Lab 3

Tousif Habib
201726429
Mohammed Shakeel
201831104

Real Time Operating Systems Lab 3

Prelab

1. pthread_t is an implementation-defined type, meaning that its underlying type can vary depending on the implementation. It is an opaque type, so you're not supposed to know what it really is. In many threads implementations, the pthread_t abstract type is implemented as an integer (4 byte) thread ID. A pthread_t is unique within a process until the thread terminates and may be reused after that.

2. A thread-safe function is a function that can be safely called simultaneously from multiple threads without causing any problems such as data corruption or race conditions. Here's an example of a simple C function that is not thread-safe:

```
int i = 0;
void increment() {
    i++;
}
```

   This function is not thread-safe because if two threads call it simultaneously, they might both read the value of i, increment it, and write it back at the same time, resulting in only one increment instead of two.

   Here's an improved version that is thread-safe:

```
#include <pthread.h>
int i = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
void increment() {
    pthread_mutex_lock(&lock);
    i++;
    pthread_mutex_unlock(&lock);
}
```

   This version uses a mutex to ensure that only one thread can access the i variable at a time.

3. a. Here's an implementation of a function void increment(struct foo *), which increments the f_count field in a thread-safe manner:

```
void increment(struct foo *f) {
    pthread_mutex_lock(&f->f_lock);
    f->f_count++;
    pthread_mutex_unlock(&f->f_lock);
}
```

   b. Here's an implementation of the function struct foo * foo_create(void):

```
struct foo * foo_create(void) {
    struct foo *f = malloc(sizeof(struct foo));
    f->f_count = 0;
    pthread_mutex_init(&f->f_lock, NULL);
    return f;
}
```

4. The document describes C blocks as a newly-supported feature of the C language family on macOS, which are segments of code that can have arguments, read-only access to local variables, be assigned to variables, and invoked using the variable name. In contrast, C++ lambda functions are anonymous functions that can capture variables from the enclosing scope by value or by reference, and can have a return type.

   While there are similarities between C blocks and C++ lambda functions in terms of their ability to capture variables and be assigned to variables, there are also notable differences in their syntax and capabilities. C blocks use the caret (^) symbol to denote a block, whereas C++ lambda functions use the square bracket ([]) syntax. Additionally, C++ lambda functions can have a return type, while C blocks cannot.

   Overall, C blocks and C++ lambda functions are similar in concept, but differ in syntax and capabilities.

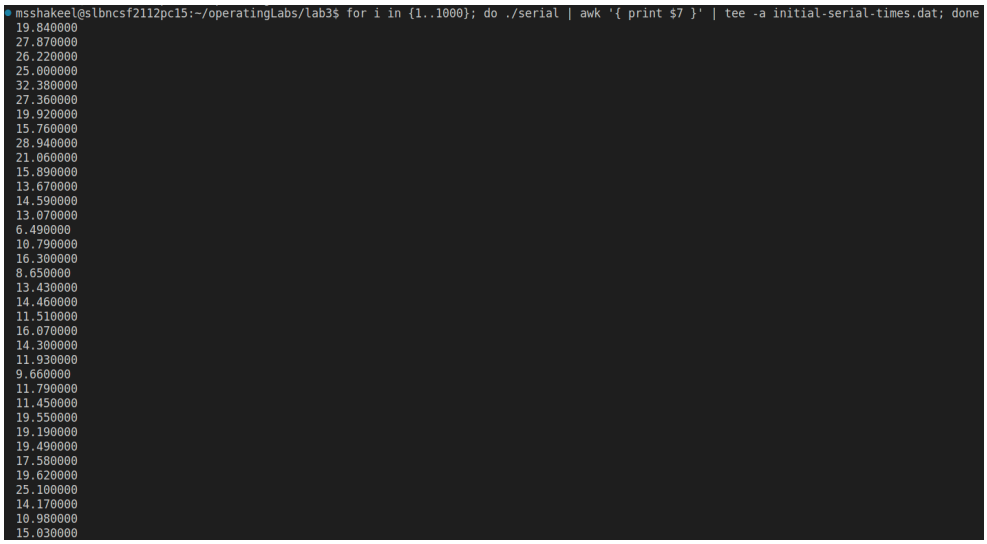
Lab 3

1. This C program serial.c defines a program that counts from 0 to a specified number of iterations, using a loop that runs a certain amount of "work" on each iteration. The program measures the time it takes to perform this counting operation and outputs the number of iterations counted, the total time elapsed, and the average time per iteration.
2. Output:

```
msshakeel@slbncsf2112pc15:~/operatingLabs/lab3$ ./serial | awk '{ print $7 }' | tee -a initial-serial-times.dat
25.690000
```

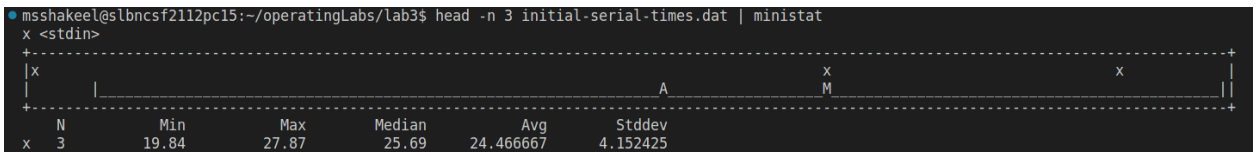
3. Use the shell to run this command 999 more times.
   command:

   for i in {1..1000}; do ./serial | awk '{ print $7 }' | tee -a initial-serial-times.dat; done

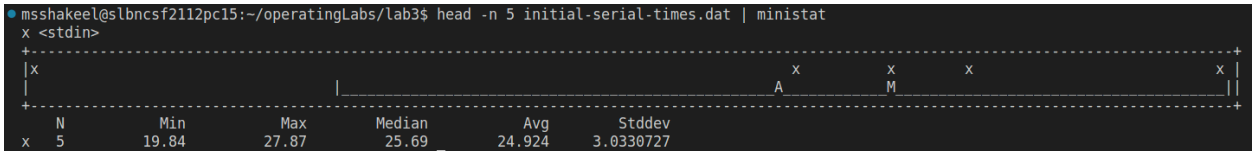


   Using head(1) and ministat(1), report the average and standard deviation for the first 3, 5, 10, 100 and 1000 runs.

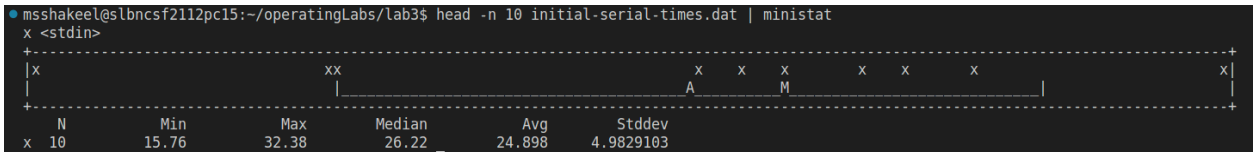
   For first 3:



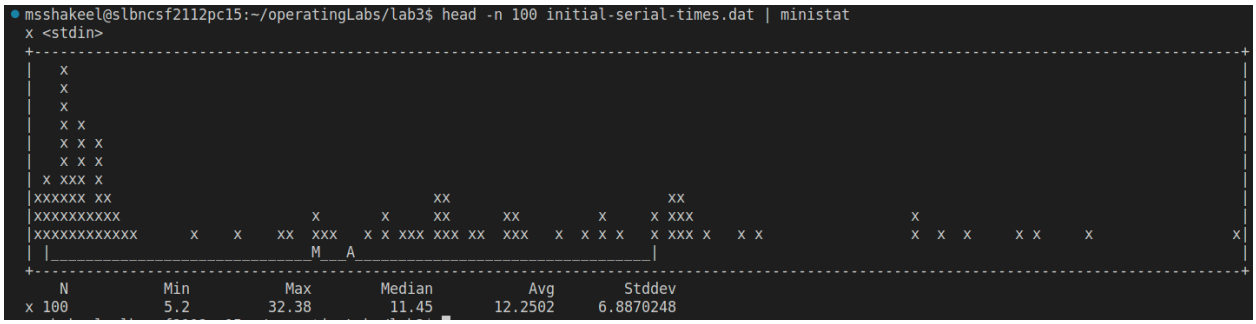
First 5:



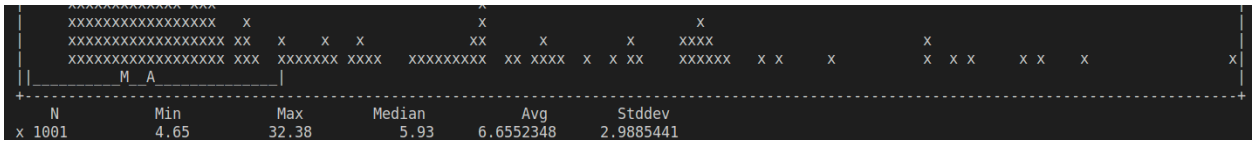
First 10:



First 100:



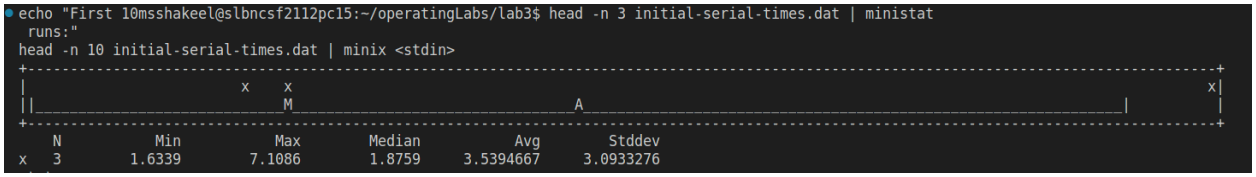
First 1000: command “ministat initial-serial-times.dat”

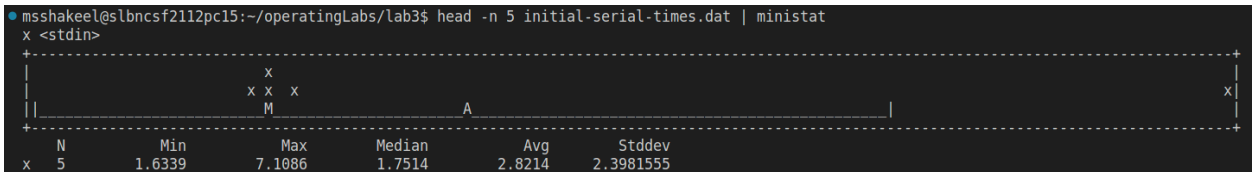


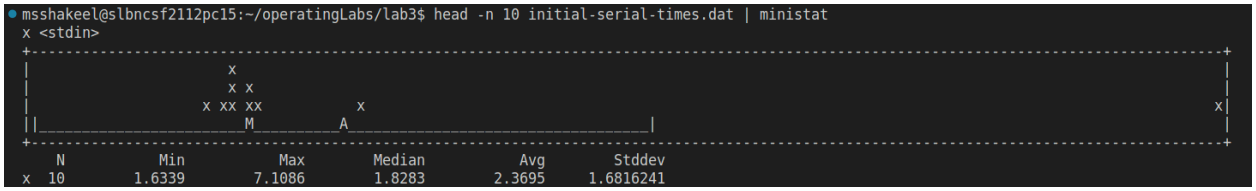
Number of runs	Min	Max	Median	Average	Stddev
3	19.84	27.87	25.69	24.47	4.15
5	19.84	27.87	25.69	24.92	3.03
10	15.76	32.38	26.22	24.89	4.98
100	5.2	32.38	11.45	12.50	6.89
1000	4.65	32.38	5.93	6.66	2.99

As the number of runs increases up to 10, the minimum time for a run remains relatively stable, ranging from 15.76 to 19.84 nanoseconds. However, when the number of runs reaches 100, the minimum time significantly decreases to 5.2 nanoseconds, and further decreases to 4.65 nanoseconds at 1000 runs. With more runs, the program should be able to cache information, making the minimum time lower. The maximum run times for 3, 5, and 10 runs are fairly consistent, ranging from 27.87 to 32.38 nanoseconds. However, as the number of runs increases to 100 and 1000, the maximum run time remains stable at 32.38 nanoseconds.

4. **Output** after running the command: make JOBS=100 WORK_PER_JOB=100 clean serial







```
msshakeel@slbncsf2112pc15:~/operatingLabs/lab3$ head -n 100 initial-serial-times.dat | ministat
x <stdin>
+--------------------------------------------------------------------------------------------------------------------------------------------+
|                                         x                                                                                                  |
|  x                                      x                                                                                                  |
|  x                             xx       x                   x    x                                                                       |
| xx x   x   x x      xxxx  x  xx      xxx  x   xxx x       xx    x     x    x      x                                                      |
|xxxxxxx x xxx x   xxxxxxxx xxxxx x xx xxx xxxxxxxxxxxxx  xxxxxx xx xxx x   xx   xx xx                           x                    x|
|              |_________________________A__________________________|                                                                        |
+--------------------------------------------------------------------------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x 100        1.5781       11.6507        4.4823      4.516168     1.9076263
```

```
|   x xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xx x xxx   x   x             x                                                     |
|   x xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xx x xxxx  xx  x x x         x                                                     |
|xx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xx xxxx xx x      x x                                                   |
|xx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  xxx x xxx                       x                    x|
|       |________________M__A_________________|                                                                                            |
+--------------------------------------------------------------------------------------------------------------------------------------------+
    N           Min           Max        Median           Avg        Stddev
x 1000       1.2065       11.6507        3.0435_     3.2374853     1.4724113
```

Execution time decreases when the number of JOBS goes up while WORK_PER_JOB remains unchanged. On the other hand, if WORK_PER_JOB increases while the number of JOBS stays the same, execution time increases.

POSIX Threads

(Running on M2 Pro Chip so results may look different)

Code for pthreads.c:

```
#include <err.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
void *increment(void *countp);
int main(int argc, char *argv[]) {
int counter = 0;
struct timespec begin, end;
     // Set C locale settings to get niceties like thousands
separators
     // for decimal numbers.
     setlocale(LC_NUMERIC, "");
 if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &begin) != 0) {
       err(-1, "Failed to get start time");
 }
pthread_t threads[JOBS];
int rc;
for (int i = 0; i < JOBS; i++)
 {
   rc = pthread_create(&threads[i], NULL, increment, &counter );
}
for(int i = 0; i < JOBS; i++) {
    pthread_join(threads[i], NULL);
}
 if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end) != 0) {
     err(-1, "Failed to get end time");
}
long diff = end.tv_nsec - begin.tv_nsec;
diff += (1000 * 1000 * 1000) * (end.tv_sec - begin.tv_sec);
printf("Counted to %'d in %'ld ns: %f ns/iter\n",
JOBS * WORK_PER_JOB, diff, ((double) diff) / counter);
 return 0; }
void *increment(void *countp) {
for (int i = 0; i < WORK_PER_JOB; i++) {
(int)countp++;
}
}
```

Bash script to create csv of jobs vs average time:

```
#!/bin/bash

JOBS_RANGE="1 2 4 8 16 32 64 128 256 512 1024"
WORK_PER_JOB=1000

# Remove any existing data file
rm -f jobs_vs_avg_time.csv

# Run the program with different values of JOBS
for jobs in $JOBS_RANGE; do
   make clean && make JOBS=$jobs WORK_PER_JOB=$WORK_PER_JOB
   result=$(./pthreads | awk '{print $7}')
   echo "$jobs,$result" >> jobs_vs_avg_time.csv
done
```

Bash script to create csv of work per job vs average time:

```
#!/bin/bash

JOBS=4
WORK_PER_JOB_RANGE="1 2 4 8 16 32 64 128 256 512 1024"

# Remove any existing data file
rm -f work_per_job_vs_avg_time.csv

# Run the program with different values of WORK_PER_JOB
for work_per_job in $WORK_PER_JOB_RANGE; do
   make clean && make JOBS=$JOBS WORK_PER_JOB=$work_per_job
   result=$(./pthreads | awk '{print $7}')
   echo "$work_per_job,$result" >> work_per_job_vs_avg_time.csv
done
```

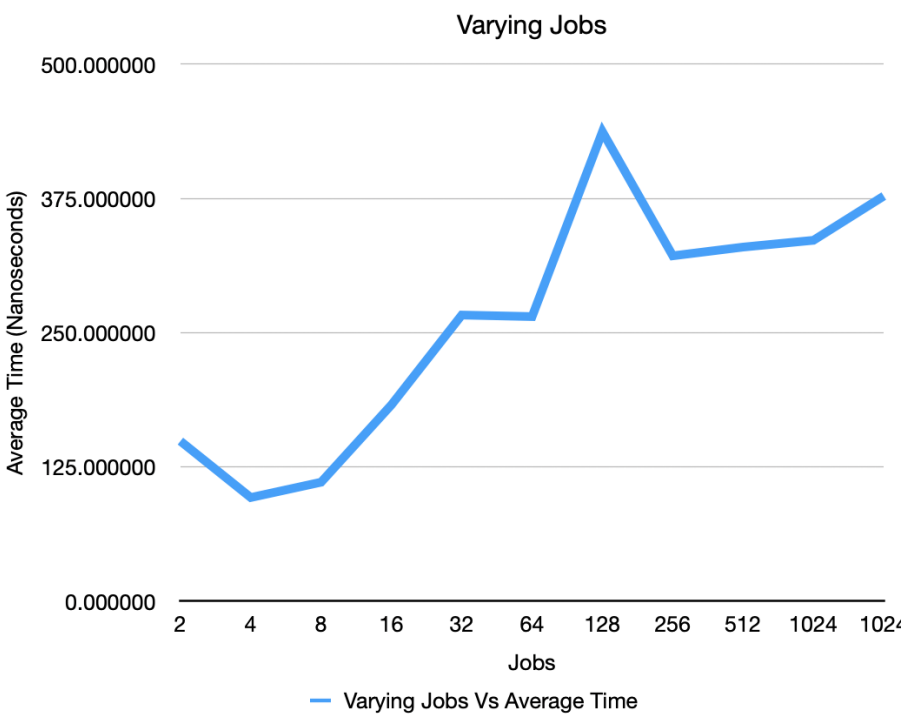
Bash script to create csv of throughput vs threads:

```bash
#!/bin/bash

# Change the following line to the desired range of values
THREADS_RANGE="1 2 4 8 16 32 64 128"
WORK_PER_JOB=1000

# Remove any existing data file
rm -f throughput_vs_threads.csv

# Run the program with different values of threads
for threads in $THREADS_RANGE; do
   make clean && make JOBS=$threads WORK_PER_JOB=$WORK_PER_JOB
   result=$(./pthreads | awk '{print $7}')
   throughput=$(echo "scale=5; $WORK_PER_JOB / $result" | bc)
   echo "$threads,$throughput" >> throughput_vs_threads.csv
done
```

Plots:

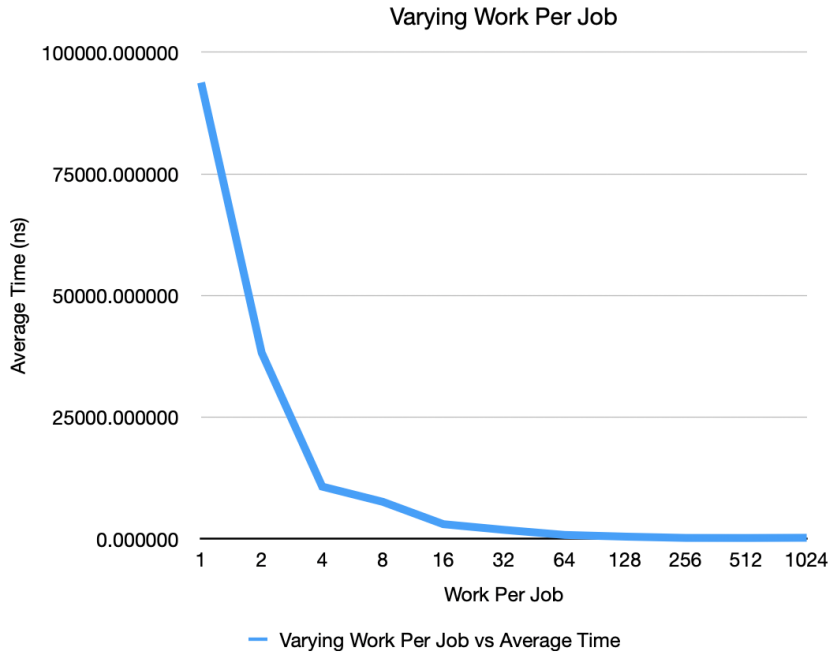
Jobs vs Avg. Time

Jobs	Time
1	149.000000
2	96.500000
4	110.750000
8	182.750000
16	266.375000
32	264.937500
64	437.046875
128	321.531250
256	329.648438
512	335.960938
1024	377.160156



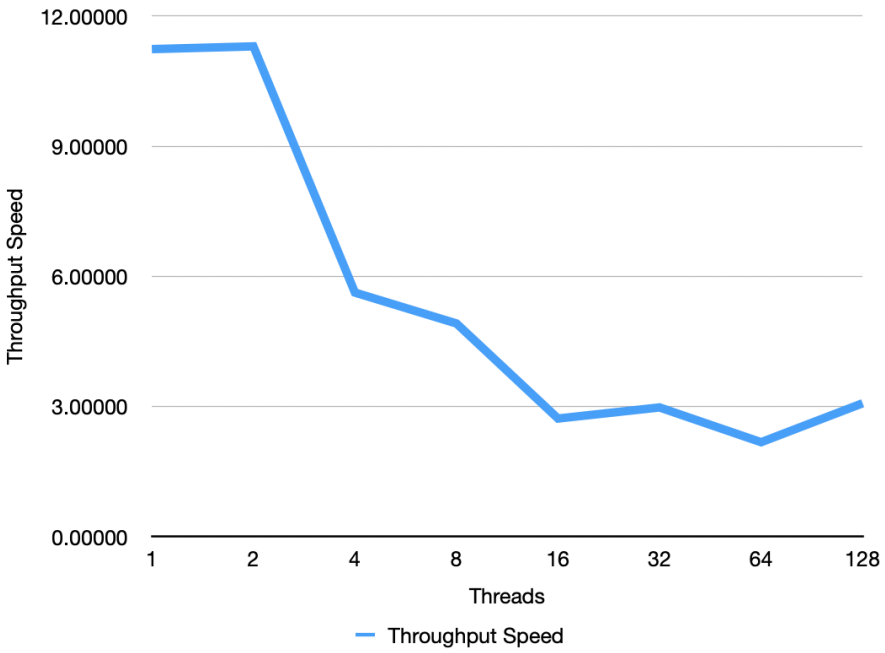
Work Per Job vs Average Time

Work Per Job	Average Time
1	93750.000000
2	38250.000000
4	10750.000000
8	7625.000000
16	3031.250000
32	1867.187500
64	824.218750
128	476.562500
256	209.960938
512	196.777344
1024	240.478516



Threads vs Throughput

Threads	Throughput
1	11.23595
2	11.29943
4	5.62587
8	4.91400
16	2.72340
32	2.97896
64	2.17790
128	3.07514



Libdispatch:

(Running on M2 Pro Chip so results may look different)

Code for libdispatch.c:

```c
#include <err.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <dispatch/dispatch.h>
#include <semaphore.h>

#define JOBS 100 // define JOBS constant
#define WORK_PER_JOB 1000 // define WORK_PER_JOB constant

int shared_resource = 0;

dispatch_function_t increment(int *countp);
void increment_wrapper(void *arg);

int main(int argc, char *argv[]) {
   int counter = 0;
   struct timespec begin, end;

   // Set C locale settings to get niceties like thousands separators
   // for decimal numbers.
   setlocale(LC_NUMERIC, "");

   if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &begin) != 0) {
       err(-1, "Failed to get start time");
   }

   dispatch_queue_t que = dispatch_queue_create(NULL, NULL);
   dispatch_group_t groups = dispatch_group_create();

   for (int i = 0; i < JOBS; i++) {
       dispatch_group_async_f(groups, que, &counter, increment_wrapper);
   }

   dispatch_group_wait(groups, DISPATCH_TIME_FOREVER);
```

```
    dispatch_release(groups);

    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end) != 0) {
        err(-1, "Failed to get end time");
    }

    long diff = end.tv_nsec - begin.tv_nsec;
    diff += (1000 * 1000 * 1000) * (end.tv_sec - begin.tv_sec);
    printf("%f\n", ((double) diff));

    return 0;
}

dispatch_function_t increment(int *countp) {
    for (int i = 0; i < WORK_PER_JOB; i++) {
        (*countp)++;
    }
}

void increment_wrapper(void *arg) {
    increment((int *)arg);
}
```

Bash script to create csv of jobs vs average time:

```
#!/bin/bash

JOBS_RANGE="1 2 4 8 16 32 64 128 256 512 1024"
WORK_PER_JOB=1000

# Remove any existing data file
rm -f jobs_vs_avg_time.csv

# Run the program with different values of JOBS
for jobs in $JOBS_RANGE; do
    make clean && make JOBS=$jobs WORK_PER_JOB=$WORK_PER_JOB
    result=$(./libdispatch)
    echo "$jobs,$result" >> jobs_vs_avg_time.csv
done
```

Bash script to create csv of work per job vs average time:

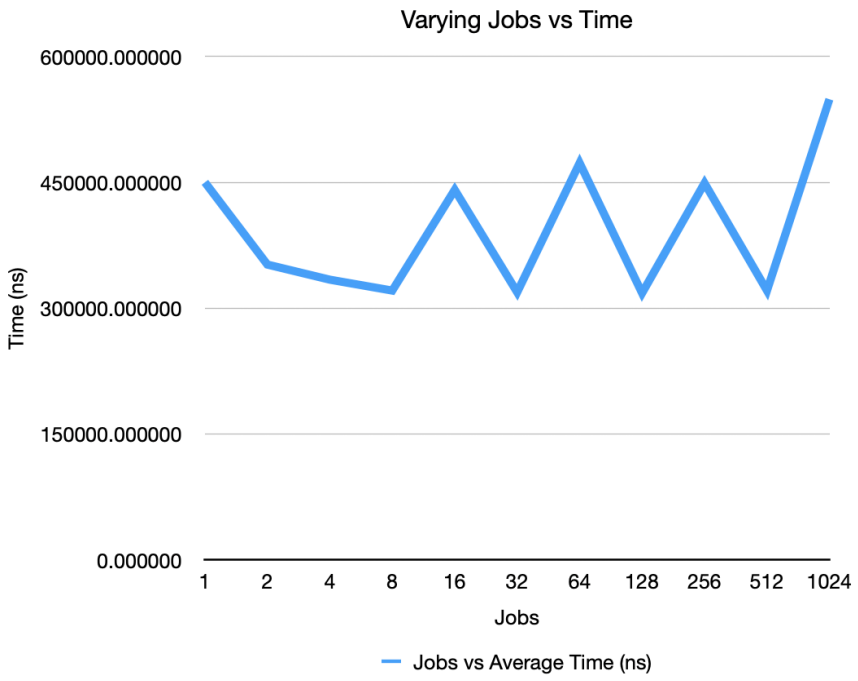
```
#!/bin/bash

JOBS=4
WORK_PER_JOB_RANGE="1 2 4 8 16 32 64 128 256 512 1024"

# Remove any existing data file
rm -f work_per_job_vs_avg_time.csv

# Run the program with different values of WORK_PER_JOB
for work_per_job in $WORK_PER_JOB_RANGE; do
   make clean && make JOBS=$JOBS WORK_PER_JOB=$work_per_job
   result=$(./libdispatch)
   echo "$work_per_job,$result" >> work_per_job_vs_avg_time.csv
done
```

Bash script to create csv of throughput vs threads:

```
#!/bin/bash

# Change the following line to the desired range of values
THREADS_RANGE="1 2 4 8 16 32 64 128"
WORK_PER_JOB=1000

# Remove any existing data file
rm -f throughput_vs_threads.csv

# Run the program with different values of threads
for threads in $THREADS_RANGE; do
   make clean && make JOBS=$threads WORK_PER_JOB=$WORK_PER_JOB
   result=$(./libdispatch)
   throughput=$(echo "scale=5; $WORK_PER_JOB / $result" | bc)
   echo "$threads,$throughput" >> throughput_vs_threads.csv
done
```

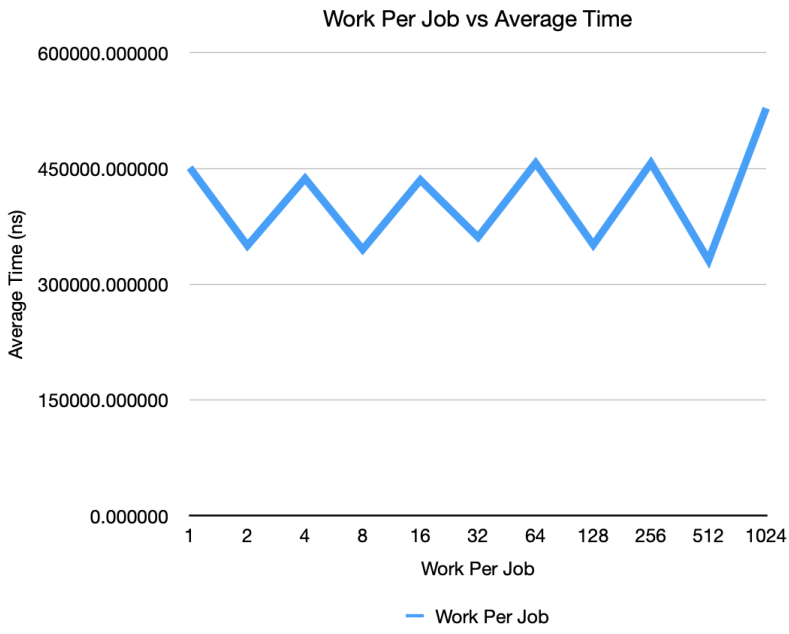
Jobs vs Average Time

Jobs	Average Time (ns)
1	450000.000000
2	352000.000000
4	334000.000000
8	321000.000000
16	441000.000000
32	319000.000000
64	473000.000000
128	318000.000000
256	449000.000000
512	321000.000000
1024	549000.000000



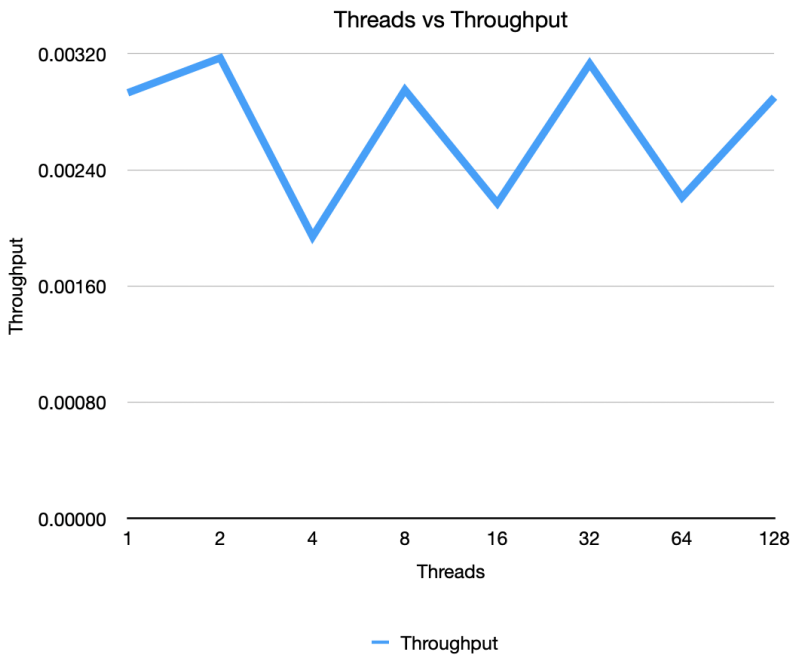
Work Per Job vs Time

Work Per Job	Time
1	451000.000000
2	350000.000000
4	437000.000000
8	345000.000000
16	435000.000000
32	361000.000000
64	457000.000000
128	351000.000000
256	457000.000000
512	331000.000000
1024	528000.000000



Threads vs Throughput

Threads	Throughput
1	0.00293
2	0.00317
4	0.00194
8	0.00295
16	0.00217
32	0.00313
64	0.00221
128	0.00290



Race Conditions:

1. Modified serial.c:

```c
#include <err.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <dispatch/dispatch.h>

void increment(int *countp);

int main(int argc, char *argv[]) {
   int counter = 0;
   struct timespec begin, end;

   setlocale(LC_NUMERIC, "");

   if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &begin) != 0) {
       err(-1, "Failed to get start time");
   }

   for (int i = 0; i < JOBS; i++) {
       increment(&counter);
```

```
    }

    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end) != 0) {
        err(-1, "Failed to get end time");
    }

    long diff = end.tv_nsec - begin.tv_nsec;
    diff += (1000 * 1000 * 1000) * (end.tv_sec - begin.tv_sec);
    printf("Counted to %'d in %'ld ns: %f ns/iter\n", counter, diff, ((double)
diff) / counter);

    return 0;
}

void increment(int *countp) {
    for (int i = 0; i < WORK_PER_JOB; i++) {
        (*countp)++;
    }
}
```

Modified pthread.c:

```
#include <err.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>

#define JOBS 100
#define WORK_PER_JOB 1000

void* worker(void* arg);
void increment(int *countp);

int counter = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int main(int argc, char *argv[])
{
    struct timespec begin, end;
    setlocale(LC_NUMERIC, "");

    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &begin) != 0)
    {
        err(-1, "Failed to get start time");
    }

    pthread_t threads[JOBS];
```

```
    for (int i = 0; i < JOBS; i++)
    {
        int* arg = malloc(sizeof(int));
        *arg = i;
        pthread_create(&threads[i], NULL, worker, arg);
    }

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

    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end) != 0)
    {
        err(-1, "Failed to get end time");
    }

    long diff = end.tv_nsec - begin.tv_nsec;
    diff += (1000 * 1000 * 1000) * (end.tv_sec - begin.tv_sec);

    printf("Counter value: %'d\n", counter);

    return 0;
}

void* worker(void* arg)
{
    int job = *(int*)arg;
    free(arg);

    int work = WORK_PER_JOB;
    if (job == JOBS - 1)
    {
        work += (JOBS * WORK_PER_JOB) % JOBS;
    }

    for (int i = 0; i < work; i++)
    {
        pthread_mutex_lock(&mutex);
        increment(&counter);
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}

void increment(int *countp)
{
```

```
    (*countp)++;
}
```

Modified libdispatch.c:

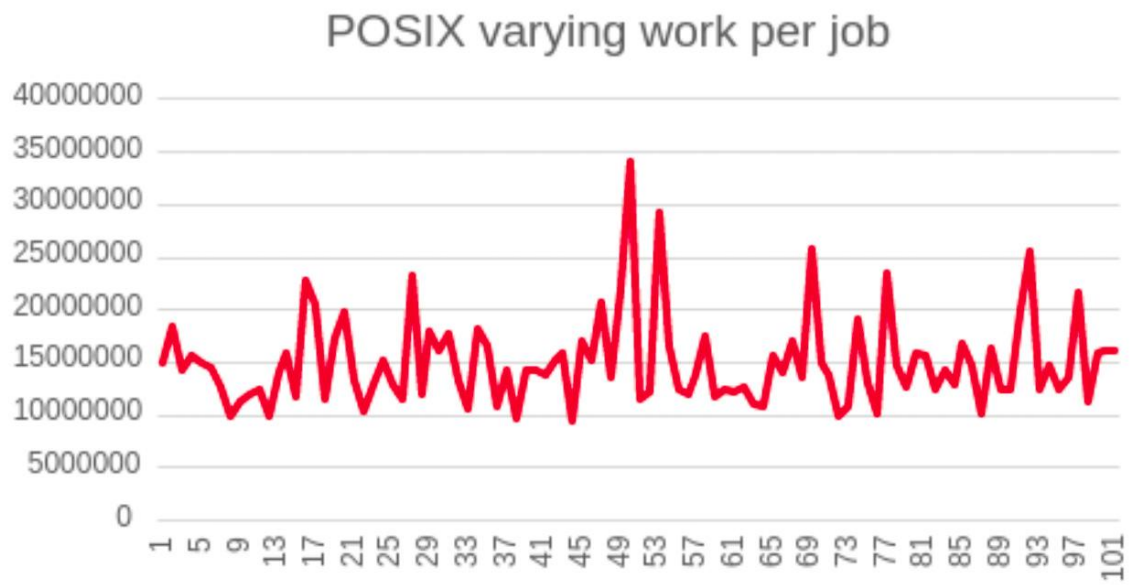
```
#include <err.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <dispatch/dispatch.h>
#include <semaphore.h>

#define JOBS 100 // define JOBS constant
#define WORK_PER_JOB 1000 // define WORK_PER_JOB constant

int shared_resource = 0;

dispatch_function_t increment(int *countp);
void increment_wrapper(void *arg);

int main(int argc, char *argv[]) {
    int counter = 0;
    struct timespec begin, end;

    // Set C locale settings to get niceties like thousands separators
    // for decimal numbers.
    setlocale(LC_NUMERIC, "");

    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &begin) != 0) {
        err(-1, "Failed to get start time");
    }

    dispatch_queue_t que = dispatch_queue_create(NULL, NULL);
    dispatch_group_t groups = dispatch_group_create();

    for (int i = 0; i < JOBS; i++) {
        dispatch_group_async_f(groups, que, &counter, increment_wrapper);
    }

    dispatch_group_wait(groups, DISPATCH_TIME_FOREVER);
    dispatch_release(groups);

    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end) != 0) {
        err(-1, "Failed to get end time");
    }

    long diff = end.tv_nsec - begin.tv_nsec;
    diff += (1000 * 1000 * 1000) * (end.tv_sec - begin.tv_sec);
```

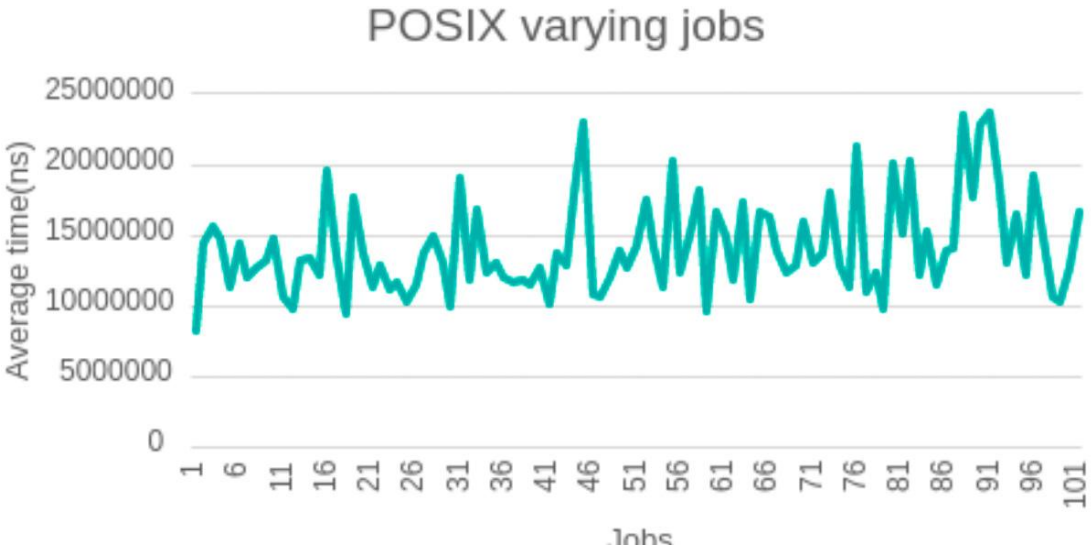
```
    printf("Time elapsed: %f\n", ((double) diff));
    printf("Counter value: %d\n", counter);

    return 0;
}

dispatch_function_t increment(int *countp) {
    for (int i = 0; i < WORK_PER_JOB; i++) {
        (*countp)++;
    }
}

void increment_wrapper(void *arg) {
    increment((int *)arg);
}
```

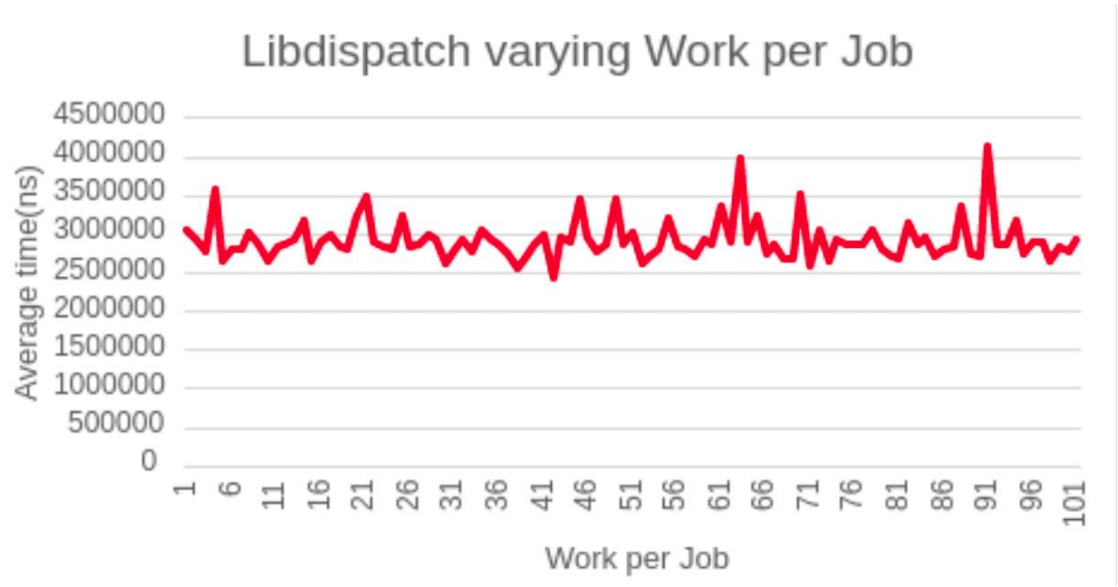
Output:

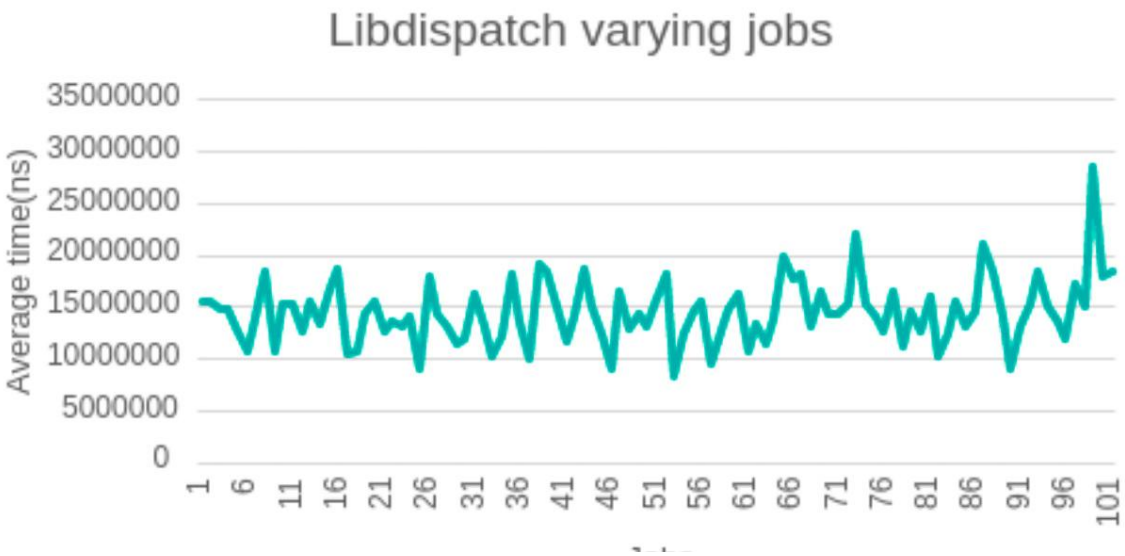
```
./pm_serial
Counter : 100
./pm_pthreads
Counter : 100
./pm_libdispatch
Counter : 100
```

2.


POSIX varying work per job
40000000
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
1
5
9
13
17
21
25
29
33
37
41
45
49
53
57
61
65
69
73
77
81
85
89
93
97
101


POSIX varying jobs
Average time(ns)
25000000
20000000
15000000
10000000
5000000
0
1
6
11
16
21
26
31
36
41
46
51
56
61
66
71
76
81
86
91
96
101
Jobs





3. Modified serial.c:

```c
#include <err.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <semaphore.h>

sem_t my_semaphore;
int shared_resource = 0;
void increment(int * countp);
```

```
int main(int argc, char * argv[]) {
   int counter = 0;
   struct timespec begin, end;
   // Set C locale settings to get niceties like thousands separators // for
decimal numbers.
   setlocale(LC_NUMERIC, "");
   // initialize semaphore
   if (sem_init( & my_semaphore, 0, 1) != 0) {
       err(-1, "Failed to initialize semaphore");
   }
   if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, & begin) != 0) {
       err(-1, "Failed to get start time");
   }
   for (int i = 0; i < JOBS; i++) {
       increment( & counter);
   }
   if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, & end) != 0) {
       err(-1, "Failed to get end time");
   }
   long diff = end.tv_nsec - begin.tv_nsec;
   diff += (1000 * 1000 * 1000) * (end.tv_sec - begin.tv_sec);
   printf("Counter: %'d\n", counter);
   return 0;
}

void increment(int * countp) {
   for (int i = 0; i < WORK_PER_JOB; i++) {
       sem_wait( & my_semaphore);
       ( * countp) ++;
       sem_post( & my_semaphore);
   }
}
```

Modified pthread:

```
#include <err.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>

#define JOBS 100 // define JOBS constant
#define WORK_PER_JOB 1000 // define WORK_PER_JOB constant
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void * increment(void * countp);
int main(int argc, char * argv[]) {
   int counter = 0;
   struct timespec begin, end;
```

```
    // Set C locale settings to get niceties like thousands separators
    // for decimal numbers.
    setlocale(LC_NUMERIC, "");
    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, & begin) != 0) {
        err(-1, "Failed to get start time");
    }
    pthread_t threads[JOBS];
    int rc;
    for (int i = 0; i < JOBS; i++) {
        rc = pthread_create( & threads[i], NULL, increment, & counter);
        if (rc != 0) {
            err(-1, "Failed to create thread");
        }
    }
    for (int i = 0; i < JOBS; i++) {
        rc = pthread_join(threads[i], NULL);
        if (rc != 0) {
            err(-1, "Failed to join thread");
        }
    }
    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, & end) != 0) {
        err(-1, "Failed to get end time");
    }
    long diff = end.tv_nsec - begin.tv_nsec;
    diff += (1000 * 1000 * 1000) * (end.tv_sec - begin.tv_sec);
    printf("Counter: %ld\n", counter);
    return 0;
}
void * increment(void * countp) {
    for (int i = 0; i < WORK_PER_JOB; i++) {

        pthread_mutex_lock( & mutex);
        ( * ((int * ) countp)) ++;
        pthread_mutex_unlock( & mutex);
    }
    return NULL;
}
```

Modified libdispatch.c:

```
#include <err.h>

#include <locale.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>
```

```c
#include <dispatch/dispatch.h>

#include <semaphore.h>

#define JOBS 100 // define JOBS constant
#define WORK_PER_JOB 1000 // define WORK_PER_JOB constant
sem_t my_semaphore;
int shared_resource = 0;
dispatch_function_t increment(int * countp);
int main(int argc, char * argv[]) {
   int counter = 0;
   struct timespec begin, end;
   // Set C locale settings to get niceties like thousands separators // for
decimal numbers.
   setlocale(LC_NUMERIC, "");
   // initialize semaphore
   if (sem_init( & my_semaphore, 0, 1) != 0) {
       err(-1, "Failed to initialize semaphore");
   }
   if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, & begin) != 0) {
       err(-1, "Failed to get start time");
   }
   dispatch_queue_t que = dispatch_queue_create(NULL, NULL);
   dispatch_group_t groups = dispatch_group_create();
   for (int i = 0; i < JOBS; i++) {
       dispatch_group_async_f(groups, que, & counter, increment( & counter));
   }
   dispatch_group_wait(groups, DISPATCH_TIME_FOREVER);
   dispatch_release(groups);
   if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, & end) != 0) {
       err(-1, "Failed to get end time");

   }
   // release semaphore
   sem_destroy( & my_semaphore);
   long diff = end.tv_nsec - begin.tv_nsec;
   diff += (1000 * 1000 * 1000) * (end.tv_sec - begin.tv_sec);
   printf("%f\n", ((double) diff));
   return 0;
}
dispatch_function_t increment(int * countp) {
   for (int i = 0; i < WORK_PER_JOB; i++) {
       sem_wait( & my_semaphore);
       ( * countp) ++;
       sem_post( & my_semaphore);
   }
}
```

Output:

```
./m_serial
Counter: 100
./m_pthread
Counter: 100000
./m_libdispatch
Counter: 100,000
```

Makefile:

```
BINARIES= \
      serial \
      pthreads \
      libdispatch \
      pm_serial \
      pm_pthread \
      m_libdispatch \
      m_serial \
    m_pthread \
    m_libdispatch \

JOBS?=            10
WORK_PER_JOB?=    10

CC=         clang
CFLAGS=           -D JOBS=${JOBS} -D WORK_PER_JOB=${WORK_PER_JOB} \
            -Weverything -Wno-unused-parameter

# We don't need to link all of the following libraries for every program
that
# we're going to compile, but there's no harm in attempting to do so
# (the linker will ignore any code it isn't looking for):
LDFLAGS= -pthread

all: ${BINARIES}

clean:
      rm -f ${BINARIES}

serial: serial.c
      ${CC} ${CFLAGS} -o $@ $^ ${LDFLAGS}
```

```
libdispatch: libdispatch.c
	${CC} ${CFLAGS} -o $@ $^ ${LDFLAGS}
```