Announcements

- My office hours 4-5PM today (schedule change for this week only).
Lecture 36: Parallelism, Continued
Other Kinds of Sorting

- Another way to sort a list is *merge sort*:

  ```python
def sort(L, first, last):
    if first < last:
      middle = (first + last) // 2
      sort(L, first, middle)
      sort(L, middle+1, last)
      L[:] = merge(L[first:middle+1], L[middle+1:last+1])
      # Merge takes two sorted lists and interleaves
      # them into a single sorted list.
  ```

- Assuming that merging takes time $\Theta(N)$ for two lists of size $N/2$, this operation takes $\Theta(\ ? \ )$ steps.

- *(Batcher’s)* bitonic sort does a different kind of merge sort that runs in $\Theta((\lg N)^2)$ time, with enough processors.
Other Kinds of Sorting

- Another way to sort a list is **merge sort**:

  ```python
def sort(L, first, last):
    if first < last:
        middle = (first + last) // 2
        sort(L, first, middle)
        sort(L, middle+1, last)
        L[:] = merge(L[first:middle+1], L[middle+1:last+1])
        # Merge takes two sorted lists and interleaves
        # them into a single sorted list.
  ```

- Assuming that merging takes time $\Theta(N)$ for two lists of size $N/2$, this operation takes $\Theta(N \lg N)$ steps.

- *(Batcher’s)* **bitonic sort** does a different kind of merge sort that runs in $\Theta((\lg N)^2)$ time, with enough processors.
Example: Bitonic Sorter

Data
Comparator
Separates parallel groups
Bitonic Sort Example (I)

77  77  77  77  77  77  77  92
16  16  47  47  47  92  92  77
47  8  16  16  8  52  52  52
1  52  52  92  92  8  8  16
52  1  92  52  16  16  8  8
6  92  1  6  6  6  6  6
92  6  6  1  1  1  1  1
24  24  24  99  99  99  99  99
7  7  99  24  35  56  56  56
99  99  7  15  48  48  48  48
15  15  15  7  56  35  35  35
13  35  48  56  7  24  24  24
35  13  56  48  15  15  15  15
56  56  13  35  24  7  13  13
48  48  35  13  13  13  7  7
Bitonic Sort Example (II)
Implementing Parallel Programs

- The sorting diagrams were abstractions.
- Comparators could be processors, or they could be operations divided up among one or more processors.
- Coordinating all of this is the issue.
- One approach is to use *shared memory*, where multiple processors (logical or physical) share one memory.
- This introduces conflicts in the form of *race conditions*: processors racing to access data.
Memory Conflicts: Abstracting the Essentials

• When considering problems relating to shared-memory conflicts, it is useful to look at the primitive read-to-memory and write-to-memory operations.

• E.g., the program statements on the left cause the actions on the right.

\[
\begin{align*}
x &= 5 \\
x &= \text{square}(x) \\
y &= 6 \\
y &= y + 1
\end{align*}
\]

\[
\begin{align*}
\text{WRITE } 5 &\rightarrow x \\
\text{READ } x &\rightarrow 5 \\
&\quad(\text{calculate } 5 \times 5 \rightarrow 25) \\
\text{WRITE } 25 &\rightarrow x \\
\text{WRITE } 6 &\rightarrow y \\
\text{READ } y &\rightarrow 6 \\
&\quad(\text{calculate } 6 + 1 \rightarrow 7) \\
\text{WRITE } 7 &\rightarrow y
\end{align*}
\]
### Conflict-Free Computation

- Suppose we divide this program into two separate processes, P1 and P2:

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x = 5</td>
<td>y = 6</td>
</tr>
<tr>
<td></td>
<td>x = square(x)</td>
<td>y += 1</td>
</tr>
<tr>
<td><strong>P1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE 5 -&gt; x</td>
<td>WRITE 6 -&gt; y</td>
</tr>
<tr>
<td></td>
<td>READ x -&gt; 5</td>
<td>READ y -&gt; 6</td>
</tr>
<tr>
<td></td>
<td>(calculate 5*5 -&gt; 25)</td>
<td>(calculate 6+1 -&gt; 7)</td>
</tr>
<tr>
<td></td>
<td>WRITE 25 -&gt; x</td>
<td>WRITE 7 -&gt; y</td>
</tr>
<tr>
<td><strong>x</strong></td>
<td>25</td>
<td><strong>y</strong> 7</td>
</tr>
</tbody>
</table>

- The result will be the same regardless of which process’s READs and WRITEs happen first, because they reference different variables.
Read-Write Conflicts

• Suppose that both processes read from \( x \) after it is initialized.

\[
\begin{array}{c|c}
    x &= 5 \\
    \hline
    x = \text{square}(x) & y = x + 1 \\
    \hline
    \text{P1} & \text{P2} \\
    \text{READ } x \rightarrow 5 & \text{READ } x \rightarrow 5 \\
    \text{(calculate } 5*5 \rightarrow 25) & \text{(calculate } 5+1 \rightarrow 6) \\
    \text{WRITE } 25 \rightarrow x & \text{WRITE } 6 \rightarrow y \\
    \text{READ } x \rightarrow 5 & \\
    \text{(calculate } 5+1 \rightarrow 6) \\
    \text{WRITE } 6 \rightarrow y & \\
    x &= 25 \\
    y &= 6
\end{array}
\]

• The statements in \text{P2} must appear in the given order, but they need not line up like this with statements in \text{P1}, because the execution of \text{P1} and \text{P2} is independent.
Read-Write Conflicts (II)

- Here's another possible sequence of events

<table>
<thead>
<tr>
<th>x = 5</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>x = square(x)</th>
<th>y = x + 1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
</table>

| READ x -> 5  | READ x -> 25 |
| (calculate 5*5 -> 25) | (calculate 25+1 -> 26) |
| WRITE 25 -> x | WRITE 26 -> y |

x = 25
y = 26
Read-Write Conflicts (III)

- The problem here is that nothing forces P1 to wait for P1 to read x before setting it.

- Observation: The “calculate” lines have no effect on the outcome. They represent actions that are entirely local to one processor.

- The effect of “computation” is simply to delay one processor.

- But processors are assumed to be delayable by many factors, such as time-slicing (handing a processor over to another user’s task), or processor speed.

- So the effect of computation adds nothing new to our simple model of shared-memory contention that isn’t already covered by allowing any statement in one process to get delayed by any amount.

- So we’ll just look at READ and WRITE in the future.
### Write-Write Conflicts

- Suppose both processes write to $x$:

  \[
  \begin{array}{c|c}
  x = 5 \\
  \hline
  x = \text{square}(x) & x = x + 1 \\
  \hline
  \text{P1} & \text{P2} \\
  \hline
  \text{READ } x \rightarrow 5 & \text{READ } x \rightarrow 5 \\
  \text{WRITE } 25 \rightarrow x & \text{WRITE } 6 \rightarrow x \\
  \end{array}
  \]

- This is a **write-write conflict**: two processes race to be the one that “gets the last word” on the value of $x$. 
### Write-Write Conflicts (II)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\textbf{x = 5}</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{x = square(x)}</td>
<td>\texttt{x = x + 1}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
</table>
| \begin{align*}
| \text{READ } x \rightarrow 5 \\
| \text{WRITE } 25 \rightarrow x \\
& \end{align*} | \begin{align*}
| \text{READ } x \rightarrow 5 \\
| \text{WRITE } 6 \rightarrow x
|]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\textbf{x = 26}</td>
</tr>
</tbody>
</table>

- This ordering is also possible; P2 gets the last word.
- There are also read-write conflicts here. What is the total number of possible final values for $x$?
Write-Write Conflicts (II)

<table>
<thead>
<tr>
<th>x = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = square(x)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ x -&gt; 5</td>
<td></td>
</tr>
<tr>
<td>WRITE 25 -&gt; x</td>
<td></td>
</tr>
<tr>
<td>READ x -&gt; 5</td>
<td></td>
</tr>
<tr>
<td>WRITE 6 -&gt; x</td>
<td></td>
</tr>
</tbody>
</table>

x = 26

- This ordering is also possible; P2 gets the last word.
- There are also read-write conflicts here. What is the total number of possible final values for x? **Four: 25, 5, 26, 36**
Coordinating Parallel Computation

Let's go back to bank accounts:

```python
class BankAccount:
    def __init__(self, initial_balance):
        self._balance = initial_balance
    @property
def balance(self): return self._balance
def withdraw(amount):
    if amount > self._balance:
        raise ValueError("insufficient funds")
    else:
        self._balance -= amount
        return self._balance
```

```python
acct = BankAccount(10)
acct.withdraw(8)
acct.withdraw(7)
```

- At this point, we'd like to have the system raise an exception for one of the two withdrawals, and to set `acct.balance` to either 2 or 3, depending on which withdrawer gets to the bank first, like this...
class BankAccount:
    def withdraw(amount):
        if amount > self._balance:
            raise ValueError("insufficient funds")
        else:
            self._balance -= amount
        return self._balance

acct = BankAccount(10)
acct.withdraw(8)
acct.withdraw(7)
READ acct._balance -> 10
WRITE acct._balance -> 2

But instead, we might get...
Undesireable Outcome

class BankAccount:
    def withdraw(amount):
        if amount > self._balance:
            raise ValueError("insufficient funds")
        else:
            self._balance -= amount
        return self._balance

acct = BankAccount(10)

acct.withdraw(8) # READ acct._balance -> 10
acct.withdraw(7) # READ acct._balance -> 10

READ acct._balance -> 10
WRITE acct._balance -> 2

Oops!
Serializability

• We define the desired outcomes as those that would happen if withdrawals happened sequentially, in some order.

• The nondeterminism as to which order we get is acceptable, but results that are inconsistent with both orderings are not.

• These latter happen when operations overlap, so that the two processes see inconsistent views of the account.

• We want the withdrawal operation to act as if it is atomic—as if, once started, the operation proceeds without interruption and without any overlapping effects from other operations.
One Solution: Critical Sections

- Some programming languages (e.g., Java) have special syntax for this. In Python, we can arrange something like this:

  ```python
def withdraw(amount):
    with CriticalSectionManager:
      if amount > self._balance:
        raise ValueError("insufficient funds")
      else:
        self._balance -= amount
    return self._balance
  ```

- The `with` construct:
  1. Calls the `__enter__()` method of its “context manager” argument (here, some object we’ll call `CriticalSectionManager`);
  2. Executes the body (indented portion);
  3. Finally, it calls the `__exit__()` method on the context manager. It guarantees that it will always do so, no matter how you exit from the body (via `return`, exception, etc.).

- The idea is that our `CriticalSectionManager` object should let just one process through at a time. How?
Locks

- To implement our critical sections, we'll need some help from the operating system or underlying hardware.

- A common low-level construct is the lock or mutex (for “mutual exclusion”): an object that at any given time is “owned” by one process.

- If $L$ is a lock, then
  - $L$.acquire() attempts to own $L$ on behalf of the calling process. If someone else owns it, the caller waits for it to be released.
  - $L$.release() relinquishes ownership of $L$ (if the calling process owns it).
Implementing Critical Regions

- Using locks, it's easy to create the desired context manager:

```python
from threading import Lock

class CriticalSection:
    def __init__(self):
        self.__lock = Lock()

    def __enter__(self):
        self.__lock.acquire()

    def __exit__(self, exception_type, exception_val, traceback):
        self.__lock.release()

CriticalSectionManager = CriticalSection()
```

- The extra arguments to `__exit__` provide information about the exception, if any, that caused the `with` body to be exited.

- (In fact, the bare `Lock` type itself already has `__enter__` and `__exit__` procedures, so you don’t really have to define an extra type).
Granularity

- We’ve envisioned critical sections as being atomic with respect to all other critical sections.
- Has the advantage of simplicity and safety, but causes unnecessary waits.
- In fact, different accounts need not coordinate with each other. We can have a separate critical section manager (or lock) for each account object:

```python
class BankAccount:
    def __init__(self, initial_balance):
        self._balance = initial_balance
        self._critical = CriticalSection()
    def withdraw(self, amount):
        with self._critical:
            ...
```

- That is, can produce a solution with finer granularity of locks.
Synchronization

• Another kind of problem arises when different processes must communicate. In that case, one may have to wait for the other to send something.

• This, for example, doesn't work too well:

```python
class Mailbox:
    def __init__(self):
        self._queue = []
    def deposit(self, msg):
        self._queue.append(msg)
    def pickup(self):
        while not self._queue:
            pass
        return self._queue.pop()
```

• Idea is that one process deposits a message for another to pick up later.

• What goes wrong?
Problems with the Naive Mailbox

```python
class Mailbox:
    def __init__(self):
        self._queue = []
    def deposit(self, msg):
        self._queue.append(msg)
    def pickup(self):
        while not self._queue:
            pass
        return self._queue.pop()
```

- **Inconsistency:** Two processes picking up mail can find the queue occupied simultaneously, but only one will succeed in picking up mail, and the other will get exception.

- **Busy-waiting:** The loop that waits for a message uses up processor time.

- **Deadlock:** If one is running two logical processes on one processor, busy-waiting can lead to nobody making any progress.

- **Starvation:** Even without busy-waiting one process can be shut out from ever getting mail.
Conditions

• One way to deal with this is to augment locks with *conditions*:

```python
from threading import Condition
class Mailbox:
    def __init__(self):
        self._queue = []
        self._condition = Condition()
    def deposit(self, msg):
        with self._condition:
            self._queue.append(msg)
            self._condition.notify()
    def pickup(self):
        with self._condition:
            while not self._queue:
                self._condition.wait()
            return self._queue.pop()
```

• *Conditions* act like locks with methods *wait*, *notify* (and others).
• *wait* releases the lock, waits for someone to call *notify*, and then reacquires the lock.
Another Approach: Messages

- Turn the problem inside out: instead of client processes deciding how to coordinate their operations on data, let the data coordinate its actions.

- From the Mailbox’s perspective, things look like this:

  ```python
  self._queue = []
  while True:
    wait for a request, R, to deposit or pickup
    if R is a deposit of msg:
      self._queue.append(msg)
      send back acknowledgement
    elif self._queue and R is a pickup:
      msg = self._queue.pop()
      send back msg
  ```
Rendezvous

- Following ideas from C.A.R Hoare, the Ada language used the notion of a **rendezvous** for this purpose:

```ada
task type Mailbox is
  entry deposit(Msg: String);
  entry pickup(Msg: out String);
end Mailbox;

task body Mailbox is
  Queue: ...
begin
  loop
    select
      accept deposit(Msg: String) do Queue.append(Msg); end;
      or when not Queue.empty =>
        accept pickup(Msg: out String) do Queue.pop(Msg); end;
    end select;
  end loop;
end;
```