Principles of Software Design
Concurrency and Parallelism

Robert Lukoťka
lukotka@dcs.fmph.uniba.sk
www.dcs.fmph.uniba.sk/~lukotka

M-255

Concurrency and Parallelism

- **Concurrency** - is the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome.
- **Parallelism** - calculations or the execution of processes are carried out simultaneously.

Concurrency allows for parallel execution.

Concurrency and Parallelism

- Concurrency is useful even without parallel computing (it makes sense to use more threads even if we have only one processor):
    - Efficient allocation of resources - while a thread waits for something, another thread may be executed.
    - You do not need threads to get concurrent behavior, see e.g. select system call. You do not know the order on which your code is executed. But it should not matter.

What could possibly go wrong?

```
int etx_rcvd = FALSE;
void WaitForInterrupt()
{
    etx_rcvd = FALSE;
    while (!ext_rcvd)
    {
        counter++;
    }
}
```

What could possibly go wrong?

Compiler does “obvious” optimization.

```
int etx_rcvd = FALSE;
void WaitForInterrupt()
{
    while (1)
    {
        counter++;
    }
}
```

Race conditions

OK, that was a bit silly, more standard examples (Python, C++):

```
//Example 1
if x == 5:
   x = x * 2

//Example2
x = x + 1

//Example3
x += 1

//Example4 (C++)
x++;
```

Race conditions

Each of the examples can lead to surprising behavior provided that another process can modify `x` concurrently. Assume that `x=5` at the start and other process concurrently sets `x=1`. Possible outcome in Example 1:

- 1 if `x = 1` was set after this code or before this code (expected)
- 2 if `x = 1` was set after evaluating if, but before the assignment (may be a bit surprising)
- 10 if `x = 1` was set after reading `x` and before assigning the resulting value to `x` in the second statement (may be very surprising).

Similar issues may happen in other examples, including Example 4 (depends on exact resulting instructions generated).

Race conditions

To be familiar with all possibilities, you need yo know something about

- Python: Python bytecode.
- C++: instructions used in the compiled code.

You do not want to make your program depend on these.

- It makes things very complicated.
- These details may change between versions of Python / C++ compiler.

Race conditions

At least, you need

- Mutual exclusion (mutex, synchronized, locks, . . .)
- Stop optimizations of the potentially affected code (volatile, . . .)

To be able to do stuff concurrently we have to

To be able to do stuff concurrently we have to run stuff.

- Executor - you can `submit` a function(s) to evaluate.
- ThreadPool, ProcessPool - Object Pool pattern, reuses threads/processes, fixes maximum number of threads/processes.
- Future - As you run stuff in other thread/process, your thread continues. Call of `submit` immediately returns a Future. Future is a proxy object where the result of the computation appears.
- Event Loop - run an event loop and the add tasks to it (you might not need locks in this case, details later).

Concurrency constructs

From Python threading module:

- Lock - only one thread can acquire the lock
- RLock - as Lock, but the same thread can acquire it multiple times and then it has to release it multiple times (useful if you do recursive calls)
- Condition - It includes a Lock and one must acquire it first. Method `wait` releases the lock and waits for a call of the `notify` method (it might have been called before the `wait` call).
- Semaphore - as Lock, but more threads can acquire it (the maximum number is fixed).

Concurrency constructs

From Python threading module:

- Event - Underlaying boolean variable can be set to true or false. Method `wait` blocks until the variable is true.
- Timer - An action that should be run after certain time has passed.
- Barrier - Method `wait` will block until at least `n` threads call it.

Many otherwise blocking calls have non-blocking variants or variants with timeout set (examples on next slides).

Concurrency constructs

Python Queue:

- Queues are very useful for interprocess communication.
- Such queues should generally have limited size, `put` may fail.
    - It is generally a good idea to have these limits, helps to show which part of the system is too slow.
- If you put/pop you may want to
    - Block until the action is possible.
    - Block until the action is possible or a timeout expires (may be 0, return value indicates the success).
    - Get an exception thrown in the case of failure.

How to acquire locks/resouces in Python

Context management protocol requires two methods `acquire` and `release`. It allows to use i.e. Lock, RLock, Condition, and Semaphore objects as follows.

```
with some_lock:
    do stuff ...
```

The lock is released whatever happens.

Issues with locks

It is not that easy to work with locks

- Performance issues
- Race Conditions
- Deadlocks
- ...

It is not only about memory, also about, files shared devices ...

Concurrent computing

Some good practices:

- Minimal locks (space and time, read/write)
- Prefer higher level constructs.
- Local variables.
- Immutable types.
- Pure functions.
- Acquire multiple locks in alphabetical order to prevent deadlock.
- **Make the use of locking constructs as simple as possible.**

Performance vs Simplicity

Minimal locking may be quite hard, but can improve the performance:

- See e.g. Double Checked Locking
- A misplaced / too strict lock may ruin your parallel execution.

On the other hand, the correctness of locks is hard to test.

- Assume two concurrent procedures both having with n atomic instructions.
- The number of orderings is $\binom{2n}{n}$, which is too much (there are tools to do some such tests but there are obvious computational limitations on what they can do).

Thus we must keep things simple to not introduce errors.

How to minimize locks cope (time)

If a lock is needed for too long (e.g. waiting for I/O), we cannot just acquire the lock for the whole time.

- acquire lock + save the state / version, release lock.
- wait for the long operation.
- acquire lock + check state / version, do change if no relevant change occurred during that time, release lock.
- Redo the operation or notify the user if the change is not successful (somebody else changed concurrently something related to this change).

We do the same stuff to limit the time of database transactions.

Architectural solutions to concurrency issues

We want to keep things simple. We may use architectural patterns to deal with concurrency.

- Let the database handle it.
    - A very common approach.
    - Databases are very advanced tools, it would be foolish not to use their power when appropriate.
- Create "bubbles" that are executed by a single thread.
    - Queue - thread(s) processes requests from the queue (many threads can put elements into the queue)
    - Reactor pattern - similar to previous one, but can handles multiple distinct events.

Architectural solutions to concurrency issues

We want to keep things simple. We may use architectural patterns to deal with concurrency.

- Immutable data structures - makes reading non-concern, allow atomic changes, . . .
- Pipes and filters - Data to share are sent through pipes (thread-safe queues). besides the queues each thread has only local variables . . .
- Async IO - Cooperative multitasking to efficiently use the time spent waiting for non-CPU bound operations . . .
- . . .

Immutable data structures approach

Basic concepts:

- Values - data that are:
    - Place independent.
    - Immutable when exposed.
    - Not necessary immutable during the building process, before the value is exposed.
- Pure functions:
    - Input/output are values.
    - No remote effect.
    - No notion of time.
    - Same arguments, same result.

Immutable data structures approach

- Instead of rewriting the state (and thus loosing the old one), time is captured as sequence of values.
- New value is created by applying a pure function to an existing one.
- Persistent data structures - previous state is still referenced, thus the structure also records its history.
- Eventually you need some mutable reference to the most current value, but you can do surprisingly large systems with just one mutable variable (in very large systems, you need more).

Immutable data structures approach

- Values allow massive concurrent observation.
- State changes (computation of the next value) have to be sequential.
- This is often a good tradeoff as reads are more frequent then writes.
- Requires efficient creation of new values.
- Garbage collector can clean the no-longer referenced “past”.

Immutable data structures efficiency

We really just need a few classes to compose arbitrary complex values:

- Several primitive types (int, bool, char, . . .)
- Array
- Set
- Map
- Structure
- String
- . . . maybe few more

All these structures are represented as trees with high branching factor. Data sharing makes creation of new values efficient

Data sharing example

To illustrate data sharing consider the following value (Python):

```
{"Name": "Robert",
 "Jobs": ( {"range": (1997, 2001), "inst": "GJGT"},
           {"range": (2001, 2006), "inst": "FMFI"},
           {"range": (2005, 2016), "inst": "MSHDO"},
           {"range": (2006, 2010), "inst": "FMFI"},
           {"range": (2011, 2015), "inst": "TRUNI"},
           {"range": (2015, 2019), "inst": "FMFI"} )
}
```

We want to change 2019 to 2020.

Data sharing example

- create pair `(2015, 2020)`
- create `{"range": (2015, 2019), "inst": "FMFI"}`
  - You do not need to create the `"range"` string nor the institution part of the structure. Just reference to existing value.
- Assume that the list is stored as a tree of depth 2. We create new versions only for the node containing the modified value and its parents. We use references to unchanged nodes.
- We create new version in the top-level dictionary.

Note that the old and the new value share structure - but this is fine as the data is immutable.

Data sharing efficiency

Consider an array containing 1000000 values implemented as a tree

- Modifying single value is logarithmic.
- The base of the logarithm is high.
- Slightly worse than for mutable object, but the complexity is not prohibitive.

Immutable / Persistent data structures

Not only we can create new values efficiently, other benefits:

- Prepared for parallel execution of stuff on the structure (trees are great for that)
- Easy to check if sub-state has changed (compare references)
  - You cannot lock the state while waiting for user input.
  - You obtain the reference to the current state and hope for the best.
  - Assume that after I/O completion the state has changed, possibly several times.
  - Our change may be still OK, if the other changes are unrelated.
  - Fortunately, we can easily check which parts were unaffected, because of structure sharing, it is sufficient to compare references instead of looking at the values deeply.

Immutable / Persistent data structures

Not only we can create new values efficiently, other benefits:

- Atomic state changes are easy to do.
    - You do one change and do not publish the resulting value, then you do the second change and publish the result.
    - From the outside world the two changes happened at once, nobody can read the intermediate state.
- Having recent history (even if there was no business value in it, which is rarely the case) stored should make it easy to track bugs.

Concurrency in Python

We have:

- Processes (multiprocessing module)
- Threads

In Python, processes can run in parallel, threads within the same process not (Global interpreter lock). If you want more threads, they can cooperate

- Preemptive multitasking (threading) - operating system switches tasks.
- Cooperative multitasking (asyncio) -The tasks decide when to give up control.

Note that if the application is I/O bound (program is slow because it waits for input/output), threads are as good as processes. Cooperative multitasking may be significantly faster than preemptive.

Asynchronous calls - low level tools

How to get a result from an asynchronous call (the execution of the calling code continues):

- Callbacks - You give function(s) to be called when the function finishes.
- Futures / Promises - The function returns a proxy object with initially unknown value. When the function finishes, it stores the resulting value there and the calling function may read the result.
  - The terminology is not unified in this area, but Futures are typically created by the called function and promises by the callee (and they get what to call as an argument)

Asynchronous calls - low level tools

Often promises allow setting callback, those can be chained:

```
# Parameter is a function that takes two functions.
# One calls one of them when the computation finishes.
promise = (Promise(lambda resolve, reject :
# We compute the result and it is 1.
    resolve(1))
# We chain another function
    .then(lambda x: x+1)
# We could continue chaining more stuff...
    )
```

Note that promises are not that common in Python, but they are common in other languages e.g. in JavaScript.

asyncio - Basic concepts

Coroutines - special asynchronous functions.

- Similar to generators, if coroutine is called you just get a coroutine object.
- Coroutines need to be run by an even loop
  - There is a singleton main event loop in `asyncio`, you can use `asyncio.run` (Python 3.7+) to start an event loop put a coroutine into it for execution and wait until the coroutine finishes.
- Defined using async keyword, may await other coroutines.

Coroutine chaining

- We can compose coroutines in sequence or in parallel.

asyncio - Coroutines example

```
async def part1(i):
    # sleep is a coroutine, dummy for stuff being done.
    await asyncio.sleep(i)
    print(i+2)
    return i+2

async def part2(i):
    await asyncio.sleep(i)
    print(i*2)
    return i*2
```

asyncio - Coroutines example

```
#in sequence
async def part12s(i):
#in sequence
  j = await part1(i)
  k = await part2(j)
#in parallel
  asyncio.gather(part1(i), part2(i))
```

asyncio - Basic concepts

Queue:

- It would be wasteful to use standard queue as it is for preemptive multitasking.
- Use `asyncio.Queue` - for cooperative multitasking.
  - Not thread safe (no need for thread safety in cooperative multitasking)
  - Several additional useful methods.
- Queues give another way to put coroutines together.
- Producers - put values into the queue. Consumers - take the values. Queue separates them.
- You can have multiple producers and multiple consumers.

asyncio - Queue example

```
#producer
async def part1(queue, i):
  for j in range(i):
      await asyncio.sleep((j+1)/i)
      await queue.put(i+1)

#consumer
async def part2(queue, j):
  while True:
    r = await queue.get()
    await asyncio.sleep(r/j)
    print(r*10 + j)
    queue.task_done()
```

asyncio - Queue example

```
async def run():
    queue = asyncio.Queue()
    #run consumers
    c1 = asyncio.create_task(part2(queue, 1))
    c2 = asyncio.create_task(part2(queue, 2))
    #run producers and wait till they finish
    await asyncio.gather(part1(queue, 3), part1(queue, 6))
    #wait until everything in queue is processed
    await queue.join()
    #cancel consumers (they are infinite loops)
    c1.cancel()
    c2.cancel()
asyncio.run(run())
```

asyncio - Basic concepts

Event loop - needed to actually run the coroutines. You typically need just one. You start it with `asyncio.run(coroutine)`.

- You can add additional tasks later `create_task`, but then it is up to you to ensure they finish.

asyncio - Advantages

Cooperative multitasking has several advantages

- No locks required. You decide when you give up control, thus there should be no strange run conditions.
- Efficient resource usage - You do not wait until OS decides to switch threads. Your coroutines give up control at a convenient place and the event loop may select next coroutine that is ready to run
- Good for I/O bound code, e.g. efficient servers. To do I/O efficiently it requires special non-blocking calls (otherwise you need to run a blocking I/O in another thread and you lose some efficiency).

Resources I

- Python concurrency
- Python threading module - check the basic objects
- Python queues - various ways to put and pop
- Python async IO
- Rich Hickey - Are we there yet - a way to design and write code using immutable data (till 48:00)
- optional video - Rob Pike - Concurrency Is Not Parallelism - Go language

References I

- Concurrency - Wikipedia
- Parallel Computing - Wikipedia
- Race Condition - Wikipedia