Concurrent Programming

The material in this lecture will not be on a quiz or a lab.

Concurrent programming is a form of programming in which several computations are executed during overlapping time periods—concurrently—instead of sequentially (one completing before the next starts).

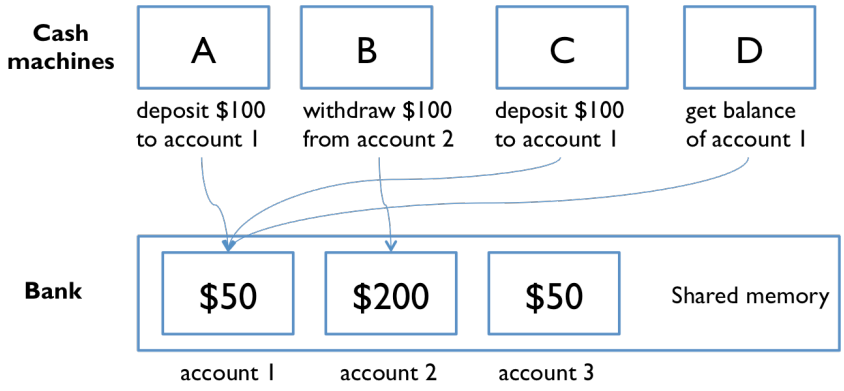
In concurrent programming, there is a separate execution point or "thread of control" for each computation ("process" or “thread”). Concurrent processes can be executed on one processor by interleaving the execution steps of each process via *time-sharing* slices: only one process runs at a time, and if it does not complete during its time slice, it is paused, another process begins or resumes, and then later the original process is resumed. In this way, multiple processes are partway through execution at a single instant, but only one process is being executed at that instant.

Not to be confused with parallelism, concurrency is when multiple sequences of operations are run in overlapping periods of time. Concurrent computations *may* be executed in parallel, for example, by assigning each process to a separate processor but we will not consider that here, and Python does not support parallel programming natively.

As a programming paradigm, concurrent computing is a form of modular programming, namely factoring an overall computation into subcomputations that may be executed concurrently.

Transferring Cash Example

It is easiest to describe concurrent programming with an example. Consider a bank server that maintains accounts and balances associated with accounts. There are Automated Teller Machines (ATMs) worldwide that connect to the server and make transaction requests corresponding to deposits, withdrawals and transfers as shown below.



We will focus on transfers of money from one account to another. There could be multiple transfers "live" at a given point of time. Of course, all of these transfers have to run on the bank server and they could run concurrently or serially as shown below. Assume that through an ATM a request is made to transfer money from account B to account A. At the same time, from a different ATM, another request is also made to transfer money from account B to A. Both of these are legitimate requests and will be run on the bank server, which is the only computer with access to the bank accounts.

Initially, account A has $0 and account B has $1000. Transfer 1 wants to transfer $500 from B to A. Transfer 2 wants to transfer $200 from B to A. Regardless of the order in which these transfers are processed, at the end we want A to have $700 and B to have $300. Obviously, at the end of the two transfers we want the total of A + B to equal $1000. Transfers neither spend money nor create money – we will assume no fees are involved.

Threads in Python

There are two separate processes or threads corresponding to transfers 1 and 2 running on the bank server concurrently. We will model these as shown in **race.py**. Here, we are running many requests each with random (potentially negative) amounts all transferring money from B to A. Run it and see what happens!

We are creating money and losing money. Why is this happening? Here is an interleaving that loses $200 across the two accounts.

```
A = 0
B = 1000

T1: newA = 500

T2: newA = 200
T2  newB = 800
T2: A = 200
```

```
T2: B = 800

T1: newB = 800 - 500 = 300
T1: A = newA = 500
T1: B = newB = 300
```

Note that `newA` and `newB` are variables that are local to each thread. That is, there are two `newA`'s in the two threads that are unrelated, similarly `newB`. However, `A` and `B` are global variables, i.e., they have scope *across* the two threads. They need to be global because we want a single value for a bank account balance. Thread `T1` runs and sets local `newA` to be `500`. Then, Thread `T2` runs to completion. Unfortunately, when Thread `T1` resumes it does not use the correct value of `newA`. The `newA` in Thread `T1` is based on a stale value of the global balance `A`, namely `0`, which has since been updated to `200` by Thread `T2`.

Races

A **race condition** or race means that the correctness of the program (the satisfaction of postconditions and invariants) depends on the relative timing of events in concurrent computations X and Y. When this happens, we say "X is in a race with Y." Threads `T1` and `T2` are racing to read and/or write global variables. If one of them allowed the other to start and finish, we would not have this issue.

Some interleavings of events may be OK, in the sense that they are consistent with what a single, nonconcurrent process would produce, but other interleavings such as the example above produce wrong answers – violating postconditions or invariants.

Fixing Races with Locks

How do we remove races from concurrent execution? One of the best ways to avoid a race condition in software and hardware applications is the use of **mutual exclusion**, which assures that only one process can handle the shared resource at a time, while other processes need to wait. We can ensure exclusive access through the use of locks.

Given a lock `lock`, two methods `lock.acquire()` and `lock.release()` ensure that at most one thread is in its section of code that modifies the global variables `A` and `B` as shown in **race-fixed.py**. This ensures that the total `A + B` is always `1000`; no money is lost or created when multiple transfers are concurrently executing. Essentially the lock ensures that once a thread enters the critical section where the global variables are being accessed, the other thread has to wait till the first thread's transfer has finished execution.

There is still a small problem associated with the execution. Every once in a while, two `print` statements are garbled in the sense that the newline of the first `print` is

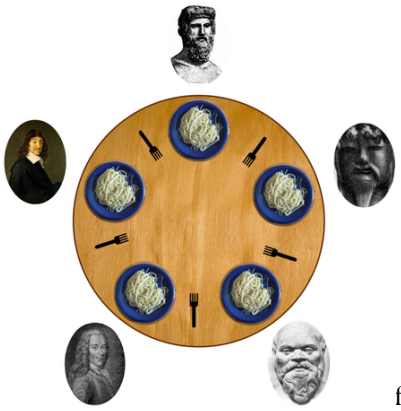
not printed before the total corresponding to the second `print` is printed, and then two newlines are printed back to back. This can be fixed by moving the `lock.release()` statement to after the `print` statement as done in **race-fixed-print.py**.

Dining Philosophers Example

Five philosophers sit at a round table with bowls of spaghetti as shown below. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Only one philosopher can hold each fork and so a philosopher can use the fork only if another philosopher is not using it. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks. We will assume that philosophers have infinite sized stomachs and there is an infinite supply of food.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.



from Wikipedia

Simulating Dining Philosophers

We'll number the philosophers 0 through 4 in counterclockwise order. We will also number the forks 0 through 4, with fork i to the left of philosopher i. The fork to the right of philosopher is then (i +1) % 5.

Consider a solution in which each philosopher is instructed to behave as follows:

- Think until the left fork is available; when it is, pick it up;
- Think until the right fork is available; when it is, pick it up;
- When both forks are held, eat for a fixed amount of time;
- Then, put the right fork down;
- Then, put the left fork down;
- Repeat from the beginning.

This algorithm is simulated in **dine.py**. Not surprisingly, since there are no locks in the implementation, when the simulation is run two philosophers pick up and eat with the same fork at the same time (yuck)!

Adding Locks to Dining Philosophers

To fix the problem of multiple philosophers eating with the same fork, we add locks. The code is shown in **dine-lock.py**.

This attempted solution fails because it allows the system to reach a **deadlock** state, in which no progress is possible. This is a state in which each philosopher has picked up the fork to the left, and is waiting for the fork to the right to become available. With the given instructions, this state can be reached, and when it is reached, the philosophers will eternally wait for each other to release a fork.

Avoiding Starvation Using Ordering

Dijkstra originally proposed one solution to the problem. It assigns a partial order to the resources (the forks, in this case), and establishes the convention that all resources will be requested in order, and that no two resources unrelated by order will ever be used by a single unit of work at the same time. Here, the resources (forks) will be numbered 0 through 4 and each unit of work (philosopher) will always pick up the lower-numbered fork first, and then the higher-numbered fork, from among the two forks they plan to use. The order in which each philosopher puts down the forks does not matter. In this case, if four of the five philosophers simultaneously pick up their lower-numbered fork, only the highest-numbered fork will remain on the table, so the fifth philosopher will not be able to pick up any fork. Moreover, only one philosopher will have access to that highest-numbered fork, so they will be able to eat using two forks.

The file **dine-order.py** contains Dijkstra's solution that uses ordering to ensure that the philosophers don't starve.

Assuming that philosophers eventually lose interest in eating, and there is an infinite supply of spaghetti, all philosophers are happy! However, in the general case, there needs to be a way of ensuring that all philosophers are able to eat with

the same regularity – i.e., they all have the same opportunity to eat. We will not tackle fairness issues in 6.009.

How Do Locks Work?

We will now describe how locks work. There are many ways that locks are implemented in modern programming languages and on different processor architectures. We will describe the simplest way of implementing locks, which does bear some conceptual resemblance to the way Python implements locks when it is run on modern processors. This way of implementing locks uses Peterson's algorithm and is implemented in **busylock.py**.

We construct a class called `BusyLock`. It uses two dictionaries, namely, `level` and `lastToEnter`. The `level` dictionary maps thread id's to level numbers. The `lastToEnter` dictionary keeps a record of the last thread that wants to enter a level `t`.

To acquire a lock, the thread will try to occupy the levels from `0` to `N-1` sequentially. For the level k, it will first set its level to k and set `lastToEnter[k]` to be its `id`. (The order here is important.) Then, it starts to frequently check if it can proceed to the next level. It waits in the current level until `lastToEnter[k] !=` its own `id`, or everyone else's level is smaller than k. If the check fails, the thread sleeps for some time and tries again.

To release a lock, the thread simply sets its level to be `-1`. In the code, we reset the whole level dictionary for simplicity (because the monitor might need to reset the state).

The code uses a static integer (shared across all instances) `N` to keep track the total number of locks. And in our dinning philosophers the number of locks equals the number of philosophers. This is not always true in other problems. We use this fact to make our `BusyLock` look more like a `threading.Lock()`.