1. Encryption & Security
2. Concurrency
Announcements

- Lab Exam 2 Tonight
- Tuesday: PA10
- Thursday: PS 12
- Friday: Final Exam
Encryption and Security
Overview

- Security issues
- Encryption and cryptanalysis
- Encryption in the digital age
  - Symmetric encryption
  - Asymmetric encryption
- Applications of encryption
- Encryption is not security!
Security issues
Networking is a security issue

Why?

If you want a really secure machine, lock it in an electromagnetically shielded room and don’t connect it to any networks or other sources of data beyond your control.

Not much fun, is it?
The Problem

- The Internet is public
  - Messages sent pass through many machines and media

- Anyone intercepting a message might
  - read it and/or
  - replace it with a different message

- The Internet is anonymous
  - IP addresses don’t establish identity

- Anyone may send messages under a false identity

Cryptography offers partial solutions to all of these problems
A Shady Example

- I want to make a purchase online and click a link that takes me to http://www.sketchystore.com/checkout.jsp

- What I see in my browser:

  Enter your credit card number: 2837283726495601
  Enter your expiration date: 0109
  Submit
When I press SUBMIT, my browser sends this:

POST /purchase.jsp HTTP/1.1

Host: www.sketchystore.com

User-Agent: Mozilla/4.0

Content-Length: 48

Content-Type: application/x-www-form-urlencoded

userid=rbd&creditcard=2837283726495601&exp=01/09
A Shady Example (cont’d)

- If this information is sent unencrypted, who has access to my credit card number?
  - Other people who can connect to my wireless ethernet
  - Other people physically connected to my wired ethernet
  - ...

- Packets are passed from router to router.
  - All those routers have access to my data.
A caveat
cryptography is not security
A CRYPTO NERD’S IMAGINATION:
His laptop’s encrypted. Let’s build a million-dollar cluster to crack it.

NO GOOD! It’s 4096-bit RSA!

BLAST! Our evil plan is foiled!

WHAT WOULD ACTUALLY HAPPEN:
His laptop’s encrypted. Drug him and hit him with this $5 wrench until he tells us the password.

GOT IT.
Encryption and cryptanalysis

basic concepts
We encrypt (encode) our data so others can’t understand it (easily) except for the person who is supposed to receive it.

We call the data to encode plaintext and the encoded data the ciphertext.

Encoding and decoding are inverse functions of each other.
Encryption/decryption

**plaintext** → Encryption algorithm → **ciphertext**

**secret key**

ATTACKATDAWN

**secret key**

Decryption algorithm → ATTACKATDAWN
Cryptanalysis

- **secret key**
- **plaintext**
- **ciphertext**
- **Encryption algorithm**
- **AGSTRMBNDO**
- **ATTACKATDAWN**

Mathematical, logical, empirical analysis
Encryption techniques

substitution and transposition
Two basic ways of altering text to encrypt/decrypt

- Substitute one letter for another using some kind of rule

  Substitution cipher

- Scramble the order of the letters using some kind of rule

  Transposition cipher
Substitution Ciphers

- Simple encryption scheme using a substitution cipher:
  - Shift every letter forward by 1:
    
    \[ A \rightarrow B, \ B \rightarrow C, \ldots, \ Z \rightarrow A \]

- Example:
  
  \[ \text{MESSAGE} \rightarrow \text{NFTTBHF} \]

- Can you decrypt TFDSFU?
Substitution Ciphers

- Simple encryption scheme using a substitution cipher:
  - Shift every letter forward by 1:
    - A → B, B → C, ..., Z → A

- Example:
  - MESSAGE → NFTTBHF

- Can you decrypt TFDSFU? SECRET
Caesar Cipher

- Shift forward $n$ letters; $n$ is the secret key

- For example, shift forward 3 letters:
  A $\rightarrow$ D, B $\rightarrow$ E, ..., Z $\rightarrow$ C
  - This is a Caesar cipher using a key of 3.

- MESSAGE $\rightarrow$ PHVVDJH

- How can we crack this encrypted message if we don’t know the key?
  DEEDUSEKBTFEIIYRBOTUSETUJXYI
How long would it take a computer to try all 25 shifts?
Vigenère Cipher

- Shift different amount for each letter. Use a key word; each letter in the key determines how many shifts we do for the corresponding letter in the message.

- Example: key word “cmu”: shift by 2, 12, 20

- Message “pittsburgh”

  cmucmcucmc

  encrypted: runvevwdaj

- Try it yourself at http://www.simonsingh.net/The_Black_Chamber/v_square.html
<table>
<thead>
<tr>
<th>Letter</th>
<th>Shifted Letters</th>
<th>Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZ</td>
<td>no shift</td>
</tr>
<tr>
<td>B</td>
<td>BCDEFGHIJKLMNOPQRSTUVWXYZA</td>
<td>shift by 1</td>
</tr>
<tr>
<td>C</td>
<td>CDEFGHIJKLMNOPQRSTUVWXYZAB</td>
<td>shift by 2</td>
</tr>
<tr>
<td>D</td>
<td>DEFGHIJKLMNOPQRSTUVWXYZABC</td>
<td>shift by 3</td>
</tr>
<tr>
<td>E</td>
<td>EFGHIJKLMNOPQRSTUVWXYZABCD</td>
<td>etc.</td>
</tr>
<tr>
<td>F</td>
<td>FGHJKLMNOPQRSTUVWXYZABCD</td>
<td>etc.</td>
</tr>
</tbody>
</table>

- **Message:** ATTACKATDAWN
- **Pick a secret key:** DECAFDECAFDE
- **Encrypted:** D

1st letter in the message is shifted by 3, 2nd letter is shifted by 4, …
• Message: ATTACKATDAWN
• Pick a secret key: DECAFDECAFDE
• Encrypted: DX

1st letter in the message is shifted by 3, 2nd letter is shifted by 4, …
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

ABCDFGHILMNOPQRSTUVWXYZ

A    ABCDFGHILMNOPQRSTUVWXYZ
B    BCDFGHILMNOPQRSTUVWXYZA
C    CDFGHILMNOPQRSTUVWXYZAB
D    DFGHIJKLMNOPQRSTUVWXYZABC
E    EFGHIJKLMNOPQRSTUVWXYZABCD
F    FGHILMNOPQRSTUVWXYZABCDE

...  

- **Message:** ATTACKATDAWN
- **Pick a secret key** DECAFDECAFDE
- **Encrypted:** DXV

1st letter in the message is shifted by 3, 2nd letter is shifted by 4, ...
<table>
<thead>
<tr>
<th></th>
<th>ABCDEFGHIJKLMNOPQRSTUVWXYZ</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZ</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>BCDEFGHIJKLMNOPQRSTUVWXYZA</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>CDEFGHIJKLMNOPQRSTUVWXYZAB</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>DEFGHIJKLMNOPQRSTUVWXYZABC</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>EFGHIJKLMNOPQRSTUVWXYZABCD</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>FGHIJKLMNOPQRSTUVWXYZABCD</td>
<td></td>
</tr>
</tbody>
</table>

...  

- **Message:** ATTACKATDAWN  
- **Pick a secret key** DECAFDECAFDE  
- **Encrypted:** DXVAHNEVDFZR  

1st letter in the message is shifted by 3, 2\(^{nd}\) letter is shifted by 4, ...
- Vigenère cipher was broken by Charles Babbage in the mid 1800s by exploiting the repeated key
  - The length of the key determines the cycle in which the cipher is repeated.
- Vernam cipher: make the key the same length as the message; Babbage’s analysis doesn’t work.
One-time Pads

- Vernam cipher is commonly referred to as a one-time pad.

- If random keys are used one-time pads are unbreakable in theory.

Alice and Bob have identical “pads” (shared keys)
Transposition ciphers

STSF…EROL…NOUA…DOTN…MPHK…OSEA…RTRN…EOND…

Encryption in computing

fast computation makes encryption usable by all of us
Encryption in computing

- One-time pads impractical on the net (why?)

- Basic assumption: the encryption/decryption *algorithm* is known; only the key is secret (why?)

- Very complicated encryptions can be computed fast:
  - typically, elaborate combinations of substitution and transposition
HTTPS

- Security protocol for the Web, the peoples’ encryption

- Purpose:
  - confidentiality (prevent eavesdropping)
  - message integrity and authentication (prevent “man in the middle” attacks that could alter the messages being sent)

- Techniques:
  - asymmetric encryption ("public key" encryption) to exchange secret key
  - certificate authority to obtain public keys
  - symmetric encryption to exchange actual messages
Symmetric vs. asymmetric encryption

- **Symmetric** (shared-key) encryption: commonly used for long messages
  - Often a complicated mix of substitution and transposition encipherment
  - Reasonably fast to compute
  - Requires a shared secret key usually communicated using (slower) *asymmetric encryption*

- **Asymmetric** encryption: different keys are used to encrypt and to decrypt
Keypspace

- *Keypspace* is jargon for the number of possible secret keys, for a particular encryption/decryption algorithm

- Number of bits per key determines *size of keyspace*
  - important because we want to make *brute force attacks* infeasible
  - brute force attack: run the (known) decryption algorithm repeatedly with *every possible key* until a sensible plaintext appears

- Typical key sizes: several hundred bits
Symmetric (Shared Key) Encryption

Ciphertext = Enc(plaintext, key)

Alice
Encrypt using key

Plaintext
Alice uses the shared key to encrypt the plaintext to produce the ciphertext

Enc() and Dec() are functions

Bob
Decrypt using key

Plaintext = Dec(Ciphertext, key)

Bob uses the shared key to decrypt the ciphertext to recover the plaintext

Ciphertext

Establishing Shared Keys

Problem: how can Alice and Bob secretly agree on a key, using a public communication system?

Solution: asymmetric encryption based on *number theory*
- Alice has one secret, Bob has a different secret; working together they establish a shared secret
- Examples: Diffie-Hellman key exchange, RSA public key encryption
One type of asymmetric encryption: RSA

- Common encryption technique for transmitting symmetric keys on the Internet (https, ssl/tls)
- Named after its inventors: Rivest, Shamir and Adleman
- Used in https (you know when you’re using it because you see the URL in the address bar begins with https://)
Asymmetric Public Key Encryption

Alice

Encrypt using pubB

plaintext

ciphertext = Enc(plaintext, pubB)

Bob’s public key pubB

Bob

Decrypt using privB

plaintext = Dec(ciphertext, privB)

Bob’s private key privB

Alice uses Bob’s public key to encrypt the plaintext to produce the ciphertext

Bob uses his private key to decrypt the ciphertext to recover the plaintext
First, we must be able to represent any message as a single number (it may already be a number as is usual for a symmetric key)

For example:

```
ATTACKATDAWN
012020010311012004012314
```
Every receiver has a **public key** \((e, n)\) and a **private key** \((d, n)\).

The transmitter encrypts a (numerical) message \(M\) into ciphertext \(C\) using the receiver’s public key:

\[ M^e \mod n \rightarrow C \ (\text{ciphertext}) \]

The receiver decodes the encrypted message \(C\) to get the original message \(M\) using the private key (which no one else knows).

\[ C^d \mod n \rightarrow M \ (\text{plaintext}) \]
**RSA Example**

- Alice’s Public Key: (3, 33) \( (e = 3, n = 33) \)
- Alice’s Private Key: (7, 33) \( (d = 7, n = 33) \)
  - Usually these are really huge numbers with many hundreds of digits!

- Bob wants to send the message 4
  - Bob encrypts the message using \( e \) and \( n \):
    \[ 4^3 \mod 33 \rightarrow 31 \]
    ... Bob sends 31

- Alice receives the encoded message 31
  - Alice decrypts the message using \( d \) and \( n \):
    \[ 31^7 \mod 33 \rightarrow 4 \]
Generating $n$, $e$ and $d$

- $p$ and $q$ are (big) random primes.
  - $p = 3$, $q = 11$
- $n = p \times q$
  - $n = 3 \times 11 = 33$
- $\varphi = (p - 1)(q - 1)$
  - $\varphi = 2 \times 10 = 20$
- $e$ is small and relatively prime to $\varphi$
  - $e = 3$
- $d$, such that:
  - $3 \times d \mod 20 = 1$
  - $d = 7$

Usually the primes are huge numbers--hundreds of digits long.
Everyone knows \((e, n)\). Only Alice knows \(d\).

If we know \(e\) and \(n\), can we figure out \(d\)?

- If so, we can read secret messages to Alice.

**We can** determine \(d\) from \(e\) and \(n\).

- Factor \(n\) into \(p\) and \(q\).
  \[
  n = p \times q \\
  \varphi = (p - 1)(q - 1) \\
  e \times d = 1 \pmod{\varphi}
  \]

  - We know \(e\) (which is public), so we can solve for \(d\).

But **only** if we can factor \(n\).
RSA is safe (for now)

- Suppose someone can factor my 5-digit $n$ in 1 ms,

- At this rate, to factor a 10-digit number would take 2 minutes.

- ... to factor a 15-digit number would take 4 months.

- ... 20-digit number ... 30,000 years.

- ... 25-digit number... 3 billion years.

- We're safe with RSA! (at least, from factoring with digital computers)
How do we know we have the right public key for someone?

Certificate Authorities sign digital certificates indicating authenticity of a sender who they have checked out in the real world.

Senders provide copies of their certificates along with their message or software.

But can we trust the certificate authorities? (only some)
Encryption is not security!
It’s just a set of techniques
How (in)secure is the Internet?

- The NSA has a budget of $11B; we know from Edward Snowden how some of it is used
- Corporations and criminals also spy on us
- What can go wrong?
  - Insecure pseudo-random number generators
  - Untrustworthy certificate authorities
  - Malware
  - “Social engineering” attacks like phishing
  - Deliberately built-in insecurity in crypto products
  - Physical tapping of Internet routers
Security is an unsolved problem

Your cyber systems continue to function and serve you not due to the expertise of your security staff but solely due to the sufferance of your opponents.

– former NSA Information Assurance Director Brian Snow (quoted by Bruce Schneier, https://www.schneier.com/blog/archives/2013/03/phishing_has_go.html)
Summary

- Cryptography is cool mathematics and protocol design
- But cryptography is not security, only a set of techniques
- Security is a broader issue involving
  - Other technology
  - Social and legal factors

“Only amateurs attack machines; professionals target people” – Bruce Schneier
Two closing thoughts

Use Signal...
Concurrency
Concurrency in Real Life

- Concurrency is the simultaneous occurrence of events.

- Most complex tasks that occur in the physical world can be broken down into a set of simpler activities
  - Building a house: bricklaying, carpentry, plumbing, electrical installation, roofing
  - Some of them can overlap and take place concurrently
Concurrency in Computing

- Computing on the Internet: independent, autonomous agents trying to achieve individual and shared goals.

- Even on our local machines, we take it for granted that we can do more than one thing at a time.
  - We continue to work in a word processor, while other applications download files, manage the print queue, and stream audio.
  - Even a single application is often expected to do more than one thing at a time.
Concurrent Programming

- The activity described by a computer program can also be divided into simpler activities (subprograms)

- **Sequential programs:** Subprograms do not overlap in time, they are executed one after another
  - In 15-110 we have been writing sequential programs.

- **Concurrent programs:** Subprograms may overlap in time, their executions proceed concurrently
  - In 15-110 we will not write concurrent programs but we will learn about what makes them tricky.
Why Do We Need It?

- Everything happens at once in the world. Inevitably, computers must deal with that world.
  - For example, traffic control, airline seat reservation, process control, banking

- Performance gain from multiprocessing hardware
  - For example, Google, Yahoo, divide each query into thousands of little queries and use thousands of small computers.
  - For example, a supercomputer with thousands of processors can compute a weather prediction much faster than a single processor.

- Increased application throughput for applications sharing computational resources. Throughput is the amount of work that a computer can do in a given time period.
  - When one application is waiting for I/O another can continue its execution.
Caution

- The advantages of concurrency may be offset by the increased complexity of concurrent programs.
  - We will be giving some examples of what may go wrong in concurrent programming.

- Notorious cases of erroneous concurrent software:
  - Therac-25 computerized radiation therapy machine
  - Mars Rover “Spirit”
DIFFERENT FLAVORS OF CONCURRENCY
A Useful Abstraction: Process

- Process: A program in execution
  - Program along with its data in memory, open files, open communication channels etc.

- Concurrency involves multiple processes running simultaneously on multiple processors or on a single processor time-sharing the processor.
Sharing a Processor

If only one processor (CPU) is available, the only way to run multiple processes is by switching between them.

Only one process is using the CPU at a given time even though they look like they are running in parallel to an observer.
Scheduling

The order in which the steps are run is determined by a scheduler. There are many possibilities.
If you have multiple CPUs, you may execute multiple processes in parallel (simultaneously). Really!

**Process 1:**
on processor 1
run

**Process 2:**
on processor 2
run

---

*step1*

*step2*

*step3*

---

**time**
Sharing Memory

- Processes may share resources such as memory.

- For example, only one processor at a time may execute an instruction that touches the shared memory. The memory hardware makes the others wait.

Shared memory can be used for communication between processes.
Distributed Computing

Processes may run on distributed systems

- For example, a cluster of workstations, communicating via sockets

Some steps are executed simultaneously but some are dependent on another.
Hardware supports parallelism. Nowadays, we have multiple processors in most computing environments such as multicore machines, clusters.

Programmers do not always support parallelism. Algorithms do not fully utilize parallelism provided by hardware.

Many programming languages offer “multithreading” libraries to support concurrent programming:

- Structuring programs where there are logically separate, naturally independent control flows.
- What is really needed is development of new languages that will enable programmers to express parallel algorithm designs.

We will not focus on parallel algorithms. We will focus on issues that arise from concurrent execution of sequential processes that cooperate to achieve a common goal.
Threads

- What most programmers think of when they hear about concurrent programming today.
- We will use Python threads to illustrate some challenges with concurrent programming.
- Thread: a (somewhat) independent computation running inside a program
- Shares resources with the main program (memory, files, network connections etc.)
Thread Basics

>>> python3 -i example.py

statement

statement

statement

statement

Python launches the “main” thread of the program. Control flows from one statement to another.
>>> python3 -i example.py

```
def foo():
    statement
    statement
    ...
```

Assume that `foo` is a function that has already been defined.
Thread Basics

```python
>>> python3 -i example.py

statement
    ↓
statement
    ↓
create thread(foo)
    ↓
statement
    ↓
statement ...
    ↓
statement ...
    ↓
... ...
```

**Concurrent execution** of the "main" thread and the function foo()
Thread Basics

>>> python3 -i example.py

```
statement
statement
create thread(foo)
statement
statement...
```

Thread is like a “process” that runs independently inside a program
Functions as Threads

The Python module `threading` allows you to create Thread objects or use functions as threads.

Below is a function that is used as a thread.

```python
import threading

def countdown(count):
    while count != 0:
        count = count-1
    return

t1= threading.Thread(target=countdown, args=(10,))
t1.start()
do your own thing
t1.join()
```
Joining a Thread

- Once you start a thread it runs independently.
- Use `t.join()` to wait for a thread `t` to exit

```python
t.start()  # launch a thread `t`

# do other work
...

# wait for thread `t` to finish and exit
```
```
Access to Shared Data

- Threads share all of the data in your program.

- We cannot assume anything about scheduling (the order of steps in an execution).

- Operations that we think of as a single step are often non-atomic (take several steps and might be interrupted).
Thread 1 and Thread 2 are separate threads.

The dashed lines indicate the points in time at which a switch occurs.

We cannot assume anything about when these switches will occur.

Thread 1:
- run
- run
- run

Thread 2:
- run
- run
- run

Thread 1 and Thread 2 are separate threads.
Consider a shared resource (variable $x$ in this example)

\[
x = 0
\]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$x = x + 1$</td>
<td>$x = x - 1$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
</tbody>
</table>
import threading
x = 0
def inc():
    global x
    for i in range(1000000):
        x = x + 1

def dec():
    global x
    for i in range(1000000):
        x = x - 1

t1 = threading.Thread(target = inc)
t2 = threading.Thread(target = dec)
t1.start()
t2.start()
t1.join()
t2.join()
print(x)
import threading
x = 0
def inc():
    global x
    for i in range(1000000):
        x = x + 1

def dec():
    global x
    for i in range(1000000):
        x = x - 1

t1 = threading.Thread(target = inc)
t2 = threading.Thread(target = dec)
t1.start()
t2.start()
t1.join()
t2.join()
print(x)

Caution: Global variables should be used sparingly. They can be modified and read in a variety of places in the code. They make it hard to read, test and debug code.
import threading
x = 0
def inc():
    global x
    for i in range(1000000):
        x = x + 1

def dec():
    global x
    for i in range(1000000):
        x = x - 1

t1 = threading.Thread(target = inc)
t2 = threading.Thread(target = dec)
t1.start()
t2.start()
t1.join()
t2.join()
print(x)
We thought of addition and subtraction as one indivisible step but Python divided their execution into smaller steps.
A Possible Interleaving of Steps

Thread 1  Thread 2

\[ x = x + 1 \quad x = x - 1 \]

Thread 1  Thread 2

Load_global x

Load_const 1

switch  Load_global x

Load_const 1

Subtract

Store_global x

Add

Store_global x

One of several possible interleavings of steps actually took place:

Think of starting execution at a state with \( x = 0 \). Can you see why the final value would be 1, not 0?
Not what the programmer intended

Thread 1    Thread 2

\[ x = x + 1 \quad x = x - 1 \]

Thread 1    Thread 2

Load_global x

Load_const 1

switch

Load_global x

Load_const 1

Subtract

Store_global x

Operations performed on a stale value of x (i.e. 0) after x has been updated to -1 by Thread 1
Knock, Knock
Race Condition!
Who’s There?
Race Condition

Thread 1       Thread 2
\[ x = x + 1 \quad x = x - 1 \]

Thread 1       Thread 2
Load\_global x
Load\_const 1

\[ \text{switch} \quad \text{Load\_global x} \]
\[ \text{Load\_const 1} \]

Subtract

Store\_global x

Add

Store\_global x

**Race condition**: two or more threads operating on the same data object without proper synchronization

The output is dependent on the timing of uncontrollable events such as scheduling decisions of the underlying system.
Concurrent programming is hard.

- Only a tiny percentage of practicing programmers can do it.
- It requires art and mathematics.
  - It’s like digital hardware design.
  - It needs proofs.
- Conventional debugging doesn’t work.
  - If you stop the program to observe, you change the behavior.
  - Testing is futile because the number of possible execution sequences for the same input explodes.
Summary

- Sequential vs. concurrent programming paradigms
  - Advantages of using concurrency: utilizing resources more efficiently, dealing with concurrent events in the computational environment
  - Challenges in concurrent programming
    - Synchronization between different tasks and access to shared data is a major source of complexity
    - Need to consider all possible executions
    - Difficulty of replicating errors

- We will NOT do any programming with threads. We looked at it only to illustrate the concepts of process scheduling, interleaving of actions, and race conditions.
Concurrency is hard...
Recap

- Process: program in execution. Unit of sequential execution.

- We can structure programs so that they can be executed as a set of concurrent processes
  - On a single processor
  - On multiple processors

- Processes may coordinate their actions using
  - Shared memory
  - Message passing

- A race condition is a situation in which multiple processes read and write a shared data item and the final result depends on the order of execution.
There are many ways to execute two processes concurrently.

The green process executes steps $S1 \ S2 \ S3$ in the given order.
The blue process executes steps $S1 \ S2 \ S3$ in the given order.

Several possible interleavings of steps.
In the rest of the lecture we will use some programs to illustrate concepts such as race conditions, interference, and deadlock. For the purposes of this lecture we assume that a single line of program is executed atomically: you can think of one line of code as corresponding to one step in the previous slide whose execution cannot be broken down into smaller steps.
Critical Sections

- Often, a process really needs exclusive access to some data.

- A **critical section** is a sequence of steps that have exclusive access to the shared resource.
  - If multiple processes are sharing a resource only one should be executing its critical region.

- Real Life Examples where critical sections are needed:
  - Crossing a traffic intersection
  - A bank with many ATMs
Critical Section Example

- Consider a bank with multiple ATM’s.
- At one, Mr. J requests a withdrawal of $10.
- At another, Ms. J requests a withdrawal of $10 from the same account.

The bank’s computer executes:

1. For Mr. J, verify that the balance is big enough.
2. For Ms. J, verify that the balance is big enough.
3. Subtract 10 from the balance for Mr. J.
4. Subtract 10 from the balance for Ms. J.

The balance went negative if it was less than $20!
Vocabulary Reminder

• **Race condition**: A behavior in concurrent processing where proper functioning depends on the timing of other uncontrollable events.

• **A critical section** is a piece of code that accesses a shared resource that must **not** be concurrently accessed by more than one process.
Critical Sections in a Program

What can we do to prevent one processor from entering the critical section while another is in it?

Location the J’s account data containing the balance

if balance < 10:
    error
else:
    balance = balance – 10

Dispense $10 from ATM

Process executed for each concurrent request
Careful Driver Method: Don’t enter the intersection unless it’s empty.

In shared memory: \texttt{free} = True \hspace{1cm} \# initially unlocked

\begin{verbatim}
# Process 1
while True :
    Non-Critical_Section
    while not free:
        pass
    free = False
    Critical_Section
    free = True
\end{verbatim}

\begin{verbatim}
# Process 2
while True :
    Non-Critical_Section
    while not free:
        pass
    free = False
    Critical_Section
    free = True
\end{verbatim}

Interference is possible!
In shared memory: \( \text{free} = \text{True} \) \# initially unlocked

# Process 1
while True :
    Non-Critical_Section
    while not free :
        pass
    free = False
Critical_Section
free = True

# Process 2
while True :
    Non-Critical_Section
    while not free :
        pass
    free = False
Critical_Section
free = True

If these two processes leave their non-critical sections at precisely the same time, then strictly alternate lines, they will both end up in the Critical_Section.
The careful driver method works in real life because

- The number of times in your life you cross the intersection is low. Twice a day for forty years is about 29,000.
- The chance of two drivers arriving at the intersection simultaneously is low.
- Cars move slowly enough that if you don’t see anyone coming, you’ll get across before anyone comes.
1. Signal your intention (by stopping).
2. Wait until cross road has no one waiting or crossing.
3. Cross intersection.
4. Renounce intention (by leaving intersection).
The Stop and Look Method

# Shared Memory
free0 = True  # P0 is not stopped at sign
free1 = True  # P1 is not stopped at sign

# Process 0
while True :
    Non-Critical_Section
    free0 = False
    while not free1 :
        pass
    Critical_Section
    free0 = True

# Process 1
while True :
    Non-Critical_Section
    free1 = False
    while not free0 :
        pass
    Critical_Section
    free1 = True

This version of the code does not suffer from interference. Does it now guarantee safe execution of the critical section?
The Stop and Look Method

# Shared Memory

free0 = True  # P0 is not stopped at sign
free1 = True  # P1 is not stopped at sign

# Process 0
while True :
    Non-Critical_Section
    free0 = False
    while not free1:
        pass
    Critical_Section
    free0 = True

# Process 1
while True :
    Non-Critical_Section
    free1 = False
    while not free0:
        pass
    Critical_Section
    free1 = True

Once again, if the two processes exit the non-critical section at the same time and strictly alternate lines they will end up stuck in their while loops. This is called Deadlock.
Deadlock

- Deadlock is the condition when two or more processes are all waiting for some shared resource, but no process actually has it to release, so all processes to wait forever without proceeding.

- It’s like gridlock in real traffic.
The Stop Sign Method with Tie Breaking

1. Signal your intention (by stopping).
2. Wait until cross road has no one else waiting or crossing.
3. If two of you are both waiting, yield to the car to your right.
4. Cross intersection.
5. Renounce intention (by leaving intersection).
Peterson’s algorithm avoids all bugs!

```python
free0 = True
free1 = True
priority = 0

# Process 0
while True :
    Non-Critical_Section0
    free0 = False
    priority = 1
    while not free1 and priority==1:
        pass
    Critical_Section0
    free0 = True

# Process 1
while True :
    Non-Critical_Section1
    free1 = False
    priority = 0
    while not free0 and priority==0:
        pass
    Critical_Section1
    free1 = True
```
Peterson’s algorithm avoids all bugs!

```python
free0 = True
free1 = True
priority = 0

# Process 0
while True :
    Non-Critical_Section0
    free0 = False
    priority = 1
    while not free1 and priority==1 :
        pass
    Critical_Section0
    free0 = True

# Process 1
while True :
    Non-Critical_Section1
    free1 = False
    priority = 0
    while not free0 and priority==0 :
        pass
    Critical_Section1
    free1 = True
```

Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section (free1 == True) or if P1 has given priority to P0 by setting priority to 0 (priority == 0).
There is a conceptually easier way to solve synchronization problem by embracing probable thinking. We just use the stop sign approach but wait for a random amount of time when a conflict occurs.
Types of HeisenBugs*

In decreasing order of seriousness:

1. Interference: multiple process in critical section.
2. Deadlock: two processes idle forever, neither entering their critical or non-critical sections.
3. Starvation: one process needlessly idles forever while the other stays in its non-critical section.
4. Unfairness: a process has lower priority for no reason.

Note: We did not discuss 3 and 4 in detail. You can learn more about them in the future.

* In computer programming jargon, a heisenbug is a software bug that seems to disappear or alter its behavior when one attempts to study it. Source: Wikipedia
Dining Philosophers’ Problem

Aristotle

Socrates

Homer

Plato
The Dining Philosophers

• Each philosopher thinks for a while, then picks up his left fork, then picks up his right fork, then eats, then puts down his left fork, then puts down his right fork, thinks for a while...
  – We assume here that each philosopher thinks and eats for random times, and a philosopher cannot be interrupted while he picks up or puts down a single fork.

• Each fork models a "resource" on a computer controlled by an OS.

• Original problem was proposed by Edsgar Dijkstra.
There are $N$ philosophers.

Philosopher $i$ does the following:
1. THINK
2. Pick up fork $i$.
3. Pick up fork $(i+1) \mod N$.
4. EAT
5. Put down fork $i$.
6. Put down fork $(i+1) \mod N$.
7. Go to step 1.

NOTE: $(i+1) \mod N = i+1$, if $0 \leq i < N-1$
$(i+1) \mod N = 0$, if $i = N-1$
Dining Philosophers’ Problem

• There are N philosophers.

• Philosopher i does the following:
  1. THINK
  2. Pick up fork i.
  3. Pick up fork \((i+1)\) modulo \(N\).
  4. EAT
  5. Put down fork i.
  6. Put down fork \((i+1)\) modulo \(N\).
  7. Go to step 1.

How can deadlock occur here?
Removing the Deadlock

Philosopher $i$ does the following:

1. THINK
2. If $i$ is not equal to $N-1$:
   a. Pick up fork $i$
   b. Pick up fork $i + 1$
3. If $i$ is equal to $N-1$:
   a. Pick up fork 0
   b. Pick up fork $N - 1$
4. EAT
5. If $i$ is not equal to $N-1$:
   a. Put down fork $i$
   b. Put down fork $i + 1$
6. If $i$ is equal to $N-1$:
   a. Put down fork 0
   b. Put down fork $N - 1$
7. Go to step 1

This philosopher picks up the right fork first

This philosopher picks up the right fork first
Afterthoughts

Some counter-intuitive ideas about bugs and risks.
This man removed all the traffic lights and signs!

It reduced accidents by 50% and reduced congestion. Pedestrian and driver anxiety increased.
Why did Jared Diamond sleep under a tree when his aborigine companion wouldn’t?

There is only a $1/10,000$ chance of a particular tree falling on any given night. But aborigines sleep in the woods every night so if they tempted fate every night for 40 years there is only a 23% chance of not dying that way, i.e. \((1-1/10,000)^{(40*365)} = 0.23\).
Why is a 1% chance of a bug biting better than a 0.1% chance?

- If there is a 1% chance of error, the bug will show up during 100 days of testing.

- If there is a 0.1% chance, the bug will show up after three years when the system is deployed.