ICS 2305: Systems Programming Make-Up Assignment: SCT211-0221/2018: Peter Kibuchi

◷ Created	@November 21, 2023 12:05 PM
◷ Last Updated	@December 8, 2023 12:00 PM
☰ Tags	3.1 Assignments

1. **What is atomic operation?**

   An atomic operation is a non-divisible, indivisible operation that is executed completely or not at all, without interruption, ensuring consistency in concurrent programming.

2. **Why will the following not work in parallel code:**

   ```
   //In the global section
   size_t a;
   //In pthread function
   for(int i = 0; i < 100000000; i++) a++;
   ```

   And this will?

   ```
   //In the global section
   atomic_size_t a;
   //In pthread function
   for(int i = 0; i < 100000000; i++) atomic_fetch_add(a, 1);
   ```

   The first code is not thread-safe because the increment operation `a++` is not atomic, and multiple threads can interfere, leading to a race condition. The second code is thread-safe because `atomic_fetch_add` ensures an atomic increment operation, preventing race conditions in parallel code.

3. **What are some downsides to atomic operations? What would be faster: keeping a local variable or many atomic operations?**

   Downsides to atomic operations:

   1. **Overhead:** Atomic operations often involve hardware-level locks or other synchronization mechanisms, introducing overhead.
   2. **Scalability:** Excessive use of atomic operations can lead to contention and reduced parallelism in multithreaded programs.
   3. **Limited Operations:** Atomic operations are typically limited to simple operations like increments and swaps, which may not cover all use cases.

For speed:

- **Keeping a local variable is generally faster than many atomic operations.** Atomic operations involve synchronization overhead, and frequent atomic operations can lead to performance bottlenecks. Use local variables when synchronization is not critical for correctness.

4. **What is the critical section?**

   A critical section is a part of a program where shared resources are accessed and must be protected to avoid data corruption in a concurrent or parallel computing environment. Only one thread or process at a time is allowed to execute the code within the critical section to ensure data consistency.

5. **Once you have identified a critical section, what is one way of assuring that only one thread will be in the section at a time?**

   One way to ensure that only one thread is in the critical section at a time is to use a mutex (mutual exclusion) lock.

6. I**dentify the critical section here:**

```
struct linked_list;
struct node;
void add_linked_list(linked_list *ll, void* elem){
    node* packaged = new_node(elem);
    if(ll->head){
         ll->head =
    }else{
         packaged->next = ll->head;
         ll->head = packaged;
         ll->size++;
    }

}

void* pop_elem(linked_list *ll, size_t index){
    if(index >= ll->size) return NULL;

    node *i, *prev;
    for(i = ll->head; i && index; i = i->next, index--){
        prev = i;
    }

    //i points to the element we need to pop, prev before
    if(prev->next) prev->next = prev->next->next;
    ll->size--;
    void* elem = i->elem;
    destroy_node(i);
    return elem;
}
```

The critical section is the block of code inside the `add_linked_list` function where the linked list is modified:

```
if (ll->head) {
    ll->head = ... // Critical section
} else {
    packaged->next = ll->head;
    ll->head = packaged;
    ll->size++;
}
```

In this block, the linked list's `head` is modified based on a condition, and the modification involves updating pointers and incrementing the size. Access to this block needs to be synchronized in a multi-threaded environment to avoid race conditions and ensure the integrity of the linked list.

How tight can you make the critical section?

The critical section should be as tight as necessary to complete the atomic operation, minimizing the time during which the shared resource is protected to reduce contention and improve parallelism.

7. **What is a producer consumer problem? How might the above be a producer consumer problem be used in the above section? How is a producer consumer problem related to a reader writer problem?**

   Producer-Consumer Problem:
   The producer-consumer problem is a classic synchronization problem in concurrent programming. It involves two types of processes: producers that generate data and place it into a shared buffer, and consumers that retrieve and consume the data from the buffer. The challenge is to ensure proper synchronization and data integrity between producers and consumers to avoid issues such as race conditions.

   Connection to the Provided Section:
   In the provided section, the `add_linked_list` function can be considered a producer as it adds elements to the linked list. Multiple producers (threads) may attempt to add elements concurrently, necessitating synchronization to ensure the linked list's consistency.

   Producer-Consumer vs. Reader-Writer:

   - **Producer-Consumer:** Involves communication and synchronization between processes where producers generate data and consumers consume it from a shared buffer.
   - **Reader-Writer:** Involves synchronization between processes where multiple readers can access a shared resource simultaneously, but exclusive access is required for writing to the resource. It deals with the challenge of balancing access between readers and writers to ensure data integrity.

   Relation:
   The producer-consumer problem and the reader-writer problem are both examples of synchronization challenges in concurrent programming. While producer-consumer focuses on data generation and consumption from a shared buffer, the reader-writer problem deals with reading and writing access to shared resources.

Both scenarios require careful synchronization mechanisms to avoid conflicts and maintain data consistency.

8. **What is a condition variable? Why is there an advantage to using one over a `while` loop?**

   A condition variable is a synchronization mechanism used in concurrent programming to enable threads to wait for a specific condition to become true before proceeding. It is typically associated with a mutex and provides a way for threads to efficiently wait for and signal changes in shared state.

   Advantage of using a condition variable over a while loop:

   - **Efficiency:** Condition variables allow a thread to release a lock and enter a waiting state efficiently, allowing other threads to acquire the lock and make progress. This is more efficient than a while loop, which would continuously check a condition in a tight loop, wasting CPU cycles and potentially causing high contention.

   In summary, condition variables provide a more efficient way for threads to wait for a specific condition to be met, reducing the need for busy-waiting (as in a while loop) and allowing better utilization of system resources.

9. **Why is this code dangerous?**

   ```
   if(not_ready){
        pthread_cond_wait(&cv, &mtx);
   }
   ```

   The code is dangerous because it lacks proper handling of spurious wake-ups. The `pthread_cond_wait` function should be used within a loop that checks the condition (`not_ready`) to prevent potential spurious wake-ups.

10. **What is a counting semaphore? Give me an analogy to a cookie jar/pizza box/limited food item.**

    A counting semaphore is a synchronization primitive that maintains an internal counter. It allows multiple threads to access a shared resource simultaneously up to a specified limit. The counter is decremented when a thread acquires the semaphore and incremented when it releases it.

    Analogy: Think of a counting semaphore as a pizza box with a limited number of slices. Each time a person takes a slice (acquires the semaphore), the slice count decreases. When the pizza box is empty (counter reaches zero), other people (threads) must wait until more slices are added (counter is incremented by releasing the semaphore). The semaphore enforces a limit on simultaneous access, analogous to controlling the number of available pizza slices.

11. **What is a thread barrier?**

A thread barrier is a synchronization mechanism that forces threads to wait until all participating threads have reached a certain point in the program before any of them can proceed further. Once all threads have arrived at the barrier, they are collectively released, allowing the program to continue execution.

12. **Use a counting semaphore to implement a barrier.**

    Here's a simple implementation of a barrier using a counting semaphore in pseudocode:

```
// Initialize semaphore with count equal to the number of threads
CountingSemaphore barrierSemaphore = new CountingSemaphore(0);

// Each thread performs its work independently

// At the barrier point
barrierSemaphore.release(); // Increment semaphore count

// Wait for all threads to reach the barrier
barrierSemaphore.acquire(); // Block if count is not equal to the number of threads

// All threads have reached the barrier, continue with the next phase
```

    In this example, each thread increments the semaphore count when it reaches the barrier point. The `acquire` operation is then used to block each thread until the semaphore count becomes equal to the total number of threads. Once all threads have reached the barrier, they are allowed to proceed to the next phase by releasing the semaphore.

    This implementation ensures that all threads synchronize at the barrier before moving forward, creating a simple barrier using a counting semaphore.

13. **Write up a Producer/Consumer queue, How about a producer consumer stack?**

    Producer/Consumer Queue:

```
#include <pthread.h>
#include <semaphore.h>

#define BUFFER_SIZE 10

int buffer[BUFFER_SIZE];
int in = 0, out = 0;

sem_t empty, full, mutex;

void init() {
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);
}

void* producer(void* arg) {
    while (1) {
        // Produce item
        int item = produce_item();
        // Wait for an empty slot in the buffer
        sem_wait(&empty);
        // Acquire the mutex
        sem_wait(&mutex);

        // Put item in buffer
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;

        // Release the mutex
        sem_post(&mutex);
        // Signal that the buffer is no longer empty
        sem_post(&full);
    }
}

void* consumer(void* arg) {
    while (1) {
        // Wait for a filled slot in the buffer
        sem_wait(&full);
        // Acquire the mutex
        sem_wait(&mutex);

        // Consume item
        int item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        // Release the mutex
        sem_post(&mutex);
        // Signal that the buffer is no longer full
        sem_post(&empty);

        // Consume the item
        consume_item(item);
    }
}
```

Producer/Consumer Stack:

```
#include <pthread.h>
#include <semaphore.h>

#define STACK_SIZE 10

int stack[STACK_SIZE];
int top = -1;

sem_t stackEmpty, stackFull, stackMutex;

void initStack() {
    sem_init(&stackEmpty, 0, STACK_SIZE);
    sem_init(&stackFull, 0, 0);
    sem_init(&stackMutex, 0, 1);
}

void push(int item) {
    sem_wait(&stackEmpty);
    sem_wait(&stackMutex);

    stack[++top] = item;

    sem_post(&stackMutex);
    sem_post(&stackFull);
}

int pop() {
    sem_wait(&stackFull);
    sem_wait(&stackMutex);

    int item = stack[top--];

    sem_post(&stackMutex);
    sem_post(&stackEmpty);

    return item;
}

void* producer(void* arg) {
    while (1) {
        int item = produce_item();
        push(item);
    }
}

void* consumer(void* arg) {
    while (1) {
        int item = pop();
        consume_item(item);
    }
}
```

These examples use semaphores for synchronization and mutual exclusion. Ensure proper error checking and handling for real-world scenarios.

14. **Give me an implementation of a reader-writer lock with condition variables, make a struct with whatever you need, it just needs to be able to support the following functions:**

```
void reader_lock(rw_lock_t* lck);
void writer_lock(rw_lock_t* lck);
void reader_unlock(rw_lock_t* lck);
void writer_unlock(rw_lock_t* lck);
```

The only specification is that in between `reader_lock` and `reader_unlock`, no writers can write. In between the writer locks, only one writer may be writing at a time.

```
#include <pthread.h>

typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t read_cond;
    pthread_cond_t write_cond;
    int readers;
    int writers;
    int active_writers;
} rw_lock_t;

void reader_lock(rw_lock_t* lck) {
    pthread_mutex_lock(&lck->mutex);
    while (lck->writers > 0 || lck->active_writers > 0) {
        pthread_cond_wait(&lck->read_cond, &lck->mutex);
    }
    lck->readers++;
    pthread_mutex_unlock(&lck->mutex);
}
```

```
void writer_lock(rw_lock_t* lck) {
    pthread_mutex_lock(&lck->mutex);
    while (lck->readers > 0 || lck->active_writers > 0) {
        pthread_cond_wait(&lck->write_cond, &lck->mutex);
    }
    lck->active_writers++;
    pthread_mutex_unlock(&lck->mutex);
}

void reader_unlock(rw_lock_t* lck) {
    pthread_mutex_lock(&lck->mutex);
    lck->readers--;
    if (lck->readers == 0) {
        pthread_cond_signal(&lck->write_cond);
    }
    pthread_mutex_unlock(&lck->mutex);
}

void writer_unlock(rw_lock_t* lck) {
    pthread_mutex_lock(&lck->mutex);
    lck->active_writers--;
    if (lck->active_writers == 0) {
        pthread_cond_broadcast(&lck->read_cond);
    } else {
        pthread_cond_signal(&lck->write_cond);
    }
    pthread_mutex_unlock(&lck->mutex);
}
```

This implementation of a reader-writer lock uses a mutex for protecting the shared state and two condition variables (`read_cond` and `write_cond`) to manage the synchronization between readers and writers. It ensures fairness by allowing waiting readers or writers to proceed in the order they arrived. The `active_writers` variable is used to track the number of active writers to prevent writers from being overtaken by new readers.

15. **Write code to implement a producer consumer using ONLY three counting semaphores. Assume there can be more than one thread calling enqueue and dequeue. Determine the initial value of each semaphore.**

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define BUFFER_SIZE 5

sem_t mutex, empty, full;
int buffer[BUFFER_SIZE];
int in = 0, out = 0;

void enqueue(int item) {
    sem_wait(&empty); // Wait if the buffer is full
    sem_wait(&mutex); // Enter critical section

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    sem_post(&mutex); // Exit critical section
    sem_post(&full);  // Signal that there is one more item in the buffer
}
```

```
int dequeue() {
    sem_wait(&full);  // Wait if the buffer is empty
    sem_wait(&mutex); // Enter critical section

    int item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    sem_post(&mutex); // Exit critical section
    sem_post(&empty); // Signal that there is one more empty slot in the buffer

    return item;
}

void* producer(void* arg) {
    int item = *((int*)arg);
    enqueue(item);
    printf("Produced: %d\\n", item);
    pthread_exit(NULL);
}

void* consumer(void* arg) {
    int item = dequeue();
    printf("Consumed: %d\\n", item);
    pthread_exit(NULL);
}

int main() {
    sem_init(&mutex, 0, 1);    // Initial value: 1
    sem_init(&empty, 0, BUFFER_SIZE); // Initial value: BUFFER_SIZE
    sem_init(&full, 0, 0);    // Initial value: 0

    pthread_t producerThread, consumerThread;

    int item = 42;

    pthread_create(&producerThread, NULL, producer, &item);
    pthread_create(&consumerThread, NULL, consumer, NULL);

    pthread_join(producerThread, NULL);
    pthread_join(consumerThread, NULL);

    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

In this implementation, the `mutex` semaphore provides mutual exclusion to the critical sections in `enqueue` and `dequeue`. The `empty` semaphore is initialized to `BUFFER_SIZE` because initially, all slots in the buffer are empty. The `full` semaphore is initialized to `0` because initially, there are no items in the buffer. The semaphores are used to synchronize access to the shared buffer, ensuring that producers and consumers don't interfere with each other.

16. **Write code to implement a producer consumer using condition variables and a mutex. Assume there can be more than one thread calling enqueue and dequeue.**

```c
#include <stdio.h>
#include <pthread.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int front = 0;
int rear = -1;
int itemCount = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;

void enqueue(int item) {
    pthread_mutex_lock(&mutex);

    while (itemCount == BUFFER_SIZE) {
        pthread_cond_wait(&not_full, &mutex);
    }

    rear = (rear + 1) % BUFFER_SIZE;
    buffer[rear] = item;
    itemCount++;

    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&mutex);
}

int dequeue() {
    int item;

    pthread_mutex_lock(&mutex);

    while (itemCount == 0) {
        pthread_cond_wait(&not_empty, &mutex);
    }

    item = buffer[front];
    front = (front + 1) % BUFFER_SIZE;
    itemCount--;

    pthread_cond_signal(&not_full);
    pthread_mutex_unlock(&mutex);

    return item;
}

void* producer(void* arg) {
    int item = *((int*)arg);
    enqueue(item);
    printf("Produced: %d\\n", item);
    pthread_exit(NULL);
}

void* consumer(void* arg) {
    int item = dequeue();
    printf("Consumed: %d\\n", item);
    pthread_exit(NULL);
}

int main() {
    pthread_t producerThreads[5], consumerThreads[5];
    int items[] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; ++i) {
        pthread_create(&producerThreads[i], NULL, producer, &items[i]);
        pthread_create(&consumerThreads[i], NULL, consumer, NULL);
    }

    for (int i = 0; i < 5; ++i) {
        pthread_join(producerThreads[i], NULL);
        pthread_join(consumerThreads[i], NULL);
    }

    return 0;
}
```

In this implementation, the `mutex` is used for mutual exclusion to protect critical sections during enqueue and dequeue operations. Two condition variables (`not_empty` and `not_full`) are used to signal when the buffer is not empty and not full, respectively. Producers wait on the `not_full` condition variable when the buffer is full, and consumers wait on the `not_empty` condition variable when the buffer is empty. The use of condition variables helps avoid busy-waiting and allows threads to efficiently wait for the desired conditions to be met.

17. **Use CVs to implement add(unsigned int) and subtract(unsigned int) blocking functions that never allow the global value to be greater than 100.**

```
#include <pthread.h>
#include <stdio.h>

#define MAX_VALUE 100

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
unsigned int global_value = 0;

void add(unsigned int value) {
    pthread_mutex_lock(&mutex);

    while (global_value + value > MAX_VALUE) {
        pthread_cond_wait(&cv, &mutex);
    }

    global_value += value;

    pthread_cond_broadcast(&cv);
    pthread_mutex_unlock(&mutex);
}

void subtract(unsigned int value) {
    pthread_mutex_lock(&mutex);

    while (global_value < value) {
        pthread_cond_wait(&cv, &mutex);
    }

    global_value -= value;

    pthread_cond_broadcast(&cv);
    pthread_mutex_unlock(&mutex);
}

int main() {
    // Example usage
    add(50); // Adds 50 to global_value (total: 50)
    subtract(30); // Subtracts 30 from global_value (total: 20)
    return 0;
}
```

In this example:

- The `add` function adds the specified value to `global_value` but waits if the addition would exceed the maximum value of 100. It uses a condition variable (`cv`) to block the thread until it's safe to perform the addition.
- The `subtract` function subtracts the specified value from `global_value` but waits if the subtraction would result in a negative value. It also uses the condition variable to block the thread until it's safe to perform the subtraction.
- The mutex ensures mutual exclusion during access to `global_value`.
- Condition variables are used to signal and wait for changes in the state of `global_value`. The `pthread_cond_broadcast` is used to wake up waiting threads when the state changes.

This ensures that the global value is never allowed to exceed 100, and threads are properly synchronized to avoid race conditions.

18. **Use CVs to implement a barrier for 15 threads.**

```
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 15

int count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t barrier_cv = PTHREAD_COND_INITIALIZER;

void barrier() {
    pthread_mutex_lock(&mutex);
    count++;

    if (count == NUM_THREADS) {
        // Last thread to arrive signals others to proceed
        count = 0;
        pthread_cond_broadcast(&barrier_cv);
    } else {
        // Wait for other threads to arrive
        pthread_cond_wait(&barrier_cv, &mutex);
    }

    pthread_mutex_unlock(&mutex);
}

void* threadFunction(void* arg) {
    // Thread performs work

    // Synchronize at the barrier
    barrier();

    // Continue with other work

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_create(&threads[i], NULL, threadFunction, NULL);
    }

    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

In this example, the `barrier` function is used to synchronize the threads. Each thread calls `barrier` before continuing with its work. The `count` variable is used to keep track of the number of threads that have arrived at the barrier. When the last thread arrives, it signals all other threads to proceed using `pthread_cond_broadcast`. Other threads wait for the signal using `pthread_cond_wait`. This creates a simple barrier synchronization point for 15 threads.

19. **How many of the following statements are true?**
    - **There can be multiple active readers**
    - **There can be multiple active writers**
    - **When there is an active writer the number of active readers must be zero**
    - **If there is an active reader the number of active writers must be zero**
    - **A writer must wait until the current active readers have finished**

    1. **There can be multiple active readers: True**
        - This is a characteristic of a reader-writer lock that allows multiple threads to read the shared resource simultaneously without mutual exclusion.
    2. **There can be multiple active writers: False**
        - In most standard implementations of a reader-writer lock, only one writer is allowed at a time to ensure exclusive access to the resource during writing.
    3. **When there is an active writer, the number of active readers must be zero: True**
        - This is typically true to ensure that no other threads are reading while a writer is modifying the shared resource. Writers usually require exclusive access.
    4. **If there is an active reader, the number of active writers must be zero: True**
        - This is also true to prevent a writer from starting to modify the shared resource while there are active readers. Reading can occur concurrently, but writing needs exclusive access.
    5. **A writer must wait until the current active readers have finished: True**

- To maintain consistency and avoid conflicts, a writer often needs to wait until all active readers have finished reading before it can acquire the lock and write to the resource.