum propagation time of messages gives us a clue as to how to proceed.

Recall that the problem with physical order was that it ordered many more pairs of events than were truly causally related. We can improve on physical order with the observation that most message delivery systems have a minimum delivery time for messages, call it δ. Thus an event occuring at time t cannot be caused by a message sent after time $t - \delta$. This observation induces us to define the relation *temporally precedes,* written $\rightarrow^{\delta}$, and defined

$$m_1 \rightarrow^{\delta} m_2 \Leftrightarrow t(send(m_2) - t(send(m_1)) > \delta$$

Notice that use of temporal order again seems to require synchronized physical clocks, the precision with which they are synchronized will affect our choice of δ.

Claim: if messages are delivered in temporal order, the delivery order will respect causality (but only if δ properly reflects the physical parameters of the system such as minimum message delivery time and the precision of the clock synchronization.

Coordination (2.8)

Mutual exclusion between processes in centralized systems

Two points in talking about mutual exclusion in centralized systems:

- CS are easier to understand in introducing the topic
- As mentioned when we talked about remote operations (Sec. 2.3), multiple threads in a single application or server may simplify the design of the code while allowing much of the benefit of using a non-blocking interaction interface. Yet, such multiple threads may well have to coordinate their actions.

Classical example: producer-consumer. Producer produces a product that the consumer needs. The consumer may use it to produce another product that another consumer needs, and so forth. Each element in such a chain is often called a *filter.*

Between the producer and consumer, the product is placed in a *buffer,* often having several slots, but let's consider only 1.

```
shared boolean slotBusy;
shared Item slot;

Producer::
local Item item;
while true {
   item = produce();
   while slotBusy;
   /* L1 */
   slotBusy = true;
   slot = item;
   /* L2 */
```

```
}

Consumer::
local Item item;
while true {
   while !slotBusy;
   /* L1 */
   item = slot;
   slotBusy = false;
   /* L2 */
   consume(item);
}
```

What goes wrong? What if there is more than one producer?

Let's try introducing some mechanism for mutual exclusion around manipulations of the shared variables: begin_mutual_exclusion at L1, end_mutual_exclusion at L2. Between the begin_ and end_mutual_exclusions only one process is allowed to execute. ??? How can mutual exclusion be implemented?

First attempt: boolean variable

```
shared boolean lock=false;
begin_mutual_exclusion() {
   while lock ;
   lock = true;
}

end_mutual_exclusion() {
   lock = false;
}
```

What's wrong now?

Test-and-set based implementation

```
shared boolean lock = false;
begin_mutual_exclusion() {
   while test-and-set(lock));
```

```
   // don't have to set lock; test-and-set did it
}
end_mutual_exclusion() {
   lock = false;
}
```

Problems now? Inefficient. While waiting a process uses processor cycles – called "busy waiting". Must only use busy waiting when the time the lock is held is very short: busy waiting in user processes is very ill-advised.

Instead, use a mechanism provided by the OS that coordinates waiting with the OS's process scheduling mechanism. One example of such is the semaphore (Dijkstra).

```
shared Semaphore sem = n;
Wait(Semaphore sem, int n) {
   // block until sem>=n, then sem =- n
}
Signal(Semaphore sem, int n) {
   // sem =+ n
}
```

Semaphore operation bodies are written as comments because they have to be specially implemented in the OS to ensure that they are executed atomically.

Notice that if initialized to 1 and only signalled after a wait, semaphores can serve as locks. They can also be used more powerfully: to track the number of free resources, or to count the occurrence of events.

```
shared Semaphore mutex = 1;
shared Semaphore slotNotBusy = 1;
shared Semaphore slotBusy = 0;
Producer::
while true {
   item = produce();
   wait(slotNotBusy);
   wait(mutex);
   slot = item;
```

```
    signal(mutex);
    signal(slotBusy);
}
Consumer::
while true {
    wait(slotBusy);
    wait(mutex);
    item = slot;
    signal(mutex);
    signal(slotNotBusy);
    consume(item);
}
```

Note: mutex is not required in our 1-slot case, but is required for the MAX-slot case in the book.

Another mechanism of considerable interest is the *monitor* construct, exemplified by Java's synchronized methods and it 's wait and signal operations (which do *not* have semaphore semantics). You can learn much more about synchronization and other issues in shared-memory multi-threaded programming in the Concurrent Programming class that I teach in the fall. (Last fall it was 483/580 but by next fall it will have a new number. The title will be Concurrent Programming, however.)

Mutual exclusion in decentralized systems

Coordinator approach - Lock Server. Problems – Lock server is a single point of failure. Failure of client holding the lock will also cause problems.

Distributed mutual exclusion

Leader election as an optimization of distributed mutual exclusion

Deadlock: a *deadlock* occurs when two or more processes are waiting for each other and no progress can be made. Conditions for deadlock:

- mutual exclusion – one process can do something that keeps another from making progress; typically, holding some resource
- hold-and-wait – it is possible to wait for a resource while holding another resource

- non-preemption – one process can't force another to let go of a resource
- circular waiting: there exists a chain of process in a hold-wait cycle

Eliminating Deadlock

Elimate one of the necessary conditions – not so easy. First two are fundamental for many algorithms and data structures.

Allow pre-emption: greatly complicates application code – may introduce asynchronous interruption into otherwise single-threaded code.

Prevent circular waiting – always acquire resources in the same order (define a total order on resources)

Deadlock detection: detect cyclical waiting and kill a process (a brutal form of pre-emption!)

Deadlock avoidance: don't acquire a lock that would cause a deadlock.

- Note: both of the last two require capturing a consistent global state of a distributed computation to do correctly. Leading us into our next topic:

Consistency

In my opinion the authors' given motivation for calling this section "consistency", in the intro to the section, is incomplete, though it is not a bad title. The consistency we are *first* concerned with here is the that required for obtaining an accurate understanding of the global state of a distributed system: an understanding that captures the notion that *state* explains the future behavior of a system. In short, what is a *consistent global state?*

We explore the quesiton from two perspectives. First, as an outside observer of a system, what does a consistent global state look like? Second, how can a participant (inside) a distributed system determine a consistent global state? The answer to the second question builds on the first.

Once we have an answer to the second question, we will be able to use it to build distributed algorithms solving useful problems such as consensus and deadlock detection.

The second sense of consistency, (which is more consistent with the author's intro to the section) concerns specification and design issues: what is the requirement for different nodes in a distributed system to maintain consistent state and provide consistent replies to clients? A broad range of answers to this question is possible: which one is chosen depends on the application and people's expecations of its behavior.

Examples:

- strongly-consistent replicated database
- eventually-consistent replicated database