Experience with Processes and Monitors in Mesa

Butler W. Lampson
David D. Redell

Presented by Priyal and Tina

Outline

- Introduction
- Monitors
- Condition Variables
  - Different ways to notify
- Processes
- Discussion Questions

Introduction

- Built at Xerox PARC in late 1970s

Introduction

- Built at Xerox PARC in late 1970s
- Problem they are trying to solve: Concurrency between lightweight processes (threads today)
  - Multiple processes running in a time sliced fashion in Pilot
  - Want to support a preemptive scheduler

Introduction

- Built at Xerox PARC in late 1970s
- Problem they are trying to solve: Concurrency between lightweight processes (threads today)
  - Multiple processes running in a time sliced fashion in Pilot
  - Want to support a preemptive scheduler
- Both the OS and programs written in Mesa, natural to design language to contain concurrency support

Introduction

- Built at Xerox PARC in late 1970s
- Problem they are trying to solve: Concurrency between lightweight processes (threads today)
  - Multiple processes running in a time sliced fashion in Pilot
  - Want to support a preemptive scheduler
- Both the OS and programs written in Mesa, natural to design language to contain concurrency support
- Two approaches
  - Shared Memory
  - Message passing

Introduction

- Built at Xerox PARC in late 1970s
- Problem they are trying to solve: Concurrency between lightweight processes (threads today)
  - Multiple processes running in a time sliced fashion in Pilot
  - Want to support a preemptive scheduler
- Both the OS and programs written in Mesa, natural to design language to contain concurrency support
- Two approaches
  - **Shared Memory - why?**
  - Message passing

Monitors

- Structure that contains a **lock**, **data**, and **code**
  - Similar to a class in Java

Monitors

- Structure that contains a **lock**, **data**, and **code**
  - Similar to a class in Java
- Synchronize processes

Monitors

- Structure that contains a **lock**, **data**, and **code**
  - Similar to a class in Java
- Synchronize processes
- All of the data is private

Monitors

- Structure that contains a **lock**, **data**, and **code**
  - Similar to a class in Java
- Synchronize processes
- All of the data is private
- Three types of procedures:
  - Entry (monitor, public)
  - Internal (monitor, private)

Monitors

- Structure that contains a **lock**, **data**, and **code**
  - Similar to a class in Java
- Synchronize processes
- All of the data is private
- Three types of procedures:
  - Entry (monitor, public)
  - Internal (monitor, private)
  - External (non-monitor, public)

Condition Variables

- Queues that keep track of processes waiting for a condition to become true

Condition Variables

- Queues that keep track of processes waiting for a condition to become true
- To enter the queue, processes call **wait()**

Condition Variables

- Queues that keep track of processes waiting for a condition to become true
- To enter the queue, processes call **wait()**
- Processes leave the queue when some other process calls **notify()** on the CV

Condition Variables

- Queues that keep track of processes waiting for a condition to become true
- To enter the queue, processes call **wait()**
- Processes leave the queue when some other process calls **notify()** on the CV
- Each CV is associated with a timeout

Condition Variables

- Queues that keep track of processes waiting for a condition to become true
- To enter the queue, processes call **wait()**
- Processes leave the queue when some other process calls **notify()** on the CV
- Each CV is associated with a timeout
- Note:
  - Condition variables do ***not*** have any mechanism to verify if the condition is true/false. That is the responsibility of the programmer.

Code Snippet

```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
                DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
        END;

Free:  ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
         <put back chunk of size words & update availableStorage>;
        NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
        pNew ← Allocate[size];
        <copy contents from old block to new block>;
        Free[pOld] END;

END.
```

Code Snippet

```
StorageAllocator: MONITOR = BEGIN
    availableStorage: INTEGER:
    moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
    UNTIL availableStorage ≥ size
        DO WAIT moreAvailable ENDLOOP;
    p ← <remove chunk of size words & update availableStorage>
    END;

Free: ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
    <put back chunk of size words & update availableStorage>;
    NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
    pNew ← Allocate[size];
    <copy contents from old block to new block>;
    Free[pOld] END;

END.
```

Code Snippet

```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
                DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
        END;

Free:  ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
         <put back chunk of size words & update availableStorage>;
        NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
        pNew ← Allocate[size];
        <copy contents from old block to new block>;
        Free[pOld] END;

END.
```

Code Snippet

```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
                DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
        END;

Free:  ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
         <put back chunk of size words & update availableStorage>;
        NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
        pNew ← Allocate[size];
        <copy contents from old block to new block>;
        Free[pOld] END;

END.
```

Code Snippet

```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
                DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
        END;

Free:  ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
         <put back chunk of size words & update availableStorage>;
        NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
        pNew ← Allocate[size];
        <copy contents from old block to new block>;
        Free[pOld] END;

END.
```

Code Snippet

```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
                DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
        END;

Free:  ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
         <put back chunk of size words & update availableStorage>;
        NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
        pNew ← Allocate[size];
        <copy contents from old block to new block>;
        Free[pOld] END;

END.
```

Code Snippet

```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
                DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
        END;

Free:  ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
        <put back chunk of size words & update availableStorage>;
        NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
        pNew ← Allocate[size];
        <copy contents from old block to new block>;
        Free[pOld] END;

END.
```

Code Snippet

```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
                DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
        END;

Free:  ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
         <put back chunk of size words & update availableStorage>;
        NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
        pNew ← Allocate[size];
        <copy contents from old block to new block>;
        Free[pOld] END;

END.
```

Code Snippet - Question

```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
                DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
        END;

Free:  ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
         <put back chunk of size words & update availableStorage>;
        NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
        pNew ← Allocate[size];
        <copy contents from old block to new block>;
        Free[pOld] END;

END.
```

Code Snippet - Answer

```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
                DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
        END;

Free:  ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
         <put back chunk of size words & update availableStorage>;
        NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
        pNew ← Allocate[size];
        <copy contents from old block to new block>;
        Free[pOld] END;

END.
```

Alternatives to NOTIFY

- TIMEOUT
  - Do processes that hit TIMEOUT throw an exception?
- ABORT
  - Do processes need to listen to an ABORT?
- BROADCAST
  - Can you replace a NOTIFY with a BROADCAST?

Calls made to the Conditional Variable:

- wait()
- notify()
- abort()
- broadcast()

Alternatives to NOTIFY

- TIMEOUT
  - Condition variables associated with some timeout variable *t*
  - Does not throw exception

Alternatives to NOTIFY

- TIMEOUT
  - Condition variables associated with some timeout variable *t*
  - Does not throw exception
- ABORT
  - When process resumes, throws *Aborted* exception
  - Aborted process does not have to listen

Alternatives to NOTIFY

- TIMEOUT
  - Condition variables associated with some timeout variable *t*
  - Does not throw exception
- ABORT
  - When process resumes, throws *Aborted* exception
  - Aborted process does not have to listen
- BROADCAST
  - All processes waiting in a condition variable’s queue wakes up
  - Can always use a BROADCAST where NOTIFY is used

Bug in Code Snippet

```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
                DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
        END;

Free:  ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
         <put back chunk of size words & update availableStorage>;
        NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
        pNew ← Allocate[size];
        <copy contents from old block to new block>;
        Free[pOld] END;

END.
```

Bug in Code Snippet

```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
                DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
        END;

Free:  ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
         <put back chunk of size words & update availableStorage>;
        NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
        pNew ← Allocate[size];
        <copy contents from old block to new block>;
        Free[pOld] END;

END.
```

Processes

- A Mesa process can be thought of as a modern-day thread

Processes

- A Mesa process can be thought of as a modern-day thread
- Represented by a 10-byte descriptor called *ProcessState* and a *frame*
    - Frame can be thought of as a stack frame, which keeps track of the procedures that a given process calls

Processes

- A Mesa process can be thought of as a modern-day thread
- Represented by a 10-byte descriptor called *ProcessState* and a *frame*
  - Frame can be thought of as a stack frame, which keeps track of the procedures that a given process calls
- Created using the *Process* module
  - Fork
  - Join
  - End
  - Detach
  - Abort
  - Yield

Processes

- At a given time, a process can be in only one of the following queues:
  - Ready Queue
  - Monitor lock Queue
  - Condition Variable Queue
  - Fault Queue

Discussion Questions

- Difference between how Hoare and Mesa monitors handle waking up after a NOTIFY?
  - What can you assume about the monitor's state after waking up?
  - How does it change the use of *while* vs *if*
- What are the different ways to deadlock in Mesa?
- How can implementing monitors defeat priority queues?
- What is the exception handling mechanism in Mesa?
  - How does it interact with Monitors?
- What are the correctness/performance benefits of providing concurrency models at different layers of the system?