Lecture 33: Coordinating Parallel Computation

Let’s go back to bank accounts:

class BankAccount:
    def __init__(self, initial_balance):
        self._balance = initial_balance
    @property
    def balance(self): return self._balance
    def withdraw(amount):
        if amount > self._balance:
            raise ValueError("insufficient funds")
        else:
            self._balance -= amount
            return self._balance

acct = BankAccount(10)
acct.withdraw(8)
acct.withdraw(7)

• At this point, we’d like to have the system raise an exception for one of the two withdrawals, and to set acct.balance to either 2 or 3, depending on with withdrawer gets to the bank first, like this...
class BankAccount:
    def withdraw(amount):
        if amount > self._balance:
            raise ValueError("insufficient funds")
        else:
            self._balance -= amount
        return self._balance

acct = BankAccount(10)
acct.withdraw(8)
acct.withdraw(7)

READ acct._balance -> 10
WRITE acct._balance -> 2

READ acct._balance -> 2
<raise exception>

But instead, we might get...
Undesireable Outcome

class BankAccount:
    def withdraw(amount):
        if amount > self._balance:
            raise ValueError("insufficient funds")
        else:
            self._balance -= amount
        return self._balance

acct = BankAccount(10)

acct.withdraw(8)
READ acct._balance -> 10
WRITE acct._balance -> 2

acct.withdraw(7)
READ acct._balance -> 10
WRITE acct._balance -> 3

Oops!
Serializability

- We define the desired outcomes as those that would happen if withdrawals happened sequentially, in *some* order.

- The *nondeterminism* as to which order we get is acceptable, but results that are inconsistent with both orderings are not.

- These latter happen when operations overlap, so that the two processes see *inconsistent* views of the account.

- We want the withdrawal operation to act as if it is *atomic*—as if, once started, the operation proceeds without interruption and without any overlapping effects from other operations.
One Solution: Critical Sections

- Some programming languages (e.g., Java) have special syntax for this. In Python, we can arrange something like this:

```python
def withdraw(amount):
    with CriticalSectionManager:
        if amount > self._balance:
            raise ValueError("insufficient funds")
        else:
            self._balance -= amount
        return self._balance
```

- The `with` construct:
  1. Calls the `__enter__()` method of its "context manager" argument (here, some object we'll call `CriticalSectionManager`);
  2. Executes the body (indented portion);
  3. Finally, it calls the `__exit__()` method on the context manager. It guarantees that it will always do so, no matter how you exit from the body (via `return`, exception, etc.).

- The idea is that our `CriticalSectionManager` object should let just one process through at a time. How?
Locks

• To implement our critical sections, we’ll need some help from the operating system or underlying hardware.

• A common low-level construct is the lock or mutex (for “mutual exclusion”): an object that at any given time is “owned” by one process.

• If $L$ is a lock, then
  - $L$.acquire() attempts to own $L$ on behalf of the calling process. If someone else owns it, the caller waits for it to be released.
  - $L$.release() relinquishes ownership of $L$ (if the calling process owns it).
Implementing Critical Regions

- Using locks, it's easy to create the desired context manager:

```python
from threading import Lock

class CriticalSection:
    def __init__(self):
        self.__lock = Lock()

    def __enter__(self):
        self.__lock.acquire()

    def __exit__(self, exception_type, exception_val, traceback):
        self.__lock.release()

CriticalSectionManager = CriticalSection()
```

- The extra arguments to `__exit__` provide information about the exception, if any, that caused the `with` body to be exited.

- (In fact, the bare `Lock` type itself already has `__enter__` and `__exit__` procedures, so you don’t really have to define an extra type).
Granularity

• We’ve envisioned critical sections as being atomic with respect to all other critical sections.

• Has the advantage of simplicity and safety, but causes unnecessary waits.

• In fact, different accounts need not coordinate with each other. We can have a separate critical section manager (or lock) for each account object:

```python
class BankAccount:
    def __init__(self, initial_balance):
        self._balance = initial_balance
        self._critical = CriticalSection()

    def withdraw(self, amount):
        with self._critical:
            ...
```

• That is, can produce a solution with finer granularity of locks.
Synchronization

• Another kind of problem arises when different processes must communicate. In that case, one may have to wait for the other to send something.

• This, for example, doesn't work too well:

```python
class Mailbox:
    def __init__(self):
        self._queue = []
    def deposit(self, msg):
        self._queue.append(msg)
    def pickup(self):
        while not self._queue:
            pass
        return self._queue.pop()
```

• Idea is that one process deposits a message for another to pick up later.

• What goes wrong?
Problems with the Naive Mailbox

```python
class Mailbox:
    def __init__(self):
        self._queue = []
    def deposit(self, msg):
        self._queue.append(msg)
    def pickup(self):
        while not self._queue:
            pass
        return self._queue.pop()
```

- **Inconsistency**: Two processes picking up mail can find the queue occupied simultaneously, but only one will succeed in picking up mail, and the other will get exception.

- **Busy-waiting**: The loop that waits for a message uses up processor time.

- **Deadlock**: If one is running two logical processes on one processor, busy-waiting can lead to nobody making any progress.

- **Starvation**: Even without busy-waiting one process can be shut out from ever getting mail.
Conditions

- One way to deal with this is to augment locks with *conditions*:

```python
from threading import Condition
class Mailbox:
    def __init__(self):
        self._queue = []
        self._condition = Condition()
    def deposit(self, msg):
        with self._condition:
            self._queue.append(msg)
            self._condition.notify()
    def pickup(self):
        with self._condition:
            while not self._queue:
                self._condition.wait()
            return self._queue.pop()
```

- Conditions act like locks with methods `wait`, `notify` (and others).
- `wait` releases the lock, waits for someone to call `notify`, and then reacquires the lock.
Another Approach: Messages

• Turn the problem inside out: instead of client processes deciding how to coordinate their operations on data, let the data coordinate its actions.

• From the Mailbox’s perspective, things look like this:

```python
self._queue = []
while True:
    wait for a request, R, to deposit or pickup
    if R is a deposit of msg:
        self._queue.append(msg)
        send back acknowledgement
    elif self._queue and R is a pickup:
        msg = self._queue.pop()
        send back msg
```
Rendezvous

- Following ideas from C.A.R Hoare, the Ada language used the notion of a *rendezvous* for this purpose:

```ada
task type Mailbox is
  entry deposit(Msg: String);
  entry pickup(Msg: out String);
end Mailbox;

task body Mailbox is
  Queue: ... 
begin 
  loop 
    select 
      accept deposit(Msg: String) do Queue.append(Msg); end;
      or when not Queue.empty =>
      accept pickup(Msg: out String) do Queue.pop(Msg); end;
    end select;
  end loop;
end;
```