6th Sep 2023: (Transition Systems)

* Never ever consider time to control/synchronize threads.

8th Sep 2023: (Transition Systems, Mutual Exclusion)

* In order to model print statement, you need to write the print command as a label on arrow.
* For looping, just add the iteration number in parenthesis, e.g., P11, Q15***(2)***.
* Missing an iteration number can lead to identical transition states.
* Types of transition system covered in class:
  + Print Statements (add print statement as annotation on state arrow)
  + For loops (add iteration number in parenthesis next to the statement)
  + Local variables (add a ***“?”*** against a local variable)
  + While loops (more or less, transition states will reappear, WHILE is to be treated as a separate statement)

11th Sep 2023: (Mutex)

* Attempt I Slide (why it fails): starvation can be there as there could be a possibility that P decides not to go into the Critical Section i.e., if Q wants to go in, it is not possible before P wants to go in
* How to check in transition Systems:
  + For Mutex: no state for P4, Q4,.
  + Absence of Livelock: either one of the thread executions change their transition.

13th Sep 2023: (Dekkers Algorithm)

* Dekker is not based on priority i.e., who requested access first, it prioritizes whose turn it is.
* Peterson focuses on priority and turn both.
* “Volatile” keyword to avoid reordering of instruction or to flush the registers.
* Concurrent Blocking Data Structure.

15th Sep 2023: (Semaphores)

* Two parts: operations, data fields.
* Acquire consumes a permission.
* Release: if no thread is waiting, it releases the resource back, but if there is a thread waiting list, then it wakes up a waiting thread.
* Mutex is a standard name given to binary semaphore.
* Parameter (true) in Semaphore initialization ensures a fair semaphore which is free from starvation.

18th Sep 2023: (Semaphore)

* Binary semaphore permission range 0-1 e.g., Mutex/Critical Section.
* Counting Semaphore permission range 0-arbitrary number e.g., Reader/Writer, Producer/Consumer.
* “!” indicates processes are not ready for execution in transition systems.
* 1 transition shows atomic state.
* How to check for Mutex in transition system? (No code line of critical section is present in a single state).
* How to check for Absence of Livelock? (When any of the processes gets executed, only one of them can get scheduled).
* How to check for Starvation? (Check for all possible executions, one assumption should be that scheduler is fair, and P/Q would eventually get executed)
* **Issue:** the main data structure for semaphore is Sets (to keep track of sleeping threads, possibility of Starvation) which can be fixed if data structure is replaced by Queue.
* **Dining Philosopher Problem**: N philosopher, each wants to eat, but for them to eat, they need 2 forks. Problem is like Granny Problem introduced in beginning classes. Possible Solutions:
  + **Allow at most N**-1 people to have eat (**Chain Semaphore**), in this one would eat all the time because we have 1 surplus fork.
  + **Allow the use of binary semaphore (Fork Semaphore)** to check each philosopher can pick one thread from the left, and one from the right.

20th Sep 2023: (Semaphore Patterns)

* Example program:
  + You cannot use negative permits.
  + Acquire/release order can be in any way.
  + The value in Semaphore(?) – Here “?” shows the number of initial permits available.
* ***Producer/Consumer Problem:***
  + Simple Case Start:
    - Just 1 producer (generate a random number) and 1 Consumer (Initializes an integer buffer and read from that) – no issues in that.
  + N producers and M consumers, in this situation producer should wait if there is already an event to be consumed, and the consumer should also wait if there is no event to be consumed.
  + Race condition variables (start/end/etc.) need to be in the critical section.
  + REVISIT VOLATILE KEYWORD.
* ***Readers/Writer Problem:***
  + When writing is in execution, no one can either read or write.
  + When reading is in execution, no can write but others can read.
  + Concept for readers: the first reader acquires the lock for all readers, and the last reader releases the lock. (the solution is not fair to the writers)
  + Introduce a new semaphore for entry protocol code.

4th October 2023: (Cyclic Barrier and Cats and Dogs Problem)

* Second barrier in cyclic barrier was added to avoid each thread racing each other i.e., each thread should wait for the other threads to arrive and then enter the barrier, example thread 0 is waiting for other thread, another thread 0 from another iteration comes in, and then another 0 comes in as well, now the threads can fall through, but thread 0 should wait for corresponding thread 1 and thread 2.
* Cyclic Barrier code breaks when # of threads is greater than # of barrier (figure out why)

10th October 2023: (Monitors)

* Monitors are higher level in comparison to semaphore.
* It is all about objects with a built-in locking mechanism.
* Use of keyword ***“Synchronized”***, - it is an intrinsic lock.
* Two types of locks:
  + Intrinsic locks: it applies to an entire object.
  + Extrinsic locks: they can be used outside.
* “***notify()” vs. “notifyAll()”*** instruction.
* Wait states are attached to lock, when a thread goes to sleep it releases the built-in lock.
* RR is W.

30th October 2023: (ERLang)

* Use of try catch.
* Higher order function.
* Spawning a server.
* Use a debugger.

13th November 2023: (Promela)

* New thread can be spawed using  
  atomic {  
   run thread\_name();  
  }
* All built in variable names start with an “\_”
* Thread/process syntax  
  proctype process\_name() { definition goes here }
* Active keyword before proctype spawn the thread immediately.
* Timeout occurs when your system is deadlocked.
* Each line of program is executed atomically, **not one whole block of code/conditional**.
* A thread will not terminate until and unless all thread having PID greater than current process PIDS are terminated.