Lecture 05: Theory of concurrency

Software System Components

Behzad Bordbar
School of Computer Science, University of Birmingham, UK
Recap

- An example of a race condition – travel agent
- Four main conditions to ensure correct concurrent programs
- Using locks – requires ensuring atomic action
- Strict alternation (not all four conditions are satisfied)
- Busy waiting and its problems
Contents

- Busy waiting vs. Sleep and wakeup
- Producer-consumer problem
- Race condition in producer-consumer
- Semaphores
- Solving producer-consumer using semaphores
- Semaphores are difficult to use!
- Monitors
- Monitors and Java
Busy waiting (recap)

When a thread wants to enter the DR, it checks to see if entry is allowed, i.e. sits in a loop waiting.

- Waste of resources
- Priority inversion problem

- Avoid waste of resources *Sleep and wakeup*

- Sleep is a system call that allows the caller to block itself, until another thread wakes it up (wakeup call)

- We must also take care of the case that the thread is not sleep and it receives a wakeup call

- How do we program this?

- This leads to the idea of *semaphores*
Producer-consumer

- Two threads share the same common buffer
- Buffer is of fixed size
- Producer puts info/items into buffer
- Consumer takes info out
- If buffer full, producer goes to sleep – it is awakened when the consumer removes an item
- If buffer is empty, consumer goes to sleep – it is awakened when producer puts an item in the buffer
- Let us use integer variable $count$ for the number of items in the buffer, $count \leq n$.
- Producer/consumer increment/decrement $count$. 
Race condition in P-C

- Suppose buffer empty: \( \text{count} = 0 \)
- Consumer reads the value count - scheduler kicks in stops running the consumer and start running the producer. Notice to consumer count is 0
- Producer adds an item, \( \text{count} = 1 \)
- Hence, producer says, now must wake up consumer, send wakeup call
- But the consumer is NOT sleep and the wake up call is discarded (lost)
- When the scheduler starts the consumer, it will go to sleep, because the value for the consumer is still 0
- Producer adds items to buffer, when it gets full, producer goes to sleep - both are sleep!
E.W. Dijkstra suggested using of a variable value to count the number of wakeups used for further use (semaphore)—designed originally for processes.

value 0 means no wakeup calls.

There are two operations:

1. **down()** invoked on a semaphore checks the value:
   - If value > 0 decrement the value (uses on wakeup) and continue.
   - If value = 0 process goes to sleep without completing the down() operation.

The step involving -Checking the value and changing it and going to sleep— is atomic (implementation issue).
semaphores (continue)

2. `up()` invoked on a semaphore increments the value

If one or more processes are sleeping on a semaphore unable to complete an earlier `down()` one of them (randomly) is chosen by the system to complete its `down()`

The stage involving – incrementing a semaphore and waking up a process – is atomic (implementation issue) - for example as system calls, while OS disabling all interrupts while dealing with semaphore
Solving P-C via semaphores

- Create three semaphores
  1. full for counting number slots full. full is initially set to 0
  2. empty for counting number slots that are empty. empty is initially set to N
  3. BS (Binary Semaphore) to ensure P-C do not access CR at the same time. It has only two values \(\{0,1\}\). BS is initially set to 1
Solving P-C via semaphores (ctd)

**Producer()**

```plaintext
item = produce_item() // generate item to put in buff
empty.down() // increment empty count
BS.down() // enter CR
item.insert2buffer() // insert the item to buffer
BS.up() // leave critical region
full.up() // increment count of full slots
```

**Consumer()**

```plaintext
full.down() // decrement down counter
BS.down() // enter CR
item.removeFromBuffer() // remove item from buffer
BS.up() // leave CR
empty.up() // increment count of empty slots
item.use_item() // do whatever you want with the item
```
Example: using Mutex

- When the semaphore ability to count is not needed
- mutex has two states: *locked* and *unlocked*

Thread 1:
```
While (true){
    execute non-CR
    down(mutex)
    enter CR;
    up(mutex)
    execute non-CR
}
```

Thread 2:
```
While (true){
    execute non-CR
    down(mutex)
    enter CR;
    up(mutex)
    execute non-CR
}
```
Semaphores: difficult to program

- Suppose a minor change: in producer code, BS was decremented before empty (change the order)

```java
Producer()
...
BS.down()
empty.down()
...
```

- If the buffer is full the producer blocks, while BS has the value 0
- Next access by buffer is BS.down(). But the value of BS is 0. Hence, consumer blocks.
- Both are blocked!
Monitors

- Semaphores are powerful
- Difficult to getting it right. Simple error results in deadlocks, livelocks, starvation,…
- Solution to raise the level of abstraction
- Tony Hoare (1974) and Brinch-Hansen (1975) suggested a high-level programming construct for the synchronisation in processes called a monitor.
Monitors

- We are not going to go through details of monitors and proving it is equivalent with Semaphore.
- We are ONLY interested in Java implementation of monitors
- Here we give a brief overview
- For further details: www.acm.org/classics/feb96/
- A monitor is essentially a *shared class with explicit queues*. 
shared class

Monitor includes a collection of
- Private variables
- Private procedures

(only can be modified with the monitor)
- Constructors to initialise the monitor
- A number of public method calls to can be invoked by processes

A monitor has no public data: processes (thread) can call a monitor, but they can not access the monitor’s internal data from outside
Syntax of a monitor

**monitorname**: monitor

**begin** ... declarations of data local to the monitor;

**procedure** **procname** *(formal parameters . . .)*;

**begin** ... procedure body ... **end**;

... declarations of other procedures local to the monitor;

... initialization of local data of the monitor ...

**end**;
Explicit queues

- At most one process (thread) can be executing within a monitor => mutual exclusion
- If a process calls a monitor procedure and the monitor has a process running, the caller will be blocked outside of the monitor.
- This requires creation of (explicit) queues.
There is "wait" operation, issued from inside a procedure of the monitor, which causes the calling program to be delayed.

and a "signal" operation, also issued from inside a procedure of the same monitor, which causes exactly one of the waiting programs to be resumed immediately.

If there are no waiting programs; the signal has no effect.
Condition variable

- In many cases, there may be more than one reason for waiting, and these need to be distinguished by both the waiting and the signalling operation.
- We therefore introduce a new type of "variable" known as a "condition"
- Condition "variable" is neither true nor false; indeed, it does not have any stored value accessible to the program.
- In practice, a condition variable will be represented by an (initially empty) queue of processes which are currently waiting on the condition
- This queue is invisible both to waiters and signallers.
Example:

- Consider a single resource, which is dynamically acquired and released by an unknown number of customer processes by calls on procedures
  
  ```pascal
  procedure acquire;
  procedure release;
  ```

- Introduce a variable `busy: Boolean` determines whether or not the resource is in use. If an attempt is made to acquire the resource when it is busy, the attempting program must be delayed by waiting on a variable

- Introduce a variable condition `nonbusy: condition` which is signalled by the next subsequent release.
The initial value of $busy$ is $false$.

```
single resource: monitor
   begin
      busy: Boolean;
      nonbusy: condition;
      procedure acquire;
         begin
            if busy then nonbusy.wait;
            busy := true
         end;
      procedure release;
         begin
            busy := false;
            nonbusy.signal
         end;
   end
   busy := false; comment initial value;
end single resource;
```
Back to Java

- Using true monitors you need two queues
  (Similarly, remember in semaphore P-C we used two semaphores full and empty)
- In Java only one queue for waiting on an object (we will see this in our future lectures)
- Better design at the program level is required!
Brinch-Hansen:

Java’s most serious mistake was the decision to use the sequential part of the language to implement the run-time support for the parallel features.

In 1975, Concurrent Pascal demonstrated that platform independent parallel programs (even small operating systems) can be written as a secure programming language with monitors. It is astounding to me that Java’s insecure parallelism is taken seriously by the programming language community a quarter of a century after the invention of monitors and Concurrent Pascal. It has no merit.

What does this mean for a Java programmer?