Announcement

Homework 5

- **Question 1**: If you declare an array myArr[0:n], you can use Flip(myArr[4],myArr[8]) to swap the entries at indices 4 and 8 of myArr.

- **Question 2**: Include at least a 1 sentence explanation of why the shown algorithm ensures mutual exclusion, avoids deadlock, and ensures eventual entry. Also explain the purpose of `in` and `last`.

- **Question 4**: refer to the book’s discussion about dissemination and butterfly barriers for hints on the number of stages needed by both barriers as a function on n workers.
From last time

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

a =

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Distance : 0

sum =

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Distance : 1

sum =

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

Distance : 2

sum =

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>18</td>
</tr>
</tbody>
</table>

Distance : 4

sum =

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>

Q: Advantage/speedup? : After \( \log_2 n \) rounds all partial sums have been computed
Q: What are three shortcomings of semaphores?
From last time

Q: What are three shortcomings of semaphores?

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

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?
From last time

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

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

```java
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
}

Process 1
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 procedures may be active at the same time.

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

At time $t=n$

At time $t=n+1$
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

```java
monitor mName{
    //variables

    procedure1(){
        // statement
    }

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

At time \( t=n \)

Process 1

At time \( t=n+1 \)

Process 2
Monitors

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?
Monitors

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?

monitor mName{
    //variables

    procedure1(){
        // statement
    }

    procedure2(){
        // statement
    }
}

An attempt is made, but mName prevents concurrent execution
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

```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.

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 \( cv \)?
Monitors

- A process can indirectly query the state of a condition variable by calling publicly available methods, such as `empty`.

```c
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

```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
    }
}
```

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
  
  ```
  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.

mon
itor mN
a
me{
    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.
Monitors

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?
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?
Monitors

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 `wait`, that process is placed into the `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

If a process issues a `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 process continues.
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 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.

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

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.
Monitors

Procedure makes a call to monitor

entry queue

Monitor is free

Execution "in" monitor

exit

signal sc

signal sw

wait

CV queue
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