Concurrency lecture 3

Event counters

Classic problems part 2 - 6.5

Different sorts of problems

Monitors

Event counters - Reed and Kanodia 1977

Another way of providing safe concurrent programs.

This time with three operations (which don't have to be atomic).

The event counter always starts at zero and has enough bits that it never overflows.

```
advance(E):
  E = E + 1;
  if any processes awaiting the new value of E then
    start these processes;

read(E):
  return the value of E;

await(E, count):
  if E < count then
    put this process to sleep, awaiting E >= count;
```

```
program ProducerConsumerRelationship;
var
  numberDeposited : eventcount;
  numberReceived : eventcount;
  numberBuffer : integer;
```

```
procedure producerProcess;
var
  i : integer;
  nextResult : integer;
begin
  i := 0;
  while true do begin
    calculate(nextResult);
    i := i + 1;
    await(numberReceived, i - 1);
    numberBuffer := nextResult;
    advance(numberDeposited)
  end
end;

procedure consumerProcess;
var
  i : integer;
  nextResult : integer;
begin
  i := 0;
  while true do begin
    i := i + 1;
    await(numberDeposited, i);
    nextResult := numberBuffer;
    advance(numberReceived);
    use(nextResult)
  end
end;

begin
  cobegin producerProcess; consumerProcess;
  coend
end.
```

Readers/Writers problem - 6.5.2

This is a common database problem. Since threads which only want to read a value can run in parallel without interfering with each other we should enable multiple readers access to a shared area of data.

However since writer threads change this value only one such thread should have access at a time.

Some solutions to this including the one in the textbook treat readers as privileged threads and only allow access to the resource to writer threads when there are no waiting readers.

Others give priority to the writers.

Either way leads to indefinite postponement.

You can try solving this without indefinite postponement and which still allows multiple reader threads simultaneous access.

See Deitel's "Operating Systems" 2nd edition.

Different sorts of problems

Correctly programming concurrent processes is difficult.

Using low level constructs like semaphores or event counters are prone to mistakes.

What is wrong with this semaphore solution to the producer/consumer problem?

```
program ProducerConsumerRelationship;
var
  exclusiveAccess : semaphore;
  numberDeposited : semaphore;
  numberBuffer : integer;
procedure producerProcess;
var
  nextResult : integer;
begin
  while true do begin
    calculate(nextResult);
    wait(exclusiveAccess);
    numberBuffer := nextResult;
    signal(exclusiveAccess);
    signal(numberDeposited)
  end
end;

procedure consumerProcess;
var
  nextResult : integer;
begin
  while true do begin
    wait(numberDeposited);
    wait(exclusiveAccess);
    nextResult := numberBuffer;
    signal(exclusiveAccess);
    use(nextResult)
  end
end;
begin
  semaphoreInitialize(exclusiveAccess, 1);
  semaphoreInitialize(numberDeposited, 0);
cobegin as usual...
```

A fast producer will lose data.

This was from a very popular OS textbook (it was fixed up in the second edition).

Another popular problem is forgetting to unlock or signal.

We want an automatic (more or less) way of helping programmers lock and unlock.

Java tries to avoid or minimise problems by implementing a form of monitor.

Monitors - 6.7 Brinch Hansen (1973) Hoare (1974)

You can think of a monitor as an object which only allows one thread to be executing inside it.

It has:

- the shared resource - it can only be accessed by the monitor
- publically accessible procedures - they do the work
- a queue to get in
- scheduler - which thread gets access next
- local state - not visible externally except via access procedures
- initialization code
- condition variables

When a thread calls one of the monitor procedures the monitor checks to see if any other thread is currently running inside.

if no

the thread can enter

if yes

the thread gets queued waiting to enter

Obviously mutual exclusion can be guaranteed by this scheme.

Here is an example in some Pascally like language which includes monitors:

```
monitor  Account;

var
  money : real; { the shared resource }

procedure Deposit(amount : real);
begin
  money := money + amount
end;

function Withdraw(amount : real) : boolean;
begin
  if amount < money then begin
    money := money - amount;
    Withdraw := true
  end
  else
    Withdraw := false
end;
```

```
function Balance : real;
begin
  Balance := money
end;

begin
  money := 0.00
end.
```

There is no longer a problem with two threads trying to change the account at the same time.

Unfortunately this is a very simple example - what happens when a monitor routine needs a resource which is not currently available? e.g. Producer/Consumer problem.

Condition variables

In the producer/consumer problem we want to hold the producer until there is some space in the buffer. When space is available we want it to be able to proceed.

A condition variable is a queue which can hold threads. We have wait and signal operations on condition variables.

wait(conditionVariable) puts the current thread to sleep on the corresponding queue

signal(conditionVariable) wakes up one thread from queue (if there are any waiting)

There is no internal state kept of how many signals and waits there have been.

Thus they are simpler than the similar instructions on semaphores.

A signal with nothing waiting does nothing.

A wait always puts a thread to sleep.

e.g.

```
monitor SimpleBuffer;
var
  buffer : integer;
  bufferFree : boolean;
  empty, full : conditionVariable;

procedure Insert(value : integer);
begin
  if not bufferFree then
    wait(empty);
  buffer := value;
  bufferFree := false;
  signal(full)
end;
```

```
function Retrieve : integer;
begin
  if bufferFree then
    wait(full);
  Retrieve := buffer;
  bufferFree := true;
  signal(empty)
end;

begin
  bufferFree := true
end.
```

But doesn’t signal mean we have two threads running in the monitor?

Either we stop the thread which called signal or we don’t start the new one until the current thread leaves the monitor. If we don’t start the woken up thread until after the current thread leaves, it may signal on other condition variables as well and we have to make scheduling decisions.

It is also possible the running thread changes the conditions again and the next thread shouldn't really run.

Java avoids this problem by only allowing one condition variable per monitor and lets running threads run to completion after notify()s and before a notified thread is started.

It solves the second problem by recommending you have a while loop with the conditional wait.

Java has a single lock varible per object (it also has one per class).

Synchronized methods must check this variable before allowing entry.

Synchronized blocks check the same variable.

```
…
synchronized (anObject) {
  do things to the object;
}
```

this way code in other objects can keep access synchronized.

How does the Java implementation of monitors differ from a classical monitor?

signal is called notify()

It doesn't provide condition variables.

wait() and notify() have a single queue for the whole object.

The object can have unsynchronized methods which are not private.

Also fields which are not private. Not a good idea.