

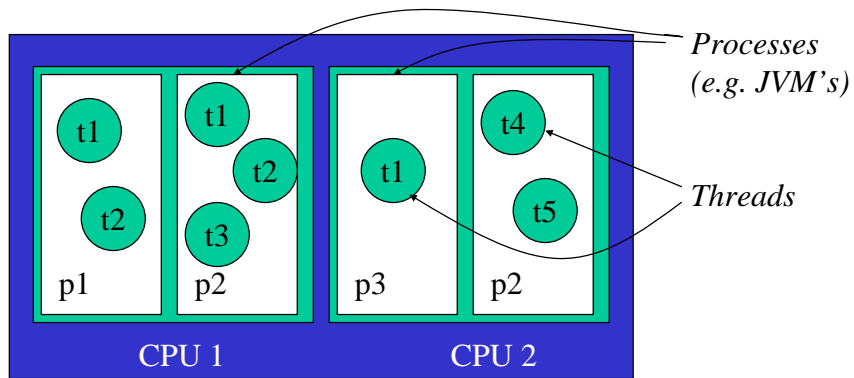
CMSC433, Fall 2002
Programming Language Technology and Paradigms
Threads and Synchronization

Michael Hicks

Overview

- What are threads?
- Thread scheduling, data races, and synchronization
- Thread mechanisms in Java

Computation Abstractions

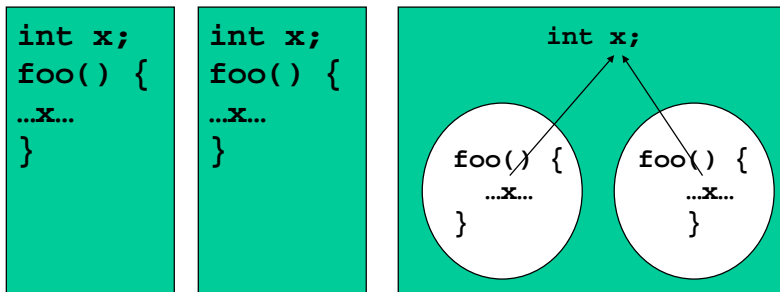


A computer

CMCS 433, Fall 2002 - Michael Hicks

3

Processes vs. Threads



Processes do not share data

Threads share data within a process

CMCS 433, Fall 2002 - Michael Hicks

4

So, what is a thread?

- Conceptually: it is a parallel computation occurring within a process
- Implementation view: it's a program counter and a stack. The heap and static area are shared among all threads
- All programs have at least one thread

CMCS 433, Fall 2002 - Michael Hicks

5

Why multiple threads?

- Performance:
 - Parallelism on multiprocessors
 - Concurrency of computation and I/O
- Can easily express some programming paradigms
 - Event processing
 - Simulations
- Keep computations separate, as in an OS
 - Java OS

CMCS 433, Fall 2002 - Michael Hicks

6

Programming Threads

- Most languages have threads
 - C, C++, Objective Caml, Java, SmallTalk ...
- The thread API differs with each, but most have the basic features we will now present

CMCS 433, Fall 2002 - Michael Hicks

7

Thread Applications

- Web browsers
 - one thread for I/O
 - one thread for each file being downloaded
 - one thread to render web page
- Graphical User Interfaces (GUIs)
 - Have one thread waiting for each important event, like key press, button press, etc.

CMCS 433, Fall 2002 - Michael Hicks

8

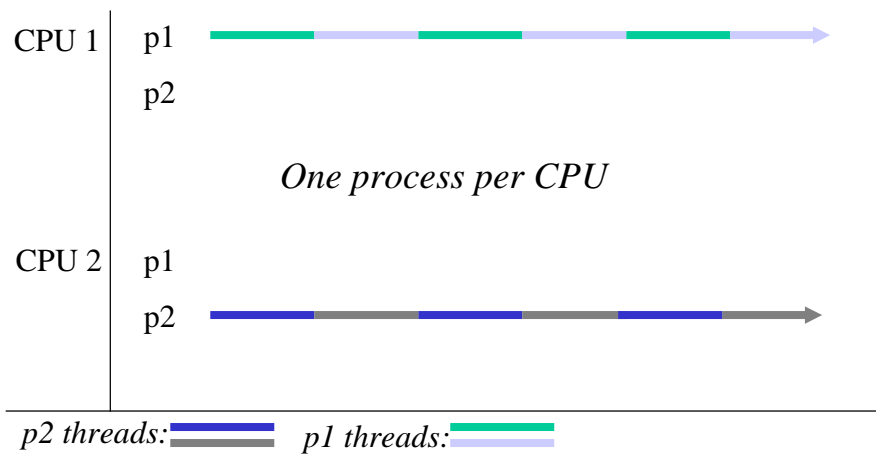
Thread Scheduling

- OS schedules a single-threaded process on a single processor
- Multithreaded process scheduling:
 - One thread per processor
 - Effectively splits a process across CPU's
 - Exploits hardware-level concurrency
 - Many threads per processor
 - Need to share CPU in slices of time

CMCS 433, Fall 2002 - Michael Hicks

9

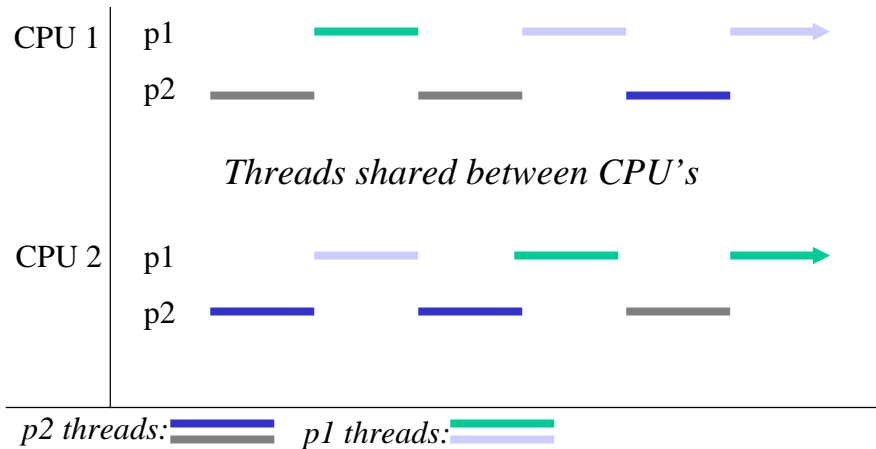
Scheduling Example (1)



CMCS 433, Fall 2002 - Michael Hicks

10

Scheduling Example (2)



CMCS 433, Fall 2002 - Michael Hicks

11

Scheduling Consequences

- Concurrency
 - Different threads from the same application can be running *at the same time* on different processors
- Interleaving
 - Threads can be “pre-empted” *at any time* in order to schedule other threads

CMCS 433, Fall 2002 - Michael Hicks

12

Data Races

- Data shared between threads can become corrupted due to “inopportune” scheduling of the sharing threads. These are called “data races.”
- Therefore need to selectively control the scheduler to avoid such data accesses. This is usually done via *synchronization*.

CMCS 433, Fall 2002 - Michael Hicks

13

Data Race Example

```
int cnt = 0;                               Shared state  cnt = 0
void thread1() {
    int y = cnt;
    cnt = y + 1;
}
void thread2() {
    int y = cnt;
    cnt = y + 1;
}
```

Start: both threads ready to run. Each will increment the global count.

CMCS 433, Fall 2002 - Michael Hicks

14

Data Race Example

```
int cnt = 0;           Shared state  cnt = 0
void thread1() {
  int y = cnt;        y = 0
  cnt = y + 1;
}
void thread2() {
  int y = cnt;
  cnt = y + 1;
}
```

■

Thread1 executes, grabbing the global counter value into y.

CMCS 433, Fall 2002 - Michael Hicks

15

Data Race Example

```
int cnt = 0;           Shared state  cnt = 0
void thread1() {
  int y = cnt;        y = 0
  cnt = y + 1;
}
void thread2() {
  int y = cnt;        y = 0
  cnt = y + 1;
}
```

■ ■

Thread1 is pre-empted. Thread2 executes, grabbing the global counter value into y.

CMCS 433, Fall 2002 - Michael Hicks

16

Data Race Example

```
int cnt = 0;
void thread1() {
    int y = cnt;
    cnt = y + 1;
}
void thread2() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state **cnt = 1**

y = 0

y = 0

■ ■ ■

Thread2 executes, storing the incremented cnt value.

CMCS 433, Fall 2002 - Michael Hicks

17

Data Race Example

```
int cnt = 0;
void thread1() {
    int y = cnt;
    cnt = y + 1;
}
void thread2() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state **cnt = 1**

y = 0

y = 0

■ ■ ■ ■

*Thread2 completes. Thread1
Executes again, storing the
old counter value (1) rather
than the new one (2)!*

CMCS 433, Fall 2002 - Michael Hicks

18

What happened?

- Thread1 was preempted after it read the counter but before it stored the new value.
- A particular way in which the execution of two threads is interleaved is called a *schedule*. We want to prevent this undesirable schedule.
- Undesirable schedules can be hard to reproduce, and so hard to debug.

CMCS 433, Fall 2002 - Michael Hicks

19

Applying synchronization

```
int cnt = 0;
lock l;
void thread1() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
void thread2() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state cnt = 0

Lock, for protecting the shared state

Acquires the lock; only succeeds if not held by another thread

Releases the lock

CMCS 433, Fall 2002 - Michael Hicks

20

Applying synchronization

```
int cnt = 0;
lock l;
void thread1() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
void thread2() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state cnt = 0



Thread1 acquires lock l

CMCS 433, Fall 2002 - Michael Hicks

21

Applying synchronization

```
int cnt = 0;
lock l;
void thread1() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
void thread2() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state cnt = 0



Thread1 reads cnt into y

CMCS 433, Fall 2002 - Michael Hicks

22

Applying synchronization

```
int cnt = 0;
lock l;
void thread1() {
  synchronized(l) {
    int y = cnt;
    cnt = y + 1;
  }
}
void thread2() {
  synchronized(l) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 0

y = 0



*Thread1 is pre-empted.
Thread2 attempts to
acquire lock 1 but fails
because it's held by
Thread1, so it blocks*

CMCS 433, Fall 2002 - Michael Hicks

23

Applying synchronization

```
int cnt = 0;
lock l;
void thread1() {
  synchronized(l) {
    int y = cnt;
    cnt = y + 1;
  }
}
void thread2() {
  synchronized(l) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 1

y = 0



*Thread1 runs, assigning
to cnt*

CMCS 433, Fall 2002 - Michael Hicks

24

Applying synchronization

```
int cnt = 0;
lock l;
void thread1() {
  synchronized(l) {
    int y = cnt;
    cnt = y + 1;
  }
}
void thread2() {
  synchronized(l) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 1



Thread1 releases the lock
and terminates

CMCS 433, Fall 2002 - Michael Hicks

25

Applying synchronization

```
int cnt = 0;
lock l;
void thread1() {
  synchronized(l) {
    int y = cnt;
    cnt = y + 1;
  }
}
void thread2() {
  synchronized(l) {
    int y = cnt;
    cnt = y + 1;
  }
}
```

Shared state cnt = 1



Thread2 now can acquire
lock l.

CMCS 433, Fall 2002 - Michael Hicks

26

Applying synchronization

```
int cnt = 0;
lock l;
void thread1() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
void thread2() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state cnt = 1



Thread2 reads cnt into y.

CMCS 433, Fall 2002 - Michael Hicks

27

Applying synchronization

```
int cnt = 0;
lock l;
void thread1() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
void thread2() {
    synchronized(l) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state cnt = 2



Thread2 assigns cnt,
then releases the lock

CMCS 433, Fall 2002 - Michael Hicks

28

Synchronization not a Panacea

- Can be expensive to acquire a lock
- Two threads can block on locks held by the other; this is called *deadlock*

```
lock l;  
lock m;  
void thread1() {  
    synchronized (l) {  
        synchronized (m) {  
            ...  
        }  
    }  
}  
  
void thread2() {  
    synchronized (m) {  
        synchronized (l) {  
            ...  
        }  
    }  
}
```

CMCS 433, Fall 2002 - Michael Hicks

29

Other Thread Operations

- **Condition variables: wait and notify**
 - Alternative synchronization mechanism
- **Yield**
 - Voluntarily give up the CPU
- **Sleep**
 - Wait for a certain length of time

CMCS 433, Fall 2002 - Michael Hicks

30

Thread Lifecycle

- While a thread executes, it goes through a number of different phases
 - **New**: created but not yet started
 - **Runnable**: either running, or able to run on a free CPU
 - **Blocked**: waiting for I/O or on a lock
 - **Sleeping**: paused for a user-specified specified interval

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.