

# COMN - Computer Communications and Networks

## Assignment 2: Implement & Analyse Sliding Window Protocols

---

**Michio Honda**

With acknowledge to Andrew Ferguson and Jon Larrea

School of Informatics

University of Edinburgh

14/02/2025



# Assignment 2 Released

- Due Monday 31th March, midday (1200 UTC / GMT)
- Available at <https://git.ecdf.ed.ac.uk/mhonda/comn25cw>
  - Linked from the Piazza announcement will be emailed

The screenshot shows a GitHub repository page for 'comn24cw'. The repository has 6 commits, 1 branch, and 0 tags. A commit titled 'Release CW2: Sliding Window Protocols' by Andrew Ferguson was authored 14 hours ago. The commit hash is 0c0ee490. Below the commit, there are buttons for History, Find file, Edit, and Code. In the sidebar, under the 'Code' section, the 'sliding\_window' folder is highlighted with a red oval. The main content area displays a table of files:

Name	Last commit	Last update
sliding_window	Release CW2: Sliding Window Protocols	14 hours ago
traffic_analysis	cw1	2 weeks ago
vm	ubuntu ver	2 weeks ago
README.md	vm setup	3 weeks ago



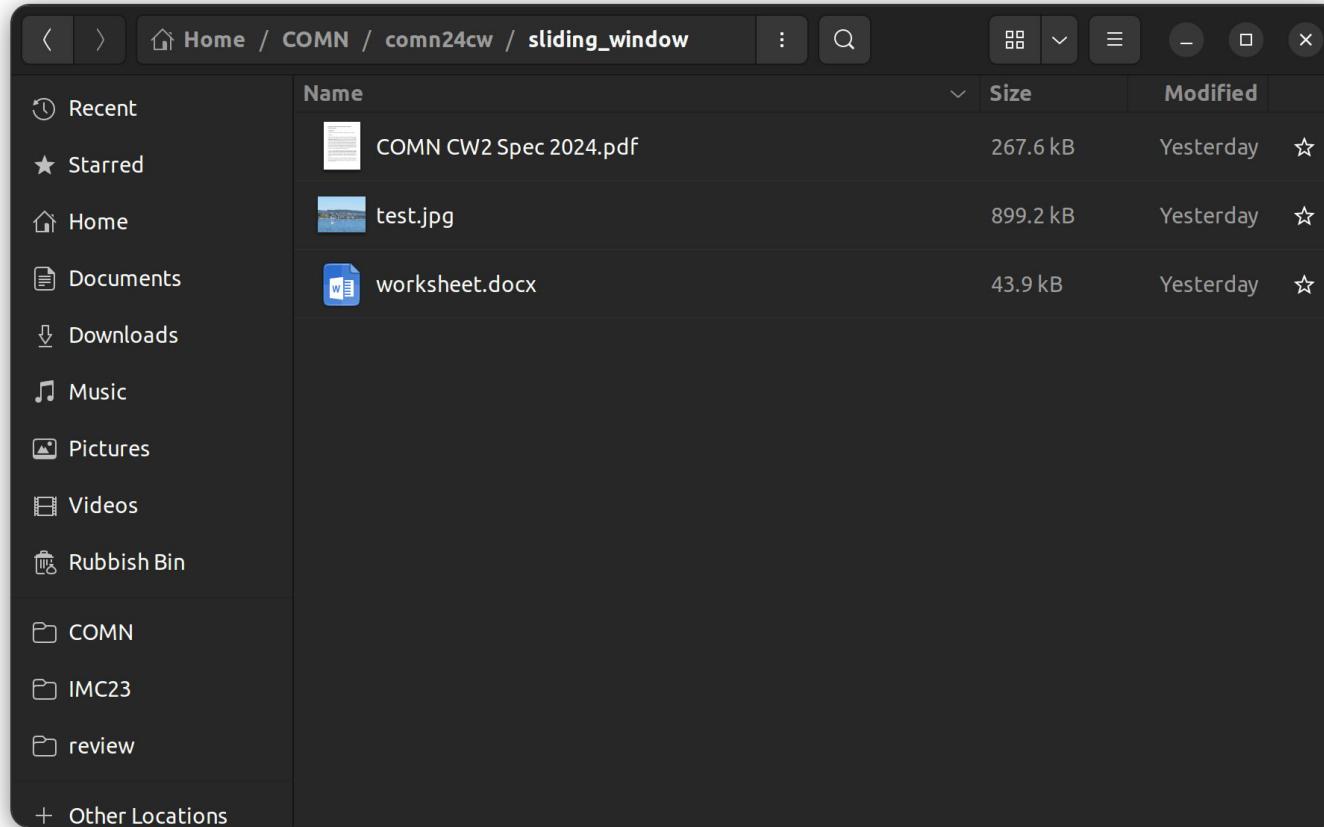
# Assignment 2 Released

---

- "I have already cloned the git repo (e.g. to setup the VM, or attempt CW1)"
  - **cd comn25cw**
  - **git pull**
  - <enter your username / password>
- "I have not yet cloned the git repo"
  - **git clone <https://git.ecdf.ed.ac.uk/mhonda/comn25cw.git>**
  - <follow the VM setup instructions in comn24cw/vm>



# Assignment 2 Released



# Coursework Overview

---

- Goal
  - Implementation and evaluation of three end-to-end reliable data transfer protocols
- Assessment: 30% of course mark
  - Part 1 (4.2%): rdt1.0
  - Part 2 (8.6%): rdt 3.0 (Stop-and-Wait)
  - Part 3 (8.6%): Go-back-N
  - Part 4 (8.6%): Selective Repeat + iPerf experiment



# Reliable Data Transport Protocols

---

- Why do we need a reliable data transport protocol?
- Consider TCP (reliable) vs UDP (unreliable)

TCP



UDP



TCP



UDP



# Reliable Data Transport Protocols

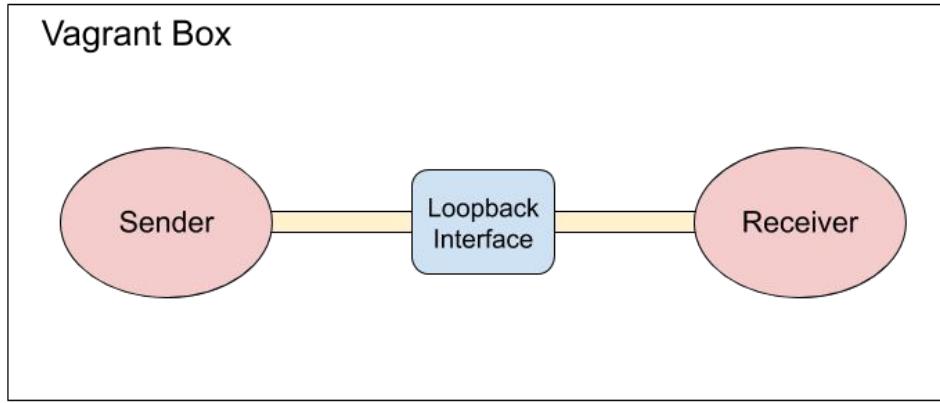
---

- TCP: Ensures each packet is received by the receiver
  - UDP: sends data without caring what the receiver is doing (or even if there is one at all)
- 
- For this assignment: "reliable" == no **lost** packets
    - Data corruption is a separate concern
    - We assume no corruption can occur
    - In the real world checksums can detect and correct errors



# Conceptual Structure

---

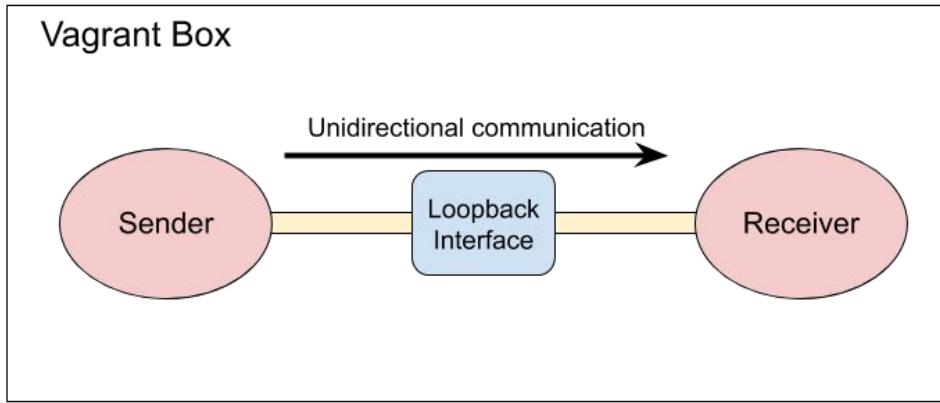


## ■ Linux Traffic Control (The tc utility)

- Allows you to modify the packet scheduler for a given interface
- Configuration of interface characteristics (bandwidth, delay, loss)
- Command-line program: tc

# Conceptual Structure

---



- Sender
  - Reads the file and breaks it into a number of packets
  - Sends the packets to a receiver over the loopback interface which has the forwarding rules modified
- Receiver
  - Receives the packets; extracts data in the packets; and saves the data in a file

# Header format

---

- The following formats should be used across all parts
  - Exception: no ACK packets needed in part 1
- Data packet
  - (Sender to Receiver)

0	1	2	3 ~ up to 1026
16-bit sequence number	8-bit EoF flag	Data	

- ACK packet
  - (Receiver to Sender)

0	1
16-bit sequence number	

# Useful Links

---

- These two links are for simulators for the go-back N / selective repeat protocols that you need to implement.
  - They are one of the best ways to improve your understanding of how the algorithm works and how it handles edge cases.
- 
- [https://media.pearsoncmg.com/aw/ecs\\_kurose\\_compnetwork\\_7/cw/content/interactiveanimations/go-back-n-protocol/index.html](https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/go-back-n-protocol/index.html)
  - [https://media.pearsoncmg.com/aw/ecs\\_kurose\\_compnetwork\\_7/cw/content/interactiveanimations/selective-repeat-protocol/index.html](https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/selective-repeat-protocol/index.html)



# Useful tools: iPerf

---

- **iPerf** is a tool used to measure network performance measurement in terms of throughput and latency.

## Client

```
openair@openair-1:~$ iperf -c 192.168.4.5 -i1 -t10
-----
[ Client connecting to 192.168.4.5, TCP port 5001
TCP window size: 85.0 KByte (default)
[ 3] local 192.168.4.10 port 34562 connected with 192.168.4.5 port 5001
[ ID] Interval Transfer Bandwidth
[ 3] 0.0- 1.0 sec 11.2 MBytes 94.4 Mbits/sec
[ 3] 1.0- 2.0 sec 11.2 MBytes 94.4 Mbits/sec
[ 3] 2.0- 3.0 sec 11.1 MBytes 93.3 Mbits/sec
[ 3] 3.0- 4.0 sec 11.2 MBytes 94.4 Mbits/sec
[ 3] 4.0- 5.0 sec 11.2 MBytes 94.4 Mbits/sec
[ 3] 5.0- 6.0 sec 11.1 MBytes 93.3 Mbits/sec
[ 3] 6.0- 7.0 sec 11.2 MBytes 94.4 Mbits/sec
[ 3] 7.0- 8.0 sec 11.1 MBytes 93.3 Mbits/sec
[ 3] 8.0- 9.0 sec 11.2 MBytes 94.4 Mbits/sec
[ 3] 9.0-10.0 sec 11.1 MBytes 93.3 Mbits/sec
[ 3] 0.0-10.0 sec 112 MBytes 93.9 Mbits/sec
```

## Server

```
openair@openair-1:~$ iperf -s
-----
[ Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
```



# Useful tools: iPerf

---

```
iperf -c 192.168.4.5 -i1 -t10
```

```
iperf -c 192.168.4.5 -i1 -n 30MB
```

```
iperf -c 192.168.4.5 -i1 -F test.jpg -M 1KB
```

- **-c** → Receiver IP address
- **-i** → Interval, seconds between periodic bandwidth reports
- **-t** → time in seconds to transmit for (default 10 secs)
- **-n** → number of bytes to transmit (instead of -t)
- **-F** → input the data to be transmitted from a file
- **-M** → set TCP maximum segment size



# Useful tools: tcpdump

---

- **tcpdump** is a data-network packet analyser tool. (Command line version of **Wireshark**).
- Due to the lack of a GUI in the COMN VM, we cannot run **Wireshark** in the VM. We can however obtain the packet capture trace from inside the VM using **tcpdump**, then examine it outside the VM on our host machine using **Wireshark**.

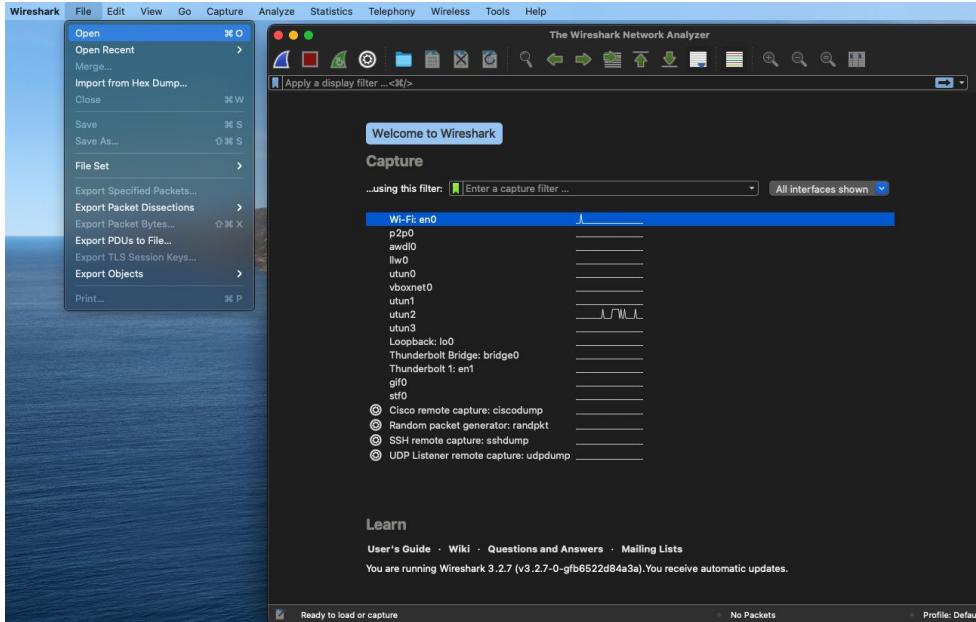
```
sudo tcpdump -i lo -w out.pcap
```

- **-i** → Interface (loopback).
- **-w** → Write the raw packets to a file.

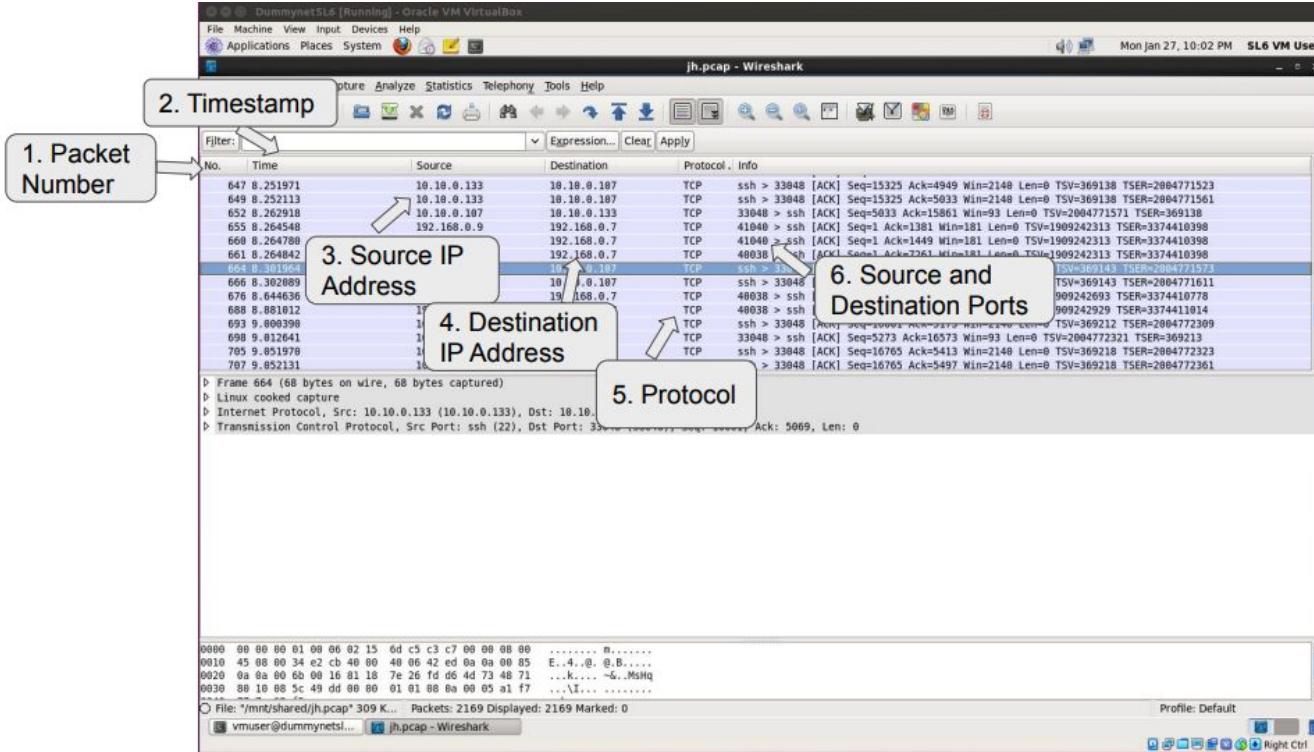


# Useful tools: Wireshark

- **Wireshark** is an open-source packet analyser tool that is used to capture network packets to understand and troubleshoot network behaviour.



# Useful tools: Wireshark



# Useful tools: Wireshark

jh.pcap - Wireshark

File Edit View Go Capture Analyze Statistics Telephony Tools Help

Filter: Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Info
647	8.251971	10.10.0.133	10.10.0.107	TCP	ssh > 33048 [ACK] Seq=15325 Ack=4949 Win=2140 Len=0 TSV=369138 TSER=2004771523
649	8.252113	10.10.0.133	10.10.0.107	TCP	ssh > 33048 [ACK] Seq=15325 Ack=5033 Win=2140 Len=0 TSV=369138 TSER=2004771561
652	8.262918	10.10.0.107	10.10.0.133	TCP	33048 > ssh [ACK] Seq=5033 Ack=15861 Win=93 Len=0 TSV=2004771571 TSER=369138
655	8.264548	192.168.0.9	192.168.0.7	TCP	41040 > ssh [ACK] Seq=1 Ack=1381 Win=181 Len=0 TSV=1909242313 TSER=3374410398
660	8.264780	192.168.0.9	192.168.0.7	TCP	41040 > ssh [ACK] Seq=1 Ack=1449 Win=181 Len=0 TSV=1909242313 TSER=3374410398
661	8.264842	192.168.0.9	192.168.0.7	TCP	40038 > ssh [ACK] Seq=1 Ack=7261 Win=181 Len=0 TSV=1909242313 TSER=3374410398
664	8.301964	10.10.0.133	10.10.0.107	TCP	ssh > 33048 [ACK] Seq=16001 Ack=5069 Win=2140 Len=0 TSV=369143 TSER=2004771573
666	8.302089	10.10.0.133	10.10.0.107	TCP	ssh > 33048 [ACK] Seq=16001 Ack=5105 Win=2140 Len=0 TSV=369143 TSER=2004771611
676	8.644636	192.168.0.9	192.168.0.7	TCP	40038 > ssh [ACK] Seq=1 Ack=7561 Win=181 Len=0 TSV=1909242693 TSER=3374410778
688	8.881012	192.168.0.9	192.168.0.7	TCP	40038 > ssh [ACK] Seq=1 Ack=7861 Win=181 Len=0 TSV=1909242929 TSER=3374411014
693	9.000390	10.10.0.133	10.10.0.107	TCP	ssh > 33048 [ACK] Seq=16001 Ack=5173 Win=2140 Len=0 TSV=369213 TSER=2004772309
698	9.012641	10.10.0.107	10.10.0.133	TCP	33048 > ssh [ACK] Seq=5273 Ack=16573 Win=93 Len=0 TSV=2004772321 TSER=369213
705	9.051970	10.10.0.133	10.10.0.107	TCP	ssh > 33048 [ACK] Seq=16765 Ack=5413 Win=2140 Len=0 TSV=369218 TSER=2004772323
707	9.052131	10.10.0.133	10.10.0.107	TCP	ssh > 33048 [ACK] Seq=16765 Ack=5497 Win=2140 Len=0 TSV=369218 TSER=2004772361

Frame 664 (68 bytes on wire, 68 bytes captured)  
Internet Protocol, Src: 10.10.0.133 (10.10.0.107)  
Transmission Control Protocol, Src Port: ssh (22), Dst Port: 33048 (33048), Seq: 16001, Ack: 5069, Len: 0  
Source port: ssh (22)  
Destination port: 33048 (33048)  
[Stream index: 1]  
Sequence number: 16001 (relative sequence number) ← 7. TCP sequence number  
Acknowledgement number: 5069 (relative ack number)  
Header length: 32 bytes  
Flags: 0x10 (ACK)  
Window size: 2140  
Checksum: 0x99dd [Verif. disabled]  
Options: (12 bytes)  
[SEQ/ACK analysis]

8. TCP window size

Frame (frame), 68 bytes Packets: 2169 Displayed: 2169 Marked: 0 Profile: Default  
vmuser@dummynets... jh.pcap - Wireshark Right Ctrl



# Miscellaneous

---

- Some useful Python libraries for the assignment:
  - sys
  - socket
  - math
  - time
  - Thread from threading
  - Lock from threading



# Miscellaneous

---

- Some useful Python libraries for the assignment:
  - sys
  - socket
  - math
  - time
  - Thread from threading
  - Lock from threading

All of these are installed by default (in Python standard library).  
**Do not install your own modules – you will FAIL.**



# Design choices for Part 3 and 4

---

- Both sender and receiver are implementable without multithreading
  - Definitely no need for multithreading at the receiver side
  - Multithreading may be useful for sender implementation
    - e.g. threads for (a) sending, (b) receiving (ACKs), (c ) checking timeouts
- Use non-blocking socket for non-multithreaded implementation
  - setblocking(0) and select()
  - Why? Whilst waiting to receive an ACK (or a timeout) you may need to send another packet
- **Many** design choices for the sender are possible



# How is CW2 marked?

---

1. We (the markers) manually run your code.
2. We check that:
  - The file is transferred correctly (byte level).
  - Your code did not crash.
  - Your code did not hang.
3. We run **tcpdump** to capture the network traffic caused by your code.
4. We use **WireShark** to check the captured packets to see if you have implemented the protocols correctly.
5. The worksheet is marked more conventionally using a marks scheme.



# Common Mistakes

---

1. Your code does not run or crashes.
  - **DO NOT INSTALL ANY ADDITIONAL PYTHON PACKAGES**
  - If in doubt, create a fresh version of the vagrant VM, and check there.
2. Your codes does not exit.
  - e.g. receiver waits for the final packet with EoF flag, that packet is lost
  - More marks lost for code that does not exit than incorrectly transferred file
3. Timeouts **are in milliseconds**
  - Not seconds!



# Less Common Mistakes

---

1. Please don't try using TCP
  - (We will notice)



# Getting Help

---

- Piazza
  - Ask general questions to the entire class
  - Ask code-specific questions to the instructors only
- Weekly Q+A Sessions
  - Will be focusing on CW2 for the next four weeks





# Questions?

Optional: slides on multithreading (*borrowed* from Operating Systems lectures)



---

# Appendix: Multithreading



# What is multithreading?

---

- Multithreading is similar to multi-processing
- A multi-processing OS can run several processes at the same time
  - Each process has its own address/memory space
  - Separate processes do not have access to each other's memory space
- In a multithreaded application, there are several points of execution  
**within the same memory space**
  - Each point of execution is called a thread
  - Threads share access to memory



# Thread support in Python

---

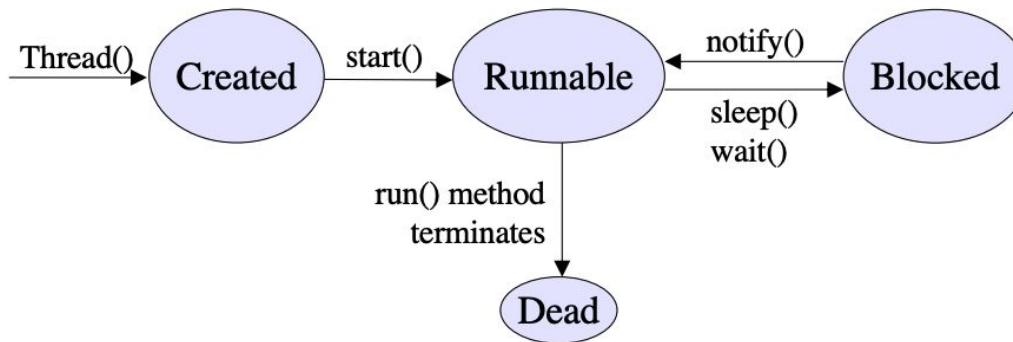
- Python threading allows you to have different parts of your program running concurrently.
- Threads are represented by a Thread object
  - A thread object maintains the state of the thread
  - It provides control methods such as start, run, sleep, join
- When an application executes, the main method is executed by a single thread
  - If the application requires more threads, the application must create them



# Threads States

---

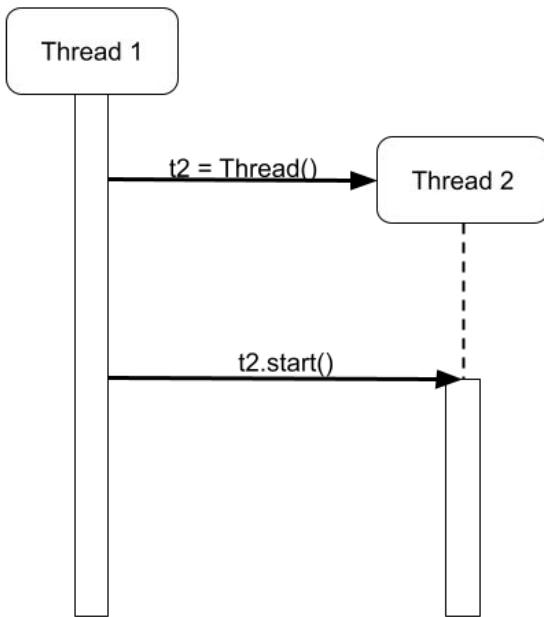
- Threads can be in one of four states as shown in the figure below
- A thread's state changes based on:
  - Control methods such as start, sleep, wait, notify
  - Termination of the run method



# How does a thread run?

---

- The thread class has a `run()` method
  - `run()` is executed when the thread's start method is invoked
- The thread terminates if the `run` method terminates
  - `run()` method often has an endless loop to prevent thread termination
- One thread starts another by calling its start method



# Creating your own thread

- The Python standard library provides **threading**, which contains all the primitives you'll need for this assignment.

```
import threading
import time

def thread_function(name):
    print('Hello from ' + name)

    for i in range(5):
        print(i)
        time.sleep(1)

if __name__ == '__main__':
    print('Before creating the thread')
    your_thread = threading.Thread(target=thread_function, args=( 'My Thread', ))
    print('Before running thread')
    your_thread.start()

    print('Wait for the thread to finish')
    your_thread.join()
    print('End')
```

```
jon@macbook:~/Desktop$ python test.py
Before creating the thread
Before running thread
Hello from My Thread
Wait for the thread to finish
0
1
2
3
4
End
```



# Synchronization: Critical Sections/Mutual Exclusion

---

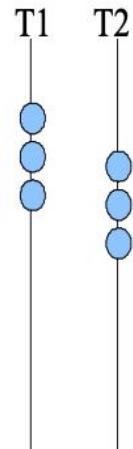
- Sequence of instructions that may get incorrect results if executed simultaneously are called **critical sections**
- We also use the term **race condition** to refer a situation in which the results depends on timing
- **Mutual exclusion** means “not simultaneous”
  - $A < B$  or  $B < A$
  - We don't care which
- Forcing mutual exclusions between two critical section executions is sufficient to ensure the correct execution – guarantees ordering
- One way to guarantee mutually exclusive executions is using **locks**



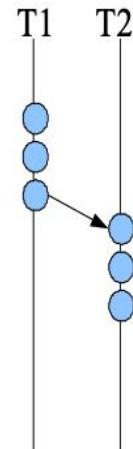
# Critical sections

---

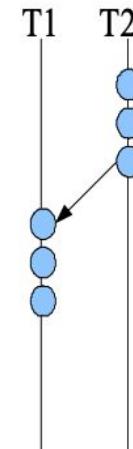
→ is the "happens-before" relation



Possibly incorrect



Correct



Correct

# When do critical sections arise?

---

- One common pattern:
  - read-modify-write of a shared value (variable) in code that can be executed concurrently
- Shared variable:
  - Globals and heap-allocated variables
  - NOT local variables (which are on the stack)



# Example: Shared bank account

---

- Suppose we have to implement a function to withdraw money from a bank account:

```
int withdraw(account, amount) {  
    int balance = get_balance(account);      // read  
    balance -= amount;                      // modify  
    put_balance(account, balance);          // write  
    spit out cash;  
}
```

- Now suppose you and your partner share a bank account with a balance of \$100.00
  - What happened if you both go to separate ATM machines, and simultaneously withdraw \$10.00 from the account?



# Example: Shared bank account

---

- Assume the bank's application is multi-threaded
- A random thread is assigned a transaction when that transaction is submitted

```
int withdraw(account, amount) {  
    int balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    spit out cash;  
}
```

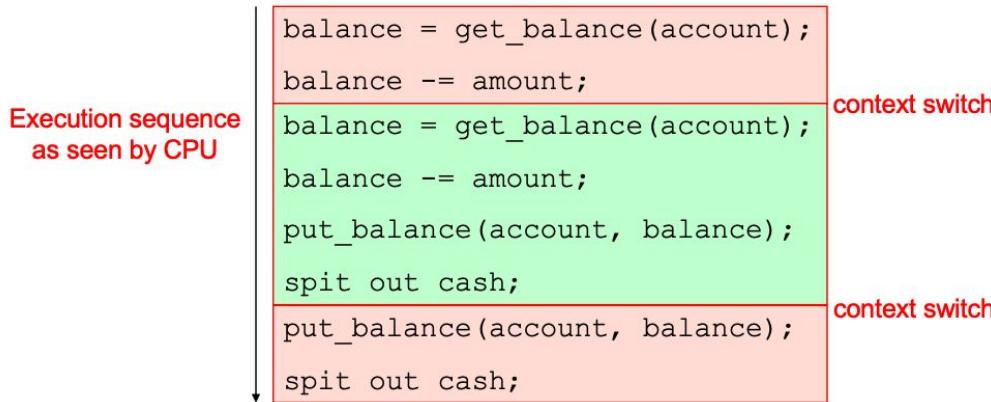
```
int withdraw(account, amount) {  
    int balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    spit out cash;  
}
```



# Interleaved schedules

---

- The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:



- What's the account's balance after this sequence?
  - Who's happy, the bank or you?

# Locks

---

- A lock is a memory object with two operations:
  - **acquire()**: obtain the right to enter the critical section
  - **release()**: give up the right to be in the critical section
- **acquire()**: prevents progress of the thread until the lock can be acquired

