



# CS 340

Daemons, Zombies, and Deadlocks  
(The spooky side of CS)

# Learning Goals

- Improve understanding of threads and other related vocabulary
- Understanding thread safety concerns
- Be able to analyze multithreading situations

# Plan for Today

- Review 
- Thread Safety
  - Deadlocks
  - Think-Pair-Share situations

Practice with Clickers (0.1% extra credit) 

# Threads Vocab I

**Thread** - Same VM, separate PC and registers

**Concurrency** - Two things both have started and neither have finished

**Parallel** - Two things making forward progress at the same time

```
1 from threading import Thread, Lock
2
3 lock = Lock()
4
5 def func(id_):
6     for i in range(500):
7         print(id_, i)
8
9
10 threads = [Thread(target=func, args=(tidx,)) for tidx in range(2)]
11 for t in threads: t.start()
12 for t in threads: t.join()
13
14 func(2)
```

Handwritten annotations:

- Red arrows pointing to lines 1, 3, 5, 6, 7, 10, 11, 12, and 14.
- Blue arrows pointing to lines 5, 6, 11, 12, and 14.
- A red box around the `func` definition (lines 5-7).
- A red box around the `func(2)` call (line 14).
- A red circle around `func` in line 10.
- A red circle around `t.start()` in line 11.
- A red circle around `t.join()` in line 12.
- A red arrow pointing down to the `range(2)` in line 10, with the text "0, 1" written next to it.
- A red arrow pointing left to the `t.join()` in line 12.

## How many threads could be running concurrently? (max)

[clicker.cs.illinois.edu](http://clicker.cs.illinois.edu)

# Q1

~Code~  
340



```

1  from threading import Thread, Lock
2
3  lock = Lock()
4  count = 0
5  def func(id_):
6      global count
7      print(id_, "running")
8      with lock:
9          print(id_, "has lock")
10         for i in range(count):
11             print(id_, i)
12             count += 1
13
14 threads = [Thread(target=func, args=(tidx,)) for tidx in range(3)]
15 for t in threads: t.start()
16 func(3)
17 for t in threads: t.join()
18 thr = Thread(target=func, args=(4,))
19 thr.start()
20
21

```

Handwritten annotations on the code:

- Red arrows pointing to lines 6 and 7, labeled "red".
- Blue arrows pointing to lines 8 and 9, labeled "blue".
- A blue circle around the number 4 in the argument of the last thread creation (line 18), with a red arrow pointing to it from the right.
- A blue circle around the list of threads in the `for` loop (line 15), with a red arrow pointing to it from the right.
- Red text "0, 1, 2" next to the list of threads.
- Red text "1" next to the thread created in line 18.
- Red text "1 2 3" and "4" next to the thread list, with a red arrow pointing to the "4" from the right.
- Red text "red, blue" at the bottom, with a red arrow pointing to the "red" label and a blue arrow pointing to the "blue" label.

How many threads could be running in parallel?  
(max)

clicker.cs.illinois.edu

Q2

~Code~  
340



```
1 from threading import Thread, Lock
2
3 lock = Lock()
4 count = 0
5 def func(id_):
6     global count
7     print(id_, "running")
8     with lock:
9         print(id_, "has lock")
10        for i in range(count):
11            print(id_, i)
12            count += 1
13
14 threads = [Thread(target=func, args=(tidx,)) for tidx in range(3)]
15 for t in threads: t.start()
16 func(3)
17 for t in threads: t.join()
18 thr = Thread(target=func, args=(4,))
19 thr.start()
20
21
```

Handwritten annotations on the code:

- Red box around line 6: `global count` with an arrow pointing to it.
- Red box around line 8: `with lock:` with an arrow pointing to it.
- Red box around lines 9-12: The loop body with an arrow pointing to it.
- Red box around line 14: `threads = [...]` with an arrow pointing to it.
- Red box around line 15: `for t in threads: t.start()` with an arrow pointing to it.
- Red box around line 16: `func(3)` with an arrow pointing to it.
- Red box around line 17: `for t in threads: t.join()` with an arrow pointing to it.
- Red box around line 18: `thr = Thread(target=func, args=(4,))` with an arrow pointing to it.
- Red box around line 19: `thr.start()` with an arrow pointing to it.

Handwritten notes:

- 1 parallel
- 4 concurrently
- 0, 1, 2
- 1, 2, 3 0

4

# Threads Vocab II

**Daemon** - A thread that isn't tied to the execution of the main program.

**Zombie** - A thread that is done running but is still holding onto resources

**Race Condition** - threads "race" to modify and access variables they share.



My code does different things everytime I run it! Which of the following could be the issue?

clicker.cs.illinois.edu

Q3

~Code~  
340



- ~~A) Multiple threads are all trying to access a local function variable at the same time.~~
- ~~B) Multiple Daemon threads are all trying to access a local function variable at the same time.~~
- ~~C) The program keeps creating zombie threads instead of releasing them.~~
- ~~D) All the code my threads run are put into critical areas through locks.~~
- E) None of the above.

My code stops making forward progress sometimes. Which of the following could be the issue?

clicker.cs.illinois.edu

Q4

~Code~  
340



- ~~A) A race condition with threads.~~
- ~~B) An infinite loop.~~
- ~~C) The program keeps creating zombie threads instead of releasing them.~~
- ☒ D) None of the above.

# Thread Safety

A system that cannot make forward progress



Deadlock - The system does not change



Livelock - The system changes but is stuck in a loop

Starvation - A part of the system never gets any resources



### Allie's Algorithm

1. get AN
2. get MW
3. Play
4. return MW
5. return AN

### Jordan's Algorithm

1. get MW
2. get AN
3. play
4. return AN
5. return MW

# Thread Safety

A system that cannot make forward progress

Deadlock - The system does not change

Livelock - The system changes but is stuck in a loop

Starvation - A part of the system never gets any resources

# Coffman Conditions

If there is a deadlock then these conditions are true

- Circular Wait

- ○ Mutual Exclusion

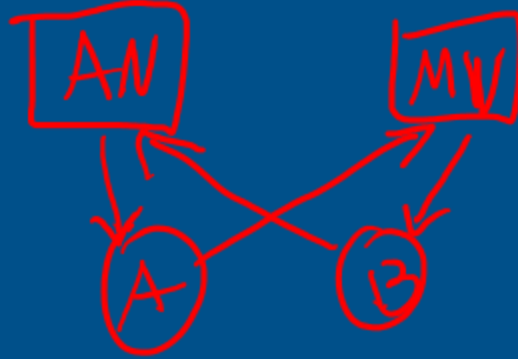
- ○ Hold and Wait

- ○ No preemption

nothing  
can force  
you to give up

→ only  
1 resource

→ once  
obtained,  
keep it







# Thread Safety

A system that cannot make forward progress

Deadlock - The system does not change

Livelock - The system changes but is stuck in a loop

Starvation - A part of the system never gets any resources



# Thread Safety

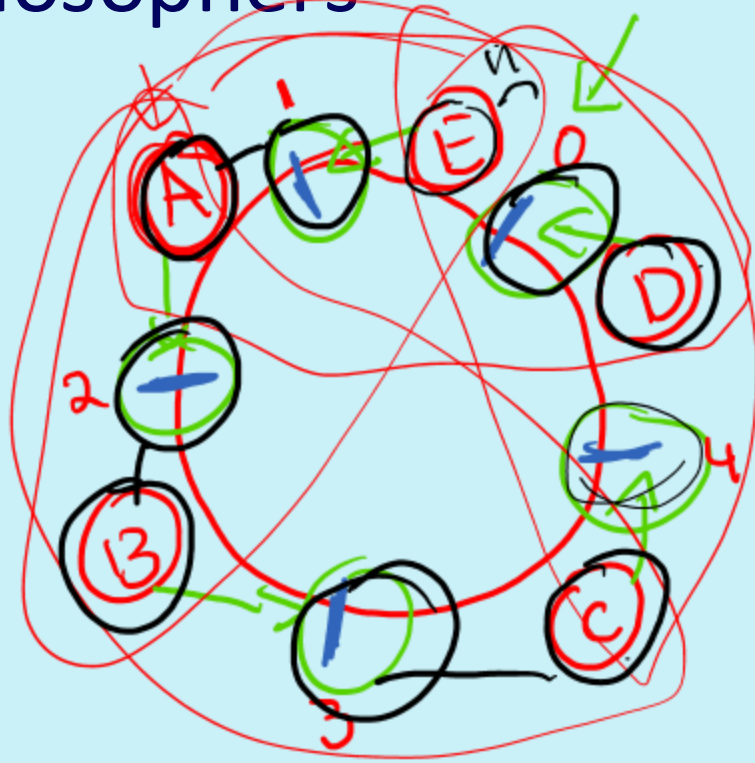
A system that cannot make forward progress

Deadlock - The system does not change

Livelock - The system changes but is stuck in a loop

Starvation - A part of the system never gets any resources

# Dining Philosophers



1. grab ~~left~~ lowest
2. grab right
3. eat
4. put down
5. think

# Dining Philosophers

# Speaker Example

Suppose you attend an event where students get to share their ideas on the main quad. The event has two tools for speakers: a microphone to help speakers be heard and a stand to help speakers be seen; each can be used by only one speaker at a time. Every speaker either wants to be seen or heard, and some want to be both seen and heard.

The event organizers are considering several policies to managing speakers, listed below.

They think a policy is **bad** if it could create deadlock with no speaker able to speak.

They think a policy is **so-so** if it isn't bad but might make people use a tool they don't want to use.

They think a policy is **good** if it ensures that everyone can speak with just the tool that they want to use.

They **don't care** if several people are talking at once, or if speakers get interrupted and have to finish their remarks later, or order people speak in.

# Speaker Example

The event organizers are considering several policies to managing speakers, listed below.

They think a policy is **bad** if it could create deadlock, with no speaker able to speak.

They think a policy is ~~so so~~ if it isn't bad but might make people use a tool they don't want to use.

They think a policy is **good** if it ensures that everyone can speak with just the tool that they want to use.

They **don't care** if several people are talking at once, or if speakers get interrupted and have to finish their remarks later, or order people speak in.

People who want the microphone and the stand must always take the microphone first.

clicker.cs.illinois.edu

Q5

~Code~  
340



# Speaker Example

The event organizers are considering several policies to managing speakers, listed below.

They think a policy is **bad** if it could create deadlock, with no speaker able to speak.

They think a policy is **so-so** if it isn't bad but might make people use a tool they don't want to use.

They think a policy is **good** if it ensures that everyone can speak with just the tool that they want to use.

They **don't care** if several people are talking at once, or if speakers get interrupted and have to finish their remarks later, or order people speak in.

If the stand is unoccupied and you want the stand, take the stand; if ~~you~~ the microphone is unheld and you want the microphone, take the microphone.

clicker.cs.illinois.edu

Q6

~Code~  
340



Bad :)



# Speaker Example

The event organizers are considering several policies to managing speakers, listed below.

They think a policy is **bad** if it could create deadlock, with no speaker able to speak.

They think a policy is ~~so-so~~ if it isn't bad but might make people use a tool they don't want to use.

They think a policy is **good** if it ensures that everyone can speak with just the tool that they want to use.

They **don't care** if several people are talking at once, or if speakers get interrupted and have to finish their remarks later, or order people speak in.

Anyone can take any unoccupied tool at any time. If you have one tool and wait for the other for a full minute, you have to release the tool you're holding and can't pick it up again until someone else has used it.

clicker.cs.illinois.edu

Q7

~Code~  
340



# Speaker Example

The event organizers are considering several policies to managing speakers, listed below.

They think a policy is **bad** if it could create deadlock, with no speaker able to speak.

They think a policy is ~~so-so~~ if it isn't bad but might make people use a tool they don't want to use.

They think a policy is **good** if it ensures that everyone can speak with just the tool that they want to use.

They **don't care** if several people are talking at once, or if speakers get interrupted and have to finish their remarks later, or order people speak in.

Anyone can take any unoccupied tool at any time. Additionally, if you have one tool and want the other tool too, you can take it from whoever has it if your UIN is a larger number than theirs.

clicker.cs.illinois.edu

Q8

~Code~  
340



preemption

# Speaker Example

The event organizers are considering several policies to managing speakers, listed below.

They think a policy is **bad** if it could create deadlock, with no speaker able to speak.

They think a policy is **so-so** if it isn't bad but might make people use a tool they don't want to use.

They think a policy is **good** if it ensures that everyone can speak with just the tool that they want to use.

They **don't care** if several people are talking at once, or if speakers get interrupted and have to finish their remarks later, or order people speak in.

You can only take the stand if you already have the microphone.

clicker.cs.illinois.edu

Q9

~Code~  
340



so-so

# Speaker Example

The event organizers are considering several policies to managing speakers, listed below.

They think a policy is **bad** if it could create deadlock, with no speaker able to speak.

They think a policy is **so-so** if it isn't bad but might make people use a tool they don't want to use.

They think a policy is **good** if it ensures that everyone can speak with just the tool that they want to use.

They **don't care** if several people are talking at once, or if speakers get interrupted and have to finish their remarks later, or order people speak in.

There's an MC who assigns speaker order. The next speaker in line takes the tools they want and then speaks.

clicker.cs.illinois.edu

Q10

~Code~  
340



# What to do if in a deadlock?

A system that cannot make forward progress

1. Analyze your threading logic outside of the code to determine the location and situation.
  - a. Remove a Coffman condition
2. Turn it off and back on again and hope for the best
3. Remove all threading



# What is coming up

- Finish HW (due 12:30pm Thursday [after spring break])
- Work on MP 6 (due 11:59pm Tuesday [after spring break])
- Read website text for more details and information
- Have a relaxing spring break!