Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Practice Problems: Concurrency

1. Answer yes/no, and provide a brief explanation.

   (a) Is it necessary for threads in a process to have separate stacks?

   (b) Is it necessary for threads in a process to have separate copies of the program executable?

   Ans:

   (a) Yes, so that they can have separate execution state, and run independently.

   (b) No, threads share the program executable and data.

2. Can one have concurrent execution of threads/processes without having parallelism? If yes, describe how. If not, explain why not.

   Ans:

   Yes, by time-sharing the CPU between threads on a single core.

3. Consider a multithreaded webserver running on a machine with N parallel CPU cores. The server has M worker threads. Every incoming request is put in a request queue, and served by one of the free worker threads. The server is fully saturated and has a certain throughput at saturation. Under which circumstances will increasing M lead to an increase in the saturation throughput of the server?

   Ans: When $M < N$ and the workload to the server is CPU-bound.

4. Consider a process that uses a user level threading library to spawn 10 user level threads. The library maps these 10 threads on to 2 kernel threads. The process is executing on a 8-core system. What is the maximum number of threads of a process that can be executing in parallel?

   Ans: 2

5. Consider a user level threading library that multiplexes $N > 1$ user level threads over $M \geq 1$ kernel threads. The library manages the concurrent scheduling of the multiple user threads that map to the same kernel thread internally, and the programmer using the library has no visibility or control on this scheduling or on the mapping between user threads and kernel threads. The N user level threads all access and update a shared data structure. When (or, under what conditions) should the user level threads use mutexes to guarantee the consistency of the shared data structure?

   (a) Only if $M > 1$.

   (b) Only if $N \geq M$.

   (c) Only if the M kernel threads can run in parallel on a multi-core machine.

(d) User level threads should always use mutexes to protect shared data.

Ans: (d) (because user level threads can execute concurrently even on a single core)

6. Which of the following statements is/are true regarding user-level threads and kernel threads?

   (a) Every user level thread always maps to a separate schedulable entity at the kernel.

   (b) Multiple user level threads can be multiplexed on the same kernel thread

   (c) Pthreads library is used to create kernel threads that are scheduled independently.

   (d) Pthreads library only creates user threads that cannot be scheduled independently at the kernel scheduler.

   Ans: (b), (c)

7. Consider a Linux application with two threads T1 and T2 that both share and access a common variable x. Thread T1 uses a `pthread` mutex lock to protect its access to x. Now, if thread T2 tries to write to x without locking, then the Linux kernel generates a trap. [T/F]

   Ans: F

8. In a single processor system, the kernel can simply disable interrupts to safely access kernel data structures, and does not need to use any spin locks. [T/F]

   Ans: T

9. In the `pthread` condition variable API, a process calling wait on the condition variable must do so with a mutex held. State one problem that would occur if the API were to allow calls to wait without requiring a mutex to be held.

   Ans: Wakeup happening between checking for condition and sleeping causing missed wakeup.

10. Consider N threads in a process that share a global variable in the program. If one thread makes a change to the variable, is this change visible to other threads? (Yes/No)

    Ans: Yes

11. Consider N threads in a process. If one thread passes certain arguments to a function in the program, are these arguments visible to the other threads? (Yes/No)

    Ans: No

12. Consider a user program thread that has locked a pthread mutex lock (that blocks when waiting for lock to be released) in user space. In modern operating systems, can this thread be context switched out or interrupted while holding the lock? (Yes/No)

    Ans: Yes

13. Repeat the previous question when the thread holds a pthread spinlock in user space.

    Ans: Yes

14. Consider a process that has switched to kernel mode and has acquired a spinlock to modify a kernel data structure. In modern operating systems, will this process be interrupted by external hardware before it releases the spinlock? (Yes/No)

    Ans: No

15. Consider a process that has switched to kernel mode and has acquired a spinlock to modify a kernel data structure. In modern operating systems, will this process initiate a disk read before it releases the spinlock? (Yes/No)

    Ans: No

16. When a user space process executes the wakeup/signal system call on a pthread condition variable, does it always lead to an immediate context switch of the process that calls signal (immediately after the signal instruction)? (Yes/No)

    Ans: No

17. Consider a process in kernel mode that acquires a spinlock. For correct operation, it must disable interrupts on its CPU core for the duration that the spinlock is held, in both single core and multicore systems. [T/F]

    Ans. T

18. Consider a process in kernel mode that acquires a spinlock in a multicore system. For correct operation, we must ensure that no other kernel-mode process running in parallel on another core will request the same spinlock. [T/F]

    Ans. F

19. Multiple threads of a program must use locks when accessing shared variables even when executing on a single core system. [T/F]

    Ans: T

20. What is the output of the multi-threaded program below?

```
int x = 3;
int a = 3;
void *print_int(void *arg) {
  int k = *((int *)arg);
  printf("k = %d\n", k);
}
void main() {
  pthread_t threads[x+1];
  while(x > 0) {
    pthread_create(&threads[x], NULL, print_int, &a);
    x--;
  }
  //suitable code to join all threads }
```

    Ans: k = 3 printed three times

21. What is the output of the multi-threaded program below? Note that this program is the same as the one above, except that the argument passed to the thread create function is not the variable `a` but the variable `x`.

```
int x = 3;
int a = 3;
void *print_int(void *arg) {
  int k = *((int *)arg);
  printf("k = %d\n", k);
}
void main() {
  pthread_t threads[x+1];
  while(x > 0) {
    pthread_create(&threads[x], NULL, print_int, &x);
    x--;
  }
  //suitable code to join all threads }
```

Ans: Output is hard to predict, as value of x changes by the time we pass the argument. Some combinations of 3, 2, 1, 0 can get printed when the three threads print k.

22. Recall that the atomic instruction compare-and-swap (CAS) works as follows:
`CAS(&var, oldval, newval)` writes `newval` into `var` and returns true if the old value of `var` is `oldval`. If the old value of `var` is not `oldval`, `CAS` returns false and does not change the value of the variable. Write code for the function to acquire a simple spinlock using the `CAS` instruction.

    Ans: while(!CAS(&lock, 0, 1));

23. The simple spinlock implementation studied in class does not guarantee any kind of fairness or FIFO order amongst the threads contending for the spin lock. A ticket lock is a spinlock implementation that guarantees a FIFO order of lock acquisition amongst the threads contending for the lock. Shown below is the code for the function to acquire a ticket lock. In this function, the variables `next_ticket` and `now_serving` are both global variables, shared across all threads, and initialized to 0. The variable `my_ticket` is a variable that is local to a particular thread, and is not shared across threads. The atomic instruction `fetch_and_increment(&var)` atomically adds 1 to the value of the variable and returns the old value of the variable.

```
acquire():
  my_ticket = fetch_and_increment(&next_ticket)
  while(now_serving != my_ticket); //busy wait
```

You are now required to write the code to release the spinlock, to be executed by the thread holding the lock. Your implementation of the release function must guarantee that the next contending thread (in FIFO order) will be able to acquire the lock correctly. You must not declare or use any other variables.

```
release(): //your code here
```

Ans:

```
release(): //your code here
now_serving++;
```

24. Consider a multithreaded program, where threads need to aquire and hold multiple locks at a time. To avoid deadlocks, all threads are mandated to use the function `acquire_locks`, instead of acquiring locks independently. This function takes as arguments a variable sized array of pointers to locks (i.e., addresses of the lock structure), and the number of lock pointers in the array, as shown in the function prototype below. The function returns once all locks have been successfully acquired.

```
void acquire_locks(struct lock *la[], int n);
//i-th lock in array can be locked by calling lock(la[i])
```

Describe (in English, or in pseudocode) one way in which you would implement this function, while ensuring that no deadlocks happen during lock acquisition. Your solution must not use any other locks beyond those provided as input. Note that multiple threads can invoke this function concurrently, possibly with an overlapping set of locks, and the lock pointers can be stored in the array in any arbitrary order. You may assume that the locks in the array are unique, and there are no duplicates within the input array of locks.

Ans. Sort locks by address struct lock *, and acquire in sorted order.

25. Consider the atomic hardware instruction `swap` that is used to implement locks in computer systems. The instruction `swap(var x, value v)` atomically replaces contents of variable `x` with value `v`, and returns the previous value of x.

    (a) Consider a basic spinlock, represented by a variable `locked`, that is set to 1 when the lock is held, and 0 otherwise. Write the line of code that is used to acquire such a spinlock by a thread, using the `swap` instruction in a busy-spin while loop.

    Ans: while(swap(locked, 1) == 1);

    (b) Now suppose we wish to implement a different kind of lock, which we will call the "Q" spinlock. The Q spinlock maintains a linked list of contending threads. To acquire a lock, a thread creates a `qnode` variable, consisting of a flag `waiting`, and a pointer to the next qnode in the list. The thread then adds its qnode to the tail of the lock's list, to indicate that it is contending for the lock. Nodes added to the lock's list have their `waiting` flag set to true initially. A contending thread continues to spin as long as this flag is true. When a thread holding the lock wishes to release the lock, it sets the `waiting` flag of the next qnode after it to false, thereby ending the next node's busy spinning, and handing it the lock.

    Shown below is the pseudocode for acquiring the Q spinlock. The arguments to this function are the `qnode` being added to the list, and the corresponding `lock` that is being acquired. The qnode has a `qnode.waiting` flag and a `qnode.next` pointer to next node. The lock has a `lock.tail` variable which points to the current tail of the lock's list. To acquire a lock, a qnode is added to the tail of the list in two steps: using the `swap` instruction to move the tail pointer to the new qnode, and setting the next pointer of the previous tail node to point to the new qnode. The thread then busily spins for the `waiting` flag of this qnode to become false. Fill in the **four** blanks below to correctly implement the functionality of the

lock acquire function described above. Write your answer in the space provided below. Note that a simplified code for releasing the lock is provided for your reference.

```
acquire(lock, qnode):
  qnode.next = NULL;
  prev_tail = swap(________ , ________); //fill both blanks
  if(prev_tail != NULL) {
    //list is not empty, someone else has lock
    qnode.waiting = true;
    prev_tail.next = ___________; //fill blank
    while(______________); //fill blank for busy wait
  }
release(lock, qnode):
  if(qnode.next != NULL) qnode.next.waiting = false;
```

Ans: `swap(lock.tail, qnode)`
`prev_tail.next = qnode`
`while(qnode.waiting);`

26. Consider a process where multiple threads share a common Last-In-First-Out data structure. The data structure is a linked list of "struct node" elements, and a pointer "top" to the top element of the list is shared among all threads. To push an element onto the list, a thread dynamically allocates memory for the struct node on the heap, and pushes a pointer to this struct node in to the data structure as follows.

```
void push(struct node *n) {
n->next = top;
top = n;
}
```

A thread that wishes to pop an element from the data structure runs the following code.

```
struct node *pop(void) {
struct node *result = top;
if(result != NULL) top = result->next;
return result;
}
```

A programmer who wrote this code did not add any kind of locking when multiple threads concurrently access this data structure. As a result, when multiple threads try to push elements onto this structure concurrently, race conditions can occur and the results are not always what one would expect. Suppose two threads T1 and T2 try to push two nodes n1 and n2 respectively onto the data structure at the same time. If all went well, we would expect the top two elements of the data structure would be n1 and n2 in some order. However, this correct result is not guaranteed when a race condition occurs.

Describe how a race condition can occur when two threads simultaneously push two elements onto this data structure. Describe the exact interleaving of executions of T1 and T2 that causes the race condition, and illustrate with figures how the data structure would look like at various phases during the interleaved execution.

Ans: One possible race condition is as follows. n1's next is set to top, then n2's next is set to top. So both n1 and n2 are pointing to the old top. Then top is set to n1 by T1, and then top is set to n2 by T2. So, finally, top points to n2, and n2's next points to old top. But now, n1 is not accessible by traversing the list from top, and n1 remains on a side branch of the list.

27. Consider the following scenario. A town has a very popular restaurant. The restaurant can hold N diners. The number of people in the town who wish to eat at the restaurant, and are waiting outside its doors, is much larger than N. The restaurant runs its service in the following manner. Whenever it is ready for service, it opens its front door and waits for diners to come in. Once N diners enter, it closes its front door and proceeds to serve these diners. Once service finishes, the backdoor is opened and the diners are let out through the backdoor. Once all diners have exited, another batch of N diners is admitted again through the front door. This process continues indefinitey. The restaurant does not mind if the same diner is part of multiple batches.

    We model the diners and the restaurant as threads in a multithreaded program. The threads must be synchronized as follows. A diner cannot enter until the restaurant has opened its front door to let people in. The restaurant cannot start service until N diners have come in. The diners cannot exit until the back door is open. The restaurant cannot close the backdoor and prepare for the next batch until all the diners of the previous batch have left.

    Below is given unsynchronized pseudocode for the diner and restaurant threads. Your task is to complete the code such that the threads work as desired. Please write down the complete synchronized code of each thread in your solution.

    You are given the following variables (semaphores and initial values, integers) to use in your solution. The names of the variables must give you a clue about their possible usage. You must not use any other variable in your solution.

```
sem (init to 0): entering_diners, exiting_diners, enter_done, exit_done
sem (init to 1): mutex_enter, mutex_exit
Integer counters (init to 0): count_enter, count_exit
```

    All changes to the counters and other variables must be done by you in your solution. None of the actions performed by the unsynchronized code below will modify any of the variables above.

    (a) Unsynchronized code for the restaurant thread is given below. Add suitable synchronization in your solution in between these actions of the restaurant.

```
openFrontDoor()
closeFrontDoor()
serveFood()
openBackDoor()
closeBackDoor()
```

    (b) Unsynchronized code for the diner thread is given below. Add suitable synchronization in your solution around these actions of the diner.

```
enterRestaurant()
eat()
exitRestaurant()
```

Ans: Correct code for restautant thread:

```
openFrontDoor()
do N times: up(entering_diners)
down(enter_done)

closeFrontDoor()
serveFood()

openBackDoor()
do N times: up(exiting_diners)
down(exit_done)
closeBackDoor()
```

Correct code for the diner thread:

```
down(entering_diners)
enterRestaurant()

down(mutex_enter)
  count_enter++
  if(count_enter == N) {
    up(enter_done)
    count_enter = 0
    }
up(mutex_enter)

eat()

down(exiting_diners)
exitRestaurant()

down(mutex_exit)
  count_exit++
  if(count_exit == N) {
    up(exit_done)
    count_exit = 0
    }
up(mutex_exit)
```

An alternate to doing up N times in restaurant thread is: restaurant does up once, and every woken up diner does up once until N diners are done. This alternate solution is shown below. Correct code for restautant thread:

```
openFrontDoor()
up(entering_diners)
down(enter_done)

closeFrontDoor()
serveFood()

openBackDoor()
up(exiting_diners)
down(exit_done)
closeBackDoor()
```

Correct code for the diner thread:

```
down(entering_diners)
enterRestaurant()

down(mutex_enter)
  count_enter++

  if(count_enter < N)
      up(entering_diners)
  else if(count_enter == N) {
    up(enter_done)
    count_enter = 0
    }
up(mutex_enter)

eat()

down(exiting_diners)
exitRestaurant()

down(mutex_exit)
  count_exit++
  if(count_exit < N)
     up(exiting_diners)
  else if(count_exit == N) {
    up(exit_done)
    count_exit = 0
    }
up(mutex_exit)
```

28. Consider a scenario where a bus picks up waiting passengers from a bus stop periodically. The bus has a capacity of K. The bus arrives at the bus stop, allows up to K waiting passengers (fewer if less than K are waiting) to board, and then departs. Passengers have to wait for the bus to arrive and then board it. Passengers who arrive at the bus stop after the bus has arrived should not be allowed to board, and should wait for the next time the bus arrives. The bus and passengers are represented by threads in a program. The passenger thread should call the function board() after the passenger has boarded and the bus should invoke depart() when it has boarded the desired number of passengers and is ready to depart.

    The threads share the following variables, none of which are implicitly updated by functions like board() or depart().

    ```
    mutex = semaphore initialized to 1.
    bus_arrived = semaphore initialized to 0.
    passenger_boarded = semaphore initialized to 0.
    waiting_count = integer initialized to 0.
    ```

    Below is given synchronized code for the passenger thread. You should not modify this in any way.

    ```
    down(mutex)
    waiting_count++
    up(mutex)
    down(bus_arrived)
    board()
    up(passenger_boarded)
    ```

    Write down the corresponding synchronized code for the bus thread that achieves the correct behavior specified above. The bus should board the correct number of passengers, based on its capacity and the number of those waiting. The bus should correctly board these passengers by calling up/down on the semaphores suitably. The bus code should also update waiting_count as required. Once boarding completes, the bus thread should call depart(). You can use any extra local variables in the code of the bus thread, like integers, loop indices and so on. However, you must not use any other extra synchronization primitives.

    Ans:

    ```
    down(mutex)
    N = min(waiting_count, K)
    for i= 1 to N
      up(bus_arrived)
      down(passenger_boarded)
    waiting_count = waiting_count - N
    up(mutex)
    depart()
    ```

29. Consider the following synchronization puzzle with a bus and passengers, which can be thought of as threads in a program. The bus arrives periodically to pick up waiting passengers at a bus stop. The bus has the capacity to seat C passengers. When it arrives at the bus stop, it boards up to C waiting passengers (fewer if less than C are waiting), and then departs. Passengers arriving at the bus stop have to wait until the arrival of the bus, and then board. Passengers who arrive after the bus has started boarding must wait for the next arrival of the bus to board. You must write code to synchronize the bus and passenger threads using mutexes and condition variables.

    You must use the following variables shared across all threads in the program: a lock `mutex`, two condition variables `cvP` and `cvB`, two counters `Pcount` and `Bcount`, both initialized to 0. You can declare additional temporary variables if required. The code for the passenger thread is given to you. You must complete the code for the bus thread. Your code must carefully update the counters as well, and keep them consistent across multiple passengers and multiple arrivals of the bus.

```
passenger_thread {
  lock(m)
  Pcount++
  wait(cvP, m)
  board()
  Bcount++
  signal(cvB)
  unlock(m)
}

bus_thread {
  // Write your code here









  depart()
}
```

Ans:

```
bus_thread {
  lock(m)
  N = min(C, Pcount)
  do N times: signal(cvP)
  Pcount -= N
  Bcount = 0
  while(Bcount < N) wait(cvB, m)
  unlock(m)

  depart()
}
```

There are other minor variants of this solution that update the counters slightly differently.

An alternate solution is shown below, but it is not fully accurate. In the below code, the bus signals each passenger one at a time, and waits to board him. When the lock is released during this wait, other new passengers can join the queue which we do not want (because passengers that arrive after the bus has started to board must not join the queue now). But otherwise, this solution is partially correct as well.

```
bus_thread {
  lock(m)
  N = min(C, Pcount)
  do N times: {
    signal(cvP)
    Pcount--
    wait(cvB, m)
    Bcount++
  }
  unlock(m)

  depart()
}
```

30. Consider a roller coaster ride at an amusement park. The ride operator runs the ride only when there are exactly N riders on it. Multiple riders arrive at the ride and queue up at the entrance of the ride. The ride operator waits for N riders to accumulate, and may even take a nap as he waits. Once N riders have arrived, the riders call out to the operator indicating they are ready to go on the ride. The operator then opens the gate to the ride and signals exactly N riders to enter the ride. He then waits until these N riders enter the ride, and then proceeds to start the ride.

    We model the operator and riders as threads in a program. You must write pseudocode for the operator and rider threads to enable the behavior described above. Shown below is the skeleton code for the operator and rider threads. Complete the code to achieve the behavior described above. You can assume that the functions to open, start, and enter ride are implemented elsewhere, and these functions do what the names say they do. You must write the synchronization logic around these functions in order to invoke these functions at the appropriate times. You must use only locks and condition variables for synchronization in your solution. You may declare, initialize, and use other variables (counters etc.) as required in your solution.

```
//operator code, fill in the missing details

....
open_ride()
....
start_ride()
....

//rider thread, fill in the missing details
....
enter_ride()
....
```

Ans:

```
//variables: int rider_count (initialized to 0)
//variables: int enter_count (initialized to 0)
//condvar cv_rider, cv_operator1, cv_operator2
//mutex

//operator
lock(mutex)
while(rider_count < N) wait(cv_operator1, mutex)

open_ride()
do N times: signal(cv_rider)
while(enter_count < N) wait(cv_operator2, mutex)

start_ride()
unlock(mutex)

//rider
lock(mutex)
rider_count++
if(rider_count == N) signal(cv_operator1)
wait(cv_rider, mutex) // all wait, even N-th guy

enter_ride()

enter_count++
if(enter_count == N) signal(cv_operator2)
unlock(mutex)
```

31. A host of a party has invited $N > 2$ guests to his house. Due to fear of Covid-19 exposure, the host does not wish to open the door of his house multiple times to let guests in. Instead, he wishes that all N guests, even though they may arrive at different times to his door, wait for each other and enter the house all at once. The host and guests are represented by threads in a multi-threaded program. Given below is the pseudocode for the host thread, where the host waits for all guests to arrive, then calls openDoor(), and signals a condition variable once. You must write the corresponding code for the guest threads. The guests must wait for all N of them to arrive and for the host to open the door, and must call enterHouse() only after that. You must ensure that all N waiting guests enter the house after the door is opened. You must use only locks and condition variables for synchronization.

    The following variables are used in this solution: lock m, condition variables cv_host and cv_guest, and integer guest_count (initialized to 0). You must not use any other variables in the guest for synchronization.

```
//host
lock(m)
while(guest_count < N)
  wait(cv_host, m)
openDoor()
signal(cv_guest)
unlock(m)
```

Ans:

```
//guest
lock(m)
guest_count++
if(guest_count == N)
  signal(cv_host)
wait(cv_guest, m)
signal(cv_guest)
unlock(m)
enterHouse()
```

32. Consider the classic readers-writers synchronization problem described below. Several processes/threads wish to read and write data shared between them. Some processes only want to read the shared data ("readers"), while others want to update the shared data as well ("writers"). Multiple readers may concurrently access the data safely, without any correctness issues. However, a writer must not access the data concurrently with anyone else, either a reader or a writer. While it is possible for each reader and writer to acquire a regular mutex and operate in perfect mutual exclusion, such a solution will be missing out on the benefits of allowing multiple readers to read at the same time without waiting for other readers to finish. Therefore, we wish to have special kind of locks called reader-writer locks that can be acquired by processes/threads in such situations. These locks have separate lock/unlock functions, depending on whether the thread asking for a lock is a reader or writer. If one reader asks for a lock while another reader already has it, the second reader will also be granted a read lock (unlike in the case of a regular mutex), thus encouraging more concurrency in the application.

    Write down pseudocode to implement the functions readLock, readUnlock, writeLock, and write-Unlock that are invoked by the readers and writers to realize reader-writer locks. You must use condition variables and mutexes only in your solution.

    Ans: A boolean variable `writer_present`, and two condition variables, `reader_can_enter` and `writer_can_enter`, are used.

```
readLock:
lock(mutex)
while(writer_present)
  wait(reader_can_enter)
read_count++
unlock(mutex)

readUnlock:
lock(mutex)
read_count--
if(read_count==0)
  signal(writer_can_enter)
unlock(mutex)

writeLock:
lock(mutex)
while(read_count > 0 || writer_present)
  wait(writer_can_enter)
writer_present = true
unlock(mutex)

writeUnlock:
lock(mutex)
writer_present = false
signal(writer_can_enter)
signal_broadcast(reader_can_enter)
unlock(mutex)
```

33. Consider the readers and writers problem discussed above. Recall that multiple readers can be allowed to read concurrently, while only one writer at a time can access the critical section. Write down pseudocode to implement the functions readLock, readUnlock, writeLock, and writeUnlock that are invoked by the readers and writers to realize read/write locks. You must use **only** semaphores, and no other synchronization mechanism, in your solution. Further, you must avoid using more semaphores than is necessary. Clearly list all the variables (semaphores, and any other flags/counters you may need) and their initial values at the start of your solution. Use the notation `down(x)` and `up(x)` to invoke atomic down and up operations on a semaphore `x` that are available via the OS API. Use sensible names for your variables.

Ans:

```
sem lock = 1; sem writer_can_enter = 1; int readCount = 0;

readLock:
down(lock)
readCount++
if(readCount ==1)
  down(writer_can_enter) //don't coexist with a writer
up(lock)

readUnlock:
down(lock)
readCount--
if(readCount == 0)
  up(writer_can_enter)
up(lock)

writeLock:
down(writer_can_enter)

writeUnlock:
up(writer_can_enter)
```

34. Consider the readers and writers problem as discussed above. We wish to implement synchronization between readers and writers, while giving **preference to writers**, where no waiting writer should be kept waiting for longer than necessary. For example, suppose reader process R1 is actively reading. And a writer process W1 and reader process R2 arrive while R1 is reading. While it might be fine to allow R2 in, this could prolong the waiting time of W1 beyond the absolute minimum of waiting until R1 finishes. Therefore, if we want writer preference, R2 should not be allowed before W1. Your goal is to write down pseudocode for read lock, read unlock, write lock, and write unlock functions that the processes should call, in order to realize read/write locks with writer preference. You must use only simple locks/mutexes and conditional variables in your solution. Please pick sensible names for your variables so that your solution is readable.

Ans:

```
readLock:
lock(mutex)
while(writer_present || writers_waiting > 0)
    wait(reader_can_enter,mutex)
readcount++
unlock(mutex)

readUnlock:
lock(mutex)
readcount--
if(readcount==0)
    signal(writer_can_enter)
unlock(mutex)

writeLock:
lock(mutex)
writer_waiting++
while(readcount > 0 || writer_present)
    wait(writer_can_enter, mutex)
writer_waiting--
writer_present = true
unlock(mutex)

writeUnlock:
lock(mutex)
writer_present = false
if(writer_waiting==0)
    signal_broadcast(reader_can_enter)
else
    signal(writer_can_enter)
unlock(mutex)
```

35. Write a solution to the readers-writers problem with preference to writers discussed above, but using only semaphores.

Ans:

```
sem rlock = 1; sem wlock = 1;
sem reader_can_try = 1; sem writer_can_enter = 1;
int readCount = 0; int writeCount = 0;

readLock:
down(reader_can_try) //new sem blocks reader if writer waiting
down(rlock)
readCount++
if(readCount ==1)
  down(writer_can_enter) //don't coexist with a writer
up(rlock)
up(reader_can_try)

readUnlock:
down(rlock)
readCount--
if(readCount == 0)
  up(writer_can_enter)
up(rlock)

writeLock:
down(wlock)
writerCount++
if(writerCount==1)
  down(reader_can_try)
up(wlock)
down(writer_can_enter)  //release wlock and then block

writeUnlock:
down(wlock)
writerCount--
if(writerCount == 0)
  up(reader_can_try)
up(wlock)

up(writer_can_enter)
```

36. Consider the famous dining philosophers' problem. N philosophers are sitting around a table with N forks between them. Each philosopher must pick up both forks on her left and right before she can start eating. If each philosopher first picks the fork on her left (or right), then all will deadlock while waiting for the other fork. The goal is to come up with an algorithm that lets all philosophers eat, without deadlock or starvation. Write a solution to this problem using condition variables.

    Ans: A variable `state` is associated with each philosopher, and can be one of EATING (holding both forks) or THINKING (when not eating). Further, a condition variable is associated with each philosopher to make them sleep and wake them up when needed. Each philosopher must call the `pickup` function before eating, and `putdown` function when done. Both these functions use a mutex to change states only when both forks are available.

```
bothForksFree(i):
return (state[leftNbr(i)] != EATING &&
        state[rightNbr(i)] != EATING)

pickup(i):
  lock(mutex)
  while(!bothForksFree(i))
      wait(condvar[i])
  state[i] = EATING
  unlock(mutex)

putdown(i):
  lock(mutex)
  state[i] = THINKING
  if(bothForksFree(leftNbr(i)))
      signal(leftNbr(i))
  if(bothForksFree(rightNbr(i)))
      signal(rightNbr(i))
  unlock(mutex)
```

37. Consider a clinic with one doctor and a very large waiting room (of infinite capacity). Any patient entering the clinic will wait in the waiting room until the doctor is free to see her. Similarly, the the doctor also waits for a patient to arrive to treat. All communication between the patients and the doctor happens via a shared memory buffer. Any of the several patient processes, or the doctor process can write to it. Once the patient "enters the doctors office", she conveys her symptoms to the doctor using a call to `consultDoctor()`, which updates the shared memory with the patient's symptoms. The doctor then calls `treatPatient()` to access the buffer and update it with details of the treatment. Finally, the patient process must call `noteTreatment()` to see the updated treatment details in the shared buffer, before leaving the doctor's office. A template code for the patient and doctor processes is shown below. Enhance this code to correctly synchronize between the patient and the doctor processes. Your code should ensure that no race conditions occur due to several patients overwriting the shared buffer concurrently. Similarly, you must ensure that the doctor accesses the buffer only when there is valid new patient information in it, and the patient sees the treatment only after the doctor has written it to the buffer. You must use **only semaphores** to solve this problem. Clearly list the semaphore variables you use and their initial values first. Please pick sensible names for your variables.

Ans:

(a) Semaphores variables:

```
pt_waiting = 0
treatment_done = 0
doc_avlbl = 1
```

(b) Patient process:

```
down(doc_avlbl)
consultDoctor()
up(pt_waiting)
down(treatment_done)
noteTreatment()
up(doc_avlbl)
```

(c) Doctor:

```
while(1) {
down(pt_waiting)
treatPatient()
up(treatment_done)
}
```

38. Consider a multithreaded banking application. The main process receives requests to tranfer money from one account to the other, and each request is handled by a separate worker thread in the application. All threads access shared data of all user bank accounts. Bank accounts are represented by a unique integer account number, a balance, and a lock of type `mylock` (much like a `pthreads` mutex) as shown below.

```
struct account {
int accountnum;
int balance;
mylock lock;
};
```

Each thread that receives a transfer request must implement the transfer function shown below, which transfers money from one account to the other. Add correct locking (by calling the `dolock(&lock)` and `unlock(&lock)` functions on a `mylock` variable) to the tranfer function below, so that no race conditions occur when several worker threads concurrently perform transfers. Note that you must use the fine-grained per account lock provided as part of the account object itself, and not a global lock of your own. Also make sure your solution is deadlock free, when multiple threads access the same pair of accounts concurrently.

```
void transfer(struct account *from, struct account *to, int amount) {

from->balance -= amount; // dont write anything...
to->balance += amount; // ...between these two lines

}
```

Ans: The accounts must be locked in order of their account numbers. Otherwise, a transfer from account X to Y and a parallel transfer from Y to X may acquire locks on X and Y in different orders and end up in a deadlock.

```
struct account *lower = (from->accountnum < to->accountnum)?from:to;
struct account *higher = (from->accountnum < to->accountnum)?to:from;
dolock(&(lower->lock));
dolock(&(higher->lock));

from->balance -= amount;
to->balance += amount;

unlock(&(lower->lock));
unlock(&(higher->lock));
```

39. Consider a process with three threads A, B, and C. The default thread of the process receives multiple requests, and places them in a request queue that is accessible by all the three threads A, B, and C. For each request, we require that the request must first be processed by thread A, then B, then C, then B again, and finally by A before it can be removed and discarded from the queue. Thread A must read the next request from the queue only after it is finished with all the above steps of the previous one. Write down code for the functions run by the threads A, B, and C, to enable this synchronization. You can only worry about the synchronization logic and ignore the application specific processing done by the threads. You may use any synchronization primitive of your choice to solve this question.

Ans: Solution using semaphores shown below. The order of processing is A1–B1–C–B2–A2. All threads run in a forever loop, and wait as dictated by the semaphores.

```
sem a1done = 0; b1done = 0; cdone = 0; b2done = 0;

ThreadA:
  get request from queue and process
  up(a1done)
  down(b2 done)
  finish with request

ThreadB:
  down(a1done)
  //do work
  up(b1done)
  down(cdone)
  //do work
  up(b2done)

ThreadC:
  down(b1done)
  //do work
  up(cdone)
```

40. Consider two threads A and B that perform two operations each. Let the operations of thread A be A1 and A2; let the operations of thread B be B1 and B2. We require that threads A and B each perform their first operation before either can proceed to the second operation. That is, we require that A1 be run before B2 and B1 before A2. Consider the following solutions based on semaphores for this problem (the code run by threads A and B is shown in two columns next to each other). For each solution, explain whether the solution is correct or not. If it is incorrect, you must also point out why the solution is incorrect.

(a)
```
sem A1Done = 0;  sem B1Done = 0;
//Thread A                //Thread B
A1                         B1
down(B1Done)               down(A1Done)
up(A1Done)                 up(B1Done)
A2                         B2
```

(b)
```
sem A1Done = 0;  sem B1Done = 0;
//Thread A               //Thread B
A1                         B1
down(B1Done)               up(B1Done)
up(A1Done)                 down(A1Done)
A2                         B2
```

(c)
```
sem A1Done = 0;  sem B1Done = 0;
//Thread A                //Thread B
A1                         B1
up(A1Done)                 up(B1Done)
down(B1Done)               down(A1Done)
A2                         B2
```

Ans:

(a) Deadlocks, so incorrect.

(b) Correct

(c) Correct

41. Now consider a generalization of the above problem for the case of N threads that want to each execute their first operation before any thread proceeds to the second operation. Below is the code that each thread runs in order to achieve this synchronization. `count` is an integer shared variable, and `mutex` is a mutex binary semaphore that protects this shared variable. `step1Done` is a semaphore initialized to zero. You are told that this code is wrong and does not work correctly. Further, you can fix it by changing it slightly (e.g., adding one statement, or rearranging the code in some way). Suggest the change to be made to the code in the snippet below to fix it. You must use only semaphores and no other synchronization mechanism.

```
//run first step

down(mutex);
count++;
up(mutex);
if(count == N)
       up(step1Done);
down(step1Done);

//run second step
```

Ans: The problem is that the semaphore is decremented N times, but is only incremented once. To fix it, we must do up N times when count is N. Or, add up after the last down, so that it is performed N times by the N threads.

42. The cigarette smokers problem is a classical synchronization problem that involves 4 threads: one agent and three smokers. The smokers require three ingredients to smoke a cigarette: tobacco, paper, and matches. Each smoker has one of the three ingredients and waits for the other two, smokes the cigar once he obtains all ingredients, and repeats this forever. The agent repeatedly puts out two ingredients at a time and makes them available. In the correct solution of this problem, the smoker with the complementary ingredient should finish smoking his cigar. Consider the following solution to the problem. The shared variables are three semaphores `tobacco`, `paper` and `matches` initialized to 0, and semaphore `doneSmoking` initialized to 1. The agent code performs `down(doneSmoking)`, then picks two of the three ingredients at random and performs `up` on the corresponding two semaphores, and repeats. The smoker with tobacco runs the following code in a loop.

```
down(paper)
down(matches)
//make and smoke cigar
up(doneSmoking)
```

Similarly, the smoker with matches waits for tobacco and paper, and the smoker with paper waits for tobacco and matches, before signaling the agent that they are done smoking. Does the code above solve the synchronization problem correctly? If you answer yes, provide a justification for why the code is correct. If you answer no, describe what the error is and also provide a correct solution to the problem. (If you think the code is incorrect and are providing another solution, you may change the code of both the agent and the smokers. You can also introduce new variables as necessary. You must use only semaphores to solve the problem.)

Ans: The code is incorrect and deadlocks. One fix is to add semaphores for two ingredients at a time (e.g., tobaccoAndPaper). The smokers wait on these and the agent signals these. So there is no possibility of deadlock.

43. Consider a server program running in an online market place firm. The program receives buy and sell orders for one type of commodity from external clients. For every buy or sell request received by the server, the main process spawns a new buy or sell thread. We require that every buy thread waits until a sell thread arrives, and vice versa. A matched pair of buy and sell threads will both return a response to the clients and exit. You may assume that all buy/sell requests are identical to each other, so that any buy thread can be matched with any sell thread. The code executed by the buy thread is shown below (the code of the sell thread would be symmetric). You have to write the synchronization logic that must be run at the start of the execution of the thread to enable it to wait for a matching sell thread to arrive (if none exists already). Once the threads are matched, you may assume that the function `completeBuy()` takes care of the application logic for exchanging information with the matching thread, communicating with the client, and finishing the transaction. You may use any synchronization technique of your choice.

```
//declare any variables here

buy_thread_function:
  //start of sync logic


  //end of sync logic
  completeBuy();
```

Ans:

```
sem buyer = 0; sem seller = 0;

Buyer thread:

up(buyer)
down(seller)
completeBuy()
```

44. Consider the following classical synchronization problem called the barbershop problem. A barbershop consists of a room with N chairs. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs and awaits his turn. The barber moves onto the next waiting seated customer after he finishes one hair cut. If there are no customers to be served, the barber goes to sleep. If the barber is asleep when a customer arrives, the customer wakes up the barber to give him a hair cut. A waiting customer vacates his chair after his hair cut completes. Your goal is to write the pseudocode for the customer and barber threads below with suitable synchronization. You must use only semaphores to solve this problem. Use the standard notation of invoking up/down functions on a semaphore variable.

    The following variables (3 semaphores and a count) are provided to you for your solution. You must use these variables and declare any additional variables if required.

    ```
    semaphore mutex = 1, customers = 0, barber = 0;
    int waiting_count = 0;
    ```

    Some functions to invoke in your customer and barber threads are:

    - A customer who finds the waiting room full should call the function `leave()` to exit the shop permanently. This function does not return.
    - A customer should invoke the function `getHairCut()` in order to get his hair cut. This function returns when the hair cut completes.
    - The barber thread should call `cutHair()` to give a hair cut. When the barber invokes this function, there should be exactly one customer invoking `getHairCut()` concurrently.

Ans:

```
Customer:

down(mutex)
if(waiting_count == N)
  up(mutex)
  leave()
waiting_count++
up(mutex)

up(customers)
down(barber)

getHairCut()

down(mutex)
waiting_count--
up(mutex)

Barber:

up(barber)
down(customers)
cutHair()
```

45. Consider a multithreaded application server handling requests from clients. Every new request that arrives at the server causes a new thread to be spawned to handle that request. The server can provide service to only one request/thread at a time, and other threads that arrive when the server is busy must wait for service using a synchronization primitive (semaphore or condition variable). In order to avoid excessive waiting times, the server does not wish to have more than N requests/threads in the system (including the waiting requests and any request it is currently serving). You may assume that $N > 2$. Given this constraint, a newly arriving thread must first check if N other requests are already in the system: if yes, it must exit without waiting and return an error value to the client, by calling the function `thr_exit_failure()`. This function terminates the thread and does not return.

    When a thread is ready for service, it must call the function `get_service()`. Your code should ensure that no more than one thread calls this function at any point of time. This function blocks the thread for the duration of the service. Note that, while the thread receiving service is blocked, other arriving threads must be free to join the queue, or exit if the system is overloaded. After a thread returns from `get_service()`, it must enable one of the waiting threads to seek service (if any are waiting), and then terminate itself succesfully by calling the function `thr_exit_success()`. This function terminates the thread and does not return.

    You are required to write pseudocode of the function to be run by the request threads in this system, as per the specification above. Your solution must use only locks and condition variables for synchronization. Clearly state all the variables used and their initial values at the start of your solution.

Ans

```
int num_requests=0;
bool server_busy = false
cv, mutex

lock(mutex)

if(num_requests == N)
  unlock(mutex)
  the_exit_failure()

num_requests++

if(server_busy)
  wait(cv, mutex)

server_busy = true
unlock(mutex)

get_service()

lock(mutex)
num_requests--
server_busy = false

if(num_requests > 0)
  signal(cv)

unlock(mutex)
thr_exit_success()
```

46. Consider the previous problem, but now assume that N is infinity. That is, all arriving threads will wait (if needed) for their turn in the queue of a synchronization primitive, get served when their turn comes, and exit successfully. Write the pseudocode of the function to be run by the threads with this modified specification. Your solution must only use semaphores for synchronization, and only the correct solution that uses the least number of semaphores will get full credit. Clearly state all the variables used and their initial values at the start of your solution.

Ans

```
sem waiting = 1

down(waiting)
get_service()
up(waiting)
thr_exit_success()
```

47. Consider the following synchronization problem. A group of children are picking chocolates from a box that can hold up to N chocolates. A child that wants to eat a chocolate picks one from the box to eat, unless the box is empty. If a child finds the box to be empty, she wakes up the mother, and waits until the mother refills the box with N chocolates. Unsynchronized code snippets for the child and mother threads are as shown below:

```
//Child
while True:
  getChocolateFromBox()
  eat()

//Mother
while True:
  refillChocolateBox(N)
```

You must now modify the code of the mother and child threads by adding suitable synchronization such that a child invokes `getChocolateFromBox()` only if the box is non-empty, and the mother invokes `refillChocolateBox(N)` only if the box is fully empty. Solve this question using only locks and condition variables, and no other synchronization primitive. The following variables have been declared for use in your solution.

```
int count = 0;
mutex m; // you may invoke lock and unlock
condvar fullBox, emptyBox; //you may perform wait and signal
                           //or signal_broadcast
```

(a) Code for child thread

(b) Code for mother thread

Ans:

```
//Child
while True:
  lock(m)
  while(count == 0)
    signal(emptyBox)
    wait(fullBox, m)
  getChocolateFromBox()
  eat()
  count--
  signal(fullBox) //optional
  unlock(m)

//Mother
while True:
  lock(m)
  if(count > 0)
    wait(emptyBox, m)
  refillChocolateBox(N)
  count += N
  signal(fullBox)
  unlock(m)
```

There are two ways of waking up sleeping children. Either the mother does a signal broadcast to all children. Or every child that eats a chocolate wakes up another sleeping child. You may also assume that signal by mother wakes up all children.

48. Repeat the above question, but your solution now must use only semaphores and no other synchronization primitive. The following variables have been declared for use in your solution.

```
int count = 0;
semaphore m, fullBox, emptyBox;
//initial values of semaphores are not specified
//you may invoke up and down methods on a semaphore
```

(a) Initial values of the semaphores

(b) Code for child thread

(c) Code for mother thread

Ans:

```
m = 1, fullBox = 0, emptyBox = 0

//Child
while True:
  down(m)
  if(count == 0)
    up(emptyBox)
    down(fullBox)
    count += N
  getChocolateFromBox()
  eat()
  count--
  up(m)

//Mother
while True:
  down(emptyBox)
  refillChocolateBox(N)
  up(fullBox)
```

Here the subtlety is the lock m. Mother can't get lock to update count after filling the box, as that will cause a deadlock. In general, if child sleeps with mutex m locked, then mother cannot request the same lock.

49. Consider the classic "barrier" synchronization problem, where N threads wish to synchronize with each other as follows. N threads arrive into the system at different times and in any order. The arriving threads must wait until all N threads have arrived into the system, and continue execution only after all N threads have arrived. We wish to write logic to synchronize the threads in the manner stated above using semaphores. Below are three possible solutions to the problem. You are told that one of the solutions is correct and the other two are wrong. Identify the correct solution amongst the three given options. Further, for each of the other incorrect solutions, explain clearly why the solution is wrong. The following shared variables are declared for use in each solution.

```
int count = 0;
sem mutex; //initialized to 1
sem barrier; //initialized to 0
```

(a)
```
down(mutex)
  count++
  if(count == N) up(barrier)
up(mutex)

down(barrier)

//wait done; proceed to actual task
```

(b)
```
down(mutex)
  count++
  if(count == N) up(barrier)
up(mutex)

down(barrier)
up(barrier)

//wait done; proceed to actual task
```

(c)
```
down(mutex)
  count++
  if(count == N) up(barrier)
  down(barrier)
  up(barrier)
up(mutex)

//wait done; proceed to actual task
```

Ans: In (a) up is done only once when many threads are waiting on down. In (c), down(barrier) is called when mutex held, so code deadlocks. (b) is correct answer.

50. Consider the barrier synchronization primitive discussed in class, where the N threads of an application wait until all the threads have arrived at a barrier, before they proceed to do a certain task. You are now required to write the code for a reusable barrier, where the N application threads perform a series of steps in a loop, and use the same barrier code to synchronize for each iteration of the loop. That is, your solution should ensure that all threads wait for each other before the start of each step, and proceed to the next step only after all threads have completed the previous step. Your solution must only use semaphores. The following functions can be invoked on a semaphore `s` used in this question: `down(s)`, `up(s)`, and `up(s,n)`. While the first two functions are as studied in class, the function `up(s,n)` simply invokes `up(s)` n times atomically.

    We have provided you some code to get started. Shown below is the code to be run by each application thread, including the code to wait at the barrier. However, this is not the correct solution, as this code only works as a single-use barrier, i.e., it only ensures that the threads synchronize at the barrier once, and cannot be used to synchronize multiple times (can you figure out why?). You are required to modify this code to make it reusable, such that the threads can synchronize at the barrier multiple times for the multiple steps to be performed.

    Your solution must only use the following variables: `int count = 0`; and semaphores (initial values as given): `sem mutex = 1`; `sem barrier1 = 0`; `sem barrier2 = 0`;

```
For each step to be executed by the threads, do:

//add code here if required to make barrier reusable


down(mutex)
  count++
  if(count == N) up(barrier1, N)
up(mutex)
down(barrier1)

... wait done, execute actual task of this step ...

//add code here if required to make barrier reusable for next step
```

    Ans: The extra code to be added is at the end of completing a step, where you make all threads wait once again.

```
down(mutex)
count--
if(count==0) up(barrier2, N)
up(mutex)
down(barrier2)
```

51. Consider a web server that is supposed to serve a batch of N requests. Each request that arrives at the web server spawns a new thread. The arriving threads wait until N of them accumulate, at which point all of them proceed to get service from the server. Shown below is the code executed by each arriving thread, that causes it to wait until all the other threads arrive. The variable `count` is initialized to N. The code also uses `wait` and `signal` primitives on a condition variable; and you may assume that the signal primitive wakes up all waiting threads (not just one of them).

```
lock(mutex)
  count--;
unlock(mutex)

if(count > 0) {
  lock(mutex)
  wait(cv, mutex)
  unlock(mutex)
}
else {
  lock(mutex)
  signal(cv)
  unlock(mutex)
}

... wait done, proceed to server ...
```

You are told that the code above is incorrect, and can sometimes cause a deadlock. That is, in some executions, all N threads do not go to the server for service, even though they have arrived.

(a) Using an example, explain the exact sequence of events that can cause a deadlock. You must write your answers as bullet points, with one event per bullet point, starting from threads arriving in the system until the deadlock.

(b) Explain how you will fix this deadlock and correct the code shown above. You must retain the basic structure of the code. Indicate your changes next to the code snippet above.

Ans: The given incorrect solution may cause a missed wakeup. For example, some thread decides to wait and goes inside the if-loop, but is context switched out before calling wait (and before it acquires the lock). Now, if count hits 0 and signal happens before it runs again, it will wait with no one to wake it up, leading to deadlock. The fix is simply holding the lock all through the condition checking and waiting.

52. Consider an application that has $K + 1$ threads running on a Linux-like OS ($K > 1$). The first K threads of an application execute a certain task T1, and the remaining one thread executes task T2. The application logic requires that task T1 is executed $N > 1$ times, followed by task T2 executed once, and this cycle of N executions of T1 followed by one execution of T2 continue indefinitely. All K threads should be able to participate in the N executions of task T1, even though it is not required to ensure perfect fairness amongst the threads.

    Shown below is one possible set of functions executed by the threads running tasks T1 and T2. You are told that this solution has two bugs in the code run by the thread performing task T2. Briefly describe the bugs in the space below, and suggest small changes to the corresponding code to fix these bugs (you may write your changes next to the code snippet). You must not change the code corresponding to task T1 in any way. All threads share a counter `count` (initialized to 0), a mutex variable `m`, and two condition variables `t1cv`, and `t2cv`. Here, the function `signal` on a condition variable wakes up only one of the possibly many sleeping threads.

```
//function run by K threads of task T1
while True {
    lock(m)
    if(count >= N) {
        signal(t2cv)
        wait(t1cv, m)
    }
    //.. do task T1 once ..
    count++
    unlock(m)
}
//function run by thread of task T2
while True {
    lock(m)
    wait(t2cv, m)
    // .. do task T2 once
    count = 0
    signal(t1cv)
    unlock(m)
}
```

Ans: (a) check count<N and only then wait (b) signal broadcast instead of signal

53. You are now required to solve the previous question using semaphores for synchronization. You are given the pseudocode for the function run by the thread executing task T2 (which you must not change). You are now required to write the corresponding code executed by the K threads running task T1. You must use the following semaphores in your solution: `mutex`, `t1sem`, `t2sem`. You must initialize them suitably below. The variable `count` (initialized to 0) is also available for use in your solution.

Ans:

```
//fill in initial values of semaphores
sem_init(mutex,   );  sem_init(t1sem,   );   sem_init(t2sem,    );
//other variables
int count = 0

//function run by thread executing T2
while True {
    down(t2sem)
    //.. do task T2 ..
    up(t1sem)
}

//function run by threads executing task T1
while True {


}
```

Ans:

```
mutex=1, t1sem=0, t2sem=0

down(mutex)
if(count == N)
  up(t2sem)
  down(t1sem)
  count = 0

do task T1 once
count++
up(mutex)
```

54. Multiple people are entering and exiting a room that has a light switch. You are writing a computer program to model the people in this situation as threads in an application. You must fill in the functions `onEnter()` and `onExit()` that are invoked by a thread/person when the person enters and exits a room respectively. We require that the first person entering a room must turn on the light switch by invoking the function `turnOnSwitch()`, while the last person leaving the room must turn off the switch by invoking `turnOffSwitch()`. You must invoke these functions suitably in your code below. You may use any synchronization primitives of your choice to achieve this desired goal. You may also use any variables required in your solution, which are shared across all threads/persons.

    (a) Variables and initial values

    (b) Code `onEnter()` to be run by thread/person entering

    (c) Code `onExit()` to be run by thread/person exiting

    Ans:

```
variables: mutex, count

onEnter():
lock(mutex)
count++
if(count==1) turnOnSwitch()
unlock(mutex)

onExit():
lock(mutex)
count--
if(count==0) turnOffSwitch()
unlock(mutex)
```

    Note that this problem is very similar to the readers-writers problem.

55. Consider a database server process that has two types of threads: N database worker threads, and one garbage collector thread. The database datastructures are divided into N disjoint slices or shards. Each of the N worker threads is assigned one of these slices, and handles all database requests on its slice. All database threads run concurrently on a multicore system. The worker threads do not need to use any locks to access the shared database datastructures when other database workers are accessing them, because each worker thread operates on its own disjoint slice. Therefore, slicing the database allows all worker threads to run in parallel without locking, leading to improved performance of the database system on multicore servers. However, the garbage collector thread that runs periodically touches the entire database, and cleans up unused memory across all slices. Therefore, the garbage collector thread cannot run concurrently when any of the worker threads is accessing the database, and similarly, the worker threads must not touch the database when the garbage collector is cleaning it up.

    You must write code to synchronize the database worker threads and the garbage collector thread using semaphores to achieve the above behavior. Your solution must use two semaphores, `semLock` (initialized to 1) and `semGC` (initialized to 1), and an integer `count` (initialized to 0). The code for the garbage collector thread is given to you and must not be changed. You must complete the code that the database worker thread must invoke when it accesses the database to service a database read/write request.

```
garbage_collector {
  down(semGC)
  //perform garbage collection
  up(semGC)
}

database_worker {
  // fill code to run before accessing database





  // actual database access to read/write its slice
  ...

  //fill code to run after accessing the database





}
```

Ans:

```
database_worker {
  down(semLock)
  count++
  if(count == 1) down(semGC)
  up(semLock)

  //access database to read/write its slice

  down(semLock)
  count--
  if(count == 0) up(semGC)
  up(semLock)
}
```

Note that this problem is very similar to the readers-writers problem.

56. Consider the following classic synchronization problem. There are two kinds of threads that arrive in a system: hydrogen threads and oxygen threads. In order to assemble water correctly, we want the arriving threads to wait until one oxygen and two hydrogen threads are ready to bond into H_2O molecules. Every arriving thread, whether hydrogen or oxygen, invokes the corresponding "entry" function shown below. You must complete the entry functions for the hydrogen and oxygen threads in such a manner that the threads return from the entry functions in groups of exactly two hydrogen threads and one oxygen thread to bond, whenever such bonding is possible. Threads that are not yet ready to bond, because the matching threads have not arrived yet, should wait in these entry functions.

You must use condition variables and mutexes to solve this problem. The variables and synchronization primitives you will need to use in your solution are: a mutex `m`, two condition variables `cvH` and `cvO`, and two integers `waitH` and `waitO` both initialized to 0. You must not use any other variables. Fill in the missing parts of the code to achieve the desired synchronization, and write your answer in the space provided within the code itself.

```
hydrogen_entry {
  lock(m);
  waitH++;
  if(waitH >= 2 && waitO >= 1) {
    waitH -= 2; waitO -= 1;
    _____________;
    _____________;
  }
  else _____________;
  unlock(m);
}

oxygen_entry {
  lock(m);
  waitO++;
  if(waitH >= 2) {
    waitH -= 2; waitO -= 1;
    _____________;
    _____________;
  }
  else _____________;
  unlock(m);
}
```

Ans: signal(cvH), signal(cvO), wait(cvH, m), signal(cvH), signal(cvH), wait(cvO, m)

57. We will now solve the above water molecule problem, but using semaphores this time. The code snippets shown below use three semaphores: `semLock` (initialized to 1), `semHydrogen` (initialized to 0), `semOxygen` (initialized to 0), and one integer `count` initialized to 0. You must not use any other variables or semaphores in your solution. You are given the code of the hydrogen thread entry function. You must write the corresponding code of the oxygen thread entry function.

```
hydrogen_entry {
  down(semLock)
  count++
  if(count % 2 == 0) up(semOxygen)
  up(semLock)
  down(semHydrogen)
}

oxygen_entry {


}
```

Ans:

```
oxygen_entry {
  down(semOxygen)
  up(semHydrogen)
  up(semHydrogen)
}
```

58. Continuing on the previous question, we provide you one possible solution of the hydrogen and oxygen entry functions, which are supposed to work together. This solution uses only two semaphores, `semHydrogen` (initialized to 0) and `semOxygen` (initialized to 0). However, this solution shown below is incorrect and will not work in all cases. Explain the bug in the solution, by providing a specific example of a scenario where the solution will fail. Write your answer in the space next to the question.

```
hydrogen_entry {
  up(semOxygen)
  down(semHydrogen)
}
oxygen_entry {
  down(semOxygen)
  down(semOxygen)
  up(semHydrogen)
  up(semHydrogen)
}
```

Ans: This solution can lead to deadlocks sometime. For example, suppose two hydrogen threads arrive, and the oxygen semaphore has increased to 2. Now, two oxygen threads arrive, each does one down operation concurrently, and the semaphore is now back to 0. Now when both threads perform the second down operation, both will end up waiting. Ideally, since two hydrogens are already present, we would have wanted one of the oxygens to complete both down operations and finish bonding. But due to interleaving, both oxygens can get deadlocked.

59. Consider the following synchronization problem. Children arrive at a playing field to play a game. But before they enter the field, the children should form themselves into teams of 10 players, with one of them being the captain. This is achieved by the children waiting outside the field until 10 of them arrive. The 10th player (that is, the last player who completes the team) is automatically designated as the captain, and enters the field together with his team of 9 other players who arrived just before him. Other players who arrive after the captain is designated (but before the team enters the field) must wait until the previous team has entered the field, and must not interfere with the previous team's composition in any way.

    In the pseudocode shown below, the children are all represented by threads and synchronization between them is achieved using locks and condition variables. The code below uses the following variables: integer `count` (initialized to 0), condition variables `cv_player` and `cv_captain`, and a lock `mutex`. The solution shown below has a bug that leads to a deadlock under certain conditions. Please identify the bug and suggest a suitable solution to fix it with minor modifications to the code (using additional variables if required).

```
lock(mutex)
count++
if(count % 10 == 0) { //captain
  do 9 times {  signal(cv_player)  }
  count--
  while(count > 0) wait(cv_captain, mutex)
}
else { //player
  wait(cv_player, mutex)
  count--
  signal(cv_captain)
}
unlock(mutex)
//team is ready, enter field
```

    (a) Description of the bug

    Ans: The code works fine when only 10 players arrive, and the next team arrives only after the previous team has entered the field. Any overlaps between teams will lead to errors. For example, after the 10 player signals 9 times, and makes the count 9, the next player to come will increment count to 10 again, and think of himself as the captain. Players of the next team incrementing count may lead to the count never hitting 0, and the captain of the previous team never waking up.

    (b) Suggested change to the code.

    Ans: One solution is to use a separate counter to track the players of a team who are entering, carefully ensuring that the next team's changes to count do not impact the current team that is entering.

    Another solution is to use a different lock, which will be held while a team is entering the field, so that two consecutive teams cannot overlap. The code above can be fixed by using a new mutex (let's call it gate), and adding a statement lock(gate) to the start of the code. This mutex is unlocked by the players before their wait (allowing other players to enter), but

is released by the captain only after all signaling and waiting is done. This ensures that the "gate" is closed for future players while one team is entering the field. The complete solution is shown below.

```
lock(gate)
lock(mutex)
count++
if(count % 10 == 0) { //captain
  do 9 times: signal(cv_player)
  count--
  while(count > 0) wait(cv_captain, mutex)
  unlock(gate)
}
else { //player
  unlock(gate)
  wait(cv_player, mutex)  count--
  signal(cv_captain)
}
unlock(mutex)
//team is ready, enter field
```

60. Now, write a solution to the above problem using semaphores. You must use the following variables in your solution: integer `count` (initialized to 0), semaphore `mutex` (initialized to 1), semaphore `sem_player` (initialized to 0), and semaphore `sem_captain` (initialized to 0). Fill in the **five** blanks in the code below to achieve the desired synchronization.

```
down(mutex)
count++
if(count % 10 == 0) { //captain
  do 9 times {
    ____________________
    ____________________
  }
  ______________________
}
else {
  up(mutex)
  _________________
  _________________
}
//team is ready, enter field
```

Ans:

```
down(mutex)
count++
if(count % 10 == 0) { //captain
  do 9 times {
    up(sem_player)
    down(sem_captain)
  }
  up(mutex) //hold this mutex till team completes
}
else {
  up(mutex) //release to allow other players
  down(sem_player)
  up(sem_captain)
}
//team is ready, enter field
```