Concurrency

Garlan Handout 12 22 October 1997

The questions that we will address in the remaining part of this semester are

- How do you model concurrent systems? (Quick answer: We will look at both *shared-memory* models and *message-passing* models.)
- What properties do we typically expect of concurrent systems? (Quick answer: We will look at *safety* and *liveness* properties.)
- How do you state and reason about those properties? (Quick answer: We will use both standard logic as well as special logic called *temporal logic*.)

Before we address any of these questions in detail, let's consider what the essential problem concurrency raises. Andrews and Schneider's paper gives much more detail about all of the issues I discuss in this handout; here, I just highlight the main points.

1 The Essence of Concurrency

Consider two processes that share a resource. They use this shared resource to communicate with each other. This resource could be a shared variable or it could be a shared message channel.

The essence of concurrency is synchronizing access to the shared resource. In the extreme, if two processes are running concurrently and do not share any resource then they cannot communicate with each other; thus, they have no reason (or need) to synchronize with each other. They don't even have any reason to know about each other. Things only get interesting if the two processes want to exchange some information, i.e., communicate with each other.

If the shared resource is a shared variable, then presumably after one process, P, writes a value to the variable the other process, Q, can read it; in this indirect manner, P communicates with Q. It's like agreeing with your friend (whom your parents forbid you to see) that you'll secretly leave each other notes under a rock to communicate. This means of communication brings up the issue of synchronization since only one person at a time can be at the rock.

If the shared resource is a shared message channel, P can send a message directly to Q. The channel is the medium through which the message travels. This means of communication brings up issues like transmission delay (it takes time for messages to travel), buffering (suppose Q is busy reading the last message P sent it?), ordering (are messages received in the same order sent?), duplicates (are duplicate messages allowed?), lossiness (what happens if a message is lost?), and failures (what happens if the channel breaks?).

Which one of the these models, *shared-memory* or *message-passing*, you choose to describe your concurrent system depends on the features of the system you want to focus on, the kind of reasoning you want to perform, or simply your taste, preference, or familiarity. Theoreticians have shown that you can always model one in terms of the other so you are not losing or gaining any expressive power by choosing one over the other.

1.1 A Simple Example

To distill the essence of concurrency into its simplest form, consider the shared integer variable x, initialized to 0, and two processes, P and Q, each of which executes the same code:

```
x := 1
x := x + 1
```

If P and Q execute *sequentially* (denoted $P;\ Q$) then I can assert with confidence that $x = 2$ after they both terminate. To be pedantic, we have:

$P;\ Q \equiv$

$\{x = 0\}$
$x := 1 \quad (P)$
$\{x = 1\}$
$x := x + 1\ (P)$
$\{x = 2\}$
$x := 1 \quad (Q)$
$\{x = 1\}$
$x := x + 1\ (Q)$
$\{x = 2\}$

where I have "tagged" each statement by the name of the process executing it. Note, of course, that doing Q and then P ($Q;\ P$) would result in the same final value for x.

However, suppose we ran P and Q concurrently, denoted as $P \,||\, Q$? Then, we have the possibility that x's final value is 3, not 2:

$\{x = 0\}$
$x := 1 \quad (P)$
$\{x = 1\}$
$x := 1 \quad (Q)$
$\{x = 1\}$
$x := x + 1\ (P)$
$\{x = 2\}$
$x := x + 1\ (Q)$
$\{x = 3\}$

where we interleaved the statements of P and Q.

Summarizing, we have:

$\{x = 0\}$
$P;\ Q$
$\{x = 2\}$

$\{x = 0\}$
$P \,||\, Q$
$\{x = 2 \lor x = 3\}$

The behavior of $P;\ Q$ is deterministic but that of $P \,||\, Q$ is nondeterministic!

1.2 Another Simple Example

Suppose again P and Q share the variable x, initialized to 0. Suppose P executes the code (as before):

$$x := 1$$
$$x := x + 1$$

and Q executes just the statement:

$$x := 59$$

You should convince yourself that the strongest assertion about the value of x you can make at the end of $P \mid\mid Q$ is:

$$\{x = 2 \vee x = 59 \vee x = 60\}$$

Each possible final value corresponds to one of the possible interleavings of the statements executed by P and Q. Note that each process executes its code sequentially; hence, it's impossible for x to be 1 at the end, which could occur if all statement interleavings were allowed. (We're also assuming each process terminates so $x \neq 0$ So as the number of statements increase, the number of possible interleavings (and hence the number of possible final values) increase.

Exercise 1: Given that P executes m statements and Q executes n statements, what is the number of sequences of interleaved statements possible? (Don't worry about distinguishing between interleavings that might lead to the same final values of any shared variables.)

Exercise 2: Given that there are i processes, P_i, each executing n statements what is the number of sequences of interleaved statements possible?

2 Issues to Consider When Modeling Concurrency

2.1 Atomicity

When you are faced with a concurrent system that you want to model, the very first question you should ask is "What are its *atomic* operations?" To answer this question, therefore, you must be absolutely clear about the level of abstraction at which you are modeling your system. Why? Because at one level what is atomic may not at all be atomic at a lower level.

Bank Account Example

Consider your Mellon bank account. At one level of abstraction, the operations of withdrawing, depositing, and determining your balance may each be considered atomic. However, the operation of transferring money from one account to another might not; it might require two atomic steps: a withdrawal from one account followed by a deposit into the other.

At a lower level of abstraction your bank account might be represented as a file in Mellon Bank's central database. Performing an operation, e.g., a withdrawal, at the higher level might actually involve various access and update operations to the file. At this lower level of abstraction, we can assume that accessing and updating a file are each atomic; thus, we might effect the atomicity of the higher-level withdrawal as a sequence of lower-level atomic accesses and updates to a file.

At an even lower level, we could imagine that the file actually lives on a physical disk. At this level, reading and writing to disk are each atomic.

Even lower, we could imagine that the file is physically spread across multiple pages on the disk; reading and writing to disk is made up of atomic reads and writes to individual disk pages.

Below this, we have atomic reads and writes to the bits under the disk head. Below that we have physics (magnets, electrons, you get the idea ...).

End of Example

At any given level of abstraction, what determines whether an operation is *atomic* is whether you consider the execution of that operation to be indivisible. If it is indivisible you can't divide it any further.

If an operation is atomic *you cannot observe any intermediate states* when it executes. If an operation is non-atomic then you can observe possible intermediate states when it executes. For example, in the transfer operation above, if it's non-atomic then I can observe the intermediate state in which the sum of my two bank accounts is less than the sum of their original (and similarly, final) values. If I choose to model the transfer operation as atomic[1], then I cannot observe this intermediate state.

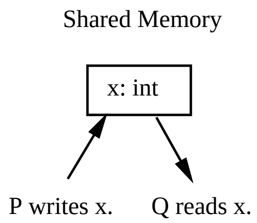
When you model a concurrent system, you certainly need to specify each of your atomic operations. Sometimes you will need to specify non-atomic operations too (e.g., the transfer operation), perhaps as a sequence of atomic operations. Remember that because there are other processes executing concurrently with the process executing the non-atomic operation, you have weaker guarantees. For example, suppose I specify f to be the non-atomic operation that is made up of the sequence of atomic g and h:

$$f = \langle\langle g \rangle\rangle; \langle\langle h \rangle\rangle$$

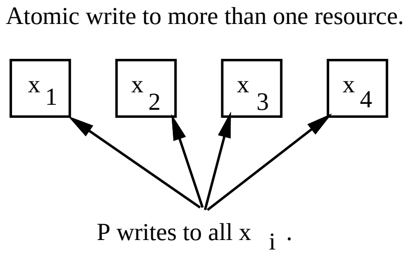
where I am using double angle brackets around g and h to emphasize that they each execute atomically. Then, in between the execution of g and h, *some other process* can do something (interleave one of its operations), therefore changing the state of the system from what g just left it in. That is, g's post-condition may not hold anymore; in particular it may not hold by the time h starts executing.

2.2 Communication

In a shared-memory model, processes communicate through *shared variables* (sometimes more generally called *shared resources*).



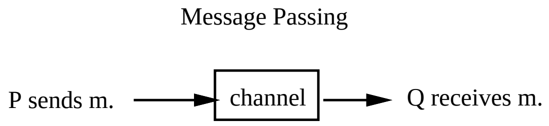
Here, the *read* and *write* operations on the shared variable are each considered atomic. Note that there are other shared- memory models, e.g., where there is a set of shared variables:



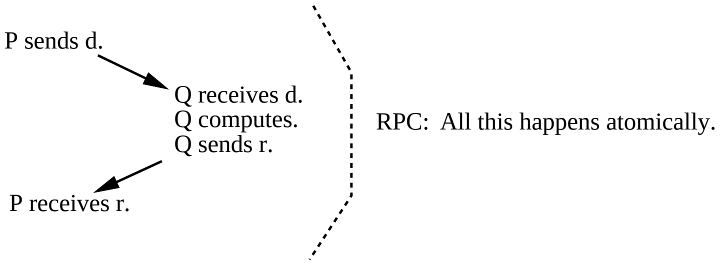
and an (atomic) *write* is a simultaneous write to each variable. This model might be appropriate for modeling a replicated database or a multiprocessor with a main memory and local caches per processor. In either case, we might want an update to a single copy of the data to be effected on all copies at the same time (at least from the outside observer's viewpoint).

In a message-passing model, processes communicate through a message channel (sometimes called a *message buffer*).

[1] as with most ATM machines.



Here, the *send* and *receive* operations are each considered atomic. Again, there are other message-passing models. For example, the multiple write operation in the second shared memory model above would be akin to an *atomic broadcast* primitive. More familiarly, we might consider the send and receive operations together to be atomic as in *remote procedure call*: P sends data to Q and blocks; Q receives and does some local computation; Q sends results to P; P receives the results.



2.3 Synchronization

In a shared-memory model, you don't want P and Q to be writing to the shared variable at the same time. This (safety) property of having at most one process access a shared variable at once is called *mutual exclusion*. To synchronize access to the shared variable, i.e., to implement mutual exclusion, you resort to using one or more *synchronization constructs* that have been designed and thoroughly studied over the years, e.g., *semaphores*, *mutexes*, *condition variables*, and *monitors*.

You also usually want Q to always read the "last" value written. Here "last" must be defined in terms of some ordering. Usually this ordering is a partial order on the set of operations performed by the concurrent processes; sometimes it's a real-time order.

A good example of shared memory is the highway system. When cars approach a four-way intersection, we do not want all cars to enter the intersection at once. We could allow cars moving parallel to each other (but in opposite directions) to enter the intersection at the same time, but certainly not cars whose paths are perpendicular to each other. So, we use traffic lights to mediate access to the shared resource.

In a message-passing model, the sending and receiving processes need to coordinate sending and receiving messages with each other so that messages sent are eventually received and that messages received have actually been sent. They synchronize access to the shared channel.

There are two kinds of communication over this shared channel: *asynchronous* and *synchronous*. In asynchronous message passing, the sender is *non-blocking*; it sends its message and proceeds immediately to do more work, not waiting for the receiver to receive the message. The sender and receiver execute independently of each other.

A good example of asynchronous message passing is the postal system where the sender drops a piece of mail in the mailbox and continues merrily along (shopping, eating, whatever); the recipient receives the mail sometime later. (It would be terrible if the sender had to wait until the mail was actually delivered to the recipient; it would get awfully tired and hungry!)

In synchronous message passing, both sender and receiver are *blocking* and the channel provides a direct link between the two processes. A process sending a message delays until the other process is ready to receive it. An exchange of a message represents a synchronization point between the two processes. Thus, communication and synchronization are tightly-coupled.

A good example of synchronous message passing is the telephone system[2] where the caller places a call

[2] at least in the good old days before all the fancy features of today's telephones.

and waits for the callee to answer; the caller is blocked (and does not do more work) until the callee answers. Note that initially the callee is blocked too; for example, it does not periodically pick up its phone to see if there's someone trying to talk to it. We can also model *buffered message passing*, in which the channel has capacity. Here, the sender delays if the channel is full; thus, if the callee is busy, then the caller has to wait until the callee is off the phone before being able to place the call successfully.

CSP, a model of concurrent systems that we will study in detail this term is based on synchronous message passing.