Announcements

HW2

• Due this coming-up Monday

Thursday, 21 July

• Optimization Lab

Wednesday, 27 July

• Final Exam
• In-class
• You can bring with you one piece of paper with notes
• No electronic help of any kind is allowed

Thursday, 28 July

• Last lab (Message passing)
From last time

Our goal: allow CS1 and CS2 to have turns gaining access to their critical sections
From last time

```c
bool lock = false;
bool TS(bool lock) {
    < bool initial = lock;
    lock = true;
    return initial;
    >
}
```

```c
process CS1{
    while(true) {
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

```c
process CS2{
    while(true) {
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

**TS (Test and Set)** is an atomic action procedure that relies on a shared variable `lock`

**Our goal**: allow CS1 and CS2 to have turns gaining access to their critical sections

**Task**: Explain the shortcomings of using TS (consider CS1 through CSn, where n is large), which motivates the following solution ...
From last time

```c
bool lock = false;

bool TS(bool lock) {
    < bool initial = lock;
    lock = true;
    return initial;
}

process CS1{
    while(true) {
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS2{
    while(true) {
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS3{
    while(true) {
        while (lock) {};
        while (TS(lock)) {
            while (lock) {};
        }
        critical section;
        lock = false;
        noncritical section
    }
}
```

Q: What are the advantages of using the code CS3 over the code for CS1 and CS2 above?
From last time

bool lock = false;

bool TS(bool lock){
    < bool initial = lock;
    lock = true;
    return initial;
>
}

process CS1{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS3{
    while(true){
        while (lock) {};
        while (TS(lock)) {
            while (lock) {};
        }
        critical section;
        lock = false;
        noncritical section
    }
}

A process only tests lock, which can be read from local cache, UNTIL there is a possibility that TS can succeed.

This is called Test and Test and Set.

Regardless of whether you use TS and Test and Test and Set, there is the chance that any process will repeatedly acquire the lock and starve all other processes
bool in1=false
bool in2=false
int last=1

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}

Task: Be able to explain why this use of global (shared) variables implements a fair access policy for CS1 and CS2 gaining access to their critical sections

Q: What does the variable `last` specify?
From last time

```
bool in1=true
bool in2=false
int last=1

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}
```

**Scenario 1:** CS1 is executing the body of its while loop, and CS2 is evaluating the condition of its outer most while loop
From last time

```cpp
bool in1 = true
bool in2 = false
int last = 1

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}
```

The while condition evaluates to False, and the spin loop does not execute
From last time

```c
bool in1=true
bool in2=true
int last=2

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}

CS1 enters its critical section

If at any time when CS1 is past its inner most while loop CS2 sets in2 to true, and last to 2 ...
From last time

```c
bool in1=true
bool in2=true
int last=2
```

```c
process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}
```

```c
process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}
```

CS1 enters its critical section

If at any time when CS1 is past its inner most while loop CS2 sets in2 to true, and last to 2 ...

The while loop condition in CS2 evaluates to true and CS2 spins
From last time

```c
bool in1=false
bool in2=true
int last=2

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}
```

CS1 eventually completes its critical section and sets the value of in1 to false

Q: As soon as CS1 sets the value of in1 to false, what happens?

If at any time when CS1 is past its inner most while loop CS2 sets in2 to true, and last to 2 ... The while loop condition in CS2 evaluates to true and CS2 spins
From last time

bool in1=false
bool in2=true
int last=2

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}

CS1 eventually completes its critical section and sets the value of in1 to false

Q: As soon as CS1 sets the value of in1 to false, what happens?

The condition of the while loop in CS2 evaluates to false ...
CS1 eventually completes its critical section and sets the value of \( \text{in1} \) to false

Q: As soon as CS1 sets the value of \( \text{in1} \) to false, what happens?

The condition of the while loop in CS2 evaluates to false ...
And CS2 enters its critical section
From last time

```java
bool in1=true
bool in2=true
int last=1

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}
```

While CS2 is in its critical section, CS1 might complete its non critical section, and proceed to set `in1` to `true` and `last` to `1` ...

The condition of the while loop in CS2 evaluates to false ...
And CS2 enters its critical section
From last time

```plaintext
bool in1=true
bool in2=true
int last=1
```

```plaintext
process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}
```

```plaintext
process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}
```

While CS2 is in its critical section, CS1 might complete its non critical section, and proceed to set `in1` to `true` and `last` to `1` ...

The condition of the while loop in CS2 evaluates to false ...
And CS2 enters its critical section

Which would cause CS1’s while loop condition to evaluate to true, causing spin
From last time

```cpp
bool in1=true
bool in2=true
int last=1

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}
```

Hence last refers to “who had the last attempt at gaining entry to their critical section”
From last time

```c
bool in1=false
bool in2=false
int last=1
```

```c
process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}
```

```c
process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}
```

**Scenario 2:** CS1 is JUST About to execute the body of its while loop, and CS2 is JUST About to execute the body of its while loop
From last time

```c
bool in1=true
bool in2=true
int last=1
```

```c
process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}
```

```c
process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}
```

CS1 and CS2 each update `in1` and `in2`. 
Both CS1 and CS2 attempt to set the value of last

Q: What are the possible outcomes?
bool in1=true
bool in2=true
int last=2

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}

Both CS1 and CS2 attempt to set the value of last

If CS1 sets last to 1, and CS2 immediately sets last to 2 ...
bool in1=true
bool in2=true
int last=2

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}

Both CS1 and CS2 attempt to set the value of last

If CS1 sets last to 1, and CS2 immediately sets last to 2 ... CS2 spins, and CS1 enters its critical section
bool in1=true
bool in2=true
int last=1

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}

Both CS1 and CS2 attempt to set the value of last

If CS1 sets last to 1
Both CS1 and CS2 attempt to set the value of last

If CS1 sets \( \text{last} \) to 1, proceeds to check its while (before CS2 has set \( \text{last} \) to 2) ...

CS1’s while condition evaluates to true, so CS1 spins
Today

Dissemination Barriers
Data Parallel Algorithms
Monitors
Dissemination Barrier

Q: Why use a dissemination barrier versus a butterfly barrier?
Q: What is/are the difference(s) between a butterfly and dissemination barrier?
Dissemination Barrier

Just as a butterfly barrier, in the dissemination barrier, at stage $s$, a process synchronizes with a process $2^{s-1}$ away.
Dissemination Barrier

At stage $s=1$, $2^{s-1} = 2^0 = 1$
Dissemination Barrier

worker worker worker worker worker worker

S1

[Diagram of a dissemination barrier with workers]
Dissemination Barrier

At stage $s=2$, $2^{s-1} = 2^1 = 2$
Dissemination Barrier

worker worker worker worker worker worker

S1

S2
Dissemination Barrier

worker  worker  worker  worker  worker  worker  worker

S1

S2

|          |          |          |          |
Dissemination Barrier

S1

S2
Dissemination Barrier

At stage $s=3$, $2^{s-1} = 2^2 = 4$
Dissemination Barrier

```
S1             worker          worker          worker          worker          worker          worker          worker
|                |                |                |                |                |                |                |
|                |                |                |                |                |                |                |
|                |                |                |                |                |                |                |
S2             worker          worker          worker          worker          worker          worker          worker
|                |                |                |                |                |                |                |
|                |                |                |                |                |                |                |
|                |                |                |                |                |                |                |
S3             worker          worker          worker          worker          worker          worker          worker
|                |                |                |                |                |                |                |
|                |                |                |                |                |                |                |
|                |                |                |                |                |                |                |
```

Q: Are the red barriers (one from stage S2 and the other from S3) the “same”?
**Dissemination Barrier**

S1

```
worker worker worker worker worker worker
```

Dissemination barrier

(Last stage not shown)

S2

```
worker worker worker worker worker worker
```

Q: Using a 2-process barrier, how do processes “communicate”
Dissemination Barrier

- Set arrival flag of worker “on the right”
- Wait on own flag
- Clear own flag
Dissemination Barrier

Q: Why use a dissemination barrier versus a butterfly barrier?
Q: What is/are the difference(s) between a butterfly and dissemination barrier?
Q: Why use a dissemination barrier versus a butterfly barrier?
Q: What is/are the difference(s) between a butterfly and dissemination barrier?
Q: Where/how are barriers used?
Several processes execute the “same” code and work on different parts of shared data.

\[ a = \begin{array}{cccccc}
    1 & 2 & 3 & 4 & 5 & 6 \\
\end{array} \]

\[ b = \begin{array}{cccccc}
    1 & 3 & 6 & 10 & 15 & 21 \\
\end{array} \]

Q: What is the relationship between arrays \( a \) and \( b \)?
Data Parallel Algorithms

Several processes execute the “same” code and work on different parts of shared data.

\[
\begin{align*}
a &= \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array} \\
\text{sum} &= \begin{array}{cccccc}
1 & 3 & 6 & 10 & 15 & 21 \\
\end{array}
\end{align*}
\]

Q: What is a sequential algorithm for computing the partial sum?

In class Exercise 1
Data Parallel Algorithms

Several processes execute the “same” code and work on different parts of shared data.

\[
\begin{align*}
\text{a} & = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \\
\text{sum} & = \begin{bmatrix} 1 & 3 & 6 & 10 & 15 & 21 \end{bmatrix}
\end{align*}
\]

Q: What is a sequential algorithm for computing the partial sum?

```java
sum[0] = a[0];
for (int i=1; i<n; i++) {
    sum[i] = sum[i-1] + a[i];
}
```

Q: How many computation steps (rounds) were required to calculate the partial sums when \( n=6 \)?

Q: Can we do better?
Data Parallel Algorithms

If we wanted to find the **sum** of all elements of \( a \), we could ...

\[
a = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6
\end{bmatrix}
\]
Data Parallel Algorithms

If we wanted to find the sum of all elements of \( a \), we could ...

\[ a = \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
3 & 7 & 11 & & & \\
\end{array} \]

Q: What’s next?
If we wanted to find the **sum** of all elements of \( a \), we could ...

\[
a = \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
3 & 7 & 11 & \quad & \quad & \quad \\
& 10 & \quad & \quad & \quad & \quad
\end{array}
\]
If we wanted to find the **sum** of all elements of \( a \), we could ...

\[
a = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6 \\
3 & 7 & 11 &  &  & \\
10 & & & & & \\
21 & & & & & \\
\end{bmatrix}
\]

Q: How many computation rounds were needed assuming parallelization?
Data Parallel Algorithms

If we wanted to find the **sum** of all elements of \( a \), we could ...

\[
a = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6 \\
\end{bmatrix}
\]

<table>
<thead>
<tr>
<th>Round 1</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Round 2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Round 3</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

Q: How many computation rounds were needed assuming parallelization?

Note that round 2 cannot proceed before round 1 has completed, and round 3 cannot proceed before round 2 has completed.

There is a clear need for barriers. **Q: How does this help us in calculating partial sums?**
Data Parallel Algorithms

If we wanted to find the \texttt{partial sum} of all elements of $a$, we could ...

Use barriers to “double the distance” at which elements are added.

\begin{align*}
a &= \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\end{align*}

Goal

\begin{align*}
\text{sum} &= \begin{array}{cccccc}
1 & 3 & 6 & 10 & 15 & 21 \\
\end{array}
\end{align*}
If we wanted to find the **partial sum** of all elements of \( a \), we could ... Use barriers to “double the distance” at which elements are added.

\[
a = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ \end{bmatrix}
\]

\[
\text{sum} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ \end{bmatrix}
\]

Set \( \text{sum}[i] = a[i] \)

**Goal**

\[
\text{sum} = \begin{bmatrix} 1 & 3 & 6 & 10 & 15 & 21 \\ \end{bmatrix}
\]
Data Parallel Algorithms

If we wanted to find the **partial sum** of all elements of \( a \), we could ...

Use barriers to “double the distance” at which elements are added.

\[
\begin{array}{c}
\text{a =} \\
1 & 2 & 3 & 4 & 5 & 6 \\
\text{sum =} \\
1 & 2 & 3 & 4 & 5 & 6 \\
\text{sum =} \\
1 & 3 & 5 & 7 & 9 & 11 \\
\end{array}
\]

Q: Are we “closer” to the having computed the partial sum?

<table>
<thead>
<tr>
<th>Goal</th>
<th>sum =</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
If we wanted to find the **partial sum** of all elements of \( a \), we could ...

Use barriers to “double the distance” at which elements are added.

\[
a = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\
\end{bmatrix}
\]

\[
\text{sum} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\
\end{bmatrix}
\]

\[
\text{sum} = \begin{bmatrix} 1 & 3 & 5 & 7 & 9 & 11 \\
\end{bmatrix}
\]

**Q:** A portion of the partial sum array has been correctly computed.

**Q:** What is the next step?

**Goal**

\[
\text{sum} = \begin{bmatrix} 1 & 3 & 6 & 10 & 15 & 21 \\
\end{bmatrix}
\]
If we wanted to find the **partial sum** of all elements of \( a \), we could ... Use barriers to “double the distance” at which elements are added.

\[
\begin{align*}
  a &= \begin{array}{cccccc}
    1 & 2 & 3 & 4 & 5 & 6 \\
  \end{array} \\
  \text{sum} &= \begin{array}{cccccc}
    1 & 2 & 3 & 4 & 5 & 6 \\
  \end{array} \\
  \text{sum} &= \begin{array}{cccccc}
    1 & 3 & 5 & 7 & 9 & 11 \\
  \end{array} \\
  \text{sum} &= \begin{array}{cccccc}
    1 & 3 & 6 & 10 & 14 & 18 \\
  \end{array} \\
\end{align*}
\]

**Goal**

\[
\begin{array}{cccccc}
  1 & 3 & 6 & 10 & 15 & 21 \\
\end{array}
\]
Data Parallel Algorithms

If we wanted to find the **partial sum** of all elements of \( a \), we could ...

Use barriers to “double the distance” at which elements are added.

\[
\begin{align*}
a &= \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \\
\text{sum} &= \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \\
\text{sum} &= \begin{bmatrix} 1 & 3 & 5 & 7 & 9 & 11 \end{bmatrix} \\
\text{sum} &= \begin{bmatrix} 1 & 3 & 6 & 10 & 14 & 18 \end{bmatrix}
\end{align*}
\]

Goal: \[
\text{sum} = \begin{bmatrix} 1 & 3 & 6 & 10 & 15 & 21 \end{bmatrix}
\]
If we wanted to find the **partial sum** of all elements of \( a \), we could ...

Use barriers to “double the distance” at which elements are added.

\[
\begin{align*}
a &= \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array} \\
\text{sum} &= \begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array} \\
\text{sum} &= \begin{array}{cccccc}
1 & 3 & 5 & 7 & 9 & 11 \\
\end{array} \\
\text{sum} &= \begin{array}{cccccc}
1 & 3 & 6 & 10 & 14 & 18 \\
\end{array} \\
\text{Goal} \quad \text{sum} &= \begin{array}{cccccc}
1 & 3 & 6 & 10 & 15 & 21 \\
\end{array}
\]
If we wanted to find the **partial sum** of all elements of \( a \), we could ...

Use barriers to “double the distance” at which elements are added.

<table>
<thead>
<tr>
<th>( a )</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>( 3 )</th>
<th>( 4 )</th>
<th>( 5 )</th>
<th>( 6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{sum} )</td>
<td>( 1 )</td>
<td>( 2 )</td>
<td>( 3 )</td>
<td>( 4 )</td>
<td>( 5 )</td>
<td>( 6 )</td>
</tr>
<tr>
<td>( \text{sum} )</td>
<td>( 1 )</td>
<td>( 3 )</td>
<td>( 5 )</td>
<td>( 7 )</td>
<td>( 9 )</td>
<td>( 11 )</td>
</tr>
<tr>
<td>( \text{sum} )</td>
<td>( 1 )</td>
<td>( 3 )</td>
<td>( 6 )</td>
<td>( 10 )</td>
<td>( 14 )</td>
<td>( 18 )</td>
</tr>
</tbody>
</table>

**Goal**

| \( \text{sum} \) | \( 1 \) | \( 3 \) | \( 6 \) | \( 10 \) | \( 15 \) | \( 21 \) |
If we wanted to find the **partial sum** of all elements of \( a \), we could ...

Use barriers to “double the distance” at which elements are added.

\[
a = \begin{array}{cccccc} 
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]

\[
\text{sum} = \begin{array}{cccccc} 
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]

\[
\text{sum} = \begin{array}{cccccc} 
1 & 3 & 5 & 7 & 9 & 11 \\
\end{array}
\]

\[
\text{sum} = \begin{array}{cccccc} 
1 & 3 & 6 & 10 & 14 & 18 \\
\end{array}
\]

**Q:** Are we “closer” to the having computed the partial sum?

**Goal**

\[
\text{sum} = \begin{array}{cccccc} 
1 & 3 & 6 & 10 & 15 & 21 \\
\end{array}
\]

**All of these can be done concurrently**
If we wanted to find the **partial sum** of all elements of \( a \), we could ...

Use barriers to “double the distance” at which elements are added.

\[
\begin{align*}
\text{a} &= \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \\
\text{sum} &= \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \\
\text{sum} &= \begin{bmatrix} 1 & 3 & 5 & 7 & 9 & 11 \end{bmatrix} \\
\text{sum} &= \begin{bmatrix} 1 & 3 & 6 & 10 & 14 & 18 \end{bmatrix}
\end{align*}
\]

All of these can be done concurrently

**Q:** What is the next step?

\[
\begin{align*}
\text{Goal} \quad \text{sum} &= \begin{bmatrix} 1 & 3 & 6 & 10 & 15 & 21 \end{bmatrix}
\end{align*}
\]
If we wanted to find the **partial sum** of all elements of \( a \), we could ...

Use barriers to “double the distance” at which elements are added.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{sum} )</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>( \text{sum} )</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>( \text{sum} )</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>( \text{sum} )</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>

**Goal**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>3</th>
<th>6</th>
<th>10</th>
<th>15</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{sum} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
If we wanted to find the **partial sum** of all elements of \( a \), we could ...

Use barriers to “double the distance” at which elements are added.

<table>
<thead>
<tr>
<th>( a = )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sum =</strong></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td><strong>sum =</strong></td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td><strong>sum =</strong></td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td><strong>sum =</strong></td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>

Q: Are we done?

Goal: **sum =** 1 3 6 10 15 21
Use barriers to “double the distance” at which elements are added.

<table>
<thead>
<tr>
<th>a</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Distance : 0

| sum | 1 | 3 | 5 | 7 | 9 | 11 |

Distance : 1

| sum | 1 | 3 | 6 | 10 | 14 | 18 |

Distance : 2

| sum | 1 | 3 | 6 | 10 | 15 | 21 |

Distance : 4

Q: Advantage/speedup? : After $\log_2 n$ rounds all partial sums have been computed
Data Parallel Algorithms

```c
int a[n], sum[n], old[n];
process Sum[i = 0 to n-1] {
    int d = 1;
    sum[i] = a[i];    /* initialize elements of sum */
    barrier(i);
    ## SUM:  sum[i] = (a[i-d+1] + ... + a[i])
    while (d < n) {
        old[i] = sum[i];    /* save old value */
        barrier(i);
        if ((i-d) >= 0)
            sum[i] = old[i-d] + sum[i];
        barrier(i);
        d = d+d;    /* double the distance */
    }
}
```

`barrier(i)` implements a barrier for process `i`, and it returns after all `n` processes have called `barrier`.  

Copy from “last” step

Perform calculation
Data Parallel Algorithms

```c
int a[n], sum[n], old[n];
process Sum[i = 0 to n-1] {
    int d = 1;
    sum[i] = a[i];  /* initialize elements of sum */
    barrier(i);
    \# SUM: sum[i] = (a[i-d+1] + ... + a[i])
    while (d < n) {
        old[i] = sum[i];  /* save old value */
        barrier(i);
        if ((i-d) >= 0)
            sum[i] = old[i-d] + sum[i];
        barrier(i);
        d = d+d;  /* double the distance */
    }
}
```

Most programming languages do not provide a barrier “class” much like the semaphore functionality that we’ve seen in Python and C++. Barriers are hand-made using signal and lock procedures.
Monitors

By this point we’ve talked about semaphores, locks, and barriers. Each of these allow us to impose mutex and process synchronization among multiple concurrent processes.
Monitors

By this point we’ve talked about semaphores, locks, and barriers. Each of these allow us to impose mutex and process synchronization among multiple concurrent processes.

Q: Think back to semaphores. What were some of their shortcomings?

Think back to the midterm exam semaphore question. Easy? Difficult?

On the board List
By this point we’ve talked about semaphores, locks, and barriers. Each of these allow us to impose mutex and process synchronization among multiple concurrent processes.

Q: Think back to semaphores. What were some of their shortcomings?

Semaphores are a low-level mechanism – it is easy to make errors when using them.

• You (a programmer) must be careful NOT to omit an increment or decrement somewhere in your code, or to use the wrong semaphore (in case there are multiple ones being used)
Monitors

By this point we’ve talked about semaphores, locks, and barriers. Each of these allow us to impose mutex and process synchronization among multiple concurrent processes.

Q: Think back to semaphores. What were some of their shortcomings?

Semaphores are a low-level mechanism – it is easy to make errors when using them.

• You (a programmer) must be careful NOT to omit an increment or decrement somewhere in your code, or to use the wrong semaphore (in case there are multiple ones being used)
• Semaphores are also global, thus you must examine the entire program to know how/when they are used
Monitors

By this point we’ve talked about semaphores, locks, and barriers. Each of these allow us to impose mutex and process synchronization among multiple concurrent processes.

Q: Think back to semaphores. What were some of their shortcomings?

Semaphores are a low-level mechanism – it is easy to make errors when using them.

• You (a programmer) must be careful NOT to omit an increment or decrement somewhere in your code, or to use the wrong semaphore (in case there are multiple ones being used)
• Semaphores are also global, thus you must examine the entire program to know how/when they are used
• Semaphores provide BOTH mutual exclusion and synchronization techniques, but what if we want to use these concepts independently?
Monitors

- Provide a more object-oriented style approach to mutex and synchronization
- More structure than a semaphore
- A data abstraction mechanism
- Can be implemented easily
- Access to monitor variables is only through interface
- Mutual exclusion of all monitor procedures is implicit (procedures in the same monitor cannot be executed concurrently)
- Condition synchronization is via condition variables
Monitors

- Provide a more object-oriented style approach to mutex and synchronization
- More structure than a semaphore
- A data abstraction mechanism
- Can be implemented easily
- Access to monitor variables is only through interface
- Mutual exclusion of all monitor procedures is implicit (procedures in the same monitor cannot be executed concurrently)
- Condition synchronization is via condition variables

![Monitor Diagram]

The object’s state … **not** accessible from the “outside”

The “methods” that are accessible from the outside

// condition variables

// procedures
Monitors

- **Active processes** (threads running concurrently) interact by calling procedures in the same monitor
- The monitors are referred to as **passive**

<table>
<thead>
<tr>
<th>Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>The object’s state ... <strong>not</strong> accessible from the “outside”</td>
</tr>
<tr>
<td>// condition variables</td>
</tr>
<tr>
<td>The “methods” that are accessible from the outside</td>
</tr>
<tr>
<td>// procedures</td>
</tr>
</tbody>
</table>
Monitors

• **Active processes** (threads running concurrently) interact by calling procedures in the same monitor
• The monitors are referred to as **passive**
• A monitor is used to group together the representation and implementation of a shared resource.

<table>
<thead>
<tr>
<th>Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>// condition variables</td>
</tr>
<tr>
<td>// procedures</td>
</tr>
</tbody>
</table>

The object’s state ... **not** accessible from the “outside”

The “methods” that are accessible from the outside
Monitors

- **Active processes** (threads running concurrently) interact by calling procedures in the same monitor
- The monitors are referred to as **passive**
- A monitor is used to group together the representation and implementation of a shared resource.
- In different languages, they are created in different ways

- The condition variables are most often static – why?

<table>
<thead>
<tr>
<th>Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>The object’s state ... <strong>not</strong> accessible from the “outside”</td>
</tr>
<tr>
<td>// condition variables</td>
</tr>
<tr>
<td>The “methods” that are accessible from the outside</td>
</tr>
<tr>
<td>// procedures</td>
</tr>
</tbody>
</table>
Monitors

- **Active processes** (threads running concurrently) interact by calling procedures in the same monitor.
- The monitors are referred to as **passive**.
- A monitor is used to group together the representation and implementation of a shared resource.
- In different languages, they are created in different ways.

Monitor

This is the abstract monitor “type”, a class that has functionality common to ALL monitors (such as signal).
Monitors

- **Active processes** (threads running concurrently) interact by calling procedures in the same monitor.
- The monitors are referred to as **passive**.
- A monitor is used to group together the representation and implementation of a shared resource.
- In different languages, they are created in different ways.

This specific monitor is the representation of the abstract Monitor object. It provides functionality and variables for representing and implementing access to a shared resource.
• **Active processes** (threads running concurrently) interact by calling procedures in the same monitor
• The monitors are referred to as **passive**
• A monitor is used to group together the representation and implementation of a shared resource.
• In different languages, they are created in different ways
Monitors

- **Active processes** (threads running concurrently) interact by calling procedures in the same monitor.
- The monitors are referred to as **passive**.
- A monitor is used to group together the representation and implementation of a shared resource.
- In different languages, they are created in different ways.

Multiple threads/workers may be using `mName` to coordinate sharing a shared resource.
Monitors

- **Active processes** (threads running concurrently) interact by calling procedures in the same monitor.
- The monitors are referred to as **passive**.
- A monitor is used to group together the representation and implementation of a shared resource.
- In different languages, they are created in different ways.

Because of the static designation of `con_var`, if there are two or more threads/processes using `mName`, there is only 1 copy of `con_var`. 

```plaintext
Worker 1
// code

Worker 2
// code

mName

Monitor

con_var

procedure 1
procedure 2
```
Monitors

- **Active processes** (threads running concurrently) interact by calling procedures in the same monitor.
- The monitors are referred to as **passive**.
- A monitor is used to group together the representation and implementation of a shared resource.
- In different languages, they are created in different ways.

- Only procedure names are visible to “outside” of the monitor.
- Monitors may not access variables declared outside of the monitor.
- Permanent variables are initialized before any procedures are called.
Monitor mName{
    // declare permanent values
    // initialization statements
    // procedures
}

To alter the internal state: call mName.opName(argument(s))
Monitors

monitor mName{

    //variables

    procedure1()
    {
        // statement
    }

    procedure2()
    {
        // statement
    }
}


Monitors

- A monitor procedure is called by an external process.
- A procedure is active if some process is executing a statement in the monitor.

```plaintext
monitor mName{
    //variables
    procedure1(){
        // statement
    }
    procedure2(){
        // statement
    }
}
```
Monitors

- A monitor procedure is called by an external process.
- A procedure is active if some process is executing a statement in the monitor.

At time $t=n$, assume Process 1 begins executing `procedure1`. No other processes are accessing the monitor `mName`.

```plaintext
monitor mName{
    //variables
    procedure1()
    // statement
    }
    procedure2()
    // statement
    }
```
Monitors

- A monitor procedure is called by an external process
- A procedure is active if some process is executing a statement in the monitor
- At most one instance of any one of a monitor’s procedure may be active at the same time

Q: Is this allowed? Two processes, each invoking a separate procedure in the same monitor?

At time \( t=n \)

- Process 1

At time \( t=n+1 \)

- Process 2
Monitors

- A monitor procedure is called by an external process
- A procedure is active if some process is executing a statement in the monitor
- At most one instance of any one of a monitor’s procedures may be active at the same time

This is not allowed because two procedures of a single monitor are active
This does not mean that multiple processes cannot TRY to invoke the same or multiple procedures concurrently.

Q: How is this same/different than if a semaphore were used?
This does not mean that multiple processes cannot try to invoke the same or multiple procedures concurrently.

Q: How is this same/different than if a semaphore were used?

Q: What mechanism(s) does a monitor possess that permits it to “stall” a process until it is its turn?
A condition variable is used to delay a process that cannot safely continue until the monitor’s state satisfies some Boolean condition.

```plaintext
monitor mName{
    cond cv;
    procedure1(){
        // statement
    }
    procedure2(){
        // statement
    }
}
```
Monitors

• A **condition variable** is used to delay a process that cannot safely continue until the monitor’s state satisfies some Boolean condition.

• **cond** is a data type – often usually an array of condition variables, which specify a queue of delayed processes

```
monitor mName{
    cond cv;

    procedure1(){
        // statement
    }
    procedure2(){
        // statement
    }
}
```
Monitors

- A **condition variable** is used to delay a process that cannot safely continue until the monitor’s state satisfies some Boolean condition.

- **cond** is a data type—often usually an array of condition variables, which specify a queue of delayed processes.

Initially, \( cv \) is empty (because the monitor is not delaying any processes) and \( cv \) is NOT visible to outside processes.

Q: Can an outside processes access the value of \( cv \)?
• A process can indirectly query the state of a condition variable by calling publicly available methods, such as `empty`.

```
monitor mName{
    cond cv;

    empty(cv){
        // return true if empty
    }

    wait(cv){
        // a process blocks; its ID is placed at the rear of the cv queue
    }
}
```

Q: Are a monitor’s blocking capabilities similar to a semaphore?
Monitors

- Upon execution of `wait`, a process is self blocked and its ID is placed at the rear of `cv`
- Upon execution of `wait`, the executing process also relinquish exclusive access to the monitor

Q: What feature of a monitor has the same functionality as the unblock (increment) feature of a semaphore?
Monitors

- Processes that are blocked are awakened by means of `signal` statements.

- For example, execution of

  ```plaintext
  signal(cv)
  ```

  examines the `cv` queue, and if it is empty, `signal` has no effect, but if there is a delayed process, then `signal` awakens the process at the **front** of the queue.

  ```plaintext
  monitor mName{
    cond cv;
    empty(cv){
      // return true if empty
    }
    wait(cv){
      // a process blocks; its ID is placed at the rear of the
      // cv queue
    }
  }
  ```

  Thus `wait` and `signal` provide by default a **First In First Out (FIFO)** signaling scheme, in which processes are delayed in the order they call `wait` and are awakened in the order that `signal` is invoked.
Q: How do two (or more) processes use a monitor?
Monitors

When a process executes `signal`, it is said to be executing “inside” or “within” a monitor, and has control of the lock implicitly associated with the monitor (because of the rule that only one process can be executing a procedure in a monitor at any one time)

Q: If one process is blocked by a monitor, and another process awakens the blocked process, which of the two processes (the one that is awakened, or the one that does the awaking) executes?

(remember that only ONE process can be executing a statement within any procedure at one time)

Q: What are the possibilities?
Monitors

Signal and continue (SC): the signaler (process that called `signal`) continues, and the signaled (process that is awakened) executes at some later time.

Signal and wait (SW): the signaler waits until some later time and the signaled process executes immediately.

A monitor is following one of these disciplines, but NOT both.
Monitors

**Signal and continue (SC):** the signaler (process that called `signal`) continues, and the signaled (process that is awakened) executes at some later time.

**Signal and wait (SW):** the signaler waits until some later time and the signaled process executes immediately.

Preemptive: serving or intended to forestall itself

Nonpreemptive: calling process retains exclusive control of the monitor

The use of the priority queue `cv`, `wait`, and `signal`, and a monitor’s behavior when either SC or SW rules are used, are best described using a state transition diagram.
Q: When a process $p$ calls a monitor’s procedure, what are the scenarios? What can/may happen?
Q: When a process \( p \) calls a monitor’s procedure, what are the scenarios? What can/may happen?

If there is already an existing other process using the monitor (executing some procedure within), then \( p \) must be stalled.
Q: If no other process is immediately executing in mName, then should p be allowed to execute the procedure it is invoking?
Monitors

Q: If no other process is immediately executing in mName, then should $p$ be allowed to execute the procedure it is invoking?

No, because there may have been OTHER processes PRIOR to $p$’s attempt who wanted to execute a procedure within mName, but who were stalled. They should have a chance to go first.
Q: What mechanism can we use (and that monitors implement) that allows calling processes to have fair turns at invoking a procedure in a monitor?
Procedure makes a call to monitor

When a process calls a monitor procedure, the caller (its ID) goes into the entry queue if another process is already executing “inside” the monitor.

Q: Does a monitor need only a single queue? Why or why not?
If the monitor is free, the caller exits the entry queue and starts executing “in” the monitor.
Monitors

When an executing process returns (completes)
When an executing process returns (completes) ANOTHER process (if it is in the entry queue) attains execution rights “in” the monitor.
Monitors

When a process issues `wait`
When a process issues \texttt{wait} that process is placed into the \texttt{cv} queue.
When a process issues `wait` that process is placed into the `cv` queue and ANOTHER process that is in the entry queue attains execution rights “in” the monitor.
Monitors

Signal and continue (SC) : the signaler (process that called signal) continues, and the signaled (process that is awakened) executes at some later time.

If a process issues signal, what happens next, assuming SC?

Execution “in” monitor

Entry queue

CV queue
Monitors

Signal and continue (SC) : the signaler (process that called signal) continues, and the signaled (process that is awakened) executes at some later time.

If a process issues signal, what happens next, assuming SC?

The process continues.

Execution “in” monitor

CV queue

entry queue
If a process issues `signal`, what happens next, assuming SC?

Signal and continue (SC) : the signaler (process that called `signal`) continues, and the signaled (process that is awakened) executes at some later time.

The awakened process is placed into the entry queue.

The process continues.

Execution “in” monitor
Monitors

Signal and wait (SW) : the signaler waits until some later time and the signaled process executes immediately.

If a process is executing, and it issues signal, what happens next, assuming SW?

Entry queue

CV queue

Execution “in” monitor
If a process is executing, and it issues signal, what happens next, assuming SW?

The signaler “waits” by being placed at the end of the entry queue.
The signaled process (that is, at the head of cv) executes immediately.

Signal and wait (SW): the signaler waits until some later time and the signaled process executes immediately.

If a process is executing, and it issues signal, what happens next, assuming SW?
Monitors

Procedure makes a call to monitor

entry queue

Monitor is free

 Execution "in" monitor

signal sw

wait

CV queue

signal sc
Monitors versus Semaphores

Back to the original question ...

**Q:** What are three shortcomings of semaphores?

**Does using a monitor “fix” any of these shortcomings of semaphores?**

1. You (a programmer) must be careful NOT to omit an increment or decrement somewhere in your code, or to use the wrong semaphore (in case there are multiple ones being used)
2. Semaphores are also **global**, thus you must examine the entire program to know how they are used
3. Semaphores provide BOTH mutual exclusion and synchronization techniques, but what if we want to use these concepts independently?
Up Next

Message Passing