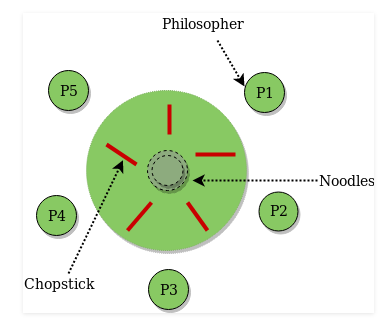
Parallel and Concurrent Programming with Java

## Chapter 4. Liveness

# Deadlock

Dining Philosophers Problem

5 (or any number N) philosophers sitting around a round table.

1 chopstick between each philosopher.

Every philosopher can only do 2 things (states): think or eat.

All philosophers are independent.

To eat, a philosopher needs 2 chopsticks to pick up food. He must acquire his left chopstick. Then, his right chopstick, seperately.

If a chopstick is available, the philosopher will acquire a lock on that chopstick.

A chopstick can only be acquired by 1 philosopher at a time.

1. Philosopher = a thread. The group of philosophers eating = process.
2. Chopstick = a mutex lock
3. The food in the middle of the table = the shared resource.
4. Action of taking food = coded by critical section

An illustration of limitations in process synchronization in multi-thread environment and how to solve them.

Problems:

1. Data race (not applicable – have mutex locks)
   * 2 or more threads (philosophers) in a single process concurrently trying to access shared resource (food).
   * At least one of the threads (philosophers) will change the shared resource (take food).
   * There are no exclusive mutex locks to control access to the shared memory.
2. Deadlock
   * Two or more threads are blocked because they are waiting forever for each other.
   * Code designs that encourage deadlock:
     + Circular chain. Thread waits for lock held by next thread in chain. Example: If all philosophers (threads) simultaneously acquire the chopstick (mutex lock) on their left, then wait to acquire the chopstick on their right, they will be blocked forever. No progress posssible.
     + Timing. Requests for new locks are made concurrently by different threads. OS happens to schedule threads to run at same time.
     + Multiple locks. Threads that are holding locks request for new locks.
3. Starvation
   * Did not get to eat.
   * Unable to access shared resource.
4. Efficiency
   * increase number of philosophers eating simultaneously (concurrency), minimize waiting/thinking.

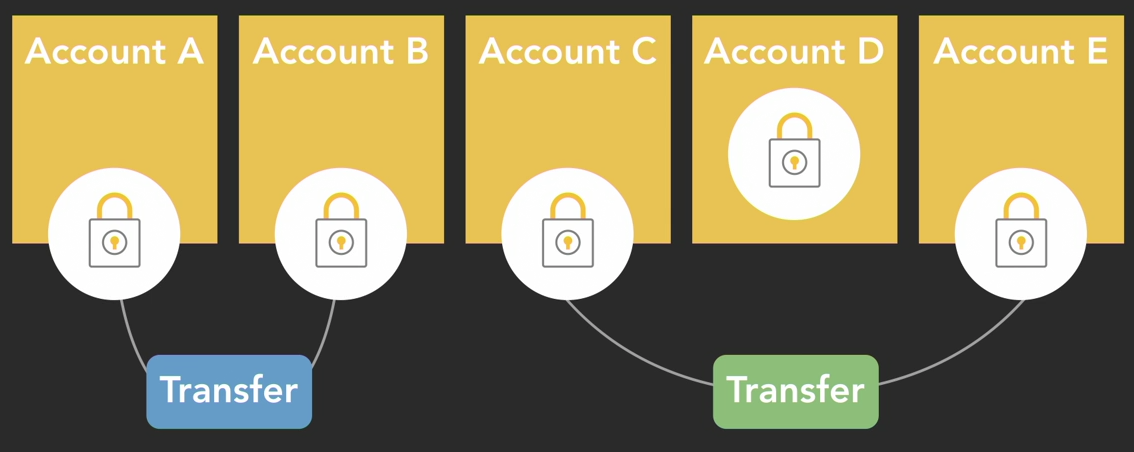
Want: Liveness

* concurrent system/process needs to make progress
* members/threads need to “take turns” in critical section

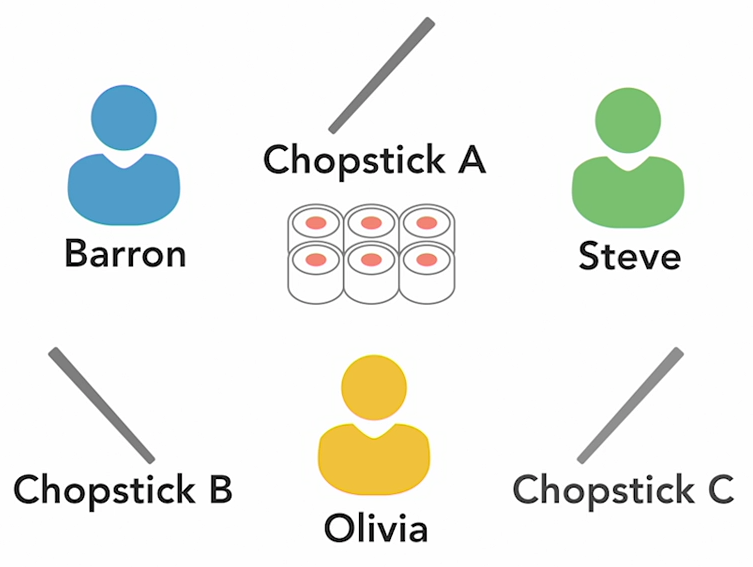
Solutions

1. define priority/order of acquiring mutex locks.
   * Locks are always acquired in the same order by any thread.
   * Problem: thread may not know all locks it needs to acquire in advance.

* Example: banking application
  + every account has its own mutex lock to ensure only 1 thread can modify its value (debit or credit) at any time.
  + Fund transfer
    - involves changing amount (memory record) in sending and receiving accounts.
    - involves acquiring locks on both (sender and receiver) accounts.
  + ? lock on receiving account before lock on sending account
  + When have multiple concurrent threads for transfer operation, have real risk of deadlock.



2. Timeout on locking attempts.

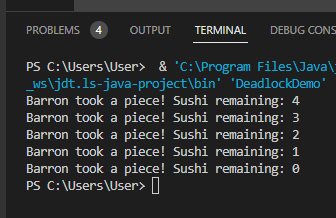
* If cannot acquire all required locks within a time limit, will have to
  + release all locks taken
  + wait for a random amount of time
  + try again.

Deadlock: Java Demo

*src*/DeadlockDemo.java

Scenario for Java demo program.

* 3 philosophers.
* 3 chopsticks.



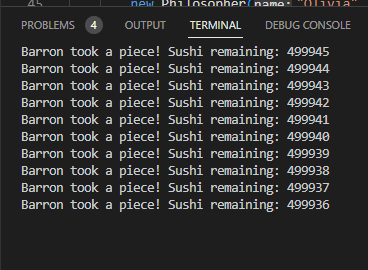
Concurrency problems are hard to discover.

Example 1:

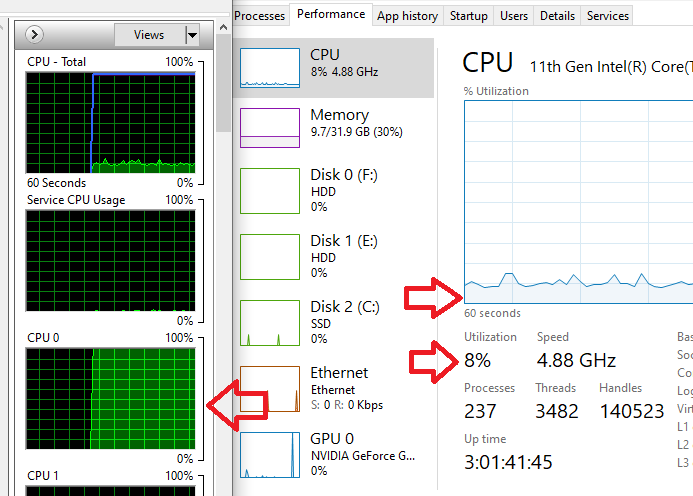
When the number of iterations is low (sushiCount = 5),

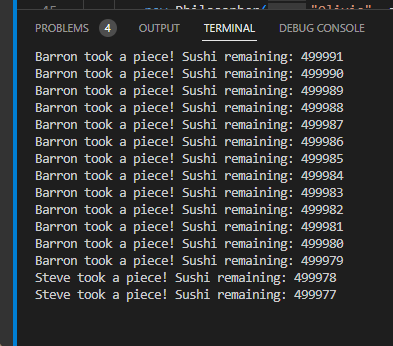
* only 1 thread (philosopher Barron in example here) accessed the critical section.
* Other threads starved.

Example 2:

When the number of iterations is high (sushiCount = 500\_000),

* deadlock
* Threads wait on each other, not actively executing codes.
* CPU cores not busy for this program/process.



Example 3:

Concurrency problems are hard to reproduce from run to run.

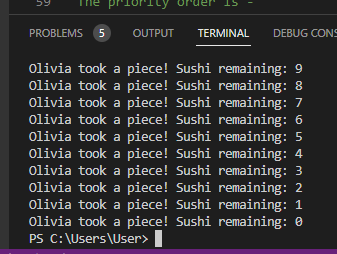
* Because threads are scheduled to run by the scheduler according to the OS’s algo every time.
* Not easy to debug.
* In screengrab, notice that threads (philosopher) involved is different from previous run (above screengrab), and sushiCount stops after a different number of iterations.

Solution to deadlock:

When the mutex locks (chopsticks) are acquired in order of priority, the deadlock is resolved.

Priority order:

ChopstickA > chopstickB > chopstickC

However, note

1. priority ordering does not even out access to shared resource by threads. 1 thread (olivia) dominates.
2. A thread may not know all the locks it needs to acquire before it starts executing. Code may have conditional logic that alters number of locks to be acquired later. May not be possible to order locks correctly then.

If check task manager when run DeadlockDemo.java, will see spike in CPU utilisation.

Note that for 1 shared resource/critical section (sushiCount), only 1 mutex lock (chopstick) is needed to protect it. The extra 2 chopsticks are just to reproduce dining philosophers problem.

Readings:

MIT software construction course

[https://web.mit.edu/6.005/www/fa15/classes/23-locks/#introduction](https://web.mit.edu/6.005/www/fa15/classes/23-locks/" \l "introduction)

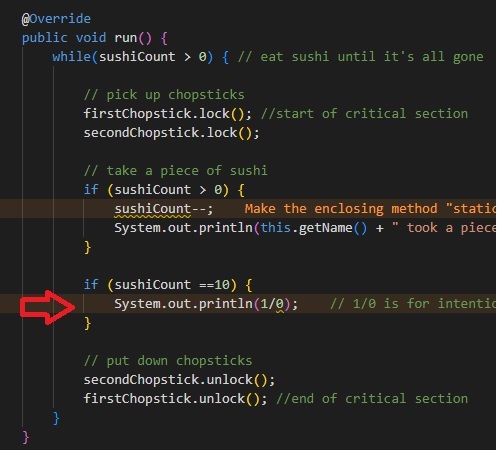
The Art of Multiprocessor Programming.

[https://books.google.com.sg/books?id=qGURkdAr42cC&lpg=PA157&dq=abandoned%20locks%20java&pg=PR12#v=onepage&q=abandoned%20locks%20java&f=false](https://books.google.com.sg/books?id=qGURkdAr42cC&lpg=PA157&dq=abandoned locks java&pg=PR12" \l "v=onepage&q=abandoned locks java&f=false)

Oracle Berkeley DB - Java Locks, Blocks and Deadlocks

https://docs.oracle.com/cd/E17276\_01/html/gsg\_xml\_txn/java/blocking\_deadlocks.html

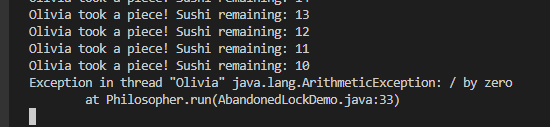
Abandoned Lock

If a thread unexpectedly terminates after acquiring a lock, the lock may be stuck in the locked state, not automatically released.

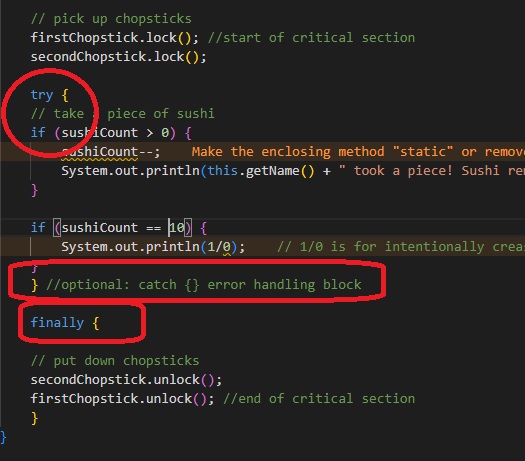
* Other waiting threads blocked.

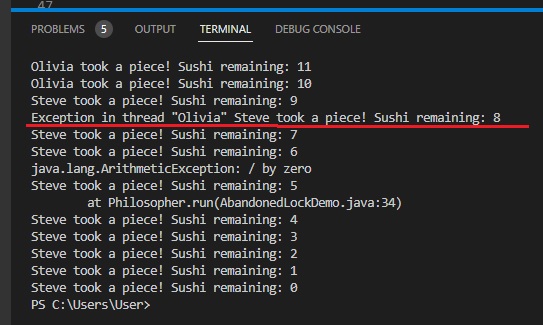
To crash a thread in demo: 1/0 math calculation.

* Olivia’s thread crashed.



To avoid abandoned lock/unexpected error, always use try-catch-finally block.

* Critical section in try block.
* Exception handling in catch block.
* Lock.unlock() in finally block

Olivia’s thread crashes when sushiCount == 10. try-catch-finally block for critical section allows other threads to continue until program ends.

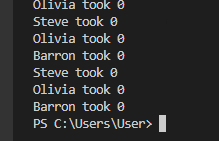
Starvation

OS Scheduler decides when each thread gets to execute on processor based on its own algo.

* By default, not all threads will have equal access.

Starvation occurs when a thread is consistently unable to gain access to shared resource, and cannot make progress.

1. Lower priority threads less likely to be scheduled for execution
   * exact handling depends on OS
2. When there are many competing concurrent high-priority threads, some can starve too.

*/src*/StarvationDemo.java

Dining Philosophers supposed to take food. They take none.

* threads starved.

LiveLock

Multiple threads are not making useful progress.

But the threads are actively executing codes.

* Their states keep changing, e.g. lock -> unlock -> lock -> unlock.
* The constantly changing state blocks progress.
* In a deadlock, the states are passive, waiting for release of mutex lock.

Program/Process does not reach completion.

Cause: Often, from codes meant to detect and recover from deadlocks.

How to avoid: Make sure that one thread takes action before others.

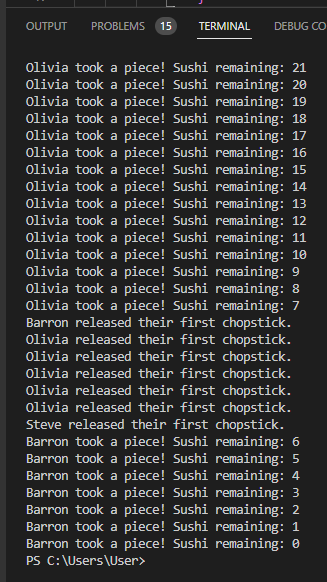
Example, by priority assignment, random selection.

How to detect: Look at CPU utilisation

deadlock: low/no %

livelock: high %, drops the moment ^C.

LiveLock: Java Demo

Refer to comments in /src/LivelockDemo.java

circular chain mutex locks – deadlock

trylock for 2nd lock & unlock 1st lock on failure – livelock

trylock for 2nd lock, unlock 1st lock and thread.sleep() for random duration – program does proceed to completion.

Random Generator

Use for thread.sleep(), so that threads do not try to enter critical section/lock at same time.

Package java.util.Random

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/random/package-summary.html

classes for “random generator”.

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Random.html

Individual values of types int, long, float, double, boolean, i.e. not just numbers.

Be careful when implementing in multi-thread environment: multiple instance of Random class, using same seed and same sequence of method calls, will generate and return identical sequence of numbers.

java.security.SecureRandom

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/security/SecureRandom.html

multi-thread safe, cryptographically secure