Learning Objectives

- Identify the differences between threads and processes, and list their distinguishing characteristics.
- Recognize the differences between deadlocks, livelocks, and starvation, and classify bad concurrent interactions as one of these.

1. Deadlock, Livelocks, and Starvation

Concurrent programming enables the illusion (or reality) of multiple logical threads of execution running at the same time. However, programming with multiple flows of execution (whether or not they are truly simultaneous) can be error-prone. Three major types of failure states, *deadlock*, *livelock*, and *starvation*, can result from errors in concurrent programs. See the Appendix for a more detailed explanation of these terms.

In this section, we will explore the situations in which deadlock, livelock, and starvation occur with an emphasis on the differences between the three. The more general and universal terms *flows* and *flows of execution* are used instead of the more specific terms thread or process, since these conditions are not specific to a particular concurrent programming framework.

Problem 1. Fill in the following table with the properties of deadlock, livelock, and starvation conditions. Some boxes have already been filled in.

Property	Deadlock	Livelock	Starvation
Flows remain in waiting state	Yes	No	Some flows
Flows are executing code	No	Yes	Some flows
Flows make meaningful progress	No	No	Some flows
System can make meaningful progress	No	No	Some flows

In the following vignettes, classify each situation as an instance of deadlock, livelock, or starvation.

Problem 2. Rashmi, Abi, and Josh go to a Mexican restaurant. Each of them is too polite to begin eating until after the others do, so no one eats for several minutes after the food arrives. Is this an instance of deadlock, livelock, or starvation?

This is an instance of deadlock. Rashmi is waiting for Abi and Josh to begin eating. Abi is waiting for Rashmi and Josh to begin eating. Josh is waiting for Rashmi and Abi to begin eating. If we drew arrows to represent the wait states, they would form a loop.

This kind of deadlock can be broken by "preempting" the resource—if any two of the three diners decides to abandon politeness and begin eating, the third will be released from the deadlock.

Problem 3. Rashmi, Abi, and Josh now go to an all-you-can-eat buffet. They sit in a booth, with Josh. seated between Rashmi and Abi. Josh is too polite to ask to be let out to get more food, so he only leaves to refill his plate when Rashmi and Abi do. However, Rashmi and Abi are very speedy and can leave the booth, refill their plates, and return before Josh has a chance to get out, so Josh never gets to refill his plate. Is this an instance of deadlock, livelock, or starvation (as it relates to concurrency)?

This is an instance of starvation, because even though Rashmi and Abi are able to exit the booth and get food as needed, Josh is not able to get sufficient food because he is either not fast enough or has the misfortune to only (want to) try to get food at times where Rashmi and Abi are seated and not getting food themselves.

Problem 4. Josh makes a hearty stew for 10 TAs over for dinner at his house. The TAs are seated around a table, and decide that if a TA wants more stew, they state their desire and the stew pot is passed around the table to them. Rashmi and Abi are seated across from one another. Rashmi asks for more stew, but while the stew pot is making its way towards her, Abi requests more stew, and the stew pot begins to make its way back towards Abi. Rashmi once again requests more stew, followed by Abi, and the stew pot is passed back and forth, never reaching Abi or Rashmi. Is this an instance of deadlock, livelock, or starvation (as it relates to concurrency)?

This is an instance of livelock, because even though Rashmi and Abi are repeatedly requesting the stew pot and the TAs between them are doing the work of passing the stew pot, no meaningful progress is made and Rashmi and Abi K. are trapped, alternating between requesting the stew pot and waiting for the stew pot to arrive.

2. Advanced Deadlock, Livelock, and Starvation

We will now consider the *dining philosophers problem* to reinforce the differences between deadlock, livelock, and starvation.

In the dining philosophers problem, five philosophers are seated around a circular table. Between each philosopher and their neighbor is a single chopstick, so that there are five chopsticks on the table and each philosopher has a left chopstick and a right chopstick. Philosophers alternate between thinking and eating. When a philosopher wishes to begin eating, they must acquire both their left and their right chopstick before beginning to eat. When a philosopher has finished eating, they return both chopsticks to the table. However:

- Philosophers can only acquire and release one chopstick at a time (i.e. these philosophers are not coordinated enough to simultaneously grab or return both their right and left chopsticks).
- Philosophers cannot acquire chopsticks that are already being held or used by their neighbors.
- A philosopher must have both chopsticks to eat — it is impossible to eat with only one chopstick.

Problem 5. Rashmi, Abi, Josh, and two other TAs go to a Chinese restaurant. Unfortunately, this restaurant has only five chopsticks for the five of them. At first, the TAs decide on the following algorithm to be carried out by each TA individually:

- If left chopstick is available, acquire it. If not, wait until it is and acquire.
- If right chopstick is available, acquire it. If not, wait until it is and acquire.
- Eat.

- Return left chopstick.
- Return right chopstick.

Describe how deadlock and starvation can occur. Can livelock occur? If it can, why? If not, what is missing?

Deadlock can occur if all TAs acquire their left chopstick at the same time. Each TA holds a single chopstick and waits for the other chopstick, which is currently held by another waiting TA. Starvation can occur if the TAs neighboring a particular TA hold their chopsticks for long periods of time, or only set down their chopsticks for brief periods before picking them back up. Livelock cannot occur, because TAs never "back off" — there is no way for TAs to, for example, get stuck in a state of acquiring a chopstick and then returning a chopstick.

Problem 6. The TAs surmise that the problem with this algorithm is that chopsticks cannot be released before eating if the other chopstick is not available. They propose the following modified algorithm:

- If left chopstick is available, acquire it. If not, wait until it is and acquire.
- If right chopstick is available, acquire it. If not, return left chopstick and restart.
- Begin eating.
- Return left chopstick.
- Return right chopstick.

Can this algorithm deadlock and/or livelock? If so, describe how. If not, argue why not. Is there a possibility for starvation?

This algorithm can no longer deadlock due to the ability for TAs to return chopsticks if they discover that they are unable to acquire their right chopstick — there is no "hold and wait", one of the necessary ingredients for deadlock. However, the algorithm can still livelock. One simple livelock situation can occur if all TAs acquire their left chopstick, realize that their right chopstick is taken, return their left chopstick, and so on. There remains a possibility for starvation, in the same way as the previous problem where one TA is starved by their neighboring TAs holding chopsticks for long periods of time and/or setting them down for brief periods of time.

Problem 7. Rashmi comes up with another solution to this problem: Each TA will be numbered 1-5. Odd-numbered TAs try to acquire their left chopstick before their right chopstick (waiting if necessary), and even-numbered TAs try to acquire their right chopstick before their left (waiting if necessary).

Can this algorithm deadlock and/or livelock? If so, describe how. If not, argue why not. Is there a possibility for starvation?

This algorithm can no longer deadlock, since circular waiting (another ingredient of a deadlock) is not present. There are not any "cycles" where TAs are holding chopsticks and waiting for resources held by TAs holding chopsticks and waiting for the chopsticks held by the first TAs.

This algorithm can also not livelock, since there are no mechanisms for leaving a waiting state besides acquiring a desired chopstick.

Starvation can occur as described in Problem 6.

Problem 8. Abi proposes the following solution: we will relax one assumption we initially made. Instead of TAs mild-mannerdly waiting for chopsticks, TAs can acquire said missing chopstick by forcibly grabbing a chopstick even if another TA is holding said chopstick! Once a TA acquires both chopsticks, the other TAs will admit defeat and the dual wielding chopsticks TA can then begin eating in peace.

Can this algorithm deadlock and/or livelock? If so, describe how. If not, argue why not. Is there a possibility for starvation?

This algorithm cannot deadlock, since preemption (grabbing the chopstick) prevents waiting from occurring.

This algorithm however can livelock. Pairs of TAs can repeatedly grab the chopstick in between from each other.

Starvation can occur as described in Problem 6.

Problem 9. *(Advanced)* Josh claims that the problem all along is that multiple TAs can carry out the chopsticks-acquisition procedure at the same time. He proposes that they follow the first algorithm, but with a catch: only one TA is allowed to attempt to acquire chopsticks at a time. This will be enforced by a single "lucky cat" figurine as follows:

- If lucky cat is available, acquire it. If not, wait.
- If left chopstick is available, acquire it. If not, return lucky cat and wait. If waiting, when left chopstick becomes available, attempt to acquire lucky cat, waiting if necessary. Then, acquire left chopstick.
- If right chopstick is available, acquire it. If not, return lucky cat and wait. If waiting, when right chopstick becomes available, attempt to acquire lucky cat, waiting if necessary. Then, acquire right chopstick.
- Return lucky cat and begin eating.
- Return left chopstick.
- Return right chopstick.

At first, it appears that the lucky cat solution has solved the problem. When a TA wishes to eat, they acquire the lucky cat, then they can acquire both chopsticks uninterrupted (if they are available). It appears that this TA dinner party has fixed the "everybody reaches left" problem. Can this algorithm still deadlock and/or livelock? If so, describe how — be precise! If not, argue why not.

2in This is not the case! There is still a possibility of deadlock, as follows: First, one TA acquires the lucky cat, both chopsticks, and begins to eat, returning the lucky cat. Then, the TA to their left acquires the lucky cat, their left chopstick, and waits on their right chopstick, currently held by the first TA, after returning the lucky cat. The TA to that TA's left likewise acquires their left chopstick and waits on the right chopstick, all around the table. The TA to the right of the first TA simply waits on their left chopstick. Now, when the first TA finishes eating, there are two scenarios — the longest-waiting TA could acquire their other chopstick and begin eating (avoiding deadlock for now), or the shortest-waiting TA could acquire their left chopstick. In the second situation, if the first TA acquires their left chopstick and waits on the right chopstick, deadlock occurs.

Livelock still cannot occur since there is no way for TAs to repeatedly alternate between a waiting state and a non-waiting state without progress (in some sense) being made.

Problem 10. *(Advanced)* You may have noticed that none of these algorithms address starvation. Propose a change to the algorithm that would prevent (or at least reduce) starvation of TAs.

One possible change to the algorithm that would help combat starvation is to have a ticket system, where each TA takes an ascending number and must check that neither neighbor holds a lower number than their number before beginning to eat. If a neighbor has a lower number, then the current TA attempting to acquire chopsticks must replace any chopsticks they already hold.

3. Threads and Processes

Processes and threads are abstractions used to capture logical control flows in concurrent programming. However, the terms *process* and *thread* are not interchangeable. Keep in mind that there are varying terminologies for these abstractions — the distinction we discuss here corresponds most closely to the Unix paradigm.

A *process* is an instance of a program, containing the resources needed to execute a program. Such resources include a separate address space, executable code sections, open handles to system objects (such as opened file descriptors), a unique process identifier, arguments and environment variables, and at least one thread.

A *thread*, on the other hand, is an entity within a process that can be scheduled for execution. All threads of a process share the process's virtual address space and system resources. Each thread maintains its own exception handlers, scheduling priority, local storage, unique thread identifier, and set of structures the system will use to save the thread context when the thread is switched out.

Each process is started with a single thread. Any thread in a process can create additional threads within the same process.

Problem 11. Fill in the following table with the different properties of processes and threads. Some boxes may require more than a simple yes or no answer.

Property	**Process**	**Thread**
Share address space	No	Yes
Share file descriptors opened before `fork` or `pthread_create`	Yes	Yes
Share file descriptors opened after `fork` or `pthread_create`	No	Yes
Share local non-static variables	No	If pointers are passed between threads
Can be scheduled	Technically, no. The threads within a process are scheduled, but processes themselves are not schedulable entities. However, every non-zombie process contains at least one thread, so it often *appears* as though processes are being scheduled.	Yes

Problem 12. Consider the following two implementations of a short program. Implementation A forks two processes which each increment and print a local, non-static variable while Implementation B creates two threads which each increment and print a local, non-static variable.

Implementation A

```
#include <stdio.h>

void *inc_print(void *arg) {
  (*(int *)arg)++;
  printf("%d ", *(int *)arg);
  exit(0);
}

int main() {
  pid_t first, second;
  int shared = 0;
  if ((first = fork()) == 0) {
    inc_print((void *)&shared);
  }
  if ((second = fork()) == 0) {
    inc_print((void *)&shared);
  }
  waitpid(first, NULL, 0);
  waitpid(second, NULL, 0);
}
```

Implementation B

```
#include <stdio.h>
#include <pthread.h>

void *inc_print(void *arg) {
  (*(int *)arg)++;
  printf("%d ", *(int *)arg);
  pthread_exit(0);
}

int main() {
  pthread_t first, second;
  int shared = 0;
  pthread_create(&first, NULL,
    &inc_print, (void *)&shared);
  pthread_create(&second, NULL,
    &inc_print, (void *)&shared);

  pthread_join(first, NULL);
  pthread_join(second, NULL);
}
```

What could be printed by Implementation A? What about by Implementation B? Will the output of the two programs be the same, or different? Explain why.

Implementatio A, using processes, can only print "1 1 " because the call to fork the child processes have completely different virtual address spaces — therefore, the local variable is not shared. Implementation B, using threads, can print either "1 2 " or "2 1 " because threads operate in the same (process's) virtual address space — the local variable is indeed "`shared`".

Problem 13. Would there be any difference in the possible output if shared were declared as a global variable instead? Why might this be?

There would be no difference in possible output, because processes still do not share global variables due to not sharing an address space while threads will still share global variables due to sharing the same process memory.

Problem 14. Implementation A′ and B′ below instead declare the integer to be printed inside the `inc_print` function.

Implementation A′

```
#include <stdio.h>

void *inc_print(void *arg) {
  int maybe_shared = 0;
  maybe_shared++;
  printf("%d ", maybe_shared);
  exit(0);
}

int main() {
  pid_t first, second;
  if ((first = fork()) == 0) {
    inc_print(NULL);
  }
  if ((second = fork()) == 0) {
    inc_print(NULL);
  }
  waitpid(first, NULL, 0);
  waitpid(second, NULL, 0);
}
```

Implementation B′

```
#include <stdio.h>
#include <pthread.h>

void *inc_print(void *arg) {
  int maybe_shared = 0;
  maybe_shared++;
  printf("%d ", maybe_shared);
  pthread_exit(0);
}

int main() {
  pthread_t first, second;
  pthread_create(&first, NULL,
    &inc_print, NULL);
  pthread_create(&second, NULL,
    &inc_print, NULL);

  pthread_join(first, NULL);
  pthread_join(second, NULL);
}
```

What do you expect will be printed by Implementation A′ and Implementation B′? If this output is different from before, why?

Implementation A′ will still print "1 1 ", but Implementation B′ can now only print "1 1 " as well because the variable `maybe_shared` is local to each thread — a different `maybe_shared` is declared on each thread stack, and each thread has its own "copy" of `maybe_shared` on its thread stack.

A. Appendix: Deadlocks, Livelocks, and Starvation

Deadlock occurs when one or more flows of execution are waiting for resources held by other flows of execution, which are in turn waiting for resources held by the initial (or even more) flows of execution, often in a cyclical manner. All involved flows of execution are thus trapped in a waiting state. No code (besides possibly to check and confirm waiting conditions) is executed, and no flows make progress.

Livelock occurs when two or more flows of execution continually repeat the same interaction in response to changes in the other flows without doing any useful work. Although these flows may enter and leave waiting states (and are "running concurrently" or even "making progress" in the sense that they are executing code), no real progress is made since these flows are trapped in an unfavorable interaction. Livelock differs from deadlock in that livelocked flows are trapped in repeating a particular interaction while deadlocked flows are trapped in waiting for other flows.

Finally, ***starvation*** occurs when one or more flows may wait indefinitely because other flows repeatedly and "unfairly" acquire a contested resource. While other flows (and perhaps even the system as a whole) may make progress, the starved flows remain in a waiting state, unable to progress due to repeatedly losing in the contest to acquire a particular shared resource.