CPS 310
Threads and Concurrency: Topics

Jeff Chase
Duke University

http://www.cs.duke.edu/~chase/cps310
Terminology and syntax

- The abstractions for concurrency control in this class are now sort-of universal at the OS/API level.

- Monitors (mutex+CV) with Mesa semantics appear in:
  - Java (e.g., on Android) and JVM languages (e.g., Scala)
  - POSIX threads or Pthreads (used on Linux and MacOS/iOS)
  - Windows, C#/.NET, and other Microsoft systems

- Terminology and APIs vary a bit.
  - mutex == lock == Java “synchronized”
  - monitor == mutex + condition variable (CV)
  - signal() == notify(), broadcast() == notifyAll()

- The slides use interchangeable terms interchangeably.
Example: the soda/HFCS machine

Delivery person (producer)

Vending machine (buffer)

Soda drinker (consumer)
Producer-consumer code

customer () {
    take a soda from machine
}

producer () {
    add one soda to machine
}
Solving producer-consumer

1. **What are the variables/shared state?**
   - Soda machine buffer
   - Number of sodas in machine (\(\leq\) MaxSodas)

2. **Locks?**
   - 1 to protect all shared state (sodaLock)

3. **Mutual exclusion?**
   - Only one thread can manipulate machine at a time

4. **Ordering constraints?**
   - Consumer must wait if machine is empty (CV hasSoda)
   - Producer must wait if machine is full (CV hasRoom)
Producer-consumer code

consumer () {
    lock
    take a soda from machine
    unlock
}

producer () {
    lock
    add one soda to machine
    unlock
}
Producer-consumer code

```plaintext
consumer () {
    lock
    wait if empty
    take a soda from machine
    notify (not full)
    unlock
}
```

```plaintext
producer () {
    lock
    wait if full
    add one soda to machine
    notify (not empty)
    unlock
}
```
Producer-consumer code

consumer () {
    lock (sodaLock)

    while (numSodas == 0) {
        wait (sodaLock, hasSoda)
    }

    take a soda from machine

    signal (hasRoom)

    unlock (sodaLock)
}

producer () {
    lock (sodaLock)

    while (numSodas == MaxSodas) {
        wait (sodaLock, hasRoom)
    }

    add one soda to machine

    signal (hasSoda)

    unlock (sodaLock)
}
Producer-consumer code

```java
synchronized consumer () {
    while (numSodas == 0) {
        o.wait();
    }
    take a soda from machine
    notify();
}

synchronized producer () {
    while (numSodas == maxSodas) {
        wait();
    }
    put a soda from machine
    notify();
}
```
producer-consumer code

```java
consumer () {
    synchronized(o) {
        while (numSodas == 0) {
            o.wait();
        }
        take a soda
    o.notify();
    }
}
```
producer-consumer code

producer () {
  lock (sodaLock)
  while (numSodas == MaxSodas) {
    wait (sodaLock, hasRoom)
  }
  fill machine with soda
  broadcast (hasSoda)
  unlock (sodaLock)
}

c consumer () {
  lock (sodaLock)
  while (numSodas == 0) {
    wait (sodaLock, hasSoda)
  }
  take a soda from machine
  signal (hasRoom)
  unlock (sodaLock)
}

The signal should be a broadcast if the producer can produce more than one resource, and there are multiple consumers.
Variations: looping producer

- **Producer**
  - Infinite loop ok?
  - Why/why not?
- Release lock in wait call

```c
producer () {
    lock (sodaLock)
    while (1) {
        while(numSodas==MaxSodas){
            wait (sodaLock, hasRoom)
        }
        add soda to machine
        signal (hasSoda)
    }
    unlock (sodaLock)
}
```
Variations: resting producer

- Producer
  - Sleep ok?
  - Why/why not?
- Shouldn’t hold locks during a slow operation

```java
producer () {
    lock (sodaLock)
    while (1) {
        sleep (1 hour)
        while(numSodas==MaxSodas){
            wait (sodaLock, hasRoom)
        }
        add soda to machine
        signal (hasSoda)
    }
    unlock (sodaLock)
}
```
Variations: one CV?

```
consumer () {
    lock (sodaLock)

    while (numSodas == 0) {
        wait (sodaLock, hasRorS)
    }

    take a soda from machine
    signal (hasRorS)
    unlock (sodaLock)
}

producer () {
    lock (sodaLock)

    while (numSodas == MaxSodas) {
        wait (sodaLock, hasRorS)
    }

    add one soda to machine
    signal (hasRorS)
    unlock (sodaLock)
}
```

Two producers, two consumers: who consumes a signal? ProducerA and ConsumerB wait while ConsumerC signals?
Variations: one CV?

consumer () {
    lock (sodaLock)

    while (numSodas == 0) {
        wait (sodaLock, hasRorS)
    }

    take a soda from machine

    signal (hasRorS)

    unlock (sodaLock)
}

producer () {
    lock (sodaLock)

    while (numSodas == MaxSodas) {
        wait (sodaLock, hasRorS)
    }

    add one soda to machine

    signal (hasRorS)

    unlock (sodaLock)
}

Is it possible to have a producer and consumer both waiting?
max=1, cA and cB wait, pC adds/signals, pD waits, cA wakes.
Variations: one CV?

consumer () {
    lock (sodaLock)

    while (numSodas == 0) {
        wait (sodaLock, hasRorS)
    }

    take a soda from machine

    signal (hasRorS)

    unlock (sodaLock)
}

producer () {
    lock (sodaLock)

    while (numSodas == MaxSodas) {
        wait (sodaLock, hasRorS)
    }

    add one soda to machine

    signal (hasRorS)

    unlock (sodaLock)
}

How can we make the one CV solution work?
Variations: one CV?

consumer () {
  lock (sodaLock)

  while (numSodas == 0) {
    wait (sodaLock, hasRorS)
  }

  take a soda from machine

  broadcast (hasRorS)

  unlock (sodaLock)
}

producer () {
  lock (sodaLock)

  while (numSodas == MaxSodas) {
    wait (sodaLock, hasRorS)
  }

  add one soda to machine

  broadcast (hasRorS)

  unlock (sodaLock)
}

Use broadcast instead of signal: safe but slow.
Broadcast vs signal

- Can I always use broadcast instead of signal?
  - Yes, assuming threads recheck condition
  - And they should: “loop before you leap”!
  - Mesa semantics requires it anyway: another thread could get to the lock before wait returns.

- Why might I use signal instead?
  - Efficiency (spurious wakeups)
  - May wakeup threads for no good reason
  - “Signal is just a performance hint”.
Locking a critical section

The threads may run the critical section in either order, but the schedule can never enter the grey region where both threads execute the section at the same time.

Holding a shared mutex prevents competing threads from entering a critical section protected by the shared mutex (monitor). At most one thread runs in the critical section at a time.
Locking a critical section

Holding a shared mutex prevents competing threads from entering a critical section. If the critical section code acquires the mutex, then its execution is serialized: only one thread runs it at a time.
How about this?

\[ x = x + 1; \quad \text{A} \]

\[ \text{mx->Acquire();} \]
\[ x = x + 1; \]
\[ \text{mx->Release();} \]
How about this?

The locking discipline is not followed: purple fails to acquire the lock mx.

Or rather: purple accesses the variable $x$ through another program section A that is mutually critical with B, but does not acquire the mutex.

A locking scheme is a convention that the entire program must follow.
How about this?

lock->Acquire();
x = x + 1;       A
lock->Release();

mx->Acquire();
x = x + 1;       B
mx->Release();
How about this?

This guy is not acquiring the right lock.

Or whatever. They’re not using the same lock, and that’s what matters.

A locking scheme is a convention that the entire program must follow.
Ucontext library routines

• The system can use `ucontext` routines to:
  – “Freeze” at a point in time of the execution
  – Restart execution from a frozen moment in time
  – Execution continues where it left off…if the memory state is right.

• The system can implement multiple independent threads of execution within the same address space.
  – Create a context for a new thread with `makecontext`: when switched in it will call a specified procedure with specified arg.
  – Modify saved contexts at will.
  – Context switch with `swapcontext`: transfer a core from one thread to another
#include <ucontext.h>

int count = 0;

ucontext_t context;

int main()
{
    int i = 0;
    getcontext(&context);

    count += 1;
    i += 1;
    sleep(2);
    printf("...", count, i);

    setcontext(&context);
}

ucontext
Standard C library routines to:

Save current register context to a block of memory (getcontext from core)

Load/restore current register context from a block of memory (setcontext)

Also: makecontext, swapcontext

Details of the saved context (ucontext_t structure) are machine-dependent.
#include <ucontext.h>

int count = 0;
ucontext_t context;

int main()
{
    int i = 0;
    getcontext(&context);
    count += 1;
    i += 1;
    sleep(1);
    printf("...", count, i);
    setcontext(&context);
}

**Save** CPU core context to memory

Loading the saved context transfers control to this block of code. (**Why?**)

**What about the stack?**

**Load** core context from memory
#include <ucontext.h>

int count = 0;

ucontext_t context;

int main()
{
    int i = 0;
    getcontext(&context);
    count += 1;
    i += 1;
    sleep(1);
    printf("...", count, i);
    setcontext(&context);
}
Reading behind the C

Disassembled code:

```assembly
movl 0x0000017a(%rip),%ecx
addl $0x00000001,%ecx
movl %ecx,0x0000016e(%rip)
movl 0xfc(%rbp),%ecx
addl $0x00000001,%ecx
movl %ecx,0xfc(%rbp)
```

On MacOS:

chase$ man otool
chase$ otool --vt context0
...

On this machine, with this cc:

Static global `_count` is addressed relative to the location of the code itself, as given by the PC register `[%rip is instruction pointer register]`

Local variable `i` is addressed as an offset from stack frame. [%rbp is stack frame base pointer]

`%rip` and `%rbp` are set “right”, then these references “work”.

```c
count += 1;
i += 1;
```
#include <ucontext.h>

int count = 0;

ucontext_t context;

int main()
{
    int i = 0;
    getcontext(&context);

    count += 1;
    i += 1;
    sleep(1);
    printf("...", count, i);

    setcontext(&context);
}

chase$ cc -O2 -o context0 context0.c
<
  warnings:
    ucontext deprecated on MacOS
>
chase$ ./context0
  1 1
  2 1
  3 1
  4 1
  5 1
  6 1
  7 1
...

What happened?
#include <ucontext.h>

int count = 0;
ucontext_t context;

int main()
{
    int i = 0;
    getcontext(&context);
    count += 1;
    i += 1;
    sleep(1);
    printf("...", count, i);
    setcontext(&context);
}