* Independent process: cannot affect or be affected by other processes in the system
* Cooperating process: can affect or be affected by other processes in the system
  + Can lead to data inconsistency problems (race conditions)
* Producer and consumer must be synchronized in the ring buffer
  + Consumer waits if the buffer is empty, producer waits if the buffer is full
* Critical Section: when a process is accessing shared data
  + When one process is executing its critical section, not other process should be allowed to execute its critical section (mutual exclusion)
  + If a process wishes to enter its critical section and no other critical section is occurring, the selection of the next process that will enter the critical section cannot be postponed indefinitely
  + There needs to be a bound on the number of times a process can enter their critical sections after a process has made its request to enter its own critical section, otherwise a process could end up stuck waiting
  + Entry section: code requesting the permission to enter critical section
  + Exit section: executed to state that a process left critical section
* Cooperating processes are supposed to alternate working/waiting
* Software Solution for critical sections:
  + Peterson’s Solution is only for 2 processes
  + Uses Boolean flags and turn numbers
* Locks:
  + All cooperating processes share a lock and when a process enters CS, the lock is true, so no other processes can enter it (binary lock is true/false)
  + Only one acquire lock operation should be executed one at a time (only one process at a time should be checking the lock)
  + The release lock operation should be performed by the process that acquired the lock and should only happen after the acquire lock operation
  + This can be done through a hardware instruction or a semaphore
  + For the bounded waiting requirement (processes taking turns):
    - Use a waiting queue (cyclic queue, so using a mod function)
    - When a process finishes its CS, it will scan through the queue in a cyclic order and allow the first waiting process to go (process P1 finishes and will go through P2, P3, etc. until it reaches a process that is waiting to enter. So if P2 isn’t ready to enter CS, but P3 is, it will check on P2 then move on to P3 and let P3 in)
    - Any waiting process will wait for at most n-1 turns
    - Think of it as waiting in line for the bathroom: if a person next in line doesn’t need to go yet, they will go to the end of the line and wait until they are next again. Only one person will be in the bathroom and locks the door as they enter (maybe all waiting processes are ready to go?)
* Hardware Solution:
  + Atomic hardware instructions (non-interruptible or nondivisible)
    - Atomic instructions execute to the end, so no context switching can occur during the hardware instruction
    - Test\_and\_set() checks if the input is open and the sets it (checks if it is locked before changing)
    - Swap() swaps two Boolean values (uses pointers)
    - Compare\_and\_swap() checks an int pointer to see if it matches an expected value, and if it matches, then it gets set to a new value (uses 0 and 1 for true and false
      * Loops until it returns 0 to indicate that no other process is in CS
* Semaphore:
  + An integer that can only be accessed by the atomic operations wait() and signal()
  + Ex: for semaphore S
    - Wait(S) {

While (S<=0); means it waits until S > 0

{

}

S--;

}

Critical section{

}

Signal(S) {

S++

}

* + With a binary semaphore, it can be either 0 or 1
    - Semaphore mutex is for mutual exclusion
      * Mutex starts at 1
      * Since mutex >= 0, P1 does not go into the wait while loop
      * Mutex gets reduced to 0, so P1 can enter CS and now all other processes will get stuck in the wait loop in wait()
      * P1 finishes CS, enters signal() and increases mutex to 1
      * Now P2 can exit the wait loop since mutex > 0, reduces mutex to 0, and then enters CS
      * Other processes can enter their wait(), but get stuck in the while loop until the current process has finished its CS and finished its signal()
  + A counting semaphore (general semaphore) starts with a higher number than 1 and is used when multiple sets of the same process are available (if 4 processes can conduct CS at the same time, mutex would start at 4)
    - One the Semaphore is 0, other processes will go into the wait queue
    - General semaphore can be a binary semaphore if it is set to 1
  + To avoid the busy loop, create a waiting queue in the semaphore for processes in the wait() step, where it won’t move to the next process in the queue until the semaphore value is above 1
    - If the count value is positive after the S++ step, then there are no waiting processes
  + A process must finish it’s wait() operation (either completing it or entering into the waiting loop) or finish it’s signal() operation before another one can
    - Means they are atomic
  + Semaphores can also be used as a synchronization tool (tell something to wait until another process has finished what it was doing and setting the semaphore)
  + Multiple binary semaphores can work similarly to counting semaphores (have semaphores S1, S2, S3, etc. to act as the number of available resources)
  + For the bounded buffer problem, two semaphores (full and empty) are needed in order to allow one process to access things at a time
    - One for exclusive access to count and one to provide a waiting queue
  + Lost update problem: the change made by one process is lost by the change of another process when sharing memory
  + Memory readers should only be able to read the memory when there is no active writer working to avoid an incorrect summary problem
  + Dining Philosophers:
    - Use wait() to pick up a chopstick ([i] for left chopstick and [(i+1] % 5] for the right chopstick)
    - Signal is used to release the chopstick once it’s finished (using same indexing)
    - Deadlocks happen when all of the resources are picked up, but no process has the full set of resources needed to operate
  + Deadlock: when a process needs multiple resources to complete its operation, but other resources have taken them all up, so no process has all the resources needed to complete, so none of them free up the resources they have
  + Starvation: when a process is endlessly waiting for resources
* Monitor:
  + ADT that has shared variables declared at the start and a series of functions that needs to be executed
  + Ensures mutual exclusion (only one process at a time can be active in the monitor at one time)
  + Processes enter into the monitor through a queue
  + Uses condition variables
    - condition x, y;
    - Invoked with x.wait() and x.signal()
      * wait() suspends the process until another process involves x.signal()
      * signal() resumes exactly one suspended process that invoked wait()
        + No effect if there aren’t any blocked processes
      * Wait queues associated with each consumer variable
  + Producers add processes to the queue, consumers read item from the queue
  + The signaling process must wait once a suspended process is allowed to resume
    - If the signaling process is allowed to continue after waking a suspended process, it would violate mutual exclusion
  + Count variables associated with the condition variables to track how many waiting processes there are
* Software solutions for an arbitrary number of processes:
* Eisenberg and McGuire’s Algorithm:
  + Before a process enters the CS, it checks the other processes to see if any others are in their CS first, if it finds one, then it goes back into its entry section and waits
    - This happens in the process’s “want-in” state, which is equivalent to the wait state
  + Turn is to track which process is currently in CS, it is also where it will start when looking for the next process that will enter the CS
* Bakery Algorithm:
  + Processes pick up a ticket number before entering CS
    - Number is 1 + largest assigned ticket number
    - Can result in processes getting the same ticket number, the tie is then decided by the process number
  + In a busy wait until it gets a ticket number, so it is a short wait
    - Then it waits until the process’s ticket number has a higher priority than the other process
  + This is a useful method for when cooperating processes are running on different machines