Concurrency: Threads & Processes

Type	📒 Lecture
Date	@February 25, 2022
Lecture #	1
Lecture URL	https://youtu.be/dR0_7AmJKKk
Notion URL	https://21f1003586.notion.site/Concurrency-Threads-Processes-eefdd659b1954818ab380430bc83f624
Week #	10

Concurrent Programming

- Multiprocessing
  - Single processor executes several computations “in parallel”
  - Time-slicing to share access

- Logically parallel actions within a single application
  - Clicking Stop terminates a download in a browser
  - User-interface is running in parallel with network access
- Process
  - Private set of local variables
  - Time-slicing involves saving the state of one process and loading the suspended state of another
- Threads
  - Operated on same local variables
  - Communicate via “shared memory”
  - Context switches are easier
- The word “process” and “thread” interchangeably

Shared variables

- Browser example: download thread and user-interface thread run in parallel
  - Shared boolean variable `terminate` indicates whether download should be interrupted
  - `terminate` is initially `false`
  - Clicking Stop sets it to `true`
  - Download thread checks the value of this variable periodically and aborts if it is set to true
- Watch out for race conditions
  - Shared variables must be updated consistently

Creating threads in Java

- Have a class extend `Thread`

```
public class Parallel extends Thread {
  private int id;

  public Parallel(int i) { id = i; }
}
```

- Define a function `run()` where execution can begin in parallel

```
public class Parallel extends Thread {
  private int id;
  public Parallel(int i) { id = i; }

  public void run() {
    for(int j = 0; j < 100; j++) {
      System.out.println("id: " + id);

      try {
        // Sleep for 1000ms
        sleep(1000);
      } catch(InterruptedException e) { ... }
    }
  }
}
```

- Invoking `p[i].start()` initiates `p[i].run()` in a separate thread

```
public class TestParallel {
  public static void main(String[] args) {
    Parallel[] p = new Parallel[5];

    for(int i = 0; i < 5; i++) {
      p[i] = new Parallel(i);
      // Start p[i].run() in a concurrent thread
      p[i].start();
    }
  }
}
```

  - Directly calling `p[i].run()` does execute in separate thread

- `sleep(t)` suspends thread for `t` milliseconds
  - Static function — use `Thread.sleep()` if current class does not extend `Thread`
  - `throws InterruptedException`

Typical Output

```
My id is 0
My id is 3
My id is 2
My id is 1
My id is 4
My id is 0
My id is 2
My id is 3
```

```
My id is 4
My id is 1
My id is 0
My id is 3
My id is 1
My id is 2
My id is 4
My id is 0
...
```

Java threads ...

- Cannot always extend `Thread`
  - Single inheritance
- Instead, implement `Runnable`

```
public class Parallel implements Runnable {
  // only the line above has changed
  private int id;
  public Parallel(int i) { ... }
  public void run() { ... }
}
```

- To use the `Runnable` class, explicitly create a `Thread` and `start()` it

```
public class TestParallel {
  public static void main(String[] args) {
    Parallel[] p = new Parallel[5];
    Thread[] t = new Thread[5];

    for(int i = 0; i < 5; i++) {
      p[i] = new Parallel(i);
      // Make a new thread t[i] from p[i]
      t[i] = new Thread(p[i]);
      // Start off p[i].run() using t[i].start(). Weird syntax but ok
      t[i].start();
    }
  }
}
```

Summary

- Common to have logically parallel actions with a single application
  - Download from one webpage while browsing another
- Threads are lightweight processes with shared variables that can run in parallel

- Use `Thread` class or `Runnable` interface to create parallel threads in Java

Race Conditions

Type	📒 Lecture
Date	@February 25, 2022
Lecture #	2
Lecture URL	https://youtu.be/a9PZhjCKm9k
Notion URL	https://21f1003586.notion.site/Race-Conditions-7c32206056624d5aa6950d18f2a1cfff
Week #	10

Maintain data consistency

- `double accounts[100]` describes 100 bank account
- Two functions that operate on `accounts: transfer()` and `audit()`

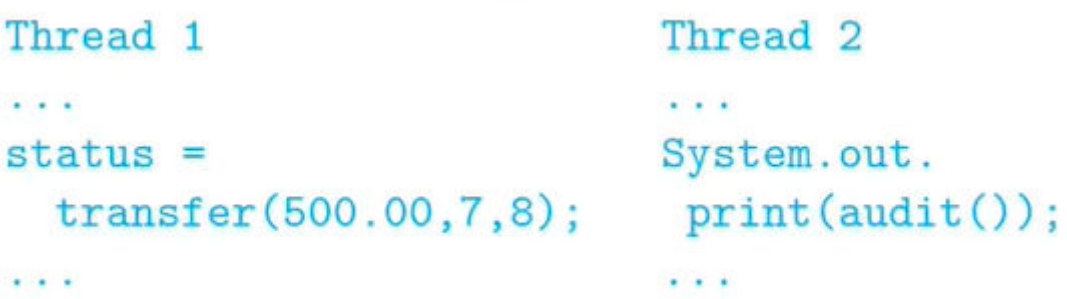
```
boolean transfer(double amount, int source, int target) {
  if(accounts[source] < amount) { return false; }

  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
```

```
}

double audit() {
  // Total balance across all accounts
  double balance = 0.00;
  for(int i = 0; i < 100; i++) {
    balance += accounts[i];
  }
  return balance;
}
```

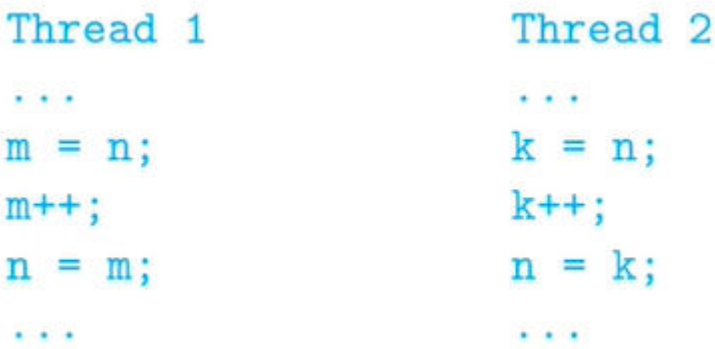
- What are the possibilities when we execute the following ...



- `audit()` can report an overall total that is `500` more or less than the actual assets
  - Depends on how actions of `transfer` are interleaved with actions of `audit`
  - Can even report an error if `transfer` happens automatically

Atomicity of updates

- Two threads increment a shared variable `n`



- Expect `n` to increase by 2
- but, time-slicing may order execution as follows

```
Thread 1: m = n;
Thread 1: m++;
Thread 2: k = n;    // k gets the original value of n
Thread 2: k++;
Thread 1: n = m;
Thread 2: n = k;    // Same value as that set by Thread 1
```

Race conditions and mutual exclusion

- Race condition — concurrent update of shared variables, unpredictable outcome
  - Executing `transfer()` and `audit()` concurrently can cause `audit()` to report more or less than the actual assets
- Avoid this by insisting that `transfer()` and `audit()` do not interleave
- Never simultaneously have current control point of one thread within `transfer()` and another thread within `audit()`
- Mutually exclusive access to critical regions of code

Summary

- Concurrent update of a shared variable can lead to data inconsistency
  - Race condition
- Control behaviour of threads to regulate concurrent updates
  - Critical sections — sections of code where shared variables are updated
  - Mutual exclusion — at most one thread at a time can be in a critical section

Mutual Exclusion

Type	📒 Lecture
Date	@February 25, 2022
Lecture #	3
Lecture URL	https://youtu.be/i6FNvH3ULMU
Notion URL	https://21f1003586.notion.site/Mutual-Exclusion-b5fa37b4a93d48f5a0256e0f19a4ed1f
Week #	10

Mutual Exclusion

- Concurrent update of a shared variable can lead to data inconsistency
  - Race condition
- Control behaviour of threads to regulate concurrent updates
  - Critical sections — sections of code where shared variables are updated
  - Mutual exclusion — at most one thread at a time can be in a critical section

Mutual exclusion for two processes

- First attempt

```
Thread 1
...
while (turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 2;
...
```

```
Thread 2
...
while (turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
turn = 1;
...
```

- Shared variable `turn` — no assumption about initial value, atomic update
- Mutually exclusive access is granted
- but one thread is locked out permanently if other thread shuts down
  - Starvation

- Second attempt

```
Thread 1
...
request_1 = true;
while (request_2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
while (request_1)
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Mutually exclusive access is granted
- but if both threads try simultaneously, they block each other
  - Deadlock

Peterson's Algorithm

```
Thread 1
...
request_1 = true;
turn = 2;
while (request_2 &&
       turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
turn = 1;
while (request_1 &&
       turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

- Combines the previous two approaches
- if both try simultaneously, `turn` decides who goes through
- If only one is alive, `request` for that process is stuck at false and `turn` is irrelevant

Beyond two processes

- Generalizing Peterson's solution to more than two processes is not trivial
- For *n* process mutual exclusion other solutions exists
- Lamport's Bakery Algorithm
  - Each new process picks up a token (increments a counter) that is larger than all waiting processes
  - Lowest token number gets served next
  - Still need to break ties — token counter is not atomic
- Need specific clevel solutions for different situations
- Need to argue correctness in each case
- Instead, provide higher level support in programming language for synchronization

Summary

- We can construct protocols that guarantee mutual exclusion to critical sections
  - Starvation and Deadlock concerns
- These protocols cleverly use regular variables
  - No assumptions about initial values, atomicity of updates
- Difficult to generalize such protocols to arbitrary situations
- Look to programming language for features that control synchronization

Test and Set

Type	📒 Lecture
Date	@February 25, 2022
Lecture #	4
Lecture URL	https://youtu.be/MbFbJy3mkqI
Notion URL	https://21f1003586.notion.site/Test-and-Set-ea3e4162ffa2430b9ca0089c717fb792
Week #	10

Test and set

- The fundamental issue preventing consistent concurrent updates of shared variables is `test-and-set`
- To increment a counter, check its current value, then add 1
- If more than one thread does this in parallel, updates may overlap and get lost
- Need to combine test and set into an atomic, indivisible step
- Cannot be guaranteed without adding this as a language primitive

Semaphores

- Programming language support for mutual exclusion
- Dijkstra's semaphores
  - Integer variable with atomic test-and-set operation
- A semaphore `S` supports two atomic operations
  - `P(s)` — from Dutch *passeren*, to pass
  - `V(s)` — from Dutch *vrygeven* to release
- `P(s)` atomically executes the following

```
if (S > 0)
  decrement S;
else
  wait for S to become positive;
```

- `V(s)` atomically executes the following

```
if (there are threads waiting
    for S to become positive)
  wake one of them up;
    //choice is nondeterministic
else
  increment S;
```

Using semaphores

- Mutual exclusion using semaphores

```
Thread 1                          Thread 2
...                               ...
P(S);                             P(S);
// Enter critical section         // Enter critical section
   ...                               ...
// Leave critical section         // Leave critical section
V(S);                             V(S);
...                               ...
```

- Semaphores guarantee
  - Mutual exclusion
  - Freedom from starvation
  - Freedom from deadlock

Problems with semaphores

- Too low level
- No clear relationship between a semaphore and the critical region than it protects
- All threads must cooperate to correctly reset semaphore
- Cannot enforce that each `P(S)` has a matching `V(S)`
- Can even execute `V(S)` without having done `P(S)`

Summary

- Test-and-set is at the heart of most race conditions

- Need a high-level primitive for atomic test-and-set in the programming language
- Semaphores provide one such solution
- Solutions based on test-and-set are low level and prone to programming errors

Monitors

Type	📒 Lecture
Date	@February 26, 2022
Lecture #	5
Lecture URL	https://youtu.be/sK1-Qkeh8mE
Notion URL	https://21f1003586.notion.site/Monitors-0213122e84ab445a817d2945e774066a
Week #	10

Atomic test-and-set

- Test-and-set is at the heart of most race conditions
- Need a high level primitive for atomic test-and-set in the programming language
- Semaphores provide one such solution
- Solutions based on test-and-set are low level and prone to programming errors

Monitors

- Attach synchronization control to the data that is being protected

- Monitors — Per Brinch Hansen and CAR Hoare
- Monitor is like a class in an OO language
  - Data definition — to which access is restricted across threads
  - Collections of functions operating on this data — all are implicitly mutually exclusive

```
monitor bank_account {
    double accounts[100];

    boolean transfer(double amount,  int source, int target) {
        if(accounts[source] < amount) {
            return false;
        }

        accounts[source] -= amount;
        accounts[target] += amount;
        return true;
    }

    double audit() {
        // compute balance across all accounts
        double balance = 0.00;
        for(int i = 0; i < 100; i++) {
            balance += accounts[i];
        }

        return balance;
    }
}
```

- Monitor guarantees mutual exclusion — if one function is active, any other function will have to wait for it to finish

Monitors: external queue

- Monitor ensures `transfer` and `audit` are mutually exclusive
- If `Thread 1` is executing `transfer` and `Thread 2` invokes `audit`, it must wait
- Implicit queue associated with each monitor
  - Contains all processes waiting for access
  - In practice, this may be just a set, not a queue

Making monitors more flexible

- Our definition of monitors may be too restrictive

`transfer(500.00, i, j);`

`transfer(400.00, j, k);`

- This should always succeed if `accounts[i] > 500`
- If these calls are reordered and `accounts[i] < 400` initially, this will fail
- A possible fix — let an account wait for pending inflows

```
boolean transfer(double amount, int source, int target) {
  if(accounts[source] < amount) {
    // wait for another transaction to transfer money
    // into accounts[source]
  }

  accounts[source] -= amount;
  accounts[target] += amount;
  return true;
}
```

Monitors — `wait()`

- All other processes are blocked out while this process waits
- Need a mechanism for a thread to suspend itself and give up the monitor
- A suspended process is waiting for monitor to change its state
- Have a separate internal queue, as opposed to external queue where initially blocked threads wait
- Dual operation to notify and wake up suspended processes

Monitors — `notify()`

```
boolean transfer(double amount, int source, int target) {
  if(accounts[source] < amount) { wait(); }

  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

- What happens when a process executes `notify()`?
- Signal and exit — notifying process immediately exits the monitor
  - `notify()` must be the last instruction

- Signal and wait — notifying process swaps roles and goes into the internal queue of the monitor
- Signal and continue — notifying process keeps control till it completes then one of the notified processes steps in

Monitors — `wait()` and `notify()`

- Should check the `wait()` condition again on wake up
  - Change of state may not be sufficient to continue — e.g. not enough inflow into the account to allow transfer
- A thread can be again interleaved betwen notification and running
  - At wake-up, the state was fine, but it has changed again due to some other concurrent action
- `wait()` should be in a `while`, not in an `if`

```
boolean transfer(double amount, int source, int target) {
  if(accounts[source] < amount) { wait(); }

  accounts[source] -= amount;
  accounts[target] += amount;
  notify();
  return true;
}
```

Condition variables

- After `transfer, notify()` is only useful for threads waiting for target account of transfer to change state
- Makes sense to have more than one internal queue

```
monitor bank_account {
    double accounts[100];
    // one internal queue for each account
    queue q[100];

    boolean transfer(double amount,  int source, int target) {
        while(accounts[source] < amount) {
            // wait in the queue associated with source
            q[source].wait();
        }

        accounts[source] -= amount;
        accounts[target] += amount;
```

```
        // notify the queue associated with target
        q[target].notify();
        return true;
    }

    // compute the balance across all accounts
    double audit() { ... }
}
```

- Monitor can have condition variables to describe internal queues

Summary

- Concurrent programming with atomic test-and-set primitives is error prone
- Monitors are like abstract datatypes for concurrent programming
  - Encapsulate data and methods to manipulate data
  - Methods are implicitly atomic, regulate concurrent access
  - Each object has an implicit external queue of processes waiting for execute a method
- `wait()` and `notify()` allow more flexible operation
- Can have multiple internal queues controlled by condition variables