WASHINGTON STATE UNIVERSITY
VANCOUVER

CS 563

Concurrent Programming

Lecture 13: Barriers

Barrier Synchronization

- A BARRIER is a point that all processes must reach before any proceed

  - very common in iterative parallelism

- Example:

```
"co inside while" style of parallelism
      while () {
         co ... oc   # "oc" is
         # essentially a barrier
      }
```

Counter Barrier - for n processes

```
int count = 0;
Barrier:  < count++ > # record arrival
          < await (count==n); > # wait for
                                # everyone to arrive
```


- Implementation:
  - increment -- use FA or critical section
  - delay loop -- use spin loop

Problems

- Reuse problem -- How do we reset count?
- Contention -- single shared counter

Solving the reuse problem

- Try counting up then counting down (called reverse sense)

```
odd barriers:   < count++ >
          < await(count==n) >
even barriers:  < count-- >
                < await(count==0) >
```

Solving the reuse problem

- Use TWO counters AND reverse their senses

```
up1, up2, down1, down2, repeat
```

- Why does this work?

Coordinator Barrier

- Idea: distribute the single counter above (a time/space tradeoff)
- Shared variables:

```
int arrive[1:n] = ([n] 0);
    continue[1:n] = ([n] 0);
```

Coordinator Barrier

- The basic signaling scheme is then implemented as follows:


```
int arrive[1:n] = ([n] 0), continue[1:n] = ([n] 0);
```


- Worker[i]:

```
process Worker[i = 1 to n] {
  while (true) {
        code to implement task i;
        arrive[i] = 1;
            ⟨await (continue[i] == 1);⟩
        . . .
  }
}
```

- Coordinator:

```
process Coordinator {
  while (true) {
        for [i = 1 to n] {
                ⟨await (arrive[i] == 1);⟩

                . . .
        }
        for [i = 1 to n] continue[i] = 1;
  }
}
```

Coordinator Barrier

- What about the reset problem?
- solve by clearing flags at the … points above

- be sure to follow the Flag Synchronization Principles:
  - Process waiting for a flag to be set should clear the flag
  - A flag should not be set until it is known that it is clear

Coordinator Barrier

- Init


```
int arrive[1:n] = ([n] 0), continue[1:n] = ([n] 0);
```


- Worker[i]:

```
process Worker[i = 1 to n] {
  while (true) {
        code to implement task i;
        arrive[i] = 1;
            ⟨await (continue[i] == 1);⟩
        continue[i] = 0;
  }
}
```

- Coordinator:

```
process Coordinator {
  while (true) {
        for [i = 1 to n] {
              ⟨await (arrive[i] == 1);⟩
                arrive[i] = 0;
        }
        for [i = 1 to n] continue[i] = 1;
  }
}
```

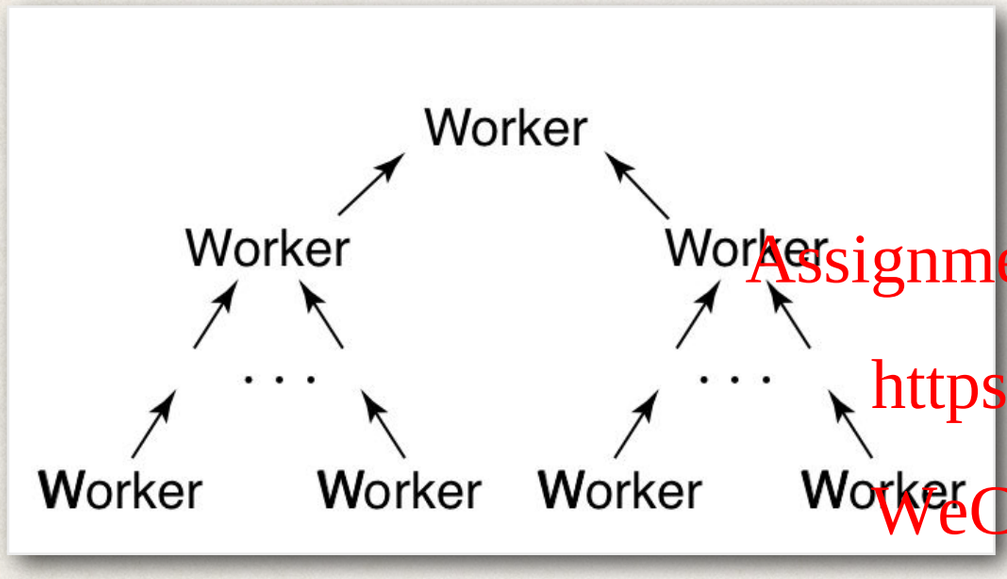
Coordinator Barrier

```
process Worker[i = 1 to n] {
    while (true) {
        code to implement task i;
        arrive[i] = 1;
        ⟨await (continue[i] == 1);⟩
        continue[i] = 0;
    }
}
```

```
process Coordinator {
    while (true) {
        for [i = 1 to n] {
            ⟨await (arrive[i] == 1);⟩
            arrive[i] = 0;
        }
        for [i = 1 to n] continue[i] = 1;
    }
}
```


- Why 2n flags?
  - Can we make it work with n flags
- What about contention?
- What about total time in best case (all workers arrive at once)

Combining Tree Barriers



leaf node `L:`
```
arrive[L] = 1;
⟨await (continue[L] == 1);⟩
continue[L] = 0;
```

interior node `I:`
```
⟨await (arrive[left] == 1);⟩
arrive[left] = 0;
⟨await (arrive[right] == 1);⟩
arrive[right] = 0;
arrive[I] = 1;
⟨await (continue[I] == 1);⟩
continue[I] = 0;
continue[left] = 1; continue[right] = 1;
```

root node `R:`
```
⟨await (arrive[left] == 1);⟩
arrive[left] = 0;
⟨await (arrive[right] == 1);⟩
arrive[right] = 0;
continue[left] = 1; continue[right] = 1;
```

Sum up

- All processes must arrive before any leave
- Flags: one per edge in the "signaling graph"
- Different types of barriers so far:
  - Counter -- symmetric, but reset problem and O(n)
  - Coordinator -- simple, but asymmetric and O(n)
  - Tree -- O(log n), but asymmetric and harder to program

Two Process Barrier

- Basic building block for two processes:


```
Worker1  <-->  Worker2    # signal each other

shared vars:    int arrive[n] = ([n] 0);

Worker[i]:      ...
            arrive[i] = 1;
            < await(arrive[j]==1); >
            ...

Worker[j]:      ...
            arrive[j] = 1;
            < await([arrive[i]==1); >
            …
```

What about reset?

Two Process Barrier


```
Worker[i]:
                arrive[i] = 1;
                < await(arrive[j]==1); >
                arrive[j] = 0;

Worker[j]:
                arrive[j] = 1;
                < await([arrive[i]==1); >
                arrive[i] = 0;
```

?

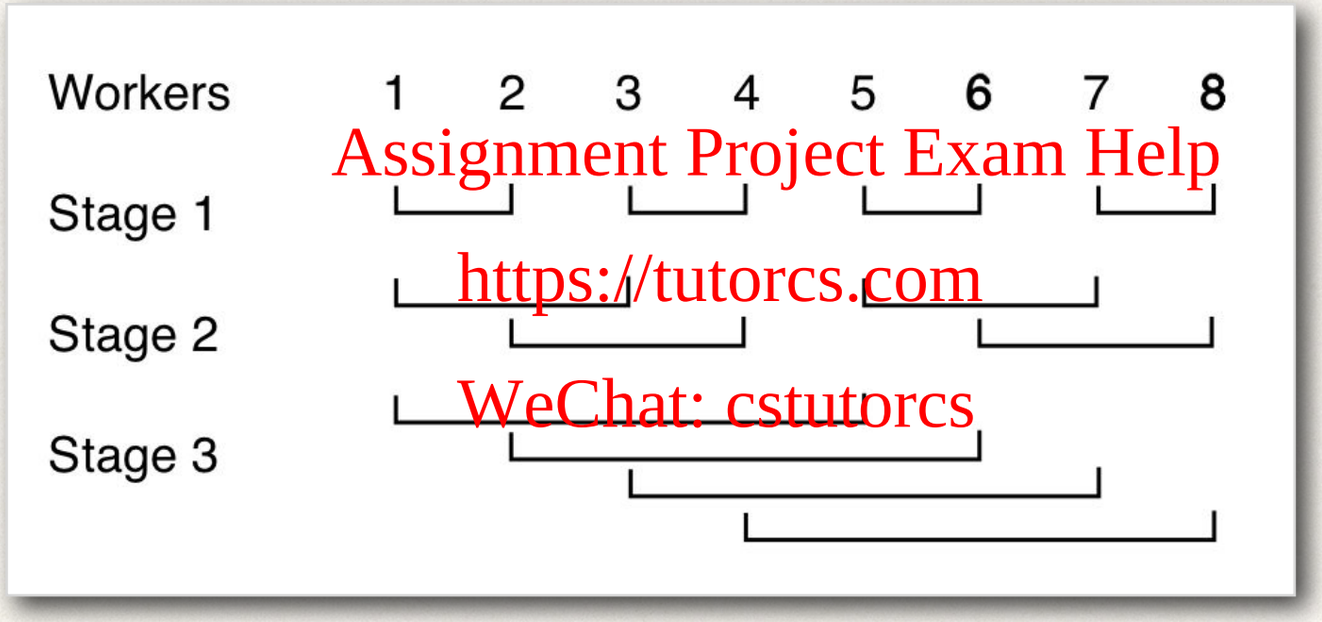
Symmetric Barriers

- What about reuse?


```
Worker[i]:      <await (arrive[i] == 0); >
                arrive[i] = 1;
                < await(arrive[j]==1); >
                arrive[j] = 0;

Worker[j]:      <await (arrive[j] == 0); >
                arrive[j] = 1;
                < await([arrive[i]==1); >
                arrive[i] = 0;
```

Butterfly Barrier



- $\log_2 n$ stages of 2 process barriers
- idea is to replicate work: each worker "barriers" with $\log_2 n$ others

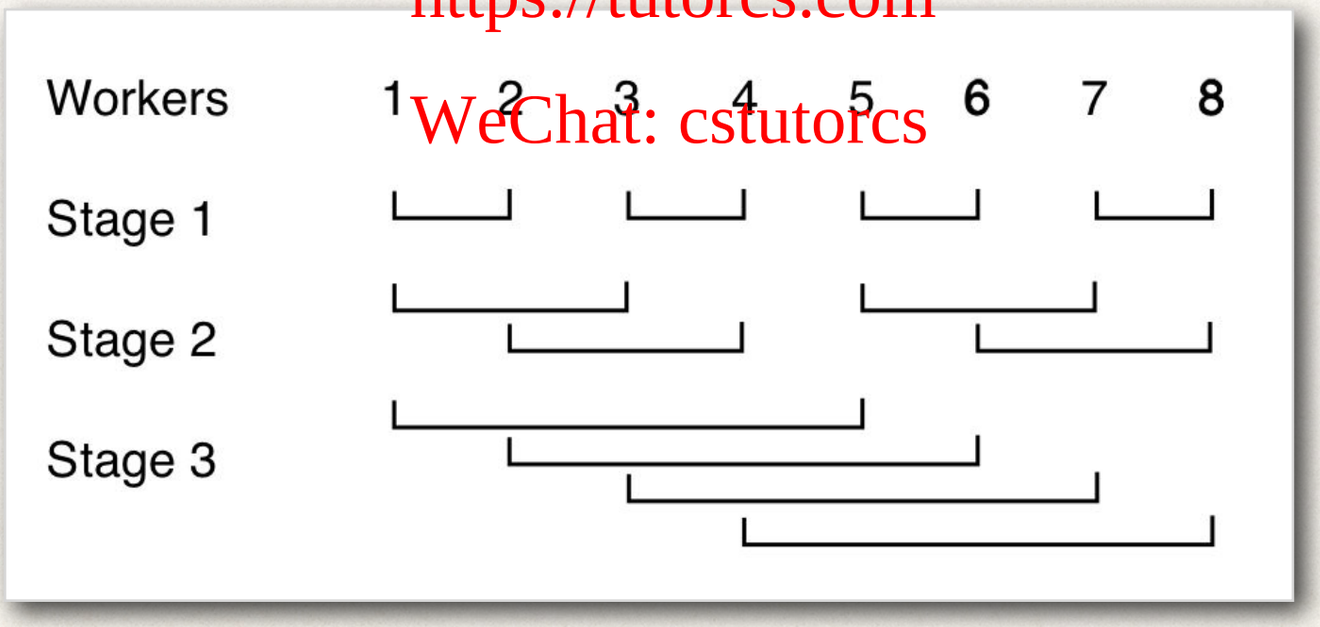
Butterfly Barrier

- Reuse:
  - Use multiple flags (arrays)
  - Or better yet, use stage counters:

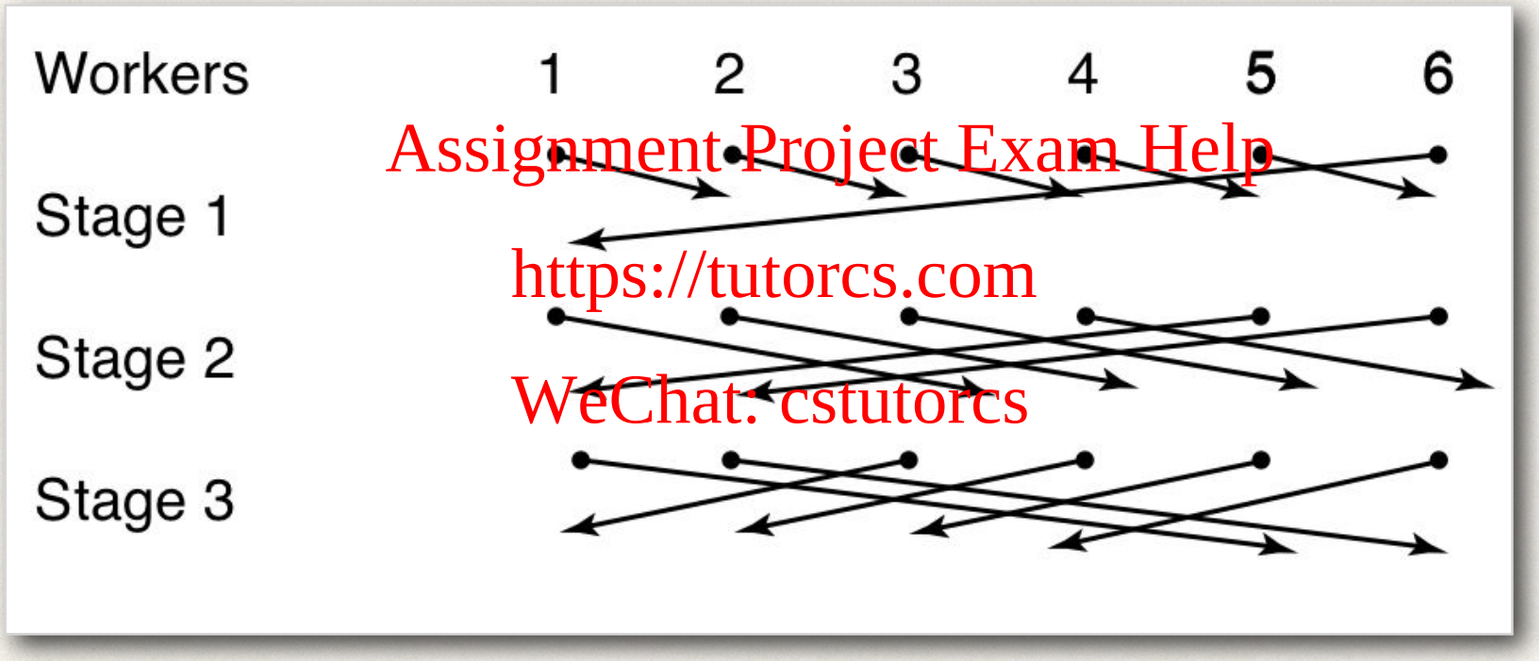

```
# barrier code for worker process I
for [s = 1 to num_stages] {
    arrive[i] = arrive[i] + 1;
    #determine neighbour j for stage s
    while (arrive[j] < arrive[i]) skip;
}
```

Butterfly Barrier

- Any disadvantages?



Dissemination Barrier



- A different way to connect the processes
- Simpler to program and works for any value of n