Producer-Consumer Problem:
- Producer puts sth into a shared buffer,
Consumer takes sth from buffer
- Buffer can store a fixed number of items (bounded buffer)
- If P&C run as separate processes (/threads), how can we ensure their concurrent execution is correct?
- P&C threads have to sync such that:
1) P does not write more than rate of read by C
2) C only attempts to read when buffer has some content

Race Condition:
- Behaviour of system where its output is **dependent on sequence or timing of uncontrollable events** (EG. thread library scheduler or kernel process scheduler is out of app developer's control)
- int val=5; (THREAD1) val++; (THREAD2) val --;
- Possible outputs: 4,5,6 dep on order of execution by scheduler.
- Count++ and count-- are not atomic in machine language.
Atomic = implemented in one clock cycle

Critical Section:
- code within Count++ have to be uninterruptible OR protected by critical sections (but critical section does not necessarily imply uninterruptibility)
- Note that CS can be pre-empted -- interrupted & resumed by same thread (or, if a thread in CS is interrupted, it has to be 'rolled' back to the state before entering the CS)

- Rules of Critical Section:
(what should be uphold if u code a library that supports CS)
1) Mutual Exclusion: preventing race condition, if T1 is exe-ing its CS, no other T can execute their CS
2) Has Progress (impt! otherwise if you just require mutex, a trival soln will be to allow NOBODY to enter CS ie. zero progress)
If no thread/process in CS, then select process in queue that can enter CS as soon as possible
3) Bounded Waiting
Each CS has finite length, there's max # times other threads/processes are allowed to enter their CS after a P/T has made a request to enter its CS & before that request is granted.
Assume that each P/T executes at nonzero speed

Synchronisation Hardware
- Easy solution for supporting CS, but less flexible
- Disable interupt during CS: although some ppl do implement CS as uninterruptible using hardware (wiring etc), this is almost impossible to do if your CS contains many lines of instr (too many different permutations of types of code developers can write)
- Atomic hardware instr implemented in the system (get(), set(), swap(), getAndSet()). Done in a single clock cycle, limited flexibility too. You def cannot merge so many lines of instr in a single cycle.

Synchronisation Problems
- CS is a section where only 1 thread at a time can access: **mutual exclusion**
- Some CS must be uninterruptible, some might not. **DO NOT CONFUSE CS WITH UNINTERRUPTIBLE INSTRUCTIONS**
- Sometimes you want a **MAX of T threads** that can access a section *at the same time* or asynchronously, instead of just 1 thread at a time. This is called **condition synchronisation**. (eg. HP of a player)

Other: Each process utilizes a resource as follows:
- request (request one instance of a resource from OS; e.g., get access to a printer)
- use (use the acquired resource privately to do its work; e.g., print a file to the acquired printer)
- release (return an acquired instance of a resource back to the OS; e.g., give back the acquired printer so the OS can give this printer to another process)

Semaphore *system calls
- Peterson's solution & hardware assisted solutions all require busy waiting
- Using semaphore, you can express a solution without busy waiting (sem is a high-level synchronisation primitive (easier to use))
- int state variable value (sem) ---> maintained & stored by kernel in UNIX systems
- Two **Atomic** operations: acquire() & release()
- Solves 2 synchronisation problems:
Mutex: if binary semaphore, x=1 (or 0)
Condition synchronisation: if counting semaphore, x>1 (0,1,2,..) {eg buffer not empty for consumer in bounded-buffer producer-consumer problem)
sem.acquire()* -> asks kernel whether there is still positive semaphore left, if yes, kernel will reduce sem & let thread continue|
Otherwise, kernel tell thread library to 'wait' (this thread) & put to the kernel's semaphore's queue
sem.release()*
- called by thread that is done, tells the kernel that the thread is done with the CS, kernel will sem++, pick some thread from queue (wake up)

Circular Need of Semaphore:
Semaphore is required to execute critical sections, but semaphore's acquire() and release() itself is a critical section.
> Circular need (sem needed for CS, CS needed for sem's acq & rel)
> Possible solutions: 1. implement acq&rel using solutions that involve busy waiting (Peterson)
2. Implement using hardware solution
3. Implement using software solution: sys call that is uninterruptible (does not work in multiprocessor system)
> Busy wait on acquire() and release() alone, no busy wait on other CS because sem is used. (busy wait is ok if its fast)

Why is Busy Waiting bad?
> Bad when it takes a long time. Otherwise, it might be beneficial if we only busy wait for a while because it prevents context switch at times in multiprocessor system.
> While loop checking condition (repeat check, wastes CPU cycle: **spin lock**)
> Generally, we do not know for sure how long a CS will take just that it is *bounded.*

Java Solution to Synchro Problems:
- Two possible ways to synchronise methods of objects accessible by multiple threads. Satisfies both cases of synchro problems: mutex & condition synchronisation.
1. **Method Synchronisation** (write synchronised between access modifier & return type) // if need condition synch use wait() then notify() within CS
2. **Block Synch** (synchronised(mutexLock)) Synchronized block is finer grained than synchronized method: allows more parallelism

Entry & Wait Set
- Entry set = queue of threads waiting to enter a (any) synchronised method for the object
- Wait set contains those threads waiting for a condition to become true (ie condition synchronisation); is per object -- threads join this list no matter what condition they are waiting for
- These sets are per object, meaning each object only has ONE lock. Each object can have many synchronized methods. These methods share one lock.
- wait() -> entry set to wait set | notify() -> wait set to entry

Deadlock
1. Mutex 2. Hold resource & wait 3. No pre-emption 4. Circular wait
If all 4 happens, deadlock MIGHT happen. Deadlock can be prevented by removing either conditions (necessary conditions but not sufficient)
Pi --> Rj (Pi req instance of Rj); Pi <-- Rj (Pi holding instance of Rj)
if graph no cycle -> no dL; if got cycle -> if only 1 instance per resource type, then dL; if several instances per resource type, possibility of dL

Handling Deadlocks
Real world OS do not handle deadlocks completely all the time
1. Deadlock Avoidance
- Pi needs to declare max resources it ever needs in the beginning
- Requests that lead to unsafe state will have to wait
- Don't give resources that might lead to future deadlocks, even if it is available now | - Always check if requests will lead to circular wait. Only grant requests with safe state.
2. Deadlock Detection
- System allows deadlock to happen and then detects with this algorithm
- After detected, 1) abort all deadlock processes, this allow them to pre-empt resources (get them back) 2) Can also abort all dL processes 1 by 1 until there's no more deadlock; by priority, time exec, resources used 3) Restart all aborted processes
3. Deadlock Prevention
- Avoids potential dL situations by design, which is to disallow either 1 of the 4 necessary conditions for dL to happen
- No resource hold & wait: Must get all resources b4 process exec, only allow request 4 resources if process has none
- Allows pre-emption: Processes must release all resources its already holding if it needs other resources that require wait; restart process, must wait for every resources again
- Disallow circular wait: must obtain resources in a certain order

1. The two basic kinds of process/thread synchronisation problems (e.g., found in the producerconsumer problem) are **_mutual exclusion_** and **_condition synchronization_** .
2. The synchronization is between two processes 0 and 1, which share the turn variable. The code shown is for Process 0; the code for Process 1 is analogous. Argue that the proposed solution satisfies mutual exclusion.
Assume a process is in the critical section (CS) and before it exits, the other process enters. Wlog, assume 0 loads the value of turn first in the inner while loop. Case 1: the loaded value is 1. In this case, 0 can't enter the CS, i.e., contradiction. Case 2: the loaded value is 0. In this case, turn must still be 0 when 1 loads its value in the inner while loop. Hence, 1 can't enter, i.e., contradiction.
3. Consider the producer-consumer problem. The excerpt producer code shown on the next page uses a named reentrant lock and named condition variables in Java.
(a) When the producer calls empty.await(), what happens to the reentrant lock mutex? **The reentrant lock mutex is released by the producer.**
(b) (b) Give a suitable actual Java statement for the <statement X> placeholder in the shown code excerpt. **full.signal()**

Peterson's Solution for Process P_i

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    critical section
    flag[i] = false;
    remainder section
}
```

There are two processes i and j.

If i (or j) can't enter CS immediately, it must be waiting at **while** loop

Satisfies **mutual exclusion**: Assume i enters first and is now inside the CS. **Case 1**. flag[j] is false when i tests it in the while loop. In this case, before j tests its while loop, it must set turn to i, so that j must wait in the while loop until i exits or otherwise flag[i] must remain true and turn must remain i. **Case 2**. flag[j] is true when i tests it in the while loop. In this case, turn must be equal to i for i to exit the while loop. When j tests its while loop, j must wait until i exits or otherwise turn must remain i.

Satisfies **progress**: **Case 1**. Only i wants to enter, so that flag[j] = false. In this case, i must be able to exit the while loop since flag[j] = false. **Case 2**. Both i and j want to enter, so that flag[i] = flag[j] = true. In this case, the process whose id is equal to turn must be able to exit the while loop and enter.

According to Peterson's Algorithm, after *i* exits the critical section (CS), before it can enter again, it must set turn to *j*. So if *j* is waiting to enter (i.e., flag[*j*] == true), *i* must now wait at the while loop until after *j* got its own chance to enter the CS. Hence, *j* can't be beaten twice in a row, and bounded waiting is satisfied with a bound of 1.

Assume LD & ST are atomic

Peterson's Solution
* You can interrupt a thread in a critical section
- Peterson's solution works only for single-core system and only for 2 threads. Inefficient cause u busy wait. Does not work for multicore systems bcus setting of/reading shared variables in multicore system is complicated (each CPU has different cache, you must make them see the latest update, need kernel's help)
Initial: Nobody in CS
A. Nobody in CS, T0 wants to enter.
> flag[0] = true (set self flag to true)
> turn = 1 (set turn to be other thread)
> check flag[1] == True && turn == 1
> escapes while since flag[1]=false
> T0 enters critical section
>>> Now, T1 wants to enter CS (T0 is paused)
> flag[1] = true
> turn = 0
> T1 checks while condition, flag[0] is true, turn == j
> (satisfies mutex) T1 is busy waiting, stuck in while loop, until thread scheduler switch back to T0 (* single core)
B. After some time, T0 is resumed and manages to finish CS
> set flag[0] = false
> when T1 is resumed again, while condition is not fulfilled anymore
> T1 breaks out of while loop & enters CS (satisfies progress)
*** NEVER THE CASE WHERE NO THREAD IS IN CS WHEN THERE IS A QUEUE
C. Both T0 & T1 wants to enter CS at the same time (interleaved interrupt)
> T0 sets flag[0] = true **> IRQ**
> T1 sets flag[1] = true, turn = 0 **> IRQ**
> T0 resumes, turn = 1 **> IRQ**
> T1 enters CS since while condition is not fulfilled. (satisfies mutex) **> IRQ**
> T0 resumes, check condition: flag[1] == True (yes), turn ==1 (yes) | fulfills, so in while loop.
(satisfies mutex)
*** the thread that goes into CS first is thread whose 'id/name' is in the turn variable (in this case, turn=1 set by T0.**

```
Object mutexLock = new Object();
   public returnType
 methodName(args){
      synchronized(mutexLock) {
        //Critical section here
        notify()
      }

      //Remainder section
 }
```

When Java threads in entry set try to enter **synchronized method / block,** the thread library will test whether the lock is free (try to acquire lock).

If yes, enter CS, if no, remain in entry set

1. **Notify()** : wakes up / pick **any** arbitrary thread from wait set to entry set.
2. **NotifyAll()** : wakes up / pick **all** thread in wait set to entry set, good for condition synchronization

Both of the above are called by a thread that's existing CS.

3. **Yield()** : gives up time to CPU scheduler but doesn't give up lock **(dangerous)**
4. **Wait()** : Thread releases object lock, gives up time to CPU scheduler, state changed to blocked / waiting, **goes to wait set.** *relies on the fact that other threads will eventually call notify() to wake you up.*

Called by thread who enters synchronized method successfully but cannot progress due to some other condition, e.g: waiting I/O, dependency on other thread's output, etc

```
Object mutexLock = new Object();
   public returnType methodName(args){
      synchronized(mutexLock) {
        while(someCondition != True)
           Thread.wait();
        //Critical section here
        notify()
      }

      //Remainder section
 }
```

JAVA NAMED CONDITION VARIABLE

For multiple conditions in ONE object

```
Lock lock = new ReentrantLock
Condition lockCondition =
    lock.newCondition()

Lock.lock()

To wait for specific condition:
    lockCondition.await()

To signal specific thread waiti
for this condition:
    lockCondition.signal()

Lock.unlock()
```

```
/**
 * myNumber is the number of the thread
 * that wishes to do some work
 */
public void doWork(int myNumber) {
  lock.lock();

  try {
    /**
     * If it's not my turn, then wait
     * until I'm signaled
     */
    if (myNumber != turn)
      condVars[myNumber].await();

    // Do some work for awhile . . .

    /**
     * Finished working. Now indicate to the
     * next waiting thread that it is their
     * turn to do some work.
     */
    turn = (turn + 1) % 5;
    condVars[turn].signal();
  } catch (InterruptedException ie) { }
  finally {
    lock.unlock();
  }
}
```

Java Synchronization

- Java provides synchronization at the language-level.
- Each Java object has an associated *binary* lock (i.e., lock is either taken or available).
- This lock is acquired by invoking a **synchronized** method.
- This lock is released when exiting the **synchronized** method.
- Hence, **mutual exclusion** is guaranteed for this object's method – at most only *one* thread can be inside it at any time.
- Threads waiting to acquire the object lock are placed in the **entry set** for the object lock.

Condition synchronization by notifyAll()

- Note that thread uses **wait()** to wait for a *condition* (e.g., buffer not empty) logically, but gets placed in a wait set that is per *object* (i.e., *not* per condition)
- Similarly, another thread uses **notify()** to signal a condition logically, but **notify()** wakes up an arbitrary thread from the *object's* wait set
  - If there are more than one conditions (e.g., the producer-consumer problem) associated with the object, **notify()** may wake up a wrong thread (one *not* waiting for the condition being notified)!
  - Solution:
    - Use **notifyAll()** to wake up *all* the threads in the wait set
    - When a thread returns from **wait()**, it *must* recheck the condition it was waiting for (the **while loop** on Slides 6.32 or 6.33 keeps rechecking the condition until the condition is true) – if thread is waked up for wrong reason, then when it rechecks the condition, it'll find the condition still false and wait again
    - Can also use fine grained Java *named condition variables* (Slide 6.37)
      - These condition variables are per logical condition, not per object

Condition synchronization by wait/notify(

- When a thread invokes **wait():**
  1. The thread releases the object lock;
  2. The state of the thread is set to blocked;
  3. The thread is placed in the **wait set** for the object.
- When a thread invokes **notify()**:
  1. An *arbitrary* thread T from the wait set is selected;
  2. T is moved from the wait to the entry set;
  3. The state of T is set to runnable.

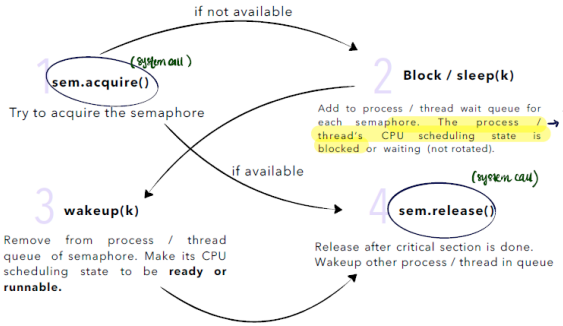
NB:

*(i) syncrhonized method solves **mutual exclusion** problem*
*(ii) wait()/notify() (together) solve **condition synchronization** problem*
(iii) Bounded buffer producer-consumer problem has two condition synchronization problems. First, for producer to wait for buffer to become non-full. Second, for consumer to wait for buffer to become non-empty.

Semaphore implementation without busy waiting

- How? By integrating semaphore implementation with CPU scheduler: can *block* and *unblock* (or wake up) processes
  - If **acquire**() can't complete (resource not available), block caller process P until semaphore becomes available (at which time wake up P)
- Associate a waiting queue (of processes) with each semaphore
- The two CPU scheduling operations:
  - block – change calling process's CPU scheduling state to *waiting/blocked*, and add it to the appropriate waiting queue
  - wakeup – remove a process from the waiting queue and change its CPU scheduling state to *ready/runnable*

NO BUSY WAITING



Java 5 named condition variables

- **Named condition variable** is created *explicitly* by first creating a reentrant lock, then invoking the lock's newCondition() method

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

  - A lock is *reentrant* is if it's safe to acquire the lock again by a caller *already holding the lock*
  - In the above code, note that the condition variable **condVar** is associated with the lock **key**; in general, this association makes sense because a thread always holds a lock when a condition is being signaled or waited for (see Slide 6.36)
- Operations on condition variables: await() and signal() methods
  - Instead of **wait**() and **notify**() for the per-object unnamed condition
- Explicit condition variables allow fine-grained condition synchronization; can solve the "threads taking turns" problem more cleanly (than the **notifyAll**() solution on Slide 6.35) …

- Another one of the Critical section implementation
- int state variable value (sem)
- Two **atomic** operations: acquire() and release()

Maintained and used by kernel in OS systems

SOLVES TWO SYNCHRONIZATION PROBLEMS:

- **Mutex**: if binary semaphore, x = 1
- **Condition synchronization**: if counting semaphore, x > 1

```
Semaphore sem = new Semaphore(x);
sem.acquire();
//Critical section
_____
_____
_____
_____
sem.release();
//Remainder section
_____
_____
_____
_____
```