Condition variables
Today

• Monitors
• Condition variables
• Solving classic problems with monitors
Predicates on shared data

• The restrictions imposed on when threads can access shared data are predicates on shared data.
  – Statements on shared data that are either true or false.
  – Examples: IsBufferEmpty?, AreThereActiveReaders?
• Threads coordinate their access to shared data based on these predicates.
  – A thread may need to check this predicate before continuing execution.
  – The execution of another thread may change the truth value of this predicate.
Encoding predicates with semaphores

Semaphores can encode any predicate, but
• we need to find the right initialization,
• we may need to use multiple semaphores and variables,
• they are low-level and thus error-prone.
Monitor

- A data abstraction mechanism, which consists of:
  - state and
  - procedures.
- The state is modeled by shared variables.
- The procedures are the only means by which the state is manipulated.
- Mutual exclusion: only one thread can execute a monitor procedure at any time.

```plaintext
Monitor monitor_name
{
    // shared variable declarations

    procedure P1(. . .) { . . . }
    . . .
    procedure PN(. . .) { . . . }
    initialization_code(. . .) { . . . }
}
```
Condition variables

• In a monitor, condition synchronization is explicitly programmed using condition variables.
• The programmer implicitly associates (encodes) a predicate on shared state with a condition variable $c$.
• The value of $c$ is a queue of threads that wait for the corresponding predicate to become true.
Condition variables

• When c’s predicate is false call wait(c):
  – block thread and add it in c’s queue.

• When c’s predicate becomes true call signal(c):
  – awake the first thread in c’s queue and remove it from the queue.

• Signal-and-continue semantics:
  – The awaken thread executes at some point in the future (when it reacquires exclusive access to the monitor).
  – The thread executing signal continues executing.
Synchronization Using Monitors

initialization code

shared data

\( x \)

\( y \)

operations

entry queue
Monitor `EventTracker` {
    int numburgers = 0;
    condition hungrycustomer;

    void customerenter() {
        while (numburgers == 0)
            hungrycustomer.wait()
        numburgers -= 1
    }

    void produceburger() {
        ++numburgers;
        hungrycustomer.signal();
    }
}
Readers and Writers

Monitor ReadersN Writers {

    int NReaders, Nwriters;
    Condition CanRead, CanWrite;

    Void BeginWrite() {

    }

    Void EndWrite() {

    }

    Void BeginRead() {

    }

    Void EndRead() {

    }

}
Readers and Writers

Monitor **ReadersNWriters** {

    int NReaders, NWriters;
    Condition CanRead, CanWrite;

    Void BeginWrite()
    {
        NWriters = 1;
    }

    Void EndWrite()
    {
        NWriters = 0;
    }

    Void BeginRead()
    {
        ++NReaders;
        Wait(CanRead);
        --WaitingReaders;
        ++NReaders;
        Signal(CanRead);
    }

    Void EndRead()
    {
        if(--NReaders == 0)
            Signal(CanWrite);
    }
}
Readers and Writers

Monitor $\text{ReadersNWriters}$

\begin{verbatim}
  int NReaders, NWriters;
  Condition CanRead, CanWrite;

  Void BeginWrite()
  {
    while(NWriters == 1 || NReaders > 0)
    {
      wait(CanWrite);
      wait(CanWrite);
    }
    NWriters = 1;
  }

  Void EndWrite()
  {
    NWriters = 0;
  }

  Void BeginRead()
  {
    while(NWriters == 1)
    {
      Wait(CanRead);
    }
    ++NReaders;
  }

  Void EndRead()
  {
    --NReaders
  }
\end{verbatim}

Readers and Writers

Monitor **ReadersNWriters** {

    int NReaders, NWriters;
    Condition CanRead, CanWrite;

    Void BeginWrite()
    {
        while(NWriters == 1 || NReaders > 0)
        {
            wait(CanWrite);
        }
        NWriters = 1;
    }

    Void EndWrite()
    {
        NWriters = 0;
        Signal(CanRead);
        Signal(CanWrite);
    }

    Void BeginRead()
    {
        while(NWriters == 1)
        {
            Wait(CanRead);
        }
        ++NReaders;
        Signal(CanRead);
        Signal(CanWrite);
    }

    Void EndRead()
    {
        if(--NReaders == 0)
        {
            Signal(CanWrite);
        }
    }
}
Semaphores VS Condition variables

• wait(c) is like P(S), and signal(c) is like V(S).
• However:
  – signal has no effect if no thread is waiting, but V has.
  – wait always blocks a thread, P does not.
Synchronization primitives

- All can encode any predicate on shared data.
- Each primitive can be used to implement another primitive.

<table>
<thead>
<tr>
<th></th>
<th>Locks (acquire, release)</th>
<th>Semaphores (init, P, V)</th>
<th>Condition Variables (wait, signal)</th>
</tr>
</thead>
</table>

Decreasing programming effort to encode predicates
class EventTracker:
    def __init__(self):
        self.hungrycustomer_lock = Lock()
        self.hungrycustomer = Condition(self.hungrycustomer_lock)
        self.numburgers = 0

    def customerenter(self):
        with self.hungrycustomer_lock:
            while self.numburgers == 0:
                self.hungrycustomer.wait()
            # check if indeed there is a burger
            assert(self.numburgers > 0)
            self.numburgers -= 1

    def produceburger(self):
        with self.hungrycustomer_lock:
            self.numburgers += 1
            self.hungrycustomer.notify()
Today

• Monitors
• Condition variables
• Solving classic problems with monitors

• [2] Implementing condition variables with semaphores, Andrew Birrell
Coming up...

• Next lecture: deadlocks
• HW2: all exercises
  – Due on Monday, 10pm
• In-class exam
  – Tuesday, last N mins of class
  – Based on HW1 and HW2