Locks and Condition Variables

Recap Lecture 13:

Concept	Key Point
Process vs Thread	Process created using fork() -> separate memory; Thread created using pthread_create() -> shared memory (code, data, heap)
Why Threads?	Parallelism, CPU efficiency, overlap I/O, and computation
Race Condition	Two threads access shared data concurrently, leading to an indeterminate result.
Mutual Exclusion	Only one thread executes the critical section at a time.
How to achieve it?	Using Locks.

Lecture 14:

Locks (Mutex Mechanism)

- Each critical section is protected by a lock.
- Only one thread can hold a lock at any moment.
- Other threads attempting to acquire it are forced to wait.
- Locks are implemented using hardware atomic instructions.

Test-and-Set (TAS) Instruction

- Hardware-provided atomic instruction.
- Performs read and set operation in one step.
- Returns the old value of a variable.

```
int TestAndSet(int *flag, int newValue) {
   int old = *flag;
   *flag = newValue;
   return old;
}
```

Usage:

- If TestAndSet(flag, 1) returns 1 -> lock already held, spin (busy-wait).
- If it returns 0 -> acquired the lock.

Spin Lock (Using Test-and-Set)

Algorithm:

```
while (TestAndSet(&lock, 1) == 1)
   ; // spin until lock becomes 0
```

This, ensures mutual exclusion (correctness) but wastes CPU cycles while spinning (performance). Under multiple threads -> unfair (some may starve).

Evaluating Spin Locks:

Property	Description
Correctness	Ensures mutual exclusion
Fairness	Not guaranteed as some threads may spin indefinitely
Performance	Very poor if many threads compete (CPU wasted in busy-wait)

Avoiding Wasteful Spinning:

Use yield(): Instead of spinning, the thread yields CPU voluntarily. Moves the thread from Running -> Ready state.

But this has some drawbacks:

- Still performance **overhead** due to frequent context switches.
- Possible **starvation** if scheduler repeatedly picks the wrong threads.
- Fairness **not guaranteed**.

A Better Approach? Sleep and Queues

- Threads sleep (block) instead of spin.
- Maintain a queue of threads waiting for the lock.

Mechanism:

1. If the lock is unavailable -> thread added to waiting queue -> sleep.
2. When the lock is released -> wake up the first sleeping thread.
3. Sleep = process moves Running -> Blocked.

4. More efficient and fairer than yield/spin.

Condition Variables (CVs): Locks provide mutual exclusion, but threads also need coordination. So we can use waiting and signaling.

Example scenario:

- Thread T1 must wait until T2 finishes a task.

Inefficient approach: busy-waiting; but a better option is to use Condition Variables (CVs).

Condition Variables: A queue where threads can wait until a condition becomes true. When another thread updates the condition, it signals the waiting threads.

Operations:

- wait(cv, lock) – releases lock and puts thread to sleep.
- signal(cv) – wakes one waiting thread.
- broadcast(cv) – wakes all waiting threads.

Example: Parent-Child Synchronization

- Thread Parent waits until Child finishes.
- Shared variable: done (0 = not done, **1 = done**)
- Child sets done = 1 and signals condition variable.

Why do we need the state variable? Because without it if child signals before parent sleeps, the signal is missed. Parent goes to sleep forever -> missed wakeup race condition.

done acts as a guard variable to check whether waiting is necessary.

Lock before wait() because if the lock is not held, it may lead to a possible race condition:

  - Parent checks done == 0, gets interrupted before sleeping.
  - Child sets done = 1, signals -> no one waiting.
  - Parent resumes and goes to sleep -> deadlock.

The correct way to use it is:

```
lock.acquire();
while (done == 0)
   wait(cv, lock);  // releases lock internally before sleeping
lock.release();
```

wait() automatically:

1. Releases the lock.
2. Puts thread to sleep.
3. Reacquires lock upon wake-up.

Questions:

Q1. The following is a description of a common synchronization problem: The Smoker's Problem. There are three smokers and a single agent (a "dealer"). The agent has a set of ingredients (tobacco, paper, and matches), and can place any two of these ingredients on the table. The smokers need all three ingredients to make a cigarette(tobacco, paper, and matches), but each smoker has a particular ingredient they need: One smoker has only tobacco, another has only paper, and the third has only matches. The agent will place two ingredients on the table (so the smokers can take what they need), and then one smoker can take the ingredients and make a cigarette. The problem is to ensure that only one smoker takes the ingredients at a time and that all smokers eventually get a chance to make their cigarettes. Given here is the code for the agent function. Explain and address the concurrency and efficiency issues in the code.

```
void* agent(void* arg) {
      while (1) {
      int ing1, ing2;
      pthread_mutex_lock(&mutex);

      printf("Agent waiting for smoker to make a cigarette\n");
      //Wait for condition 0 to be signalled
      pthread_cond_wait(&condition[0], &mutex);
      // Randomly choose two different ingredients
      ing1 = rand() % 3;
      do {ing2 = rand() % 3;} while (ing1 == ing2);
      // Update the buffer with the two ingredients
      buffer[0] = ing1; buffer[1] = ing2;
      printf("Agent placed ingredients %d and %d on the
      table.\n",ing1, ing2);

      // Signal the appropriate smoker to make the cigarette
      for (int i = 0; i < 3; i++) {
            if (i != ing1 && i != ing2) {
                  pthread_cond_wait(&condition[i]);}}
      pthread_mutex_unlock(&mutex);
      // Simulate time taken by agent to prepare next
ingredients
      usleep(1000);}
      return NULL;}
```

Ans. Issues with the given code:

1. **Deadlock:** The agent waits for a smoker (specifically smoker 0) to finish making the cigarette before placing new ingredients.
`pthread_cond_wait(&condition[0], &mutex);`
But the smoker cannot proceed because the agent is not placing ingredients. This creates a deadlock, where the agent and the smoker are waiting on each other, introducing a circular dependency.

2. **Efficiency:** Furthermore, the ingredients need not be selected inside the critical section as this is a non-critical task. Only the buffer values must be placed inside the critical section for maximal efficiency.

3. **Semantic Error:** `pthread_cond_signal(&condition[i])` must be used instead of waiting when signaling the appropriate smoker.

The code may be rewritten as follows:

```
void* agent(void* arg) {
      while (1) {

      //NO NEED TO PLACE INGREDIENT INITIALIZATION IN CRITICAL SECTION

            // Randomly choose two different ingredients
            int ing1 = rand() % 3;
            int ing2;
            do { ing2 = rand() % 3; } while (ing1 == ing2);

      //AGENT DOES NOT WAIT FOR SMOKER

      pthread_mutex_lock(&mutex);
      // Update the buffer with the two ingredients
      buffer[0] = ing1;
      buffer[1] = ing2;
      printf("Agent placed ingredients %d and %d on the table.\n",
ing1, ing2);

      // Signal the appropriate smoker to make the cigarette
      for (int i = 0; i < 3; i++) {
            if (i != ing1 && i != ing2) {
            // SIGNAL THE SMOKER, DO NOT WAIT
            pthread_cond_signal(&condition[i]);
            }
      }
      pthread_mutex_unlock(&mutex);
      usleep(1000); // Simulate time taken by the agent to prepare
the next ingredients
      }
return NULL;
}
```

Q2. 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 into 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 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.

Let’s assume:

Initial state:
top -> A   (A is the old top node)

Two threads:

- T1 pushing node n1
- T2 pushing node n2

Both execute:

```
n->next = top;
top = n;
```

Step-by-step Interleaving:

Step 1: Both threads read top
T1 reads top = A
T2 reads top = A

Step 2: Both set their next pointers
T1: n1->next = A
T2: n2->next = A

(Current structure: not yet reflected in shared variable top)
n1 -> A
n2 -> A

Step 3: T1 sets top = n1
top -> n1 -> A

Step 4: T2 sets top = n2
top -> n2 -> A

Final Result (Race Condition)
top -> n2 -> A
n1 -> A   (but n1 is now disconnected)

Illustration:

```
    (before)
top -> A
     (ideal result)
top -> n1 -> n2 -> A   or   top -> n2 -> n1 -> A
     (actual result due to race)
top -> n2 -> A
      n1 (lost, not in stack)
```

Therefore:

- *Both threads read the top before either updated it.*
- *Both linked their new nodes to the same old top (A).*
- *The last write to the top (from T2) overwrote the earlier one.*
- *Result: One node (n1) is lost, no longer reachable from the stack.*