**Problems of Concurrency**

*The concurrency problems arise from the scenario of concurrent computing, where the execution of a program is conducted in multiple processes (or threads) simultaneously.*

By simultaneousness, the processes or threads are not necessarily running independently in different physical CPUs, but more often they interleave in the same physical CPU. Note that, the concurrency could apply to either process or thread, we use the words of "process" and "thread" interchangeably in the following sections.

The concurrency is designed to above all enable multitasking, yet it could easily bring some bugs into the program if not applied properly. Depending on the consequences, the problems caused by concurrency can be categorized into three types:

* **race conditions**: the program ends with an undesired output, resulting from the sequence of execution among the processes.
* **deadlocks**: the concurrent processes wait for some necessary resources from each other. As a result, none of them can make progress.
* **resource** starvation: a process is perpetually denied necessary resources to progress its works.

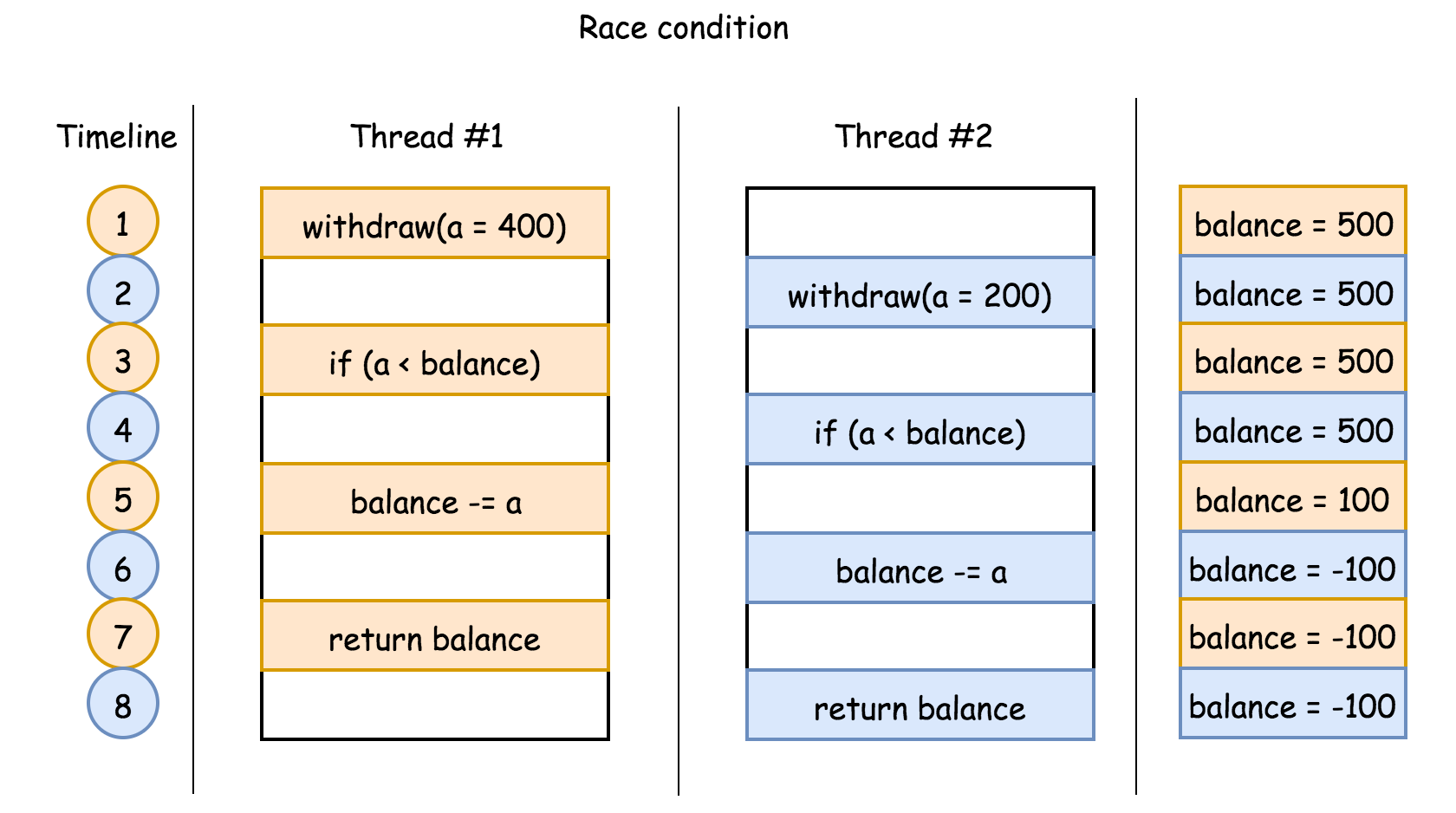
In particular, our problem here can be attributed to the race conditions. Before diving into the solutions, we show an example of race condition.

Suppose we have a function called withdraw(amount) which deduces certain amount of money from the balance, if the demanding amount is less than the current balance. At the end, the function returns the remaining balance. The function is defined as follows:

|  |
| --- |
| int balance = 500;  int withdraw(int amount) {    if (amount < balance) {      balance -= amount;    }    return balance;  } |

As we can see, in the normal case, we expect that the balance would never become negative after the execution of the function, which is also the desired behavior of the function.

However, unfortunately we could run into a race condition where the balance becomes negative. Here is how it could happen. Imagine we have two threads invoking the function at the same time with different input parameters, e.g. for thread #1, withdraw(amount=400) and for thread #2, withdraw(amount=200). The execution of the two threads is scheduled as the graph below, where at each time instance, we run exclusively only a statement from either threads.



As one can see, at the end of the above execution flow, we would end up with a negative balance, which is not a desired output.

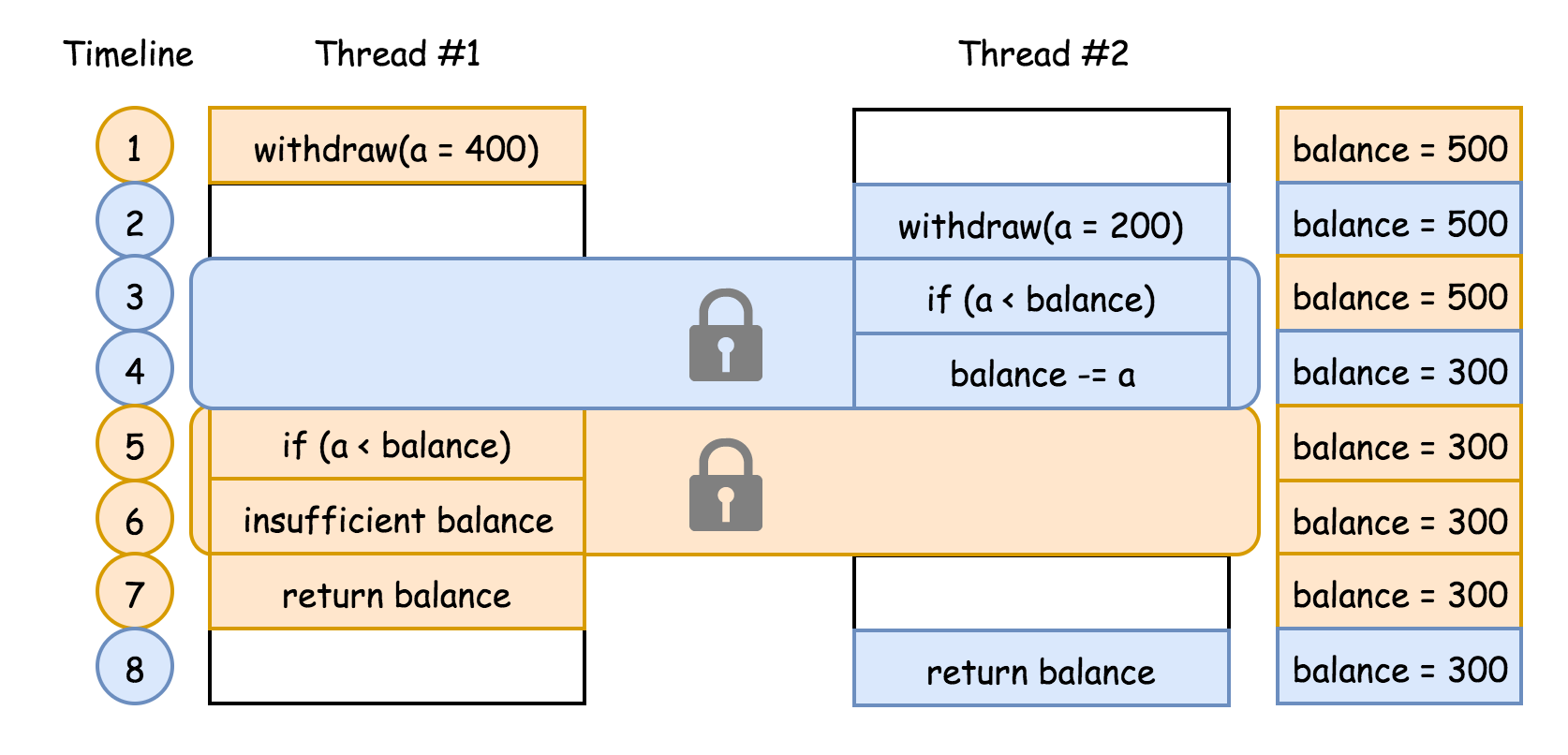
**Race-free Concurrency**

The concurrency problems share one common characteristic: multiple processes/threads share some resources (e.g. the variable balance). Since we cannot eliminate the constraint of resource sharing, the key to prevent the concurrency problems boils down to the coordination of resource sharing.

The idea is that if we could ensure the exclusivity of certain critical code section (e.g. the statements to check and deduce the balance), we could prevent the program from running into certain inconsistent states.

*The solution to the race condition becomes clear: we need certain mechanism that could enforce the exclusivity of certain critical code section, i.e. at a given time, only one thread can enter the critical section.*

One can consider the mechanism as a sort of lock that restricts the access of the critical section. Following the previous example, we apply the lock on the critical section, i.e. the statements of balance check and balance deduction. We then rerun the two threads, which could lead to the following flow:



With the mechanism, once a thread enters the critical section, it would prevent other threads from entering the same critical section. For example, at the timestamp #3, the thread #2 enters the critical section. Then at the next timestamp #4, the thread #1 could have sneaked into the dangerous critical section if the statement was not protected by the lock. At the end, the two threads run concurrently, while the consistency of the system is maintained, i.e. the balance remains positive.

If the thread is not granted with the access of the critical section, we can say that the thread is blocked or put into sleep, e.g. the thread #1 is blocked at the timestamp #4. As one can imagine, once the critical section is released, it would be nice to notify the waiting threads. For instance, as soon as the thread #2 releases the critical section at the timestamp #5, the thread #1 got notified to take over the critical section.

*As a result, it is often the case that the mechanism also comes with the capability to wake up those waiting peers.*

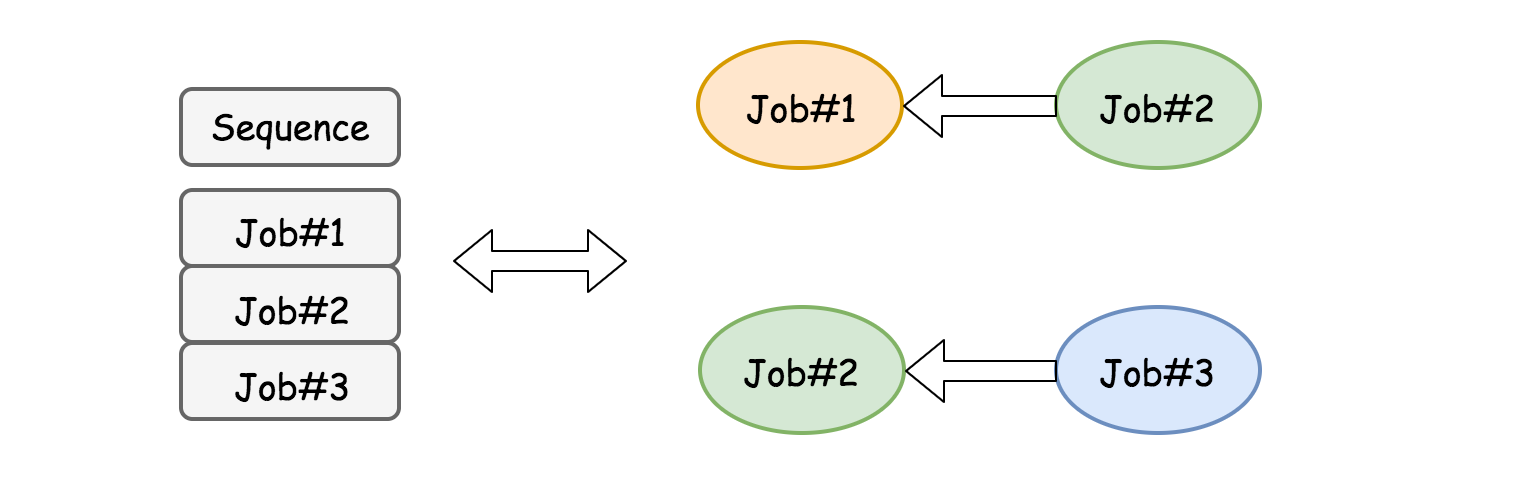
To summarize, in order to prevent the race condition in concurrency, we need a mechanism that possess two capabilities: 1). access control on critical section. 2). notification to the blocking threads.

**Approach 1: Pair Synchronization**

**Intuition**

The problem asks us to complete three jobs in order, while each job is running in a separated thread. In order to enforce the execution sequence of the jobs, we could create some dependencies between pairs of jobs, i.e. the second job should depend on the completion of the first job and the third job should depend on the completion of the second job.

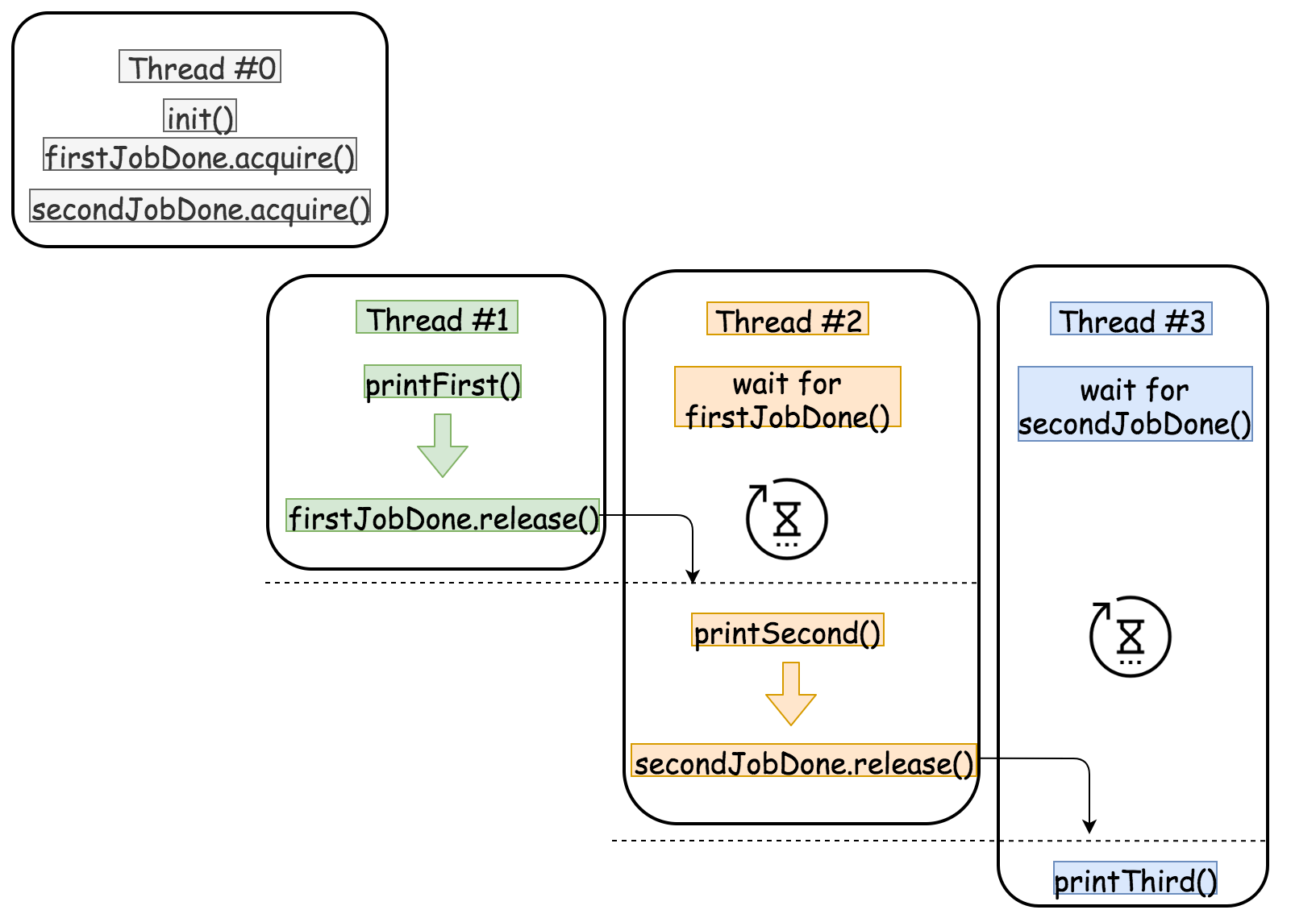
*The dependency between pairs of jobs construct a partial order on the execution sequence of all jobs, e.g. with A < B, B < C, we could obtain the sequence of A < B < C.*



The dependency can be implemented by the concurrency mechanism as we discussed in the previous section. The idea is that we could use a shared variable named firstJobDone to coordinate the execution order between the first job and the second job. Similarly, we could use another variable secondJobDone to enforce the order of execution between the second and the third jobs.

**Algorithm**

* First of all, we initialize the coordination variables firstJobDone and secondJobDone, to indicate that the jobs are not done yet.
* In the first() function, we have no dependency so that we could get straight down to the job. At the end of the function, we then update the variable firstJobDone to indicate that the first job is done.
* In the second() function, we check the status of firstJobDone. If it is not updated, we then wait, otherwise we proceed to the task of the second job. And at the end of function, we update the variable secondJobDone to mark the completion of the second job.
* In the third() function, we check the status of the secondJobDone. Similarly as the second() function, we wait for the signal of the secondJobDone, before proceeding to the task of the third job.



**Implementation**

The implementation of the above algorithm heavily depends on the programming language that one chooses, since different languages provide different constructs for the concurrency mechanism. Though some of the constructs such as mutex and semaphore are present across several programming languages including Java, C++ and Python.

Here we provide a few examples using different constructs across the languages. In particular, one could find a nice summary in the Discussion forum about the concurrency constructs in Python.

|  |
| --- |
| #include <semaphore.h>  class Foo {  protected:      sem\_t firstJobDone;      sem\_t secondJobDone;  public:      Foo() {          sem\_init(&firstJobDone, 0, 0);          sem\_init(&secondJobDone, 0, 0);      }      void first(function<void()> printFirst) {          // printFirst() outputs "first".          printFirst();          sem\_post(&firstJobDone);      }      void second(function<void()> printSecond) {          sem\_wait(&firstJobDone);          // printSecond() outputs "second".          printSecond();          sem\_post(&secondJobDone);        }      void third(function<void()> printThird) {          sem\_wait(&secondJobDone);          // printThird() outputs "third".          printThird();      }  }; |