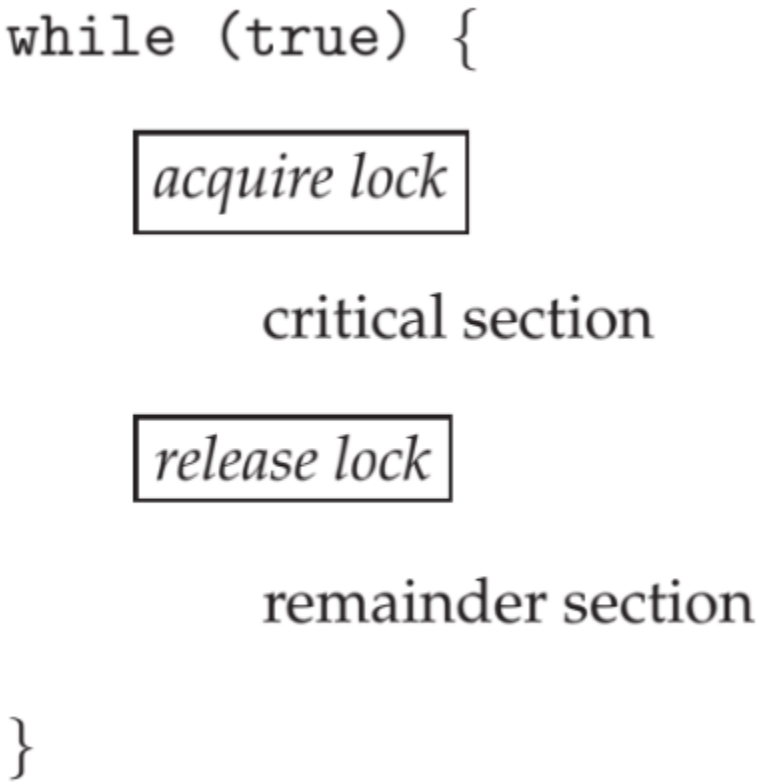
Content after Mid 2:

Hardware instructions are low-level commands that a computer's **CPU** (Central Processing Unit) can execute directly.

Atomic operations are operations that are **indivisible** — meaning they are executed **completely or not at all**, with **no chance of interruption** by other operations (especially from other threads or processors).

MUTEX LOCKS:

Simplest is mutex lock to solve Critical section problem.
Boolean variable indicating if lock is available or not.
Protect a critical section by:
First acquire() a lock
Then release() the lock
Calls to acquire() and release() must be atomic.
Usually implemented via hardware atomic instructions such as compare-and-swap.
But this solution requires busy waiting….This lock therefore called a spinlock.



Busy waiting: is a synchronization technique where a thread repeatedly checks(spins) for a condition to become true, consuming CPU cycles while waiting.

Why Busy waiting? Busy waiting has a few advantages despite its inefficiency in certain scenarios. Here are some situations where it proves beneficial:

1. **Simplicity**: Busy waiting is straightforward to implement and requires minimal overhead since no process state transitions (like switching to a wait or sleep state) are involved.
2. **Low Latency**: For scenarios where the wait time is expected to be very short, busy waiting can be faster than putting the process to sleep and then waking it up. Switching between states incurs some overhead, so spinning might be more efficient for quick waits.
3. **No Context Switching**: In busy waiting, the process doesn't relinquish the CPU, so there's no need for a costly context switch. This is particularly useful in real-time systems where the waiting process might need to act immediately after gaining the resource.

Technical Details:

1. CPU Usage: Consumes 100% CPU on the waiting core, wasting resources.
2. No OS Involvement: Does not involve the OS scheduler, reducing latency for very short waits.
3. Starvation Risk: If the condition is never met, the thread spins indefinitely.

Use Cases for Busy Waiting / Spinlocks

1. **Short Waits:**
   Ideal for very short critical sections or scenarios with low contention, where the overhead of blocking and waking up a thread outweighs the cost of busy waiting.
2. **Real-Time Systems:**
   Suitable for real-time applications where **predictable and minimal latency** is required, and waiting for a short duration is preferable over context switching.
3. **Spinlocks in Kernel-Level Programming:**
   Frequently used in **operating system kernels** where threads run on different CPUs and context switching is expensive. Spinlocks provide **lightweight synchronization** for quick operations.

Alternatives to Busy Waiting

1. **Blocking Waits:**
   Instead of consuming CPU cycles, threads can be suspended using OS-level blocking primitives like:
   - pthread_cond_wait()

- Semaphores (sem_wait(), etc.)

2. These mechanisms **free CPU resources** while waiting for a condition or resource.

3. **Mutexes with Condition Variables:**
   Use a combination of mutexes and condition variables to allow efficient and safe synchronization across threads.

SOLUTION TO THE CRITICAL SECTION PROBLEM USING MUTEX LOCKS:

Mutex with Dynamic Initialization:

```c
#define NUM_THREADS 5
#define INCREMENTS_PER_THREAD 10

pthread_mutex_t mutex;
int shared_data = 0;

void* thread_func(void* arg) {
    for (int i = 0; i < INCREMENTS_PER_THREAD; i++) {
        pthread_mutex_lock(&mutex);       // Lock
        shared_data++;                    // Critical section
        pthread_mutex_unlock(&mutex);     // Unlock
    }
    return NULL;
}
```

```c
int main() {
    pthread_t threads[NUM_THREADS];
    pthread_mutex_init(&mutex, NULL);

    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_create(&threads[i], NULL, thread_func, NULL) != 0) {
            perror("pthread_create");
            exit(1);
        }
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    // Destroy mutex
    pthread_mutex_destroy(&mutex);
    printf("Final value of shared_data: %d\n", shared_data);
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc synchro.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Final value of shared_data: 50
```

MUTEX WITH TRYLOCK:

```c
void* thread_func(void* arg) {
    int tid = *(int*)arg;  // Thread ID
    if (pthread_mutex_trylock(&mutex) == 0) {  // Try to lock
        // Critical section
        printf("Thread %d entered critical section.\n", tid);
        shared_data++;
        sleep(1);
        printf("Thread %d leaving critical section.\n", tid);
        pthread_mutex_unlock(&mutex);          // Unlock
    } else {
        // Couldn't acquire lock
        printf("Thread %d could not enter critical section (mutex busy).\n", tid);
    }
    return NULL;
}
```

MUTEX WITH TIMEOUT:

```c
//mutex with timeout
void* thread_func(void* arg) {
    struct timespec timeout;
    clock_gettime(CLOCK_REALTIME, &timeout);  // Get current time
    timeout.tv_sec += 2;                      // Wait for up to 2 seconds

    int tid = *((int*)arg);
    printf("Thread %d trying to lock mutex...\n", tid);

    int ret = pthread_mutex_timedlock(&mutex, &timeout);
    if (ret == 0) {
        printf("Thread %d acquired the lock.\n", tid);
        sleep(3);  // Simulate work in critical section
        pthread_mutex_unlock(&mutex);
        printf("Thread %d released the lock.\n", tid);
    } else {
        // Timeout occurred
        printf("Thread %d could not acquire the lock (timeout).\n", tid);
    }
    return NULL;
}
```

LOCK CONTENTION:

Contended vs. Uncontended Locks:

- A lock is **contended** if a thread is blocked while trying to acquire it.
- A lock is **uncontended** if it is available when a thread attempts to acquire it.

Contended Lock Categories:

- **Low contention**: Few threads compete for the lock.
- **High contention**: Many threads compete for the lock, leading to performance issues.

Impact on Performance:

- **Highly contended locks** tend to reduce the performance of concurrent applications.

Optimization Strategies:

- Minimize the size or duration of critical sections.
- Employ lock-free data structures or algorithms for scalability.
- Use adaptive locking mechanisms to adjust behavior based on contention levels.

SEMAPHORE: Semaphore S – integer variable
Can only be accessed via two indivisible (atomic) operations
wait() and signal()
Originally called P() and V()

```
wait(S) {
    while (S <= 0);  // busy wait
    S--;
}

signal(S) {
    S++;
}
```

1. Binary Semaphore : Also known as a **mutex-like** semaphore.
Definition:

- Can take only **two values**: 0 or 1.
- Used to **enforce mutual exclusion** (only one thread or process enters the critical section at a time).

Working:

- 0: resource is **locked** (occupied).
- 1: resource is **unlocked** (available).

Use Case:

- Accessing a shared file, memory, or variable where **only one thread should be active at a time**.

Example (in real life): Only one person can enter a bathroom. The "occupied" sign flips when someone enters (locked = 0) and flips back when they leave (unlocked = 1).

2. Counting Semaphore: Can take values from **0 to N**, where N is the number of available resources.

Definition:

- Tracks **multiple resources**.
- Allows **N threads/processes** to access the critical section **simultaneously**.

Working:

- wait() decrements the value → thread can proceed if the value > 0.
- signal() increments the value → releases a resource.

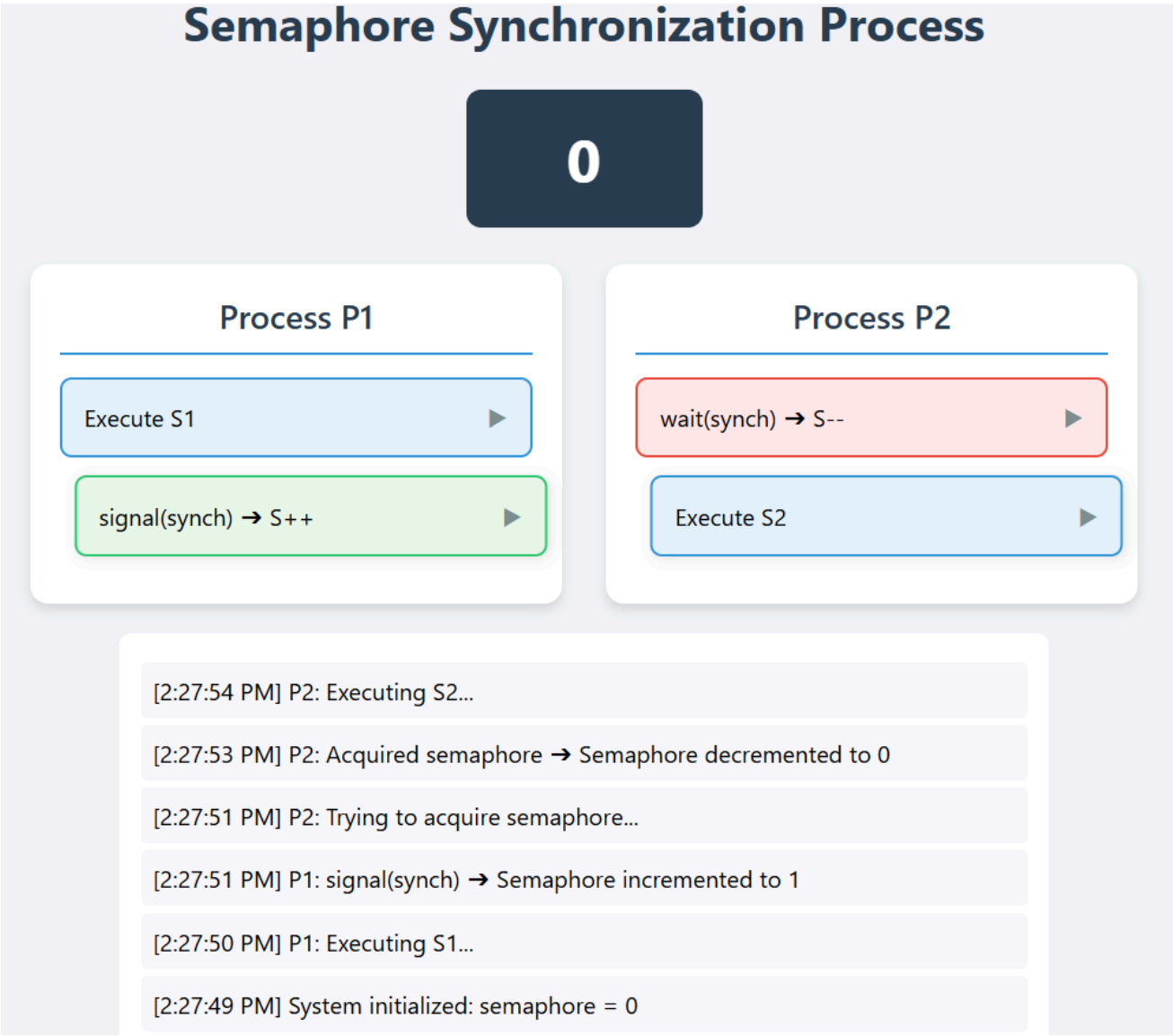
Use Case:

- Managing a connection pool (like a database), limited parking spots, or print queues.

Example (in real life): There are 3 chairs in a barber shop. If all are full (count = 0), the next customer waits. When one leaves, the count increases, and a new customer can enter.

1. **Synchronization Problem Example**:
    - Two processes: **P1 (with statement S1)** and **P2 (with statement S2)**.
    - Objective: Ensure **S2** executes only after **S1** has completed.
    - Solution:
        - Share a common semaphore synch initialized to 0.
        - In **P1**:
            - Execute S1.
            - Perform signal(synch).
        - In **P2**:
            - Perform wait(synch).
            - Execute S2.

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```



The sem_init function in C initializes a semaphore and its arguments are:

1. *sem_t sem*: A pointer to the semaphore variable that you want to initialize.
2. int pshared: Specifies if the semaphore is shared between processes or within threads of a single process.
    - If pshared is 0, the semaphore is shared only within threads of a single process.
    - If pshared is non-zero, the semaphore is shared between processes and typically resides in shared memory.
3. unsigned int value: The initial value of the semaphore. This determines how many threads (or processes) can access the shared resource concurrently.
    - For example, an initial value of 1 ensures mutual exclusion, while a higher value allows multiple threads to enter simultaneously.

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

```c
#include <pthread.h>
#include <semaphore.h>
#define NUM_THREADS 2
sem_t semaphore;
void* thread_function(void* arg) {
    int thread_id = *((int*)arg);
    // Wait on the semaphore
    sem_wait(&semaphore);
    // Critical section
    printf("Thread %d is in the critical section.\n", thread_id);
    // Signal the semaphore
    sem_post(&semaphore);
    return NULL;
}
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];
    // Initialize the semaphore with a value of 1
    sem_init(&semaphore, 0, 1);
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, thread_function, &thread_ids[i]);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    // Destroy the semaphore
    sem_destroy(&semaphore);
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ ./out
Thread 0 is in the critical section.
Thread 1 is in the critical section.
```

LIMITING CONCURRENT ACCESS:

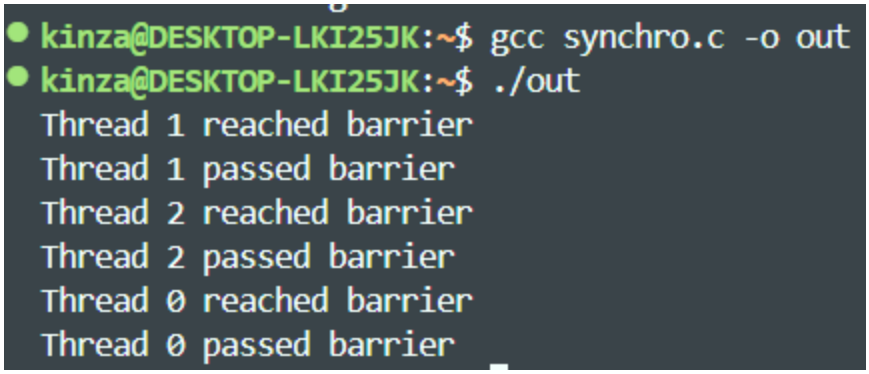
```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define MAX_THREADS 3 // Maximum threads allowed in the critical section
sem_t sem;
void* thread_func(void* arg) {
    long thread_id = (long)arg;
    // Wait to enter critical section
    sem_wait(&sem);
    // Critical section
    printf("Thread %ld: Entering critical section\n", thread_id);
    sleep(1); // Simulating work in the critical section
    printf("Thread %ld: Leaving critical section\n", thread_id);
    // Signal to leave critical section
    sem_post(&sem);
    return NULL;
}
```

```c
int main() {
    pthread_t threads[MAX_THREADS * 2];
    // Create more threads than allowed in critical section
    // Initialize the semaphore with MAX_THREADS
    sem_init(&sem, 0, MAX_THREADS);
    // Create threads
    for (long i = 0; i < MAX_THREADS * 2; i++) {
        pthread_create(&threads[i], NULL, thread_func, (void*)i);
    }
    // Wait for all threads to complete
    for (int i = 0; i < MAX_THREADS * 2; i++) {
        pthread_join(threads[i], NULL);
    }
    // Destroy the semaphore
    sem_destroy(&sem);
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc synchro.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Thread 0: Entering critical section
Thread 1: Entering critical section
Thread 2: Entering critical section
Thread 0: Leaving critical section
Thread 3: Entering critical section
Thread 1: Leaving critical section
Thread 5: Entering critical section
Thread 2: Leaving critical section
Thread 4: Entering critical section
Thread 3: Leaving critical section
Thread 4: Leaving critical section
Thread 5: Leaving critical section
```

BARRIER SYNCHRONIZATION:

```c
#define NUM_THREADS 3 // Number of threads
sem_t barrier; // Semaphore for synchronization
void* thread_func(void* arg) {
    long thread_id = (long)arg;
    // Signal arrival at the barrier
    printf("Thread %ld reached barrier\n", thread_id);
    sem_post(&barrier); // Increment semaphore count
    // Wait for all threads to arrive
    sem_wait(&barrier); // Block until other threads signal
    printf("Thread %ld passed barrier\n", thread_id);
    return NULL;
}
int main() {
    pthread_t threads[NUM_THREADS];
    // Initialize semaphore with a value of 0
    sem_init(&barrier, 0, 0);
    for (long i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, thread_func, (void*)i);
    }
    // Wait for all threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    // Destroy semaphore
    sem_destroy(&barrier);
    return 0;
}
```



Semaphore Initialization:

- sem_init(&barrier, 0, 0) initializes the semaphore with a value of 0. This ensures that threads cannot pass the barrier initially until all have arrived.

Thread Function:

- Each thread signals its arrival by incrementing the semaphore (sem_post(&barrier)).
- The thread then waits for other threads to reach the barrier using sem_wait(&barrier).

Barrier Synchronization:

- Each thread blocks after reaching the barrier until all threads arrive and signal the semaphore.
- The threads then pass the barrier collectively, ensuring synchronized execution.

Thread Creation:

- pthread_create starts multiple threads, each running the thread_func code.
- pthread_join ensures the main program waits for all threads to complete.

Cleanup:

- The semaphore is destroyed using sem_destroy(&barrier) to release system resources.

SEMAPHORE IMPLEMENTATION WITHOUT BUSY WAITING:

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  Value (of type integer)
  Pointer to next record in the list
- Two operations:

block – place the process invoking the operation on the appropriate waiting queue
wakeup – remove one of processes in the waiting queue and place it in the ready queue

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

```
typedef struct {
  int value;
  struct process *list;
  } semaphore;
```

🔄 LIVENESS IN CONCURRENT SYSTEMS

Definition: Liveness ensures that **the system continues to make progress**, i.e., something good eventually happens. It means that every process will **eventually get its chance** to execute, access resources, or reach completion.

Common Liveness Problems

1. Deadlock

- **What happens**: A set of threads are each waiting for a resource held by another, forming a cycle of dependencies.
- **Effect**: No thread can proceed; the system halts.

Example:
Thread A holds Resource 1, needs Resource 2
Thread B holds Resource 2, needs Resource 1
=> Circular wait ⇒ Deadlock

2. Starvation

- **What happens**: A thread **waits indefinitely** to acquire a resource because other threads are continually favored.
- **Effect**: The thread never makes progress, even though resources are available.

- **Example**:
  A low-priority thread keeps getting preempted by high-priority threads, so it never gets CPU time.

3. Livelock

- **What happens**:Threads keep retrying an operation but make no progress due to repeated interference.
- **Effect**: Infinite loop of "avoiding" but never succeeding.
- **Example**: Two threads repeatedly back off and retry an operation to avoid conflict, but they keep interfering with each other.

Summary:

Issue	Definition	System Behavior	Is CPU active?
Deadlock	Circular waiting, no thread can proceed	All threads stuck	❌ Inactive
Starvation	Thread never gets resource	Some threads progress, one waits	✅ Partially
Livelock	Threads retry endlessly with no success	All threads run, no progress	✅ Fully

DEADLOCK:

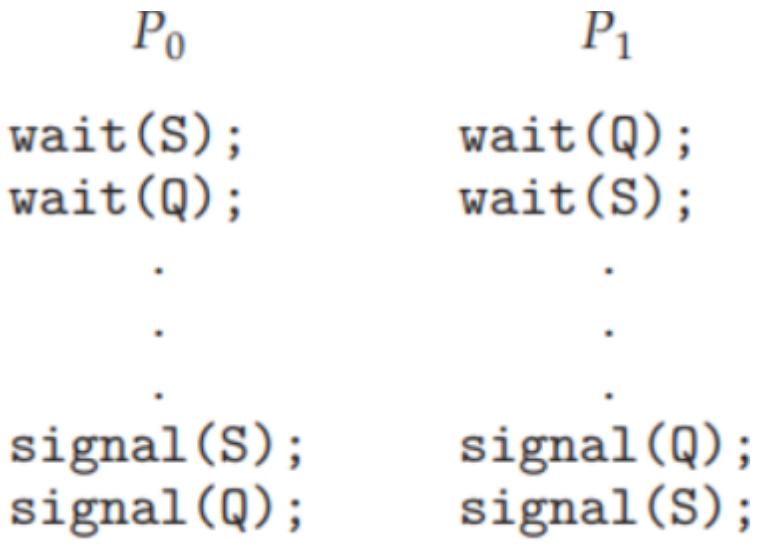
Two Processes: P0 and P1

- Both want to access **two shared resources** protected by semaphores:
  - Semaphore S
  - Semaphore Q
- Initially, both semaphores have value 1 (meaning available).

Step	P0	P1	Semaphore State	Result
1	wait(S) → acquires S		S=0, Q=1	P0 holds S
2		wait(Q) → acquires Q	S=0, Q=0	P1 holds Q
3	wait(Q)		Q=0 → already held by P1	P0 **blocks**
4		wait(S)	S=0 → already held by P0	P1 **blocks**

Result: Deadlock

- P0 is waiting for Q to be released → held by P1
- P1 is waiting for S to be released → held by P0
- **Neither process can proceed to signal()** (release) since both are blocked
- **Deadlock occurs**: Mutual waiting with no way forward.



Pthread C code for deadlock:

Thread 1 locks sem 1 and waits for sem 2.
Thread 2 locks sem 2 and waits for sem 1.
Both threads wait indefinitely, causing a deadlock.

```c
void* thread1(void* arg) {
    sem_wait(&sem1); // Lock sem1
    sem_wait(&sem2); // Wait for sem2 (deadlock)
    printf("Thread 1\n");
    sem_post(&sem2);
    sem_post(&sem1);
    return NULL;
}
```

```c
void* thread2(void* arg) {
    sem_wait(&sem2); // Lock sem2
    sem_wait(&sem1); // Wait for sem1 (deadlock)
    printf("Thread 2\n");
    sem_post(&sem1);
    sem_post(&sem2);
    return NULL;
}
```

How to Prevent It?

- **Resource Ordering**: Always acquire resources in a fixed global order (e.g., always S then Q).
- **Deadlock Detection & Recovery**: System checks for cycles in wait-for graphs.
- **Timeouts / Retry with backoff**: Avoid indefinite waits.

CHAPTER 7

7.1 CLASSICAL PROBLEMS OF SYNCHRONIZATION:

PRODUCER CONSUMER PROBLEM:

Semaphore Purpose:

- mutex – ensures only one process (producer/consumer) accesses the buffer at a time.
- empty – ensures producer waits if no empty slots.
- full – ensures consumer waits if no full slots.

```c
int n;                    // size of buffer
semaphore mutex = 1;      // binary semaphore for mutual exclusion
semaphore empty = n;      // counts empty slots in buffer
semaphore full = 0;       // counts filled slots in buffer
```

```
Producer Logic
while (true) {
    /* produce an item in next_produced */
    wait(empty);     // wait until there's space in the buffer
    wait(mutex);     // enter critical section
    /* add next_produced to buffer */
    signal(mutex);   // exit critical section
    signal(full);    // notify consumer a new item is available
}
Consumer Logic
while (true) {
    wait(full);
    wait(mutex);     // enter critical section
    /* remove an item from buffer to next_consumed */
    signal(mutex);   // exit critical section
    signal(empty);   // notify producer there's space in the buffer
    /* consume the item in next_consumed */
}
```

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

#define BUFFER_SIZE 10

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

int main() {
    pthread_t prod, cons;
    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);
    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);
    return 0;
}
```

```c
void* producer(void* arg) {
    int item = 0;
    while (1) {
        sem_wait(&empty);
        sem_wait(&mutex);
        buffer[in] = item++;
        in = (in + 1) % BUFFER_SIZE;
        sem_post(&mutex);
        sem_post(&full);
        printf("Produced: %d\n", item - 1);
    }
    return NULL;
}
void* consumer(void* arg) {
    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);
        int item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        sem_post(&mutex);
        sem_post(&empty);
        printf("Consumed: %d\n", item);
    }
    return NULL;
}
```

READERS WRITERS PROBLEM:

Two Variants of the Readers-Writers Problem

1️⃣ First Readers-Writers Problem (Readers Preference)

- **Readers have priority**.
- No reader should wait unless a writer has already obtained access.
- Writers may **starve** if readers keep arriving continuously.

> 💬 "No reader should wait for other readers to finish simply because a writer is waiting."

2️⃣ Second Readers-Writers Problem (Writers Preference)

- **Writers have priority**.
- Once a writer is ready, it should write **as soon as possible**.
- Readers may **starve** if writers keep coming.

> 💬 "If a writer is waiting to access the object, no new readers may start reading."

Starvation Issue "A solution to either problem may result in starvation."

- In the **first case**, **writers may starve**.
- In the **second case**, **readers may starve**.

That's why **other variants** (like fair solutions using reader-writer queues) have been proposed to prevent starvation altogether.

Variant	Priority Given To	Starvation Risk
First Readers-Writers	Readers	Writers may starve
Second Readers-Writers	Writers	Readers may starve

WRITER:

```
do {
     wait(wrt);
     // writing is performed
     signal(wrt);
} while (TRUE);
```

READER:

```
do {
    wait(mutex);
    readcount++;
    if (readcount == 1) {  // first reader locks the resource
        wait(wrt);
    }
    signal(mutex);

    // reading is performed

    wait(mutex);
    readcount--;
    if (readcount == 0) {  // last reader unlocks the resource
        signal(wrt);
    }
    signal(mutex);
} while (TRUE);
```

Q) Does the given code fragment ensures mutual exclusion. If no, then identify the error in the given code fragment.

Semaphore mutex initialized to 1
Semaphore wrt initialized to 1
Integer readcount initialized to 0

WRITER

```
do {
 wait (wrt) ;
   //    writing is performed
 signal (wrt) ;
     } while (TRUE);
```

READER

```
    do {
                          wait (mutex) ;
                          readcount ++ ;
                            signal(wrt) ;
                       signal (mutex)
             // reading is performed
                       wait (mutex) ;
                            readcount  -- ;
                            wait(wrt) ;
                           signal (mutex) ;
       } while (TRUE);
```

ANSWER: GIVEN CODE FRAGMENT DOESN'T GUARANTEE MUTUAL EXCLUSION. IF ONE READER ENTERS IN THE CRITICAL REGION IT ALLOWS SEVERAL WRITERS TO COME INSIDE THE CRITICAL REGION. ORDER OF SIGNAL (WRT) AND WAIT(WRT) SHOULD BE SWAPPED. AND ONLY FIRST AND LAST READER SHOULD BE ALLOWED TO DECREASE AND INCREASE THE VALUE OF WRT SEMAPHORE.

Corrected code fragment:

WRITER:

```
do {
    wait(wrt);
    // writing is performed
    signal(wrt);
} while (TRUE);
```

READER:

```
do {
    wait(mutex);
    readcount++;
    if (readcount == 1) {  // first reader locks the resource
        wait(wrt);
    }
    signal(mutex);

    // reading is performed

    wait(mutex);
    readcount--;
    if (readcount == 0) {  // last reader unlocks the resource
        signal(wrt);
    }
    signal(mutex);
} while (TRUE);
```

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

sem_t rw_mutex, mutex;
int read_count = 0;

void* writer(void* arg) {
    while (1) {
        sem_wait(&rw_mutex);
        printf("Writer is writing\n");
        sem_post(&rw_mutex);
    }
    return NULL;
}
```

C program for the first readers-writers problem

```
void* reader(void* arg) {
    while (1) {
        sem_wait(&mutex);
        read_count++;
        if (read_count == 1)
            sem_wait(&rw_mutex);
        sem_post(&mutex);
        printf("Reader %ld is rea
        sem_wait(&mutex);
        read_count--;
        if (read_count == 0)
            sem_post(&rw_mutex);
        sem_post(&mutex);
    }
    return NULL;
}
```

C program for the first readers-writers problem

```c
int main() {
    pthread_t w, r1, r2;
    sem_init(&rw_mutex, 0, 1);
    sem_init(&mutex, 0, 1);
    pthread_create(&w, NULL, writer, NULL);
    pthread_create(&r1, NULL, reader, (void*)1);
    pthread_create(&r2, NULL, reader, (void*)2);
    pthread_join(w, NULL);
    pthread_join(r1, NULL);
    pthread_join(r2, NULL);
    sem_destroy(&rw_mutex);
    sem_destroy(&mutex);
    return 0;
}
```

DINING PHILOSOPHERS PROBLEM:

The Dining Philosophers Problem is a classic synchronization problem that illustrates the challenges of concurrency control. In this scenario, five philosophers sit around a circular table with a bowl of rice in the center and a single chopstick placed between each pair of neighboring philosophers. Each philosopher alternates between thinking and eating.

- Thinking: When a philosopher is thinking, they do not interact with others.
- Eating: When a philosopher becomes hungry, they try to pick up the two chopsticks closest to them—one on the left and one on the right. A philosopher can only pick up one chopstick at a time and cannot take a chopstick already held by a neighbor.
- Eating condition: Once a philosopher has both chopsticks, they eat without releasing them.
- Returning chopsticks: After eating, the philosopher puts down both chopsticks and resumes thinking.

The problem highlights synchronization issues in concurrent systems, especially regarding resource sharing and deadlock avoidance. It's a metaphor for problems that arise when multiple processes need exclusive access to shared resources, and the solution requires careful management to prevent conflicts and ensure progress.

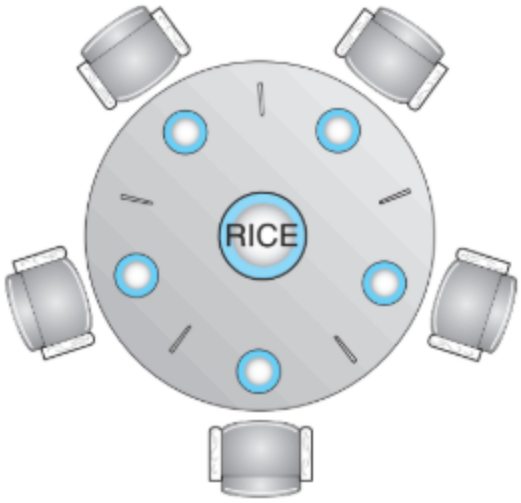


Figure 7.5 The situation of the dining philosophers.

```
while (true) {
wait(chopstick[i]);
wait(chopstick[(i+1) % 5]);
...
/* eat for a while */
...
signal(chopstick[i]);
signal(chopstick[(i+1) % 5]);
...
/* think for awhile */
...
}
```

The provided code snippet is part of a solution to the Dining Philosophers Problem, where each philosopher picks up two chopsticks (one on the left and one on the right) to eat. The philosopher then releases the chopsticks after eating and resumes thinking. However, this approach can lead to **deadlock** (if all philosophers pick up their left chopstick at the same time and wait for the right one) and **starvation** (where a philosopher may never get to eat).

To avoid these issues, solutions like:

1. **Resource allocation ordering** (ensuring chopsticks are picked up in a consistent order).
2. **Using a mutex** for the whole table to control access to the chopsticks.
3. **Introducing a priority or waiting queue** for fairness.
4. **Odd-even picking strategy** (where philosophers alternate the order in which they pick up chopsticks).

These methods help ensure **deadlock-free** and **starvation-free** resource allocation.

How to solve deadlocks in Dining philosophers' problem?

Several possible remedies to the deadlock problem are the following:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

CHAPTER 8: DEADLOCK

Deadlock in Multithreaded Applications

In multithreaded programs using **POSIX mutex locks**, **deadlocks** can occur when multiple threads try to acquire the same set of locks in different orders.

Example Scenario:

- Two mutex locks: first_mutex and second_mutex
- Two threads:
    - **Thread One** locks first_mutex then second_mutex
    - **Thread Two** locks second_mutex then first_mutex

If **Thread One** locks first_mutex and **Thread Two** locks second_mutex at the same time, both threads will be waiting for the other to release the second lock—causing a **deadlock**.

Key Points:

- **Deadlock is possible but not guaranteed**—it depends on the thread scheduling order.
- **Hard to detect and test** since it only occurs under specific timing conditions.
- Highlights the difficulty of managing deadlocks in concurrent programs.

Main Cause:

- **Inconsistent lock acquisition order** between threads.

Solution Tip:

- Avoid deadlocks by ensuring **all threads acquire locks in the same global order**.

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}


/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

NECESSARY CONDITIONS:

A **deadlock** can only occur if **all four of the following conditions** hold **simultaneously** in a system:

1. **Mutual Exclusion**:
   At least one resource is non-sharable—only one thread can use it at a time.
2. **Hold and Wait**:
   A thread is holding at least one resource and waiting to acquire additional resources held by others.
3. **No Preemption**:
   Resources cannot be forcibly taken from a thread; they must be released voluntarily.

4. **Circular Wait**:
   There exists a cycle of threads where each thread is waiting for a resource held by the next thread in the cycle.

All four conditions must be present for a deadlock to happen. If even one is prevented, deadlock can be avoided.

Deadlock Prevention vs. Deadlock Avoidance

To ensure **deadlocks never occur**, systems can use:

1. Deadlock Prevention

- **Goal**: Ensure **at least one** of the four necessary deadlock conditions **never holds**.
- **How**: Apply rules or restrictions on how resources are requested.
- **Example**:
  - Force threads to request all required resources at once (prevents *hold and wait*).
  - Impose a strict order in which resources must be requested (prevents *circular wait*).

2. Deadlock Avoidance

- **Goal**: Allow resource requests only if they **won't lead to a deadlock**.
- **How**: The system must have **advance knowledge** of each thread's maximum resource needs.
- **What it checks**:
  - Resources currently available
  - Resources currently allocated
  - Future resource requests
- **Example**: **Banker's Algorithm** checks if granting a resource keeps the system in a safe state.

✅ **Prevention** blocks deadlock from happening by limiting behavior.
🧠 **Avoidance** makes smart decisions using extra information to stay safe.

6.5 Illustrate how a binary semaphore can be used to implement mutual exclusion among n processes.

```c
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define N 5
sem_t mutex;
void* process(void* arg) {
    int id = *((int*)arg);
    sem_wait(&mutex); // Request access to critical section
    // Critical section
    printf("Process %d ENTERED critical section.\n", id);
    sleep(2); // Simulate time spent in critical section
    printf("Process %d EXITING critical section.\n", id);
    sem_post(&mutex); // Release access to critical section
    return NULL;
}
```

```c
int main() {
    pthread_t threads[N];
    int process_ids[N];
    sem_init(&mutex, 0, 1);
    for (int i = 0; i < N; i++) {
        process_ids[i] = i + 1;
        pthread_create(&threads[i], NULL, process, &process_ids[i]);
    }
    for (int i = 0; i < N; i++) {
        pthread_join(threads[i], NULL);
    }
    sem_destroy(&mutex);
    return 0;
}
```

```
kinza@DESKTOP-LKI25JK:~$ gcc synchro.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Process 1 ENTERED critical section.
Process 1 EXITING critical section.
Process 2 ENTERED critical section.
Process 2 EXITING critical section.
Process 3 ENTERED critical section.
Process 3 EXITING critical section.
Process 4 ENTERED critical section.
Process 4 EXITING critical section.
Process 5 ENTERED critical section.
Process 5 EXITING critical section.
```

6.3 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

In a single-CPU system:

1. Only one process/thread can run at a time.
2. If Thread A holds the lock and Thread B is spinning (waiting for the lock),
3. Thread B will consume CPU time by spinning, preventing Thread A from running and releasing the lock.
4. As a result, the system deadlocks or wastes CPU cycles. Spinlocks waste CPU time on single-core systems because the spinning thread blocks the one that could release the lock.

In multiprocessor/multi-core systems:

1. Multiple threads can run simultaneously on different CPUs.
2. If Thread A is holding the lock on CPU 1, and Thread B is spinning on CPU 2:
    - Thread A continues executing and will eventually release the lock.
    - Thread B will detect the change and acquire the lock — no CPU time wasted.
3. Context switches (sleeping and waking threads) are expensive compared to just spinning for a short time — so spinlocks are faster for very short waits.

Spinlocks are efficient when the lock is held for a short duration and there are multiple processors to do the spinning and actual work simultaneously.

6.6 Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: deposit(amount) and withdraw(amount). These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the withdraw() function, and the wife calls deposit(). Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

Two people (e.g., husband and wife) are using the same bank account at the same time. One tries to **withdraw** money while the other tries to **deposit** money. Both actions involve updating a shared balance.

Race Condition: What Could Go Wrong? If both functions access the balance **without coordination**, they might read the same initial balance and update it independently. This can lead to **incorrect final results** (e.g., showing more or less money than there actually is).

Step	Thread	Operation	Value of balance
1	Husband	Reads balance (`1000`)	1000
2	Wife	Reads balance (`1000`)	1000
3	Husband	`balance = 1000 - 500 = 500`	500
4	Wife	`balance = 1000 + 200 = 1200`	1200 ❌ WRONG

> 💥 **Error:** Final balance should be `700` (`1000 - 500 + 200`), but we got `1200`!

Solution: Prevent Race Conditions using Mutual Exclusion

To avoid this issue, we must ensure that **only one operation can update the balance at a time**. This is known as **mutual exclusion**. It prevents race conditions by allowing only one thread into the critical section (code that modifies shared data).

```c
pthread_mutex_t balance_lock = PTHREAD_MUTEX_INITIALIZER;  // Mutex for synchronization
// Deposit function
void deposit(int amount) {
    pthread_mutex_lock(&balance_lock);  // Lock to ensure mutual exclusion
    account_balance += amount;  // Update balance
    pthread_mutex_unlock(&balance_lock);  // Unlock after the operation is done
}
// Withdraw function
void withdraw(int amount) {
    pthread_mutex_lock(&balance_lock);  // Lock to ensure mutual exclusion
    if (account_balance >= amount) {
        account_balance -= amount;  // Update balance after withdrawal
    } else {
        printf("Insufficient funds\n");
    }
    pthread_mutex_unlock(&balance_lock);  // Unlock after the operation is done
}
```

6.7 The pseudocode of Figure 6.15 illustrates the basic push() and pop() operations of an array-based stack. Assuming that this algorithm could be used in a concurrent environment, answer the following questions:
a. What data have a race condition?
b. How could the race condition be fixed?

```
push(item) {
   if (top < SIZE) {
      stack[top] = item;
      top++;
   }
   else
      ERROR
}

pop() {
   if (!is_empty()) {
      top--;
      return stack[top];
   }
   else
      ERROR
}

is_empty() {
   if (top == 0)
      return true;
   else
      return false;
}
```

a. **Data with Race Conditions**:
The variables top and stack[] are subject to race conditions. Specifically:

- **top**: Concurrent push() or pop() operations may interleave non-atomically. For example:
    - Two threads executing push() could both read the same top value, write to stack[top], and increment top incorrectly, causing overwrites or exceeding SIZE.
    - Two threads executing pop() could both decrement top after checking is_empty(), leading to invalid indices (e.g., top becoming negative) or returning the same element twice.
- **stack[]**: Concurrent writes to stack[top] (during push()) or reads during overlapping pop() operations may result in corrupted or inconsistent data.

```c
int stack[SIZE];
int top = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
// Push function
void push(int element) {
    pthread_mutex_lock(&m);
    if (top == SIZE) {
        printf("Stack Overflow\n");
    } else {
        stack[top] = element;
        top++;
    }
    pthread_mutex_unlock(&m);
}
// Pop function
int pop() {
    pthread_mutex_lock(&m);
    int element;
    if (top == 0) {
        printf("Stack Underflow\n");
        element = -1; // or handle as needed
    } else {
        top--;
        element = stack[top];
    }
    pthread_mutex_unlock(&m);
    return element;
}
```

6.8 Race conditions are possible in many computer systems. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the bid(amount) function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
void bid(double amount) {
   if (amount > highestBid)
      highestBid = amount;
}
```

Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

Race Condition Description:

A race condition occurs when two or more threads concurrently execute the bid() function. Here's how it unfolds:

1. **Initial State**: Suppose the highestBid is $100.
2. **Thread A** (bidding $150) reads highestBid (still $100), evaluates 150 > 100 as true, and proceeds to update highestBid.
3. **Thread B** (bidding $200) *simultaneously* reads highestBid (also $100), evaluates 200 > 100 as true, and proceeds to update highestBid.
4. **Result**: Depending on the order of execution:
    - If Thread B updates highestBid to $200 first, Thread A will overwrite it with $150, **incorrectly setting the final highestBid to $150**.
    - This happens because the **check (if (amount > highestBid) and update (highestBid = amount)** are not atomic.

1. Mutual Exclusion (Locking)

```c
pthread_mutex_t bid_lock = PTHREAD_MUTEX_INITIALIZER;  // Mutex for synchronization
void bid(int amount) {
    pthread_mutex_lock(&bid_lock);  // Lock the critical section
    if (amount > current_highest_bid) {
        current_highest_bid = amount;  // Update highest bid
    }
    pthread_mutex_unlock(&bid_lock);  // Unlock the critical section
}
```

6.9 The following program example can be used to sum the array values of size N elements in parallel on a system containing N computing cores (there is a separate processor for each array element): This has the effect of summing the elements in the array as a series of partial sums, as shown in Figure 6.16. After the code has executed, the sum of all elements in the array is stored in the last array location. Are there any race conditions in the above code example? If so, identify where they occur and illustrate with an example. If not, demonstrate why this algorithm is free from race conditions.

```
for j = 1 to log_2(N) {
   for k = 1 to N {
      if ((k + 1) % pow(2,j) == 0) {
         values[k] += values[k - pow(2,(j-1))]
      }
   }
}
```

The code uses a parallel prefix sum approach, where each iteration of the outer loop j combines partial sums in a way that avoids overlapping writes or reads within the same iteration. Specifically:

1. **Outer Loop Sequential, Inner Loop Parallel**:
    - The outer loop runs sequentially from j = 1 to $\log_2(N)$. Each iteration of j must complete before the next begins.
    - The inner loop processes all k values in parallel (one per core).
2. **No Overlapping Writes**:
    - For a fixed j, the condition (k + 1) % 2^j == 0 ensures that each k writes to a **unique** array index values[k].
    - Example: For j=1 and N=4, valid k values are 1 and 3. Threads write to values[1] and values[3], which are distinct.
3. **Reads Do Not Overlap with Writes**:
    - For a fixed j, the read operation values[k - 2^(j-1)] accesses indices from previous iterations.
    - Example: For j=2 and k=3, the read from values[1] (updated in j=1) is safe because j=1 has already completed.
4. **Dependencies Are Resolved Across Iterations**:
    - Reads in iteration j depend on writes from iteration j−1, ensuring no intra-iteration conflicts.
    - Example: In j=3, values[7] reads values[3] (updated in j=2), which is safe because j=2 is fully completed.

Why No Race Conditions Exist

- **No Concurrent Writes**: Each k in a given j iteration writes to a unique array location.
- **Reads Are from Prior Iterations**: All read operations in iteration j access data finalized in earlier iterations (j' < j).
- **Sequential Outer Loop**: The outer loop ensures dependencies between iterations are respected.

This algorithm is **race-condition-free** because it enforces:

- Non-overlapping writes within the same iteration.
- Sequential progression of partial sums across iterations.
- Safe read-after-write dependencies between iterations.

6.1 In Section 6.4, we mentioned that disabling interrupts frequently can affect the system's clock. Explain why this can occur and how such effects can be minimized.

Why Disabling Interrupts Affects the System Clock:

The system clock relies on *timer interrupts* to maintain accurate timekeeping. Each timer interrupt triggers the OS to increment its internal clock counter. When interrupts are disabled:

- Missed Timer Interrupts: The CPU cannot service timer interrupts, causing the OS to "lose track" of time.
- Clock Drift: Prolonged interrupt disabling delays clock updates, leading to discrepancies between system time and real time.

Minimizing the Effects:

1. Short Critical Sections: Ensure interrupt-disabling periods are as brief as possible (e.g., only during atomic operations).
2. Local Interrupt Control: On multi-core systems, disable interrupts only on the current CPU core instead of globally.
3. Alternative Synchronization: Use lock-free algorithms or atomic instructions (e.g., CAS) to avoid disabling interrupts entirely.
4. Compensation Mechanisms: Track time spent with interrupts disabled and adjust the system clock retroactively (complex but feasible in some OS designs).

6.2 What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

Busy Waiting:

- Definition: A thread repeatedly checks a condition (e.g., a lock) in a loop without relinquishing the CPU.
- Example: Spinlocks (while (lock != 0);).
- Drawback: Wastes CPU cycles, increasing power consumption and reducing system efficiency.

Other Types of Waiting:

1. Blocking (Sleep-Waiting):
    - The thread is moved to a waiting queue and yields the CPU until the resource is available (e.g., using semaphores or condition variables).
    - Efficient for long waits but incurs context-switch overhead.
2. Hybrid Approaches:
    - Spin briefly before blocking (adaptive mutexes in Linux/Windows).

Can Busy Waiting Be Avoided Altogether?

- No, but it can be minimized:
    1. User-Level Code: Prefer blocking mechanisms (e.g., pthread_cond_wait()).
    2. Kernel/Real-Time Systems: Busy waiting (spinlocks) is sometimes necessary for short critical sections to avoid scheduler overhead.
    3. Hardware Support: Use atomic operations or hardware-assisted locks (e.g., TSX) to reduce busy-wait duration.

6.24 In Section 6.7, we use the following illustration as an incorrect use of semaphores to solve the critical-section problem:

wait(mutex);

...

critical section

...

wait(mutex);

Explain why this is an example of a liveness failure.

The example provided is a liveness failure due to the incorrect use of semaphores. The problem occurs when a thread tries to acquire a mutex twice without releasing it in between. After the first wait(mutex) call, the thread enters the critical section but then tries to acquire the mutex again with a second wait(mutex). Since the thread already holds the mutex, it becomes stuck, waiting for itself to release the mutex, resulting in deadlock or indefinite blocking.

In correct usage, the thread should release the mutex after the critical section using signal(mutex) before attempting to acquire it again. This ensures proper synchronization and prevents the thread from getting stuck.

Thus, the failure happens because the thread cannot make progress, which is a form of liveness failure.

6.12 Some semaphore implementations provide a function getValue() that returns the current value of a semaphore. This function may, for instance, be invoked prior to calling wait() so that a process will only call wait() if the value of the semaphore is > 0, thereby preventing blocking while waiting for the semaphore. For example: if (getValue(&sem) > 0) wait(&sem);
Many developers argue against such a function and discourage its use. Describe a potential problem that could occur when using the function getValue() in this scenario.

Using a getValue() function to check a semaphore's value before calling wait() can cause a race condition. Between the time the value is checked and the wait() operation is performed, the semaphore's value can change due to other processes. This leads to the process making decisions based on outdated information, potentially causing incorrect behavior like unnecessary blocking. To avoid this, semaphore operations should be atomic, and wait() should be called directly without checking the value beforehand.