

9/29 CS240 - Mesa

Announcements

For next class (Thursday 10/1)

1. Read:

[Cooperative Task Management without Manual Stack Management](#) **Only §1–3**
[Why Threads Are a Bad Idea \(for most purposes\)](#)
[Why Events Are a Bad Idea \(for high-concurrency servers\)](#)

2. No reading questions

Slides available on website Schedule for the day

When saying something in class, please state your first name the first time you say something.

Paper Background

- Xerox PARC utopia in the 1970s
 - Workstations, Ethernet, laser printers, ...
 - Custom hardware, software stack, new language & OS
 - Turing Awards



5.88Mhz CPU Microcoded
100KB RAM
2.5 MB Disk

**FUMBLING
THE
FUTURE**
HOW XEROX INVENTED,
THEN IGNORED,
THE FIRST PERSONAL
COMPUTER

Douglas K. Smith and Robert C. Alexander

Paper Overview

- Academic research meets real world needs
 - Semaphores - Edsger Dijkstra 1960s $P(\text{sema})/V(\text{sema})$
 - Monitors - Per Brinch Hansen and C. A. R. Hoare 1970s
- Why monitors?
- Experience paper - Proof?

What are the races? What bad things could happen?

```
struct BoundedStackWithRace {  
    var items: array[0..N-1] of T  
    var top: int := 0           // number of elements currently in stack  
  
    procedure push(x: T)  
        if top == N do  
            return with Error("Stack is full")  
        items[top] := x  
        top := top + 1  
  
    procedure pop(): T  
        if top == 0 do  
            return with Error("Stack is empty")  
        top := top - 1  
        x := items[top]  
        return x  
}
```

Example of early Unix critical section

```
struct buf *getblk(dev_t dev) {
    struct buf *bp;

    /* enter critical section */
    spl6();          // raise interrupt priority to block disk interrupts

    while ( (bp = bfreelist.av_forw) == NULL ) {
        /* no free buffer, sleep */
        sleep(&bfreelist, PRIBIO);
        spl6();      // re-raise level after sleep
    }

    /* remove from free list */
    bfreelist.av_forw = bp->av_forw;
    bp->b_flags |= B_BUSY;

    /* exit critical section */
    spl0();          // restore interrupts

    return bp;
}
```

Why does this work?

What changes with making this a monitor?

```
monitor BoundedStack {  
    var items: array[0..N-1] of T  
    var top: int := 0           // number of elements currently in stack  
  
    procedure push(x: T)  
        if top == N do  
            return with Error("Stack is full")  
        items[top] := x  
        top := top + 1  
  
    procedure pop(): T  
        if top == 0 do  
            return with Error("Stack is empty")  
        top := top - 1  
        x := items[top]  
        return x  
}
```

What are the monitor invariants?

```
monitor BoundedStack {  
    var items: array[0..N-1] of T  
    var top: int := 0           // number of elements currently in stack  
  
    procedure push(x: T)  
        if top == N do  
            return with Error("Stack is full")  
        items[top] := x  
        top := top + 1  
  
    procedure pop(): T  
        if top == 0 do  
            return with Error("Stack is empty")  
        top := top - 1  
        x := items[top]  
        return x  
}
```

Add waiting - Why doesn't this work?

```
monitor BoundedStack {  
    var items: array[0..N-1] of T  
    var top: int := 0                // number of elements currently in stack  
  
    procedure push(x: T)  
        while top == N do           // wait until someone pops  
            sleep(1)  
        items[top] := x  
        top := top + 1  
  
    procedure pop(): T  
        while top == 0 do           // wait until someone pushes  
            sleep(1)  
        top := top - 1  
        x := items[top]  
        return x  
}
```

Add condition variables

```
monitor BoundedStack {  
    var items: array[0..N-1] of T  
    var top: int := 0           // number of elements currently in stack  
    condition notFull, notEmpty  
  
    procedure push(x: T)  
        while top == N do  
            wait(notFull)       // wait until someone pops  
        items[top] := x  
        top := top + 1  
        signal(notEmpty)  
  
    procedure pop(): T  
        while top == 0 do  
            wait(notEmpty)      // wait until someone pushes  
        top := top - 1  
        x := items[top]  
        signal(notFull)  
        return x  
}
```

Should we make signal() conditional?

Hoare monitors

```
monitor BoundedStack {  
  var items: array[0..N-1] of T  
  var top: int := 0           // number of elements currently in stack  
  condition notFull, notEmpty  
  
  procedure push(x: T)  
    if top == N do  
      wait(notFull)           // wait until someone pops  
    items[top] := x  
    top := top + 1  
    signal(notEmpty)  
  
  procedure pop(): T  
    if top == 0 do  
      wait(notEmpty)          // wait until someone pushes  
    top := top - 1  
    x := items[top]  
    signal(notFull)  
    return x  
}
```

LockSets and monitors

If you ran the Eraser lockset algorithm on a program written in Mesa, would it be possible for a lockSet to ever have more than one lock in it?

Would Eraser be able to detect a race condition in a Mesa program?

Nested Monitors

- What is the problem with nested monitors? What choices are there?
- Interaction with software engineering
- Deadlock

Monitor routine types

- Entry
- Internal
- External

Describe each type.

Would it make sense to call from one type to another?

Exceptions

What do these keyword do:

- RETURN WITH ERROR[(arguments)]
- UNWIND

What happens for you forget an UNWIND handler on an entry routine?

Weird stack

Machine microcode support

Execution Speed

<i>Construct</i>	<i>Time (ticks)</i>
simple instruction	1
call + return	30
monitor call + return	50
process switch	60
WAIT	15
NOTIFY, no one waiting	4
NOTIFY, process waiting	9
FORK+JOIN	1,100

ProcessState queues

Ready queue

Monitor lock queue

Condition variable queue

Fault queue

Naked Notify

Purpose

Monitor records

Priorities

Reading questions

1. Explain from the [Mesa](#) paper: "[...] while any procedure suitable for forking can also be called sequentially, the converse is not generally true."
2. Consider the memory allocation code in the Mesa paper.
 1. The paper states this code has a bug. What is it and what is the fix?
 2. Intuitively, how would you change this code to work with Hoare wakeup semantics?
 3. What happens if we make *Expand* an *Entry* routine?
 4. What happens if we make the `WAIT` call just put the current thread at the end of the run queue?
 5. Give the main monitor invariant for this code.