LECTURE 42

LOCKING AND SYNCHRONIZATION

MCS 260 Fall 2021
Emily Dumas

REMINDERS

- You're done with MCS 260 homework!
- Project 4 is due Friday at 6pm central.
- Worksheet 15 is available. It is the last worksheet.
- No TA or instructor office hours in final exam week (Dec 6-10).

THREADING

Today we'll talk more about multi-threaded programming, which we introduced in Lecture 39.

Last time our threads coordinated using only the `queue.Queue` class.

Today we'll discuss other methods for coordinating multiple threads.

PROBLEM

In a multi-threaded program, you never know when the switch to another thread will happen.

I.e. code in distinct threads has *uncertain execution order*.

Planning for all possible concurrent execution scenarios is very hard!

WHAT NOT TO DO

Two threads share mutable variables without attempting to coordinate their actions.

(This usually leads to bugs, e.g. thread switch during update leads to incorrect action.)

WHAT TO DO

Use **concurrency primitives** provided by the OS or language.

These are objects designed to be accessed by multiple threads and behave in a predictable way.

Build on the behavior of primitives to ensure certain operations happen in the required order, i.e. to achieve **synchronization**.

LOCK

A **lock** or **mutex** represents a right of exclusive access. (Think: checking a book out of the library.) In Python it is provided by `threading.Lock` with methods:

- `.acquire()` — obtain the access right; if another thread has it, block (wait) until it is available
- `.release()` — give up the access right; another thread waiting in `.acquire()` will wake up

Typical use: Hold the lock while accessing shared variables (i.e. acquire before, release after).

SIMPLE EXAMPLE

A program stores a number as both an integer (e.g. 5) and a string ("five"), as values in a dictionary.

One thread prints these values on a regular basis.

The main thread updates the values, also on a regular basis.

PROBLEM

Sometimes, the printing thread wakes up in the middle of an update by the main thread and prints incorrect information.

SOLUTION 1

Only update the integer, and have the thread look up the name.

SOLUTION 2

Lock the data structure while it is in use, so only one thread can read or write.

DEADLOCKS

What if thread 1 holds a lock that thread 2 is waiting for **and** thread 2 holds a lock that thread 1 is waiting for?

Both threads stop indefinitely. This is called a **deadlock** and must be avoided.

EVENT

`threading.Event` provides a shared boolean threads can modify or wait for.

- `.wait()` — pause until the variable becomes True
- `.set()` — set the variable to True
- `.clear()` — set the variable to False
- `.is_set()` — immediately return the value

Typical use: A dedicated thread handles a specific type of event. It waits (`.wait()`) until some other thread signals the event has happened (`.set()`).

EXAMPLE

Suppose a worker thread handles a certain long-running calculation that can't be interrupted.

The main thread often changes the input of that calculation. Whenever possible, the calculation should be run again using the latest input.

Note: This is like producer-consumer but where the queue only holds one object (the one most recently submitted).

TOA/TOU

In a multi-threaded program, be careful about checking a shared variable, and later doing something that depends on the value. The value could be changed by another thread in between!

This type of bug is called *time of access / time of use*.

Holding a lock from access to use is the usual solution.

OTHER CONCURRENCY PRIMITIVES

We didn't cover:

- Condition variables (lock with notification)
- Barriers (all threads synchronize)
- Semaphores (check out a limited resource)

REFERENCES

- Python module documentation:
  - threading

REVISION HISTORY

- 2021-12-01 Initial publication