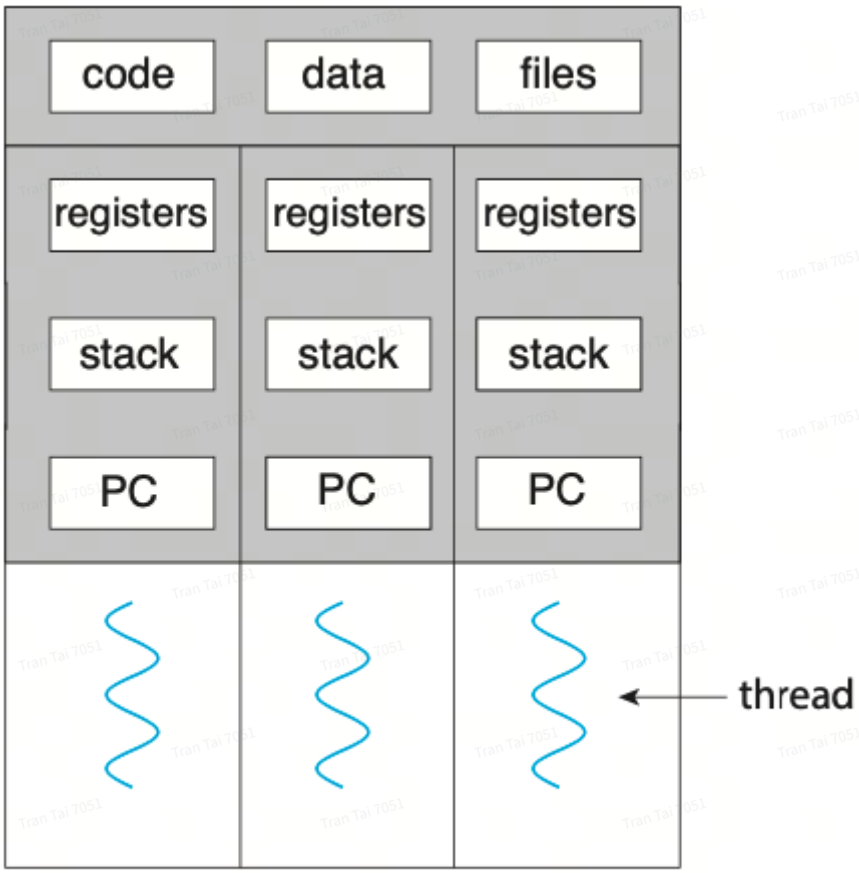
Threads and Concurrencies

Revised about threads

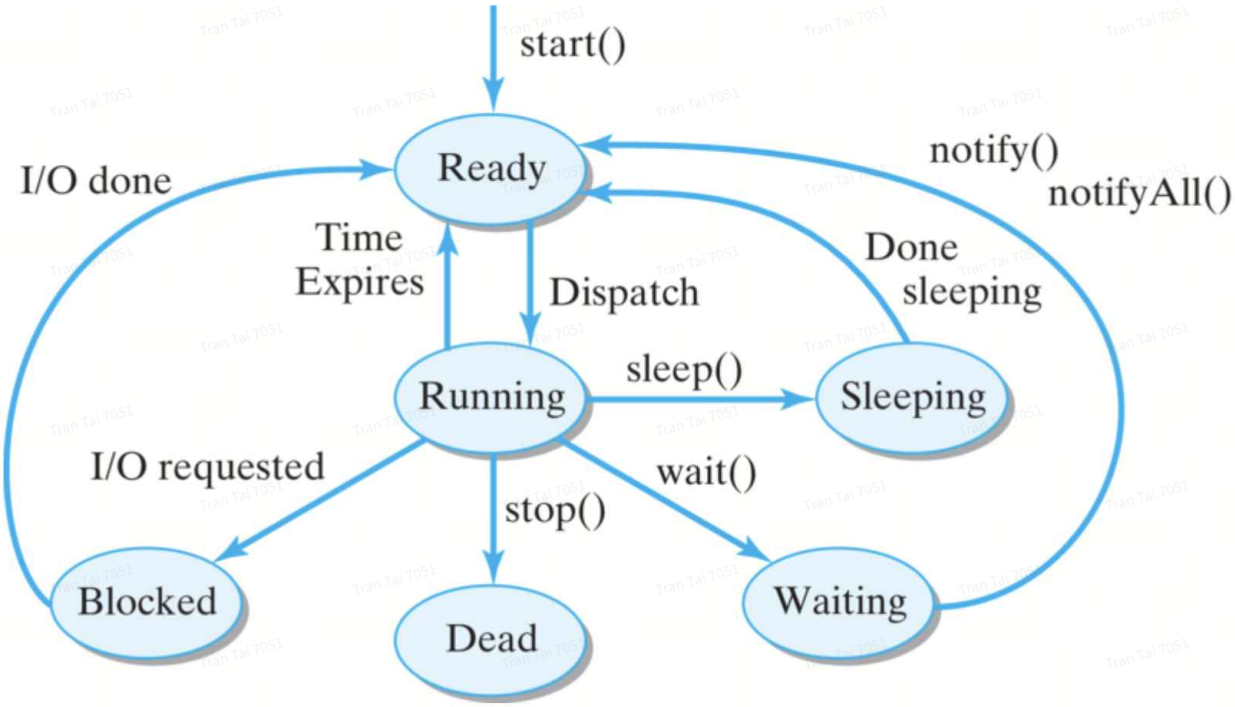
- Thread is a basic unit of CPU utilization. Thread creation is more efficient compared to process creation in terms of **time** and **resource intensive**.
- Each thread contains a **threadID**, **program counter**, **register set** and **stack**. Within the same process, they share **code section, data section** and **some OS resources** like files and signals.



multithreaded process

- Each user created thread will be assigned to a kernel thread in OS by 3 types of relationship: 1-to-1, many-to-1, many-to-many.
- Thread life cycle:
  - **Ready**: thread is ready to be executed and wait for CPU.
  - **Running**: thread is executed on CPU.
  - **Sleeping**: thread is being suspended.
  - **Waiting**: thread is waited for some events to trigger.

- **Blocked**: thread is waiting for I/O to finish.
- **Dead**: thread is finished executing.



Synchronizations

- Concurrent access to shared data might lead to data inconsistency.
- Those primatives below provide some mechanism to resolve that issue.

Lock

- When a thread wants to | enter the critical section, it should **acquire()** the lock.
  | exit the critical section, it should **release()** the lock.
- This pseudocode is called **spinlock**:

```
acquire() {
    while (!available) {
        // waiting
    }
    available = false
}

release(){
    available = true
}
```

- Lock can be implemented using CAS (Compare-And-Swap) operation. // TODO: implement
- **Lock contention**
  - If a lock is available when a thread attempts to acquire it, the lock is considered **uncontended**. Else, it is **contended**.
  - If high contention occurs (many threads try to acquire the lock at the same time), the performance of the application will be degraded.
- SpinLock?

Semaphore

- An integer that is accessed via 2 atomic operations: **signal()** and **wait()**

```
wait() {
    while (S <= 0){
        // busy waiting
    }

    S--
}

signal() {
    S++
}
```

- From the pseudo-code, binary semaphore (S = 2) behaves exactly like mutex lock.
- Counting semaphores can be used to control the **access to a given resource with fixed number of instances**.
- Semaphore implementations pseudocode
  - **sleep()** and **awake()** here are system calls.

```
type Semaphore struct {
    value int
    Thread*[] waitingThread
}

func (sem *Semaphore) wait(thread *Thread) {
    while (sem.S <= 0) {
```

```
        // busy waiting;
    }

    sem.S--

    sleep(thread)
    waitingThreads = append(waitingThreads, thread) // insert to thread list;
}

func (sem *Semaphore) signal(thread *Thread) {
     sem.value++

     wakeup(thread)
     erase(waitingThreads, thread) // remove from the thread list;
}


```

Monitor?

- The misuse of semaphores and locks can lead to some errors like the critical section is accessed by 2 threads at the same time or the threads will be blocked.
- Monitor incorporates simple synchronization tool as high-level language constructs.
- Pseudocode syntax of a monitor

```
monitor monitor name
{ /* shared variable declarations */
    function P1 ( . . . ) { ...
    }
    function P2 ( . . . ) { ...
    }
    function Pn ( . . . ) { ...
    }
    initialization code ( . . . ) {
    ...
    }
}
```

  - A function declared on the monitor can only access those variables declared inside monitor or the function params.

- Monitor ensures that at one time only one process is active in the monitor.

// TODO implement it;

Side questions

1. Difference between thread and goroutines (in Golang)
   - Thread managed by OS while goroutines managed by go-runtime.
   - Goroutine will be scheduled and execute in a thread.
   - Will be answered in detail in next sharing.

2. Difference between binary semaphore and lock.

3. Different between multi-core, multi-threaded

Multiple-core -> system characteristic

Multi-threaded -> program characteristic

4. When sleep() operation occurs, how is that thread awake?

- When the thread sleeps, it will be

5. How differences between OS (kernel) memory, heap mem, stack mem.

6. Condition variables? [OK]

7. When using Read/Write Mutex and Wait/Notify Conditional Variables.

8. Compare and Swap - strong mode and weak mode. For strong mode -> has a for loop and reassigned expected value to the actual value. For weak mode just return false. [TODO]

9. How to avoid spinlock behaviour.

- **BackOff strategy**: increase delay between each delay.
- Fine-Grained Locking: smaller sections each with locks. And reduce contention.
- Some kind of blocking mechanism.

10. What is Reentrant Lock
    - Allow threads to reeneter into a lock on a resource (multiple times) without deadlock.
    - Is there anything similar in Golang

11. How to guarantee lock will be unlock() finally.
    https://en.cppreference.com/w/cpp/language/raii

- Basically is to attach the lock to an object life cycle so that when the object is terminated (destructor func is call, the lock also unlocked)

12. std::lock for n locks to prevent deadlocks. Make sure the order of locks acquired consistent between different threads.

- Will dive deep into later.

13. Difference between the lock, semaphore and conditional variable.

- Lock: 1 thread to enter. Locking mechanism.
- Semaphore: allows x threads to enter, limit no of resource used.
- Cond: allow thread suspension and has a mechanism to wake it up later. Have option to notify most longest waiting thread or all threads.

// TO add-on

Reference

- Operating System Dinosaur book
- https://codedost.com/java/multithreading-in-java/life-cycle-of-a-thread-in-java/

Next sharing

- Deep dive into goroutine.