ECE 570/670

David Irwin

Lecture 5

Administrative Details

- Everyone should have VM account and should have tested using the autograder
  - A few people do not yet have accounts
  - Some people have already submitted
  - Assignment 0 due next week
- Reminders
  - 1. Must submit from campus network (eduroam, UMASS, etc.)
  - 2. Test soon to figure out if there are issues
    - If issues, come by office hours

Administrative Details

- 670 Project proposals due 3/24
  - Think if you want to work in a group
    - Fill out form with group members if you have decided
    - Will send reminder email to mailing list
  - I am open to any group size from 1 person to 4 people
  - Will send out calendar link for meeting signup next week

Administrative Details

- Assignment 0 review
  - Compile with `g++ inverter.cc -o inverter`
  - Run with `./inverter input.txt`
  - Submit inverter.cc file
  - Note: C++ files can be .cpp, .cc, .cxx, .whatever
    - **Autograder only recognizes .cc files!**
    - No header files (everything in one file)
  - Due date on calendar

- If you are having problems with P0, think about why:
  - C++ language/Unix environment issues?
- **Fix any non-language issues before Assignment 1**
  - (Language issues will get fixed with practice)

Administrative Details

- Many questions on assignment 0
  - Need to submit as *.cc file
  - Uses g++ compiler
  - Need to ensure there are no unseen spaces/newlines at end
  - Need to make sure it runs correctly on Ubuntu VM
  - Make sure you upgrade to latest version of VirtualBox (if your VM doesn't work) or UTM
  - Need to make sure you understand what "correct" is
  - Need to make sure EOL characters are UNIX not Windows
  - **The autograder is precise…so you need to be precise!**

Administrative Details

- Will post Assignment 1 next class
  - Focuses on threads
  - First part – implement threaded program (hard)
  - Second part – implement thread library (really hard)

- Will go over Assignment 1 in detail on Thursday

Administrative Details

- Post general assignment questions on Piazza
  - Much quicker response as myself (and TA?), and other students are monitoring it
  - Everyone can benefit from the answer

- Only email me (or the TAs?) if you have a question that is highly specific or personal to you
  - Or for scheduling a meeting time outside office hours

Today's Outline

- Discuss UNIX paper
- More synchronization!

- **Paper for Wednesday posted:**

"Virtualization Considered Harmful: OS Design Directions for Well-Conditioned Services"

"The UNIX Time-sharing System"

- Both previously worked on Multics
- Thompson wrote the B programming language
- Ritchie followed up with the C programming language
- Ken created Belle, world champion chess computer
- Ken was at Google
  - Co-designer of Go language
- Dennis was head of Lucent Research Dept until 2007 (when he retired)
- Dennis said that creating the C language "looked like a good thing to do"

Ken Thompson and Dennis Ritchie

1983 Turing award recipients

1999 Natl Medal of Technology

"The UNIX Time-sharing System"

- Most important part?
  - File System
  - Ordinary files, directories, special files (I/O)
  - File permissions, removable file systems
- Also built in…
  - Processes, e.g., forking
  - Pipes, e.g., passing output of one process to input of another
  - Process synchronization, e.g., wait
- Shell implementation
  - Standard input, standard output, standard error
  - I/O redirection
  - Multi-tasking (background processes)

Does it work?

```
leave noteJeannie
while (noteDavid){
  do nothing
}
if (noMilk){
  buy milk;
}
remove noteJeannie
```

```
leave noteDavid
if (no noteJeannie){
  if (noMilk){
    buy milk;
  }
}
remove noteDavid
```

Yes! It does work! Can you prove it?

Downside of solution

- Complexity
  - Hard to convince yourself it works
- Asymmetric
  - Jeannie and David run different code
- Not clear if this scales to > 2 people
- Jeannie consumes CPU while waiting
  - `while (noteDavid) { do nothing }`
  - **Busy-waiting**
- **Note:** only needed atomic load/store

Raising the level of abstraction

- Mutual exclusion with atomic load/store
  - Painful to program
  - Wastes resources (busy waiting)
    - Need more HW support to fix this
    - Will be covered later
- OS can provide higher level abstractions

Raising the level of abstraction

- **Locks**
  - Also called **mutexes**
  - Provide mutual exclusion
  - Prevent threads from entering a critical section
- Lock operations
  - Lock (aka `Lock::acquire`)
  - Unlock (aka `Lock::release`)

Lock operations

- Lock: wait until lock is free, then acquire it

```
do {
  if (lock is free) {
    acquire lock
    break
  }
} while (1)
```

Must be atomic with respect to other threads calling this code

  - This is a busy-waiting implementation
    - We'll improve on this later
- Unlock: atomic `release lock`

Too much milk, solution 2

```
if (noMilk) {
   if (noNote){
      leave note;
      buy milk;
      remove note;
   }
}
```

Block is not atomic.

Must atomically:

- check if lock is free
- grab it

Why doesn't the note work as a lock?

Elements of locking

1. The lock is initially free
2. Threads acquire lock before an action
3. Threads release lock when action completes
4. Lock() must wait if someone else has lock

- **Key idea:** All synchronization involves waiting
- Threads are either **running** or **blocked**

Too much milk with locks?

```
lock ()
if (noMilk) {
  buy milk
}
unlock ()
```

```
lock ()
if (noMilk) {
  buy milk
}
unlock ()
```

- Problem?
  - One person waits for lock while other buys milk
  - Works, but inefficient

Too much milk "w/o waiting"?

```
lock ()
if (noNote && noMilk){
   leave note "at store"
   unlock ()
   buy milk
   lock ()
   remove note
   unlock ()
} else {
   unlock ()
}
```

Not holding lock

```
lock ()
if (noNote && noMilk){
   leave note "at store"
   unlock ()
   buy milk
   lock ()
   remove note
   unlock ()
} else {
   unlock ()
}
```

Only hold lock while handling shared resource.

What about this?

```
lock ()
if (noMilk && noNote){
   leave note "at store"
   unlock ()
   buy milk
   stock fridge
   remove note
} else {
   unlock ()
}
```

```
lock ()
if (noMilk && noNote){
   leave note "at store"
   unlock ()
   buy milk
   stock fridge
   remove note
} else {
   unlock ()
}
```

Too much milk!!

What about this?

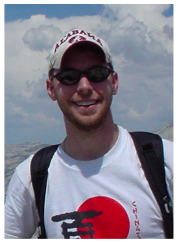
```
lock ()
if (noNote && noMilk){
  leave note "at store"
  unlock ()
  buy milk
  stock fridge
  remove note
} else {
  unlock ()
}
```

```
lock ()
if (noNote && noMilk){
  leave note "at store"
  unlock ()
  buy milk
  stock fridge
  remove note
} else {
  unlock ()
}
```

Assuming that I stock the fridge completely before removing the note, and assuming short-circuited boolean checks, technically this would work in Java, but not with all other languages or compilers. Still not safe!

Too Much Milk
Solution 1

```
leave noteJeannie
while (noteDavid){
  do nothing
}
if (noMilk){
  buy milk;
}
remove noteJeannie
```



```
leave noteDavid
if (no noteJeannie){
  if (noMilk){
    buy milk;
  }
}
remove noteDavid
```

Too much milk "w/o waiting" Solution 2

```
lock ()
if (noNote && noMilk){
  leave note "at store"
  unlock ()
  buy milk
  stock fridge
  lock ()
  remove note
  unlock ()
} else {
  unlock ()
}
```

Not holding lock

```
lock ()
if (noNote && noMilk){
  leave note "at store"
  unlock ()
  buy milk
  stock fridge
  lock ()
  remove note
  unlock ()
} else {
  unlock ()
}
```

What about Java?

```
synchronized (obj){
  if (noMilk) {
    buy milk
  }
}
```

```
synchronized (obj){
  if (noMilk) {
    buy milk
  }
}
```

- Every object is a lock
- Use **synchronized** key word
  - Lock : “{“
  - Unlock: “}”

Synchronizing methods

```
public class CubbyHole {
  private int contents;

  public int get() {
    return contents;
  }

  public synchronized void put(int value) {
    contents = value;
  }
}
```

- What does this mean? What is the lock?
  - "this" is the lock

Synchronizing methods

```
public class CubbyHole {
  private int contents;

  public int get() {
    return contents;
  }

  public void put(int value) {
    synchronized (this) {
      contents = value;
    }
  }
}
```

- Equivalent to "synchronized (this)" block

Another Example: Thread-safe queue

```
enqueue () {
  lock (qLock);
  // ptr is private
  // head is shared
  node *ptr;
  // find queue tail
  for (ptr=head;
       ptr->next!=NULL;
       ptr=ptr->next){}

  ptr->next=new_element;
  new_element->next=NULL;
  unlock(qLock);
}
```

```
dequeue () {
  lock (qLock);
  element=NULL;
  // if queue non-empty
  if (head!=NULL) {
    // remove head
    element=head;
    head=head->next;
  }
  unlock (qLock);
  return element;
}
```

Thread-safe queue

- Can enqueue unlock anywhere?
  - No
- Must leave shared data…
  - …in a consistent/sane state
- Data **invariant**
  - "consistent/sane state"
  - "always" true
  - *Formal definition:* a feature that remains unchanged when a particular transformation is applied to it

```
enqueue () {
   lock (qLock);
   node *ptr;
   for (ptr=head;
        ptr->next!=NULL;
        ptr=ptr->next){}

   ptr->next=new_element;
   unlock(qLock);
   // safe?
   new_element->next=NULL;
}
```

Invariants

- What are the queue invariants?
  - Each node appears once (from head to null)
  - Queue ends in a null
  - Enqueue results in prior list + new element
  - Dequeue removes exactly one element
- Can invariants ever be false?
  - Must be
  - Otherwise you could never change states

More on invariants

- So when are invariants broken?
  - Can only be broken **while lock is held**
  - And only by the thread holding the lock
- Really a "public" invariant (that is "always" true)
  - What state is the data in when the lock is free
  - Like having a room tidy before guests arrive
- Hold a lock whenever manipulating shared data
  - Necessary but not always sufficient
- Instead, must hold a lock whenever "breaking" invariants

More on invariants

- What about reading shared data?
  - Still must hold lock
  - Else another thread could break invariant
  - (Thread A prints queue as Thread B enqueues)