**Exercise 1**

1. Recent CPUs of desktop PCs had single cores for process execution, however, they were supporting concurrency via simulation. Concurrency became important today because the current trend in CPUs to have multiple cores for process execution and also, distributed systems became widespread with cloud computing, software applications are trying to benefit from these architectures, so software applications are undergoing concurrency optimization.
2. Safety means ensuring consistency within process execution, so nothing bad happens. Processes must have no reachable error/stop state and shared resources mustn’t be corrupted. Having deadlock is a violation of safety. For example, a race condition can cause a deadlock. Thread 2 executes some process with resource and waits for notification from Thread 1 to execute process again. Thread 1 updates a resource and sends Thread 2 a wake up notification. If a race condition occurs like Thread 1 sends notification before Thread 2 registers for a wait condition then Thread 2 cannot get wake up notification. If Thread 1 doesn’t send wake up notification anymore then deadlock condition happens.
3. Liveness means ensuring progress within process execution, so something good happens. Processes must access shared resources eventually, and they mustn’t starve. For example, Thread 1 and Thread 2 sharing a resource and it’s protected with mutual exclusion. To have progress Thread 2 should wait for long to Thread 1 to finish or vice versa. To prevent starvation, they shouldn’t be waiting on a resource.
4. A binary semaphore cannot cause a starvation or a deadlock by itself if implemented correctly. It all depends how you implement the semaphore in your code. If the critical section of your code takes time to execute than it leads to starvation. If you only call P operation of semaphore and forget to call V operation after the critical section, then it can lead to a deadlock.
5. We need synchronization mechanisms in concurrent programs to ensure safety.
6. Monitors are more structured than semaphores. Monitors are classes which encapsulate internal data and execute synchronized operations on that data.
7. Message passing and monitors are both structured, but message passing uses communication.

**Exercise 2**

Thread 1 starts and finishes after that Thread 2 starts and finishes = 40

Thread 1 starts early and finishes late, Thread 1 overwrites x value = 8

Thread 2 starts early and finishes late, Thread 2 overwrites x value = 5

Thread 2 starts and finishes after that Thread 1 starts and finishes = 12

Thread 1 starts early, but Thread 2 reads value before Thread 1 stores value to x and Thread 2 is the last to store value in x = 5

Thread 2 starts early, but Thread 1 reads value before Thread 2 store value to x and Thread 1 is the last to store value in x = 8

**Exercise 3**

We can use semaphores where wait and signal operations invoked in monitors. Replacing condition variables with semaphores. I’m going to use monitor implementation from the lecture:

type buffer(T) = monitor

var

slots : array [0..N-1] of T;

head, tail : 0..N-1;

size : 0..N;

**notfull, notempty:semaphore**;

procedure deposit(p : T);

begin

if size = N then

**notfull.P**

slots[tail] := p;

size := size + 1;

tail := (tail+1) mod N;

**notempty.V**

end

procedure fetch(var it : T);

begin

if size = 0 then

**notempty.P**

it := slots[head];

size := size - 1;

head := (head+1) mod N;

**notfull.V**

end

begin

size := 0;

head := 0;

tail := 0;

end