# Seminar 2: Parallelization with Pthreads

Laboratories 08 and 09

Sotirios Oikonomou

Group 17

February 18, 2026

## Laboratory 8 — Introduction to Pthreads

### Task 1

### Code

Listing 1: threadtest1.c

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

void* the_thread_func(void* arg) {
  printf("thread_func() starting doing some work.\n");
  long int i;
  volatile double sum = 0.0;
  for (i = 0; i < 2000000000L; i++)
    sum += 1e-7;
  printf("Result of work in thread_func(): sum = %f\n", sum);
  return NULL;
}

void* the_thread_func_B(void* arg) {
  printf("thread_func_B() starting doing some work.\n");
  long int i;
  volatile double sum = 0.0;
  for (i = 0; i < 2000000000L; i++)
    sum += 1e-7;
  printf("Result of work in thread_func_B(): sum = %f\n", sum);
  return NULL;
}

int main() {
  printf("This is the main() function starting.\n");

  pthread_t thread;
  printf("the main() function now calling pthread_create().\n");
  pthread_create(&thread, NULL, the_thread_func, NULL);

  pthread_t threadB;
  pthread_create(&threadB, NULL, the_thread_func_B, NULL);

  printf("This is the main() function after pthread_create()\n");

  printf("main() starting doing some work.\n");
  long int i;
  volatile double sum = 0.0;
  for (i = 0; i < 2000000000L; i++)
    sum += 1e-7;
  printf("Result of work in main(): sum = %f\n", sum);

  printf("the main() function now calling pthread_join().\n");
  pthread_join(thread, NULL);
  pthread_join(threadB, NULL);

  return 0;
}
```

## Compilation and Execution

```
gcc -O2 -Wall -Wextra -pthread threadtest1.c -o threadtest1
./threadtest1
```

## Runtime Results

```
PID   COMMAND      %CPU   TIME      #TH   STATE
4115  threadtest1   300.4  00:21.65  3/3   running
```

## Reflection

- Three active threads (main + 2 workers) were running concurrently.
- CPU usage reached 300%, confirming real parallel execution across multiple cores.
- Output messages were interleaved, showing that main continued after `pthread_create()`.
- Using `volatile` prevented the compiler from optimizing away the workload.

## Task 2

## Code

Listing 2: thread data.c

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

void* the_thread_func(void* arg) {
  double* data = (double*)arg;
  printf("thread_func() got data: %f, %f, %f\n", data[0], data[1], data[2]);
  return NULL;
}

```

```c
int main() {
  printf("main() starting.\n");

  double data_for_thread[3] = {1.1, 2.2, 3.3};
  double data_for_thread_B[3] = {10.0, 20.0, 30.0};

  pthread_t threadA, threadB;

  pthread_create(&threadA, NULL, the_thread_func, (void*)data_for_thread);
  pthread_create(&threadB, NULL, the_thread_func, (void*)data_for_thread_B);

  pthread_join(threadA, NULL);
  pthread_join(threadB, NULL);

  printf("main() done.\n");
  return 0;
}
```

## Compilation and Execution

```
gcc -O2 -Wall -Wextra -pthread thread_data.c -o thread_data
./thread_data
```

## Answer

Yes — the thread can access the values from `main()` through the pointer passed as the 4th argument to `pthread_create()`, as long as the pointed-to data is still valid while the thread runs (here ensured by `pthread_join()` before `main()` returns).

## Task 3

## Code

Listing 3: threaded computation.c

```c
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>

double get_wall_seconds(void)
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (double)tv.tv_sec + (double)tv.tv_usec * 1e-6;
}

const long int N1 = 300000000;
const long int N2 = 500000000;

void* the_thread_func(void* arg) {
  long int i;
  long int sum = 0;
  for(i = 0; i < N2; i++)
    sum += 7;

  long int * resultPtr;
  resultPtr = (long int *)arg;
  *resultPtr = sum;
  return NULL;
}

int main() {
  printf("This is the main() function starting.\n");

  long int thread_result_value = 0;

  pthread_t thread;
  printf("the main() function now calling pthread_create().\n");
  double t0 = get_wall_seconds();
  pthread_create(&thread, NULL, the_thread_func, &thread_result_value);

  printf("This is the main() function after pthread_create()\n");

  long int i;
  long int sum = 0;
  for(i = 0; i < N1; i++)
    sum += 7;

  printf("the main() function now calling pthread_join().\n");
  pthread_join(thread, NULL);
  double t1 = get_wall_seconds();

  printf("Wall time: %f seconds\n", t1 - t0);
  printf("sum computed by main() : %ld\n", sum);
  printf("sum computed by thread : %ld\n", thread_result_value);
  long int totalSum = sum + thread_result_value;
  printf("totalSum : %ld\n", totalSum);

  return 0;
}
```

## Compilation and Execution

```
gcc -std=c11 -Wall -Wextra -O0 threaded_computation.c -o task3 -lpthread
./task3
```

## Runtime Results

```
------ N1 = 700000000, N2 = 100000000 -------
Wall time: 0.363968 seconds
```

```
sum computed by main() : 4900000000
sum computed by thread : 700000000
totalSum : 5600000000

------ N1 = 600000000, N2 = 200000000 -------
Wall time: 0.312483 seconds
sum computed by main() : 4200000000
sum computed by thread : 1400000000
totalSum : 5600000000

------ N1 = 500000000, N2 = 300000000 -------
Wall time: 0.261112 seconds
sum computed by main() : 3500000000
sum computed by thread : 2100000000
totalSum : 5600000000

------ N1 = 400000000, N2 = 400000000 -------
Wall time: 0.214026 seconds
sum computed by main() : 2800000000
sum computed by thread : 2800000000
totalSum : 5600000000

------ N1 = 300000000, N2 = 500000000 -------
Wall time: 0.271641 seconds
sum computed by main() : 2100000000
sum computed by thread : 3500000000
totalSum : 5600000000
```

## Answer

We get the minimum runtime when both threads do approximately the same amount of work (here: `N1 = N2 = 400000000`).

## Task 4

## Code

Listing 4: task4.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>

typedef struct {
    long start;
    long end;
    long count;
} thread_data_t;

```

```
int is_prime(long n)
{
    if(n<2) return 0;
    for (long i = 2; i < n; i++)
    {
        if (n % i == 0)
        {
            return 0;
        }
    }
    return 1;
}

void* thread_count_primes(void* arg)
{
    thread_data_t* data = (thread_data_t*)arg;
    long primes = 0;
    for (long i = data->start; i < data->end; i++)
    {
        if (is_prime(i))
        {
            primes++;
        }
    }
    data->count = primes;
    return NULL;
}

int main(int argc, char const *argv[])
{
    long M = atol(argv[1]);
    if (argc != 2) {
        printf("Usage: %s M\n", argv[0]);
        return 1;
    }

    pthread_t t1, t2;
    thread_data_t A = {.start = 1, .end = M/2, .count = 0};
    thread_data_t B = {.start = M/2 + 1, .end = M, .count = 0};
    pthread_create(&t1, NULL, thread_count_primes, &A);
    pthread_create(&t2, NULL, thread_count_primes, &B);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    long primes = A.count + B.count;
    printf("There are %ld primes!\n", primes);
    return 0;
}
```

## Compilation and Execution

```
gcc -O2 -Wall -Wextra -pthread task4.c -o task4
./task4 M
```

## Runtime Results

```
$ /usr/bin/time -p ./task4 100000
There are 9592 primes!
real 0.18
user 0.26
sys  0.00
```

## Task 5

## Code

Listing 5: task5.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>
#include <unistd.h> // sleep()

typedef struct {
    long index;
} thread_data_t;

void* thread_func(void* arg)
{
    thread_data_t *data = (thread_data_t*)arg;
    printf("Thread %ld\n", data->index);
    return NULL;
}

int main(int argc, char const *argv[])
{
    if (argc != 2) {
        printf("Usage: %s M\n", argv[0]);
        return 1;
    }

    long N = atol(argv[1]);
    pthread_t *threads = malloc(N * sizeof(pthread_t));
    thread_data_t *data = malloc(N * sizeof(thread_data_t));

    for (long i = 0; i < N; i++)
    {
        data[i].index = i;
        int rc = pthread_create(&threads[i], NULL, thread_func, &data[i]);
        if (rc != 0) {
            printf("pthread_create failed at i=%ld (rc=%d)\n", i, rc);
            break;
        }
    }

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

    free(data);
    free(threads);
    return 0;
}
```

## Compilation and Execution

```
gcc -O2 -Wall -Wextra -pthread task5.c -o task5
./task5 N
```

## Task 6

## Code

Listing 6: task6.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>

typedef struct {
    long index;
    long start;
    long end;
    long count;
} thread_data_t;

double get_wall_seconds(void)
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (double)tv.tv_sec + (double)tv.tv_usec * 1e-6;
}

int is_prime(long n)
{
    if(n<2) return 0;
    for (long i = 2; i < n; i++)
    {
        if (n % i == 0)
        {
            return 0;
        }
    }
    return 1;
}

void* thread_count_primes(void* arg)
{
    thread_data_t* data = (thread_data_t*)arg;
    long primes = 0;
    for (long i = data->start; i < data->end; i++)
    {
        if (is_prime(i))
        {
            primes++;
        }
    }
    data->count = primes;
    return NULL;
}

int main(int argc, char const *argv[])
{
    if (argc != 3) {
        printf("Usage: %s M N\n", argv[0]);
        return 1;
    }

    long M = atol(argv[1]);
    long N = atol(argv[2]);
    if (N < 1) { printf("N must be >= 1\n"); return 1; }

    pthread_t *threads = malloc(N * sizeof(pthread_t));
    thread_data_t *data = malloc(N * sizeof(thread_data_t));

    long chunk = M / N;
    long rem = M % N;

    double t0 = get_wall_seconds();

    for (long t = 0; t < N; t++)
    {
        data[t].index = t;
        long extra = (t < rem) ? 1 : 0;

        data[t].start = 1 + t*chunk + (t < rem ? t : rem);
        data[t].end = data[t].start + chunk + extra;
        data[t].count = 0;

```

```c
        pthread_create(&threads[t], NULL, thread_count_primes, &data[t]);
    }

    long total = 0;
    for (long t = 0; t < N; t++) {
        pthread_join(threads[t], NULL);
        total += data[t].count;
    }

    double t1 = get_wall_seconds();
    printf("Wall time: %f seconds\n", t1 - t0);
    printf("There are %ld primes!\n", total);

    free(data);
    free(threads);
    return 0;
}
```

## Compilation and Execution

```
gcc -O2 -Wall -Wextra -pthread task6.c -o task6
./task6 M N
```

## Runtime Results

```
M = 200000

N= 1  Wall time: 0.885137 s
N= 2  Wall time: 0.662642 s
N= 3  Wall time: 0.497856 s
N= 4  Wall time: 0.376413 s
N= 5  Wall time: 0.329214 s
N= 6  Wall time: 0.286538 s
N= 7  Wall time: 0.246784 s
N= 8  Wall time: 0.226419 s
N= 9  Wall time: 0.199653 s
N=10  Wall time: 0.180217 s
N=11  Wall time: 0.162019 s
N=12  Wall time: 0.155053 s
N=13  Wall time: 0.155564 s
N=14  Wall time: 0.144189 s
N=15  Wall time: 0.135020 s
N=16  Wall time: 0.134437 s
N=17  Wall time: 0.132338 s
N=18  Wall time: 0.132321 s
N=19  Wall time: 0.134466 s
N=20  Wall time: 0.122703 s
N=21  Wall time: 0.123618 s
N=22  Wall time: 0.124768 s
```

```
N=23  Wall time: 0.116083 s
N=24  Wall time: 0.120019 s

(Each run reported: There are 17984 primes!)
```

## Answer

After we reach the number of cores, adding more threads gives little benefit because threads must share cores. Any small extra speedup comes mainly from better load balancing, but overhead (context switching/scheduling) prevents further major gains.

## Task 7

## Code

Listing 7: task7.c

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

void* child_thread_func(void* arg)
{
    printf("Hello I am a child thread\n");
    return NULL;
}

void* thread_func(void* arg)
{
    printf("Hello I am a parent thread\n");
    pthread_t ct1, ct2;
    pthread_create(&ct1, NULL, child_thread_func, NULL);
    pthread_create(&ct2, NULL, child_thread_func, NULL);

    pthread_join(ct1, NULL);
    pthread_join(ct2, NULL);
    return NULL;
}

int main(int argc, char const *argv[])
{
    pthread_t pt1, pt2;
    pthread_create(&pt1, NULL, thread_func, NULL);
    pthread_create(&pt2, NULL, thread_func, NULL);

    pthread_join(pt1, NULL);
    pthread_join(pt2, NULL);
    return 0;
}
```

## Compilation and Execution

```
gcc -O2 -Wall -Wextra -pthread task7.c -o task7
./task7
```

## Task 8

## Code

Listing 8: mutextest.c

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

long int sum = 0;
const int WORK = 100000;
pthread_mutex_t sum_mutex;

void* the_thread_func(void* arg) {
  for (int i = 1; i <= WORK; ++i) {
    pthread_mutex_lock(&sum_mutex);
    sum += 1;
    pthread_mutex_unlock(&sum_mutex);
  }
  return NULL;
}

int main(int argc, char **argv) {
  if (argc != 2) { printf("Usage: %s N\n", argv[0]); return -1; }

  printf("This is the main() function starting.\n");

  int num_threads = atoi(argv[1]);

  pthread_mutex_init(&sum_mutex, NULL);

  printf("the main() function now calling pthread_create().\n");
  pthread_t threads[num_threads];
  for (int i = 0; i < num_threads; i++)
    pthread_create(&threads[i], NULL, the_thread_func, NULL);

  printf("This is the main() function after pthread_create()\n");

  printf("the main() function now calling pthread_join().\n");
  for (int i = 0; i < num_threads; i++)
    pthread_join(threads[i], NULL);

  printf("sum = %ld\n", sum);

  pthread_mutex_destroy(&sum_mutex);
  return 0;
}
```

## Compilation and Execution

```
gcc -O2 -Wall -Wextra -pthread mutextest.c -o mutextest
./mutextest 3
```

## Runtime Results

```
--- Without the mutex object ---
sum = 113725
sum = 207745
sum = 111536

--- With the mutex object ---
sum = 300000
sum = 300000
sum = 300000
```

## Answer

Without a mutex, the result changes between runs (race condition) and is often smaller than the expected value. After adding a mutex around the update of the shared variable `sum`, the program produces the same (correct) result every time (here: 3 threads × 100000 increments = 300000).

# Laboratory 9 — Advanced Thread Synchronization

## Task 1

## Code

Listing 9: threadtest.c

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

void* the_thread_func(void* arg) {
  (void)arg;

  int *p = (int*)malloc(3 * sizeof(int));
  if (p == NULL) {
    printf("ERROR: malloc failed in thread.\n");
    return NULL;
  }

  p[0] = 11;
  p[1] = 22;
  p[2] = 33;

  return (void*)p;
}

int main() {
  printf("This is the main() function starting.\n");

  pthread_t thread;
  printf("the main() function now calling pthread_create().\n");
  if(pthread_create(&thread, NULL, the_thread_func, NULL) != 0) {
    printf("ERROR: pthread_create failed.\n");
    return -1;
  }

  printf("This is the main() function after pthread_create()\n");

  printf("the main() function now calling pthread_join().\n");

  void *ret = NULL;
  if(pthread_join(thread, &ret) != 0) {
    printf("ERROR: pthread_join failed.\n");
    return -1;
  }

  int *p = (int*)ret;
  if (p == NULL) {
    printf("ERROR: thread returned NULL.\n");
    return -1;
  }

  printf("Values from thread: %d %d %d\n", p[0], p[1], p[2]);

  free(p);

  return 0;
}
```

## Compilation and Execution

```
gcc -O2 -Wall -Wextra -pthread threadtest.c -o threadtest
./threadtest
valgrind --leak-check=full ./threadtest
```

## Runtime Results

```
This is the main() function starting.
the main() function now calling pthread_create().
This is the main() function after pthread_create()
```

```
the main() function now calling pthread_join().
Values from thread: 11 22 33

== HEAP SUMMARY:
== in use at exit: 0 bytes in 0 blocks
== total heap usage: 3 allocs, 3 frees, 1,308 bytes allocated
== All heap blocks were freed -- no leaks are possible
== ERROR SUMMARY: 0 errors from 0 contexts
```

## Answer

After the thread finishes, the main thread retrieves the pointer returned by the thread using the second argument of `pthread_join()`. The returned pointer refers to dynamically allocated memory created inside the thread function.

The values stored in the thread (11, 22, 33) can be accessed in `main()` after `pthread_join()`, confirming that data was successfully passed from the thread back to the main thread.

Since the memory was allocated using `malloc()` inside the thread, it must be released using `free()` in `main()`. Valgrind confirms that all allocated memory is properly freed and that there are no memory leaks.

## Task 2

## Code

Listing 10: wait_test.c

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

/* Shared state: worker thread must wait until main sets this to 1. */
int DoItNow = 0;

/* Mutex protects DoItNow; condition variable allows efficient waiting. */
pthread_mutex_t m;
pthread_cond_t c;

void* thread_func(void* arg) {
  (void)arg;

  printf("This is thread_func() starting, now entering loop to wait until
      DoItNow is set...\n");

  /* Condition variables are used with a mutex guarding the predicate (DoItNow)
      . */
  pthread_mutex_lock(&m);

  /* Use while (not if) to handle spurious wakeups safely. */
  while (DoItNow == 0) {
    /* Atomically: release m and sleep; re-acquire m before returning. */
    pthread_cond_wait(&c, &m);
  }

  pthread_mutex_unlock(&m);

  printf("This is thread_func() after the loop.\n");
  return NULL;
}

int main() {
  printf("This is the main() function starting.\n");

  pthread_mutex_init(&m, NULL);
  pthread_cond_init(&c, NULL);

  /* Start thread. */
  pthread_t thread;
  printf("the main() function now calling pthread_create().\n");
  pthread_create(&thread, NULL, thread_func, NULL);
  printf("This is the main() function after pthread_create()\n");

  /* Here we let the main thread do some work. */
  long int k;
  double x = 1;
  for(k = 0; k < 2000000000; k++)
    x *= 1.00000000001;
  printf("main thread did some work, x = %f\n", x);

  /* Set predicate and wake the waiting thread. */
  pthread_mutex_lock(&m);
  DoItNow = 1;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);

  /* Wait for thread to finish. */
  printf("the main() function now calling pthread_join().\n");
  pthread_join(thread, NULL);
  printf("This is the main() function after calling pthread_join().\n");

  pthread_cond_destroy(&c);
  pthread_mutex_destroy(&m);

  return 0;
}
```

## Compilation and Execution

```
gcc -O2 -Wall -Wextra -pthread wait_test.c -o wait_test
./wait_test
time ./wait_test
```

## Runtime Results

```
This is the main() function starting.
the main() function now calling pthread_create().
This is the main() function after pthread_create()
This is thread_func() starting, now entering loop to wait until DoItNow is set...
main thread did some work, x = 1.020201
the main() function now calling pthread_join().
This is thread_func() after the loop.
This is the main() function after calling pthread_join().

./wait_test  1.70s user 0.07s system 99% cpu 1.771 total

Experiment A (removed: DoItNow = 1;)
Program hangs at: the main() function now calling pthread_join().
(Worker keeps waiting; join never returns.)

Experiment B (removed: pthread_cond_signal(&c);)
Program hangs at: the main() function now calling pthread_join().
(Worker sleeps in pthread_cond_wait() and is never woken up.)
```

## Answer

Removing `DoItNow = 1` means the condition never becomes true, so the worker thread keeps waiting and `pthread_join()` blocks forever. If `pthread_cond_signal()` is removed, the condition may become true but the worker is never woken up from `pthread_cond_wait()`, so `pthread_join()` again blocks forever. Using a condition variable avoids the busy-wait loop (and the corresponding wasted CPU) because the waiting thread sleeps until it is signaled.

## Task 3

## Code

Listing 11: join.c

```c
/******************************************************************************
 * FILE: join.c
 ******************************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS 4

void *BusyWork(void *t)
{
  int i;
  long tid;
  double result = 0.0;

  tid = (long)t;
  printf("Thread %ld starting...\n", tid);

  for (i = 0; i < 20000000; i++) {
    result = result + sin(i) * tan(i);
  }

  printf("Thread %ld done. Result = %e\n", tid, result);
  pthread_exit((void*)t);
}

int main(int argc, char *argv[])
{
  pthread_t thread[NUM_THREADS];
  pthread_attr_t attr;
  int rc;
  long t;
  void *status;

  pthread_attr_init(&attr);

  /* Variant A: PTHREAD_CREATE_JOINABLE */
  /* Variant B/C: PTHREAD_CREATE_DETACHED */
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

  for (t = 0; t < NUM_THREADS; t++) {
    printf("Main: creating thread %ld\n", t);
    rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  pthread_attr_destroy(&attr);

  for (t = 0; t < NUM_THREADS; t++) {
    rc = pthread_join(thread[t], &status);
    if (rc) {
      printf("ERROR; return code from pthread_join() is %d\n", rc);
      exit(-1);
    }
    printf("Main: completed join with thread %ld having a status of %ld\n",
           t, (long)status);
  }

  printf("Main: program completed. Exiting.\n");
  pthread_exit(NULL);
}
```

## Compilation and Execution

```
make clean
make
./join
```

## Runtime Results

### Variant A (Joinable + pthread_join)

```
Main: creating thread 0
Main: creating thread 1
Thread 0 starting...
Thread 1 starting...
Main: creating thread 2
Main: creating thread 3
Thread 2 starting...
Thread 3 starting...
Thread 1 done. Result = 4.931540e+06
Thread 3 done. Result = 4.931540e+06
Thread 0 done. Result = 4.931540e+06
Main: completed join with thread 0 having a status of 0
Main: completed join with thread 1 having a status of 1
Thread 2 done. Result = 4.931540e+06
Main: completed join with thread 2 having a status of 2
Main: completed join with thread 3 having a status of 3
Main: program completed. Exiting.
```

### Variant B (Detached, no join, pthread_exit in main)

```
Main: creating thread 0
Main: creating thread 1
Thread 0 starting...
Main: creating thread 2
Thread 1 starting...
Thread 2 starting...
Main: creating thread 3
Main: program completed. Exiting.
Thread 3 starting...
Thread 0 done. Result = 4.931540e+06
Thread 1 done. Result = 4.931540e+06
Thread 2 done. Result = 4.931540e+06
Thread 3 done. Result = 4.931540e+06
```

### Variant C (Detached, no join, no pthread_exit)

```
Main: creating thread 0
Main: creating thread 1
Thread 0 starting...
Main: creating thread 2
```

```
Thread 1 starting...
Main: creating thread 3
Main: program completed. Exiting.
```

## Answer

In Pthreads, a **joinable thread** allows another thread to wait for its termination using `pthread_join()`, and retrieve its exit status. Resources associated with the thread are released only after a successful join.

A **detached thread** cannot be joined. Its resources are automatically released upon termination, and its exit status cannot be collected.

In Variant A, the main thread blocks at each `pthread_join()`, ensuring that all worker threads complete before program termination.

In Variant B, threads are detached and the join calls are removed. However, because `pthread_exit(NULL)` is still called in `main()`, the process remains alive until all threads finish execution. Therefore, all worker completion messages are printed.

In Variant C, removing `pthread_exit(NULL)` causes the program to terminate immediately when `main()` returns. As a result, worker threads are terminated prematurely and their completion messages are not printed.

This demonstrates the semantic difference between thread detachment and process termination behavior in Pthreads.

## Task 4

## Code

Listing 12: synch.c

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

#define NUM_THREADS 8

pthread_mutex_t lock;
pthread_cond_t mysignal;
int waiting = 0;
int state = 0;

void barrier()
{
  int mystate;

  pthread_mutex_lock(&lock);
  mystate = state;
  waiting++;

  if (waiting == NUM_THREADS) {
    waiting = 0;
    state = 1 - mystate;
    pthread_cond_broadcast(&mysignal);
  }

  while (mystate == state) {
    pthread_cond_wait(&mysignal, &lock);
  }

  pthread_mutex_unlock(&lock);
}

void* HelloWorld(void* arg)
{
  long id = (long)arg;

  printf("Hello World! %ld\n", id);
  barrier();
  printf("Bye Bye World! %ld\n", id);

  return NULL;
}

int main(int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  long t;

  pthread_cond_init(&mysignal, NULL);
  pthread_mutex_init(&lock, NULL);

  for (t = 0; t < NUM_THREADS; t++)
    pthread_create(&threads[t], NULL, HelloWorld, (void*)t);

  for (t = 0; t < NUM_THREADS; t++)
    pthread_join(threads[t], NULL);

  pthread_cond_destroy(&mysignal);
  pthread_mutex_destroy(&lock);

  return 0;
}
```

Listing 13: spinwait.c

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

#define NUM_THREADS 8

pthread_mutex_t lock;
int waiting = 0;
volatile int state = 0;
```

```

void barrier()
{
  int mystate;

  pthread_mutex_lock(&lock);
  mystate = state;
  waiting++;

  if (waiting == NUM_THREADS) {
    waiting = 0;
    state = 1 - mystate;
  }

  pthread_mutex_unlock(&lock);

  while (mystate == state) ;
}

void* HelloWorld(void* arg)
{
  long id = (long)arg;

  printf("Hello World! %ld\n", id);
  barrier();
  printf("Bye Bye World! %ld\n", id);

  return NULL;
}

int main(int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  long t;

  pthread_mutex_init(&lock, NULL);

  for (t = 0; t < NUM_THREADS; t++)
    pthread_create(&threads[t], NULL, HelloWorld, (void*)t);

  for (t = 0; t < NUM_THREADS; t++)
    pthread_join(threads[t], NULL);

  pthread_mutex_destroy(&lock);

  return 0;
}
```

## Compilation and Execution

```
# synch.c (condition-variable barrier)
```

```
gcc -Wall -O2 -o synch synch.c -lpthread
./synch

# synch.c without barrier (barrier() call removed in HelloWorld)
gcc -Wall -O2 -o synch synch.c -lpthread
./synch

# spinwait.c without optimization
gcc -Wall -O0 -o spinwait spinwait.c -lpthread
./spinwait

# spinwait.c with -O3 (no volatile): hangs
gcc -Wall -O3 -o spinwait spinwait.c -lpthread
./spinwait

# spinwait.c with -O3 (volatile fix): works
gcc -Wall -O3 -o spinwait spinwait.c -lpthread
./spinwait
```

## Runtime Results

### synch.c with barrier

```
Hello World! 0
Hello World! 3
Hello World! 1
Hello World! 2
Hello World! 5
Hello World! 4
Hello World! 6
Hello World! 7
Bye Bye World! 7
Bye Bye World! 3
Bye Bye World! 2
Bye Bye World! 5
Bye Bye World! 4
Bye Bye World! 0
Bye Bye World! 6
Bye Bye World! 1
```

### synch.c without barrier (barrier removed)

```
Hello World! 0
Bye Bye World! 0
Hello World! 1
Bye Bye World! 1
Hello World! 2
Bye Bye World! 2
Hello World! 4
```

```
Bye Bye World! 4
Hello World! 5
Bye Bye World! 5
Hello World! 3
Bye Bye World! 3
Hello World! 6
Bye Bye World! 6
Hello World! 7
Bye Bye World! 7
```

**spinwait.c compiled with -O0**

```
Hello World! 0
Hello World! 4
Hello World! 1
Hello World! 5
Hello World! 2
Hello World! 6
Hello World! 7
Hello World! 3
Bye Bye World! 3
Bye Bye World! 2
Bye Bye World! 1
Bye Bye World! 7
Bye Bye World! 6
Bye Bye World! 4
Bye Bye World! 5
Bye Bye World! 0
```

**spinwait.c compiled with -O3 (without volatile)**

```
Hello World! 0
Hello World! 4
Hello World! 1
Hello World! 5
Hello World! 2
Hello World! 7
Hello World! 6
Hello World! 3
Bye Bye World! 3
(hangs; no further progress)
```

**spinwait.c compiled with -O3 (with volatile)**

```
Hello World! 0
Hello World! 1
Hello World! 3
Hello World! 4
Hello World! 2
Hello World! 5
Hello World! 6
Hello World! 7
Bye Bye World! 7
Bye Bye World! 2
Bye Bye World! 5
Bye Bye World! 4
Bye Bye World! 0
Bye Bye World! 3
Bye Bye World! 1
Bye Bye World! 6
```

## Answer

**Condition-variable barrier (synch.c):**

With the barrier enabled, all threads print `Hello World!` first and only after every thread reaches the barrier do they print `Bye Bye World!`. This shows correct barrier synchronization: execution proceeds in two distinct phases.

When the barrier is removed, each thread prints `Hello` and immediately prints `Bye` without waiting for others. The two phases become interleaved, meaning no synchronization is enforced.

**Spin-wait barrier (spinwait.c):**

With `-O0`, the program works because the busy-wait loop repeatedly reloads the shared variable from memory.

With `-O3` and without `volatile`, the program hangs. The compiler optimizes the loop by caching the shared variable in a register, so updates made by other threads are not observed.

Declaring the shared variable as `volatile` forces a memory read on every loop iteration. As a result, updates become visible and the program completes correctly.

## Task 5

## Code

Listing 14: pi.c

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

typedef struct {
  long start;
  long end;
  double dx;
  double *out; /* per-thread storage to avoid mutex on global sum */
} thread_arg_t;

static void *worker(void *arg)
{
  thread_arg_t *a = (thread_arg_t *)arg;
  double sum = 0.0;

  /* Each thread integrates a contiguous block of intervals */
  for (long i = a->start; i <= a->end; i++) {
    double x = a->dx * ((double)i - 0.5);
    sum += a->dx * 4.0 / (1.0 + x * x);
  }

  *(a->out) = sum;
  return NULL;
}

int main(int argc, char *argv[])
{
  const long intervals_default = 500000000;
  int num_threads = 4;
  long intervals = intervals_default;

  if (argc > 1) num_threads = atoi(argv[1]);
  if (argc > 2) intervals = atol(argv[2]);

  pthread_t *threads = malloc(num_threads * sizeof(*threads));
  thread_arg_t *args = malloc(num_threads * sizeof(*args));
  double *partial = malloc(num_threads * sizeof(*partial));

  double dx = 1.0 / (double)intervals;

  long base = intervals / num_threads;
  long rem = intervals % num_threads;

  long next = 1;
  for (int t = 0; t < num_threads; t++) {
    long chunk = base + (t < rem ? 1 : 0);

    args[t].start = next;
    args[t].end = next + chunk - 1;
    args[t].dx = dx;
    args[t].out = &partial[t];
    partial[t] = 0.0;

    pthread_create(&threads[t], NULL, worker, &args[t]);
    next = args[t].end + 1;
  }

  /* Join ensures all partial sums are computed before reduction */
  for (int t = 0; t < num_threads; t++)
    pthread_join(threads[t], NULL);

  double sum = 0.0;
  for (int t = 0; t < num_threads; t++)
    sum += partial[t];

  printf("PI is approx. %.16f\n", sum);

  free(threads);
  free(args);
  free(partial);

  return 0;
}
```

## Compilation and Execution

```
make clean
make
time ./pi 1 10000000
time ./pi 4 10000000
```

## Runtime Results

### 1 thread (serial equivalent)

```
PI is approx. 3.1415926535904357
0.01s user 0.00s system 6% cpu 0.249 total
```

### 4 threads (parallel)

```
PI is approx. 3.1415926535897656
0.01s user 0.00s system 121% cpu 0.007 total
```

## Answer

The parallel implementation divides the integration interval into contiguous blocks and assigns one block to each thread. Each thread computes a local partial sum, and the main thread performs the final reduction after joining all threads.

The numerical results are not exactly identical between the 1-thread and 4-thread runs. This is expected due to floating-point rounding: addition is not associative, and changing the summation order alters the rounding error.

Neither implementation is strictly more accurate. The dominant error source is the discretization error (number of intervals). The small difference observed arises solely from the different accumulation order in floating-point arithmetic.

## Task 6

## Code

Listing 15: enumsort.c - Enumeration sort with two threading strategies

```c
/* Only relevant parallel parts shown */

```

```c
void *findrank(void *arg)
{
  int rank;
  long i;
  long j = (long)arg;

  rank = 0;
  for (i = 0; i < len; i++)
    if ((indata[i] < indata[j]) ||
        (indata[i] == indata[j] && i < j))
      rank++;

  outdata[rank] = indata[j];
  pthread_exit(NULL);
}

/* Improved strategy: one thread computes ranks for multiple elements */
void *findranks_chunk(void *arg)
{
  chunk_arg_t *a = (chunk_arg_t *)arg;

  for (long j = a->start; j < a->end; j++) {
    int rank = 0;
    for (long i = 0; i < len; i++)
      if ((indata[i] < indata[j]) ||
          (indata[i] == indata[j] && i < j))
        rank++;
    outdata[rank] = indata[j];
  }

  return NULL;
}
```

## Compilation and Execution

```
make clean
make
./enumsort
```

## Runtime Results

```
Approach1 (many create/join): Time: 1.299254  NUM_THREADS: 5
Approach2 (chunk per thread): Time: 0.478580  NUM_THREADS: 5
```

## Answer

### Comparison of approaches:

Approach 1 creates and joins threads repeatedly for small tasks. The overhead of frequent thread creation dominates execution time, leading to poor performance.

Approach 2 creates a fixed number of threads once, and each thread computes the ranks of multiple elements. This significantly reduces thread management overhead and results in substantially better performance (more than 2× speedup in our measurements).

**Computational complexity:**

Enumeration sort has time complexity $O(n^2)$, since for each element it compares against all other elements.

**Comparison with merge sort:**

Merge sort has time complexity $O(n \log n)$ and is asymptotically much more efficient. Although enumeration sort is naturally parallelizable (each rank computation is independent), its quadratic complexity makes it inefficient for large inputs. Therefore, compared to merge sort, enumeration sort is not a good general-purpose sorting algorithm.

## Task 7

## Code

Listing 16: matmul.c - Row-block parallel matrix multiplication

```
/* Each thread computes a disjoint block of rows of C.
   No synchronization is required during multiplication. */

static void *matmul_rows(void *arg)
{
  thread_arg_t *a = (thread_arg_t *)arg;

  for (int i = a->row_start; i < a->row_end; i++)
    for (int j = 0; j < n; j++) {
      double sum = 0.0;
      for (int k = 0; k < n; k++)
        sum += A[i][k] * B[k][j];
      C[i][j] = sum;
    }

  return NULL;
}
```

## Compilation and Execution

```
make clean
make

time ./matmul 1000 1
time ./matmul 1000 2
time ./matmul 1000 4
time ./matmul 1000 8
```

## Runtime Results (n=1000)

```
Threads=1  total  1.159 s
```

```
Threads=2  total  0.499 s
Threads=4  total  0.281 s
Threads=8  total  0.185 s
```

Maximum speedup (8 threads):

$$S_{max} = \frac{1.159}{0.185} \approx 6.27$$

## Small Matrix Sweep

```
n=50   → parallel slower (overhead dominates)
n=100  → parallel slower
n=200  → small improvement
n=300  → noticeable speedup
n=400+ → parallel clearly beneficial
```

## Answer

**Parallelism:**

The computation of each row of matrix $C$ is independent. We divide the rows among threads (static row-block distribution). No locks are required because each thread writes to distinct rows.

**Maximum speedup (1000×1000):**

Best observed speedup with 8 threads:

$$S \approx 6.3$$

This is below ideal linear speedup due to thread overhead, memory bandwidth limits, and cache effects.

**When is parallelization worthwhile?**

For small matrices ($n \leq 100$), parallel execution is slower due to thread creation and synchronization overhead.

From approximately $n \geq 300$, computation cost dominates overhead and parallelization yields clear speedup.

**Limiting factors:**

Speedup is limited by: - Thread management overhead - Memory bandwidth constraints - Poor cache locality of naive matrix multiplication - Amdahl's Law