Introduction to Parallel Computing

a.y. 2025 – 2026

Dr. Brahmaiah Gandham

Lab 2

Shared Memory Programming with Pthreads

Overview

- **Pthread**
- **Mutex**
- **Semaphores**
- **Condition Variables**
- **Read-Write Lock**

Pthread

```c
#include <pthread.h>
// Create thread
pthread_t tid;
pthread_create(&tid, NULL, thread_func, arg);
// Wait for thread
pthread_join(tid, NULL);
// Thread function signature
void* thread_func(void* arg) {
    // work here
    return NULL;
}
// Detach thread (runs independently)
pthread_detach(tid);
// Exit thread
pthread_exit(NULL);
// Get thread ID
pthread_t self_id = pthread_self();
```

Mutex

```c
pthread_mutex_t lock;
// Initialize
pthread_mutex_init(&lock, NULL);
// OR static: pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
// Lock (blocks if already locked)
pthread_mutex_lock(&lock);
// Try lock (non-blocking, returns 0 if success)
if (pthread_mutex_trylock(&lock) == 0) {
    // got lock
}
// Unlock
pthread_mutex_unlock(&lock);
// Destroy
pthread_mutex_destroy(&lock);
```

Semaphores

```c
#include <semaphore.h>
sem_t sem;
// Initialize (pshared=0 for threads, initial value)
sem_init(&sem, 0, 1);  // binary semaphore
sem_init(&sem, 0, 5);  // counting semaphore (max 5)
// Wait (decrement, blocks if 0)
sem_wait(&sem);
// Try wait (non-blocking)
if (sem_trywait(&sem) == 0) {
    // got semaphore
}
// Post (increment, signals)
sem_post(&sem);
// Get value
int val;
sem_getvalue(&sem, &val);
// Destroy
sem_destroy(&sem);
```

Condition Variables

```c
pthread_cond_t cond;
pthread_mutex_t lock;
// Initialize
pthread_cond_init(&cond, NULL);
pthread_mutex_init(&lock, NULL);
// WAITING THREAD:
pthread_mutex_lock(&lock);
while (!condition) {  // ALWAYS use while, not if!
    pthread_cond_wait(&cond, &lock);  // atomically unlocks and waits
}
// condition is true, lock is held
// do work
pthread_mutex_unlock(&lock);
// SIGNALING THREAD:
pthread_mutex_lock(&lock);
// change condition
condition = true;
pthread_cond_signal(&cond);   // wake one thread
// OR
pthread_cond_broadcast(&cond); // wake all threads
pthread_mutex_unlock(&lock);
// Destroy
pthread_cond_destroy(&cond);
```

Condition Variables

```c
pthread_cond_t cond;
pthread_mutex_t lock;
// Initialize
pthread_cond_init(&cond, NULL);
pthread_mutex_init(&lock, NULL);
// WAITING THREAD:
pthread_mutex_lock(&lock);
while (!condition) {  // ALWAYS use while, not if!
    pthread_cond_wait(&cond, &lock);  // atomically unlocks and waits
}
// condition is true, lock is held
// do work
pthread_mutex_unlock(&lock);
// SIGNALING THREAD:
pthread_mutex_lock(&lock);
// change condition
condition = true;
pthread_cond_signal(&cond);   // wake one thread
// OR
pthread_cond_broadcast(&cond); // wake all threads
pthread_mutex_unlock(&lock);
// Destroy
pthread_cond_destroy(&cond);
```

UNITRENTO

HICREST

Busy Waiting

```c
// Manual busy wait (DON'T DO THIS - wastes CPU)
volatile int flag = 0;
while (flag == 0);  // BAD: burns CPU
// Better: use atomic operations
#include <stdatomic.h>
atomic_int flag = 0;
while (atomic_load(&flag) == 0);
// Best: pthread spinlock
pthread_spinlock_t spinlock;
pthread_spin_init(&spinlock, 0);
pthread_spin_lock(&spinlock);
// critical section
pthread_spin_unlock(&spinlock);
pthread_spin_destroy(&spinlock);
```

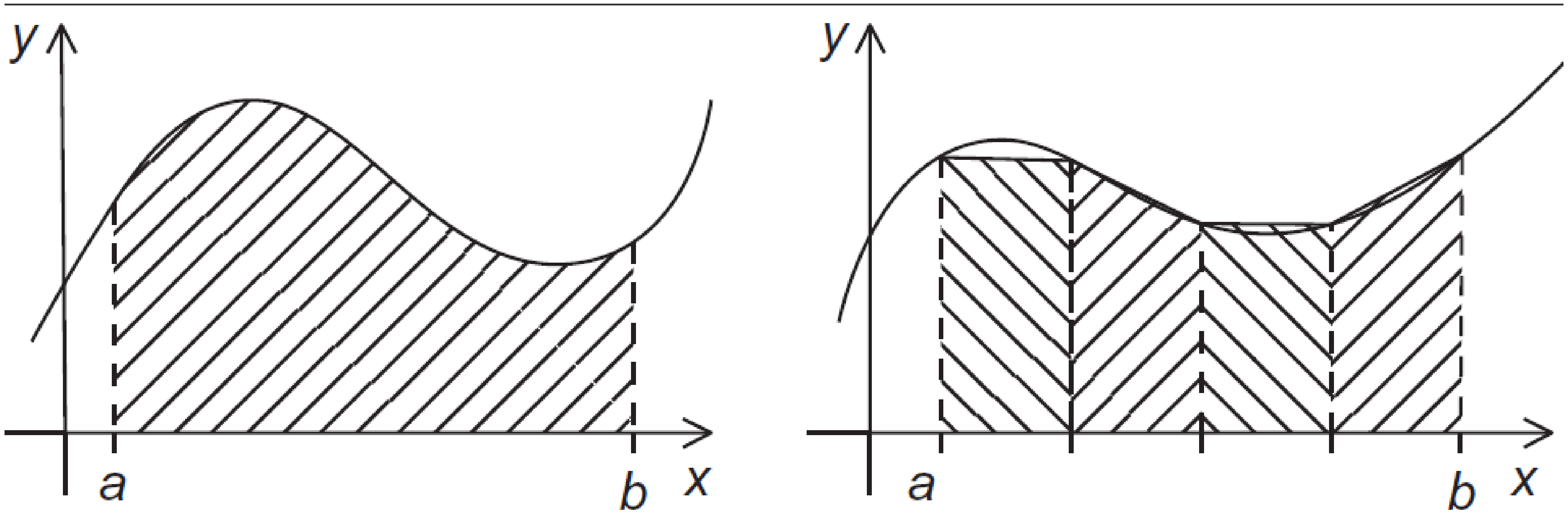
Read-Write Lock

```c
pthread_rwlock_t rwlock;
pthread_rwlock_init(&rwlock, NULL);
// Readers (multiple simultaneous)
pthread_rwlock_rdlock(&rwlock);
// read data
pthread_rwlock_unlock(&rwlock);
// Writer (exclusive)
pthread_rwlock_wrlock(&rwlock);
// write data
pthread_rwlock_unlock(&rwlock);
```

Exercise

- Write a Pthreads program that finds the average time required by your system to create and terminate a thread. Does the number of threads affect the average time? If so, how?
- Write a Pthreads program that implements the trapezoidal rule. Use a shared variable for the sum of all the threads' computations. Use busy-waiting, mutexes, and semaphores to enforce mutual exclusion in the critical section. What advantages and disadvantages do you see with each approach?
- Write a Pthreads program that uses two condition variables and a mutex to implement a read-write lock. Download the linked list program that uses Pthreads read-write locks (Lecture 7), and modify it to use your read-write locks. Now compare the performance of the program when readers are given preference with the program when writers are given preference. Can you make any generalizations?

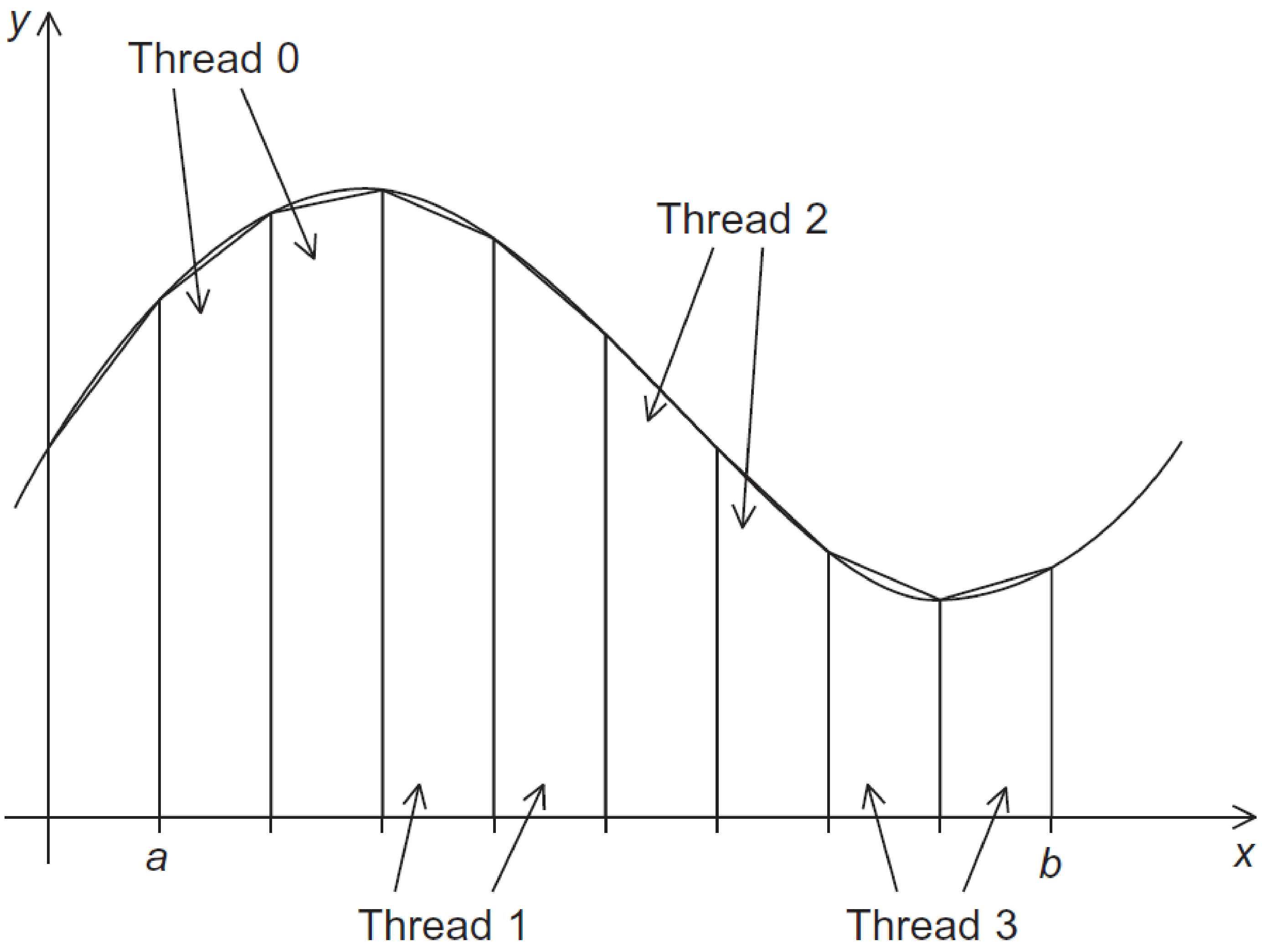
The trapezoidal rule



Serial algorithm

```
/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

Assignment of trapezoids to threads



Key Takeaway

- Shared memory allows multiple threads to work on the same data for faster processing.
- Pthreads help create and manage threads in parallel programs.
- Synchronization tools like mutexes, semaphores, and condition variables prevent data conflicts.
- Busy waiting is simple but inefficient because it wastes CPU time.
- Read–write locks manage access so multiple readers can work together, but only one writer at a time.