Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Concurrency

Introduction to concurrency

Josh Bloch Charlie Garrod

Administrivia

- Homework 5 team sign-up deadline tomorrow at 5 p.m. EDT
- Midterm exam tomorrow/Thursday (4/7-8)
  - Practice exam is out
  - Exam review session tonight 7-9 p.m.
  - Exam released Wednesday night, due Thursday 11:59 p.m.

Today's lecture: concurrency motivation and primitives

- Why concurrency?
  - Motivation, goals, problems, …
- Concurrency primitives in Java
- Coming soon (not today):
  - Higher-level abstractions for concurrency
  - Program structure for concurrency
  - Frameworks for concurrent computation

Moore's Law (1965) – number of transistors on a chip doubles every two years

Transistor count

50,000,000,000
10,000,000,000
5,000,000,000
1,000,000,000
500,000,000
100,000,000
50,000,000
10,000,000
5,000,000
1,000,000
500,000
100,000
50,000
10,000
5,000
1,000

1970 1972 1974 1976 1978 1980 1982 1984 1986 1988 1990 1992 1994 1996 1998 2000 2002 2004 2006 2008 2010 2012 2014 2016 2018

72-core Xeon Phi
Centriq 2400
GC2 IPU
SPARC M7
32-core AMD Epyc
IBM z13 Storage Controller
Apple A12X Bionic
18-core Xeon Haswell-E5
Tegra Xavier SoC
Qualcomm Snapdragon 8cx/SCX8180
Xbox One main SoC
HiSilicon Kirin 980 + Apple A12 Bionic
61-core Xeon Phi
HiSilicon Kirin 710
12-core POWER8
8-core Xeon Nehalem-EX
10-core Core i7 Broadwell-E
Six-core Xeon 7400
Qualcomm Snapdragon 835
Dual-core Itanium 2
Dual-core + GPU Iris Core i7 Broadwell-U
Quad-core + GPU GT2 Core i7 Skylake K
Pentium D Presler
POWER6
Quad-core + GPU Core i7 Haswell
Apple A7 (dual-core ARM64 "mobile SoC")
Itanium 2 with 9 MB cache
Core i7 (Quad)
AMD K10 quad-core 2M L3
Itanium 2 Madison 6M
Core 2 Duo Wolfdale
Pentium D Smithfield
Core 2 Duo Conroe
Itanium 2 McKinley
Cell
Core 2 Duo Wolfdale 3M
Pentium 4 Prescott-2M
Core 2 Duo Allendale
Pentium 4 Cedar Mill
AMD K8
Pentium 4 Prescott
Pentium 4 Northwood
Barton
Pentium 4 Willamette
Atom
Pentium III Tualatin
Pentium II Mobile Dixon
AMD K7
Pentium III Coppermine
ARM Cortex-A9
AMD K6-III
AMD K6
Pentium III Katmai
Pentium II Deschutes
Pentium Pro
Pentium II Klamath
Pentium
AMD K5
SA-110
Intel 80486
R4000
TI Explorer's 32-bit Lisp machine chip
ARM700
Intel 80386
Intel i960
ARM 3
Motorola 68020
DEC WRL MultiTitan
Intel 80286
ARM 9TDMI
Motorola 68000
Intel 80186
Intel 8086
Intel 8088
ARM 2
ARM 6
WDC 65C816
ARM 1
Novix NC4016
TMS 1000
Zilog Z80
Motorola 6809
WDC 65C02
RCA 1802
Intel 8085
Intel 8008
Intel 8080
Motorola 6800
MOS Technology 6502
Intel 4004

CPU Performance and Power Consumption

- *Dennard Scaling* (1974) – each time you double transistor density:
  - Speed (frequency) goes up by about 40% (Why 40%?)…
  - While power consumption of the chip stays constant (proportional to area)
- Combined w/ Moore's law, every 4 years the number of transistors quadruples, speed doubles, and power consumption stays constant
- It was great while it lasted…
  - Started breaking down in the mid '90s and broke down completely in the mid '90s due to *leakage currents* ☹
  - More power is required at higher frequency, generating more heat
  - And there's a limit to how much heat a chip can tolerate

One option: fix the symptom

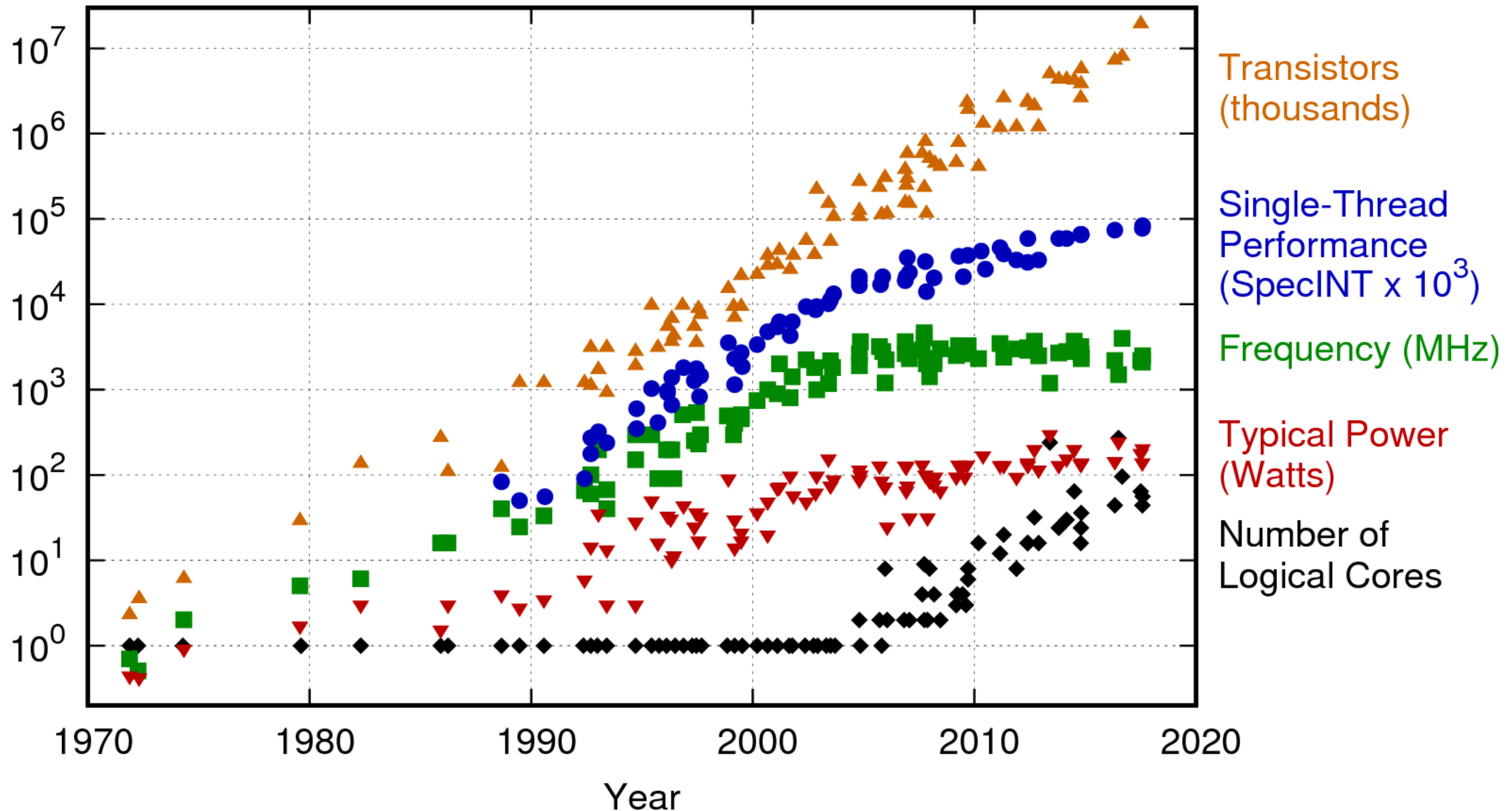
- Dissipate the heat

One option: fix the symptom

- Better(?): Dissipate the heat with liquid nitrogen

by **Paul Lilly** — Monday, December 16, 2019, 10:14 AM EDT

AMD Ryzen 9 3900X 12-Core Beast Chip Hits 5.6GHz To Claim World Record



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Concurrency then and now

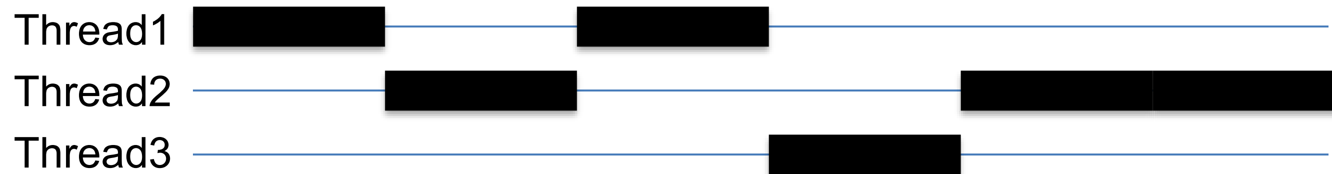
- In the past, multi-threading just a convenient abstraction
  - GUI design: event dispatch thread
  - Server design: isolate each client's work
  - Workflow design: isolate producers and consumers
- Now: required for scalability and performance

We are all concurrent programmers

- Java is inherently multithreaded
- To utilize modern processors, we *must* write multithreaded code
- Good news: a lot of it is written for you
  - Excellent libraries exist (e.g., `java.util.concurrent`)
- Bad news: you still must understand fundamentals
  - …to use libraries effectively
  - …to debug programs that make use of them

Aside: Concurrency vs. parallelism, visualized

- Concurrency without parallelism:



- Concurrency with parallelism:

Basic concurrency in Java

Review

- An interface representing a task

```
public interface Runnable {
    void run();
}
```

- A class to execute a task in a CPU thread

```
public class Thread {
    public Thread(Runnable task);
    public void start();
    public void join();
    …
}
```

Example: Money-grab (1/2)

```
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static void transferFrom(BankAccount source,
                             BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance   += amount;
    }

    public long balance() {
        return balance;
    }
}
```


institute for SOFTWARE RESEARCH

Example: Money-grab (2/2)

What would you expect this program to print?

```
public static void main(String[] args) throws InterruptedException {
    BankAccount bugs  = new BankAccount(100);
    BankAccount daffy = new BankAccount(100);

    Thread bugsThread = new Thread(()-> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(daffy, bugs, 100);
    });

    Thread daffyThread = new Thread(()-> {
        for (int i = 0; i < 1_000_000; i++)
            transferFrom(bugs, daffy, 100);
    });

    bugsThread.start(); daffyThread.start();
    bugsThread.join();  daffyThread.join();
    System.out.println(bugs.balance() + daffy.balance());
}
```

institute for SOFTWARE RESEARCH

What went wrong?

- Daffy & Bugs threads had a *race condition* for shared data
  - Transfers did not happen in sequence
- Reads and writes interleaved randomly
  - Random results ensued

The challenge of concurrency control

- Not enough concurrency control: *safety failure*
  - Incorrect computation
- Too much concurrency control: *liveness failure*
  - Possibly no computation at all (*deadlock* or *livelock*)

Shared mutable state requires concurrency control

- Three basic choices:
  1. Don't share: isolate state in individual threads
  2. Don't mutate: share only immutable state
  3. If you must share mutable state: *synchronize to achieve safety*

institute for SOFTWARE RESEARCH

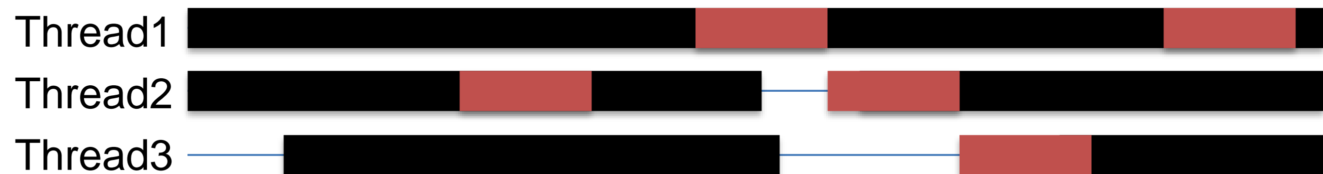
An easy fix for our BankAccount program:

```
public class BankAccount {
    private long balance;

    public BankAccount(long balance) {
        this.balance = balance;
    }

    static synchronized void transferFrom(BankAccount source,
                                          BankAccount dest, long amount) {
        source.balance -= amount;
        dest.balance   += amount;
    }

    public long balance() {
        return balance;
    }
}
```

Concurrency control with Java's *intrinsic* locks

with an explicit lock

- `synchronized (lock)` { … }
  - Synchronizes entire block on object `lock`; cannot forget to unlock
  - Intrinsic locks are *exclusive*: One thread at a time holds the lock
  - Intrinsic locks are *reentrant*: A thread can repeatedly get same lock



isr institute for SOFTWARE RESEARCH

Concurrency control with Java's *intrinsic* locks

with an implicit lock

- `synchronized (lock) { … }`
  - Synchronizes entire block on object `lock`; cannot forget to unlock
  - Intrinsic locks are *exclusive*: One thread at a time holds the lock
  - Intrinsic locks are *reentrant*: A thread can repeatedly get same lock
- `synchronized` on an instance method
  - Equivalent to `synchronized (this) { … }` for entire method
- `synchronized` on a static method in class `Foo`
  - Equivalent to `synchronized (Foo.class) { … }` for entire method

Another example: serial number generation

What would you expect this program to print?

```
public class SerialNumber {
    private static long nextSerialNumber = 0;

    public static long generateSerialNumber() {
        return nextSerialNumber++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads)
             thread.join();
        System.out.println(generateSerialNumber());
    }
}
```


institute for SOFTWARE RESEARCH

What went wrong?

- An action is *atomic* if it is indivisible
  - Effectively, it happens all at once
    - No effects of the action are visible until it is complete
    - No other actions have an effect during the action
- Java's ++ (increment) operator is not atomic!
  - It reads a field, increments value, and writes it back
- If multiple calls to `generateSerialNumber` see the same value, they generate duplicates

Again, the fix is easy

```
public class SerialNumber {
    private static long nextSerialNumber = 0;

    public static synchronized long generateSerialNumber() {
        return nextSerialNumber++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads)
             thread.join();
        System.out.println(generateSerialNumber());
    }
}
```


isr institute for SOFTWARE RESEARCH

But you can do better!

`java.util.concurrent` *is your friend*

```
public class SerialNumber {
    private static final AtomicLong nextSerialNumber = new AtomicLong();
    public static long generateSerialNumber() {
        return nextSerialNumber.getAndIncrement();
    }

    public static void main(String[] args) throws InterruptedException{
        Thread[] threads = new Thread[5];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1_000_000; j++)
                    generateSerialNumber();
            });
            threads[i].start();
        }
        for(Thread thread : threads) thread.join();
        System.out.println(generateSerialNumber());
    }
}
```

Some actions *are* atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

isr institute for SOFTWARE RESEARCH

Some actions *are* atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for ans?

i: 00000…00000111

⋮

i: 00000…00101010

ans: 00000…00101111

isr institute for SOFTWARE RESEARCH

Some actions *are* atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

i: 00000…00000111

⋮

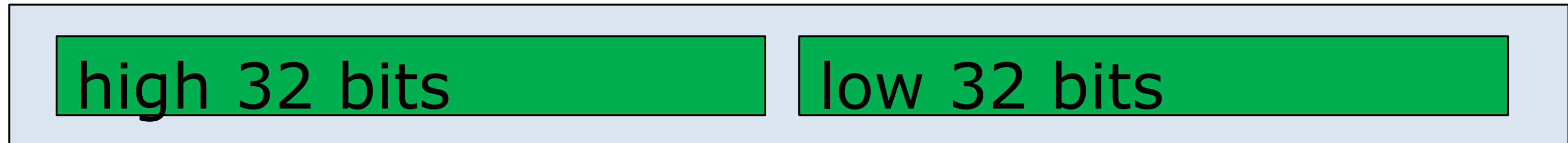
i: 00000…00101010

- In Java:
  - Reading an `int` variable is atomic
  - Writing an `int` variable is atomic
  - Thankfully, ans: ~~00000…00101111~~ is not possible

isr institute for SOFTWARE RESEARCH

Bad news: some simple actions are *not* atomic

- Consider a single 64-bit `long` value



  - Concurrently:
    - Thread A writing high and low bits
    - Thread B reading high and low bits

Precondition:

```
long i = 10_000_000_000;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

`ans:` 01001…00000000 (10,000,000,000)

`ans:` 00000…00101010 (42)

`ans:` 01001…00101010 (10,000,000,042)

All are possible!

isr institute for SOFTWARE RESEARCH

Yet another example: cooperative thread termination

How long would you expect this program to run?

```
public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested)
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(5);
        stopRequested = true;
    }
}
```

What went wrong?

- In the absence of synchronization, there is no guarantee as to when, **if ever**, one thread will see changes made by another
- **JVMs can and do perform this optimization ("hoisting"):**

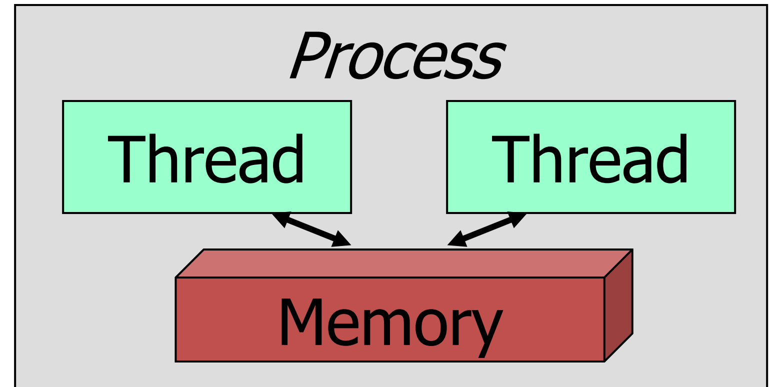
```
while (!done)
    /* do something */ ;
```

becomes:

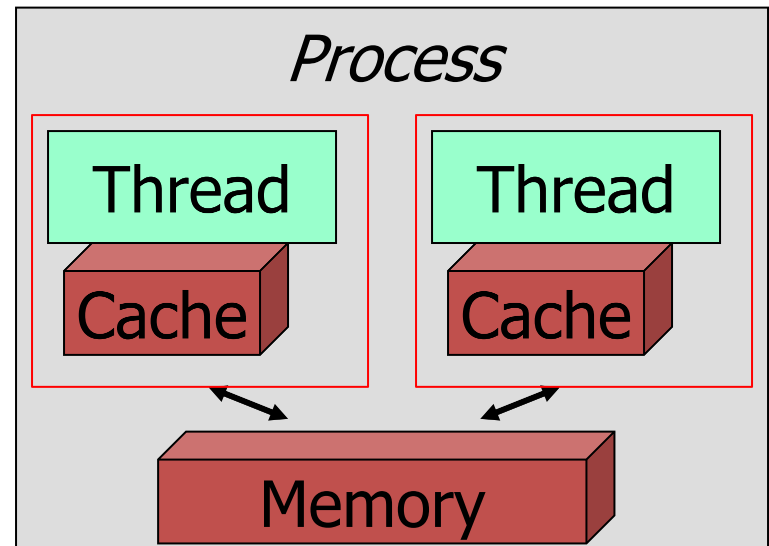
```
if (!done)
    while (true)
        /* do something */ ;
```

Why is synchronization required for communication among threads?

- Naively:
  - Thread state shared in memory



- A (slightly) more accurate view:
  - Separate state stored in registers and caches, even if shared



How do you fix it?

```
public class StopThread {
    private static boolean stopRequested;
    private static synchronized void requestStop() {
        stopRequested = true;
    }
    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested())
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(5);
        requestStop();
    }
}
```

A better(?) solution

`volatile` is synchronization without mutual exclusion

```
public class StopThread {
    private static volatile boolean stopRequested;

    public static void main(String[] args) throws Exception {
        Thread backgroundThread = new Thread(() -> {
            while (!stopRequested)
                /* Do something */ ;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(5);
        stopRequested = true;
    }
}
```


institute for SOFTWARE RESEARCH

Summary

- Like it or not, you're a concurrent programmer
- Ideally, avoid shared mutable state
  - If you can't avoid it, synchronize properly
- Some things that look atomic aren't (e.g., `val++`)
- Even atomic operations require synchronization
  - e.g., `stopRequested = true`
  - Synchronization is required for communication as well as mutual exclusion