Multi-threading using Thread Pool Pattern

Dr. Suhail Yousaf

Intelligent Systems Design Lab,
National Center of Artificial Intelligence, Pakistan

syousaf@uetpeshawar.edu.pk

December 11, 2022

Outline

1. The Problem
2. The Solution Design
3. The Example Code
4. Conclusion

Outline

1. The Problem
2. The Solution Design
3. The Example Code
4. Conclusion

The Problem

factorize – a multi-threaded program

- Factorizes natural numbers concurrently
- Reads numbers from standard input
- Writes result to the standard output
- Use thread pool pattern: main and workers
- Number of workers as command line argument $-t$
- Input numbers space: unsigned 64-bit integers

Development tools

- The **C** language
- POSIX pthread library

Sample invocation - I

```
$ cat <<_end_ | ./factorize -t 5
> a 12
> b 234
> _end_
a 12 = 2 2 3
b 234 = 2 3 3 13
```

Number **tag**

- Every number is prefixed with a tag
- Result to request matching
- Tag content is arbitrary
- No white space and control characters

Sample invocation - II

```
$ cat <<_end_ | ./factorize -t 5
> a 18446744
> b 18446744
> c 18446744
> _end_
c 18446744 = 2 2 2 349 6607
a 18446744 = 2 2 2 349 6607
b 18446744 = 2 2 2 349 6607
```

Observation

- The same number is factorized multiple times concurrently
- The results appear in different order on the standard output

Detailed requirements

Main thread

- Reads numbers from standard input
- Places the numbers into a queue
- Receives results from each worker
- Writes result to the standard output

Worker thread

- Picks up a number from the queue
- Computes factors for the picked number
- Passes the result back to the main thread

Program exit condition

- End of standard input **and** no busy threads anymore

Outline

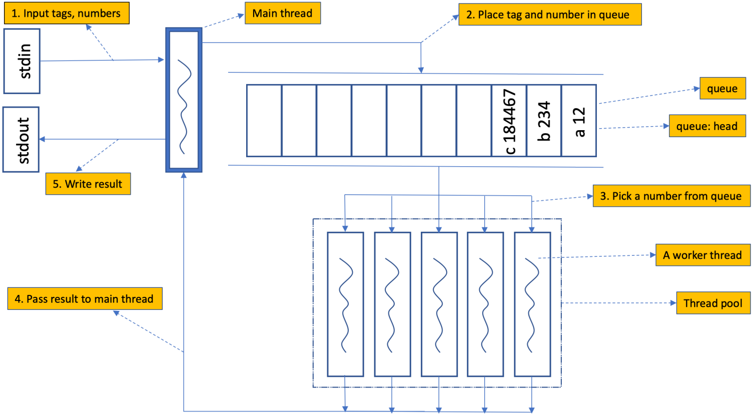
1 The Problem

2 The Solution Design

3 The Example Code

4 Conclusion

Architectural Design



Design considerations

The Queue

- The queue is accessed by multiple threads – Mutual exclusion: mutex
- What if the queue is empty – Condition variable: full
- What if the queue is full – Condition variable: empty
- How to detect the end of standard input?

The factorize function

- The seive of Eratosthenes method

Outline

1 The Problem

2 The Solution Design

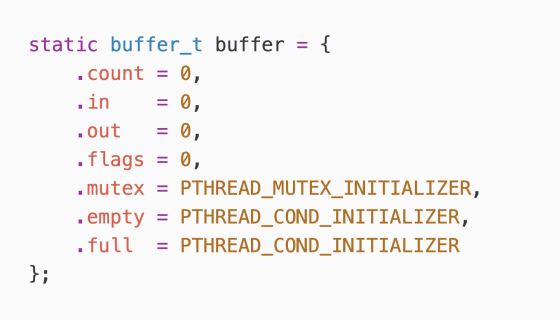
3 The Example Code

4 Conclusion

The data structures (buffer_t alias queue)

```
typedef struct {
    unsigned int    count;
    char            tag[BUFFER_SIZE][TAG_SIZE];
    unsigned long   data[BUFFER_SIZE];
    int             in;
    int             out;
    int             flags;
    pthread_mutex_t mutex;
    pthread_cond_t  empty;
    pthread_cond_t  full;
} buffer_t;
```

The data structures (queue initialization)



```c
static buffer_t buffer = {
    .count = 0,
    .in    = 0,
    .out   = 0,
    .flags = 0,
    .mutex = PTHREAD_MUTEX_INITIALIZER,
    .empty = PTHREAD_COND_INITIALIZER,
    .full  = PTHREAD_COND_INITIALIZER
};
```

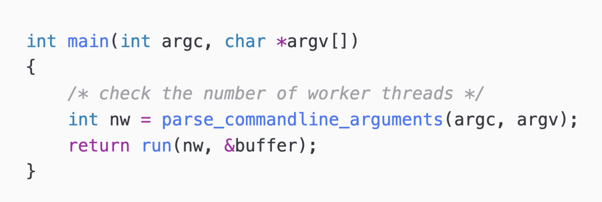
The data structures (completed task, thread_yield)

```
typedef struct {
    char* tag;
    unsigned long int nat_num;
    unsigned long int* factors;
    int count;
} completed_task_t;

typedef struct {
    completed_task_t** completed_tasks_list;
    int count; /* completed tasks by a thread */
} thread_yield_t;
```

The function prototypes

```c
/* function prototypes */
int main(int, char **);

static int parse_commandline_arguments(int, char**);
static int run(int, buffer_t*);

static int create_thread_pool(int, pthread_t**, buffer_t*);
static void read_numbers_from_stdin(buffer_t*);
static int thread_join(int, pthread_t*, thread_yield_t**);
static void write_to_standard_output(int, thread_yield_t**);
static void free_heap_memory(int, pthread_t*, thread_yield_t**);

static void enqueue(buffer_t*, char*, unsigned long);
static int dequeue(buffer_t*, char**, unsigned long**);

static void* worker(void*);
static completed_task_t* compute_factors(unsigned long);
```

The main(...) function



```
int main(int argc, char *argv[])
{
    /* check the number of worker threads */
    int nw = parse_commandline_arguments(argc, argv);
    return run(nw, &buffer);
}
```

The parse_commandline_arguments(...) function

```c
/* parse command line arguments */
static int parse_commandline_arguments(int argc, char *argv[])
{
    int c;
    int nw = 3;
    while ((c = getopt(argc, argv, "t:h")) >= 0) {
        switch (c) {
            case 't':
                if ((nw = atoi(optarg)) <= 0) {
                    fprintf(stderr, "Number of worker threads must be > 0\n");
                    exit(EXIT_FAILURE);
                }
                break;
            case 'h':
                printf("Usage: %s [-t num-worker] [-h]\n", progname);
                exit(EXIT_SUCCESS);
        }
    }
    return nw;
}
```

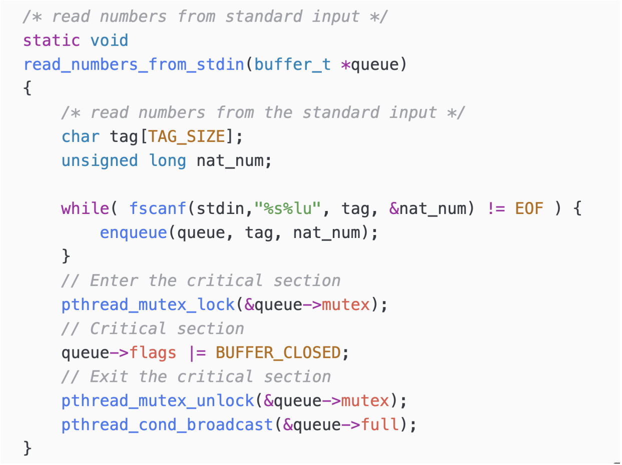
The run(...) function

```c
static int
run(int nw, buffer_t *queue)
{
    pthread_t* thread;
    thread_yield_t* yield_of_thread[nw];
    //
    if(create_thread_pool(nw, &thread, queue) == EXIT_FAILURE) return EXIT_FAILURE;
    // take input from stdin
    read_numbers_from_stdin(queue); // take input from stdin and populate the queue
    // thread join
    if(thread_join(nw, thread, (thread_yield_t**)&yield_of_thread) == EXIT_FAILURE)
        return EXIT_FAILURE;;
    // display output
    write_to_standard_output(nw, (thread_yield_t**)&yield_of_thread);
    // release dynamically acquired memory
    free_heap_memory(nw, thread, (thread_yield_t**)&yield_of_thread);
    //
    return EXIT_SUCCESS;
}
```

The create_thread_pool(...) function

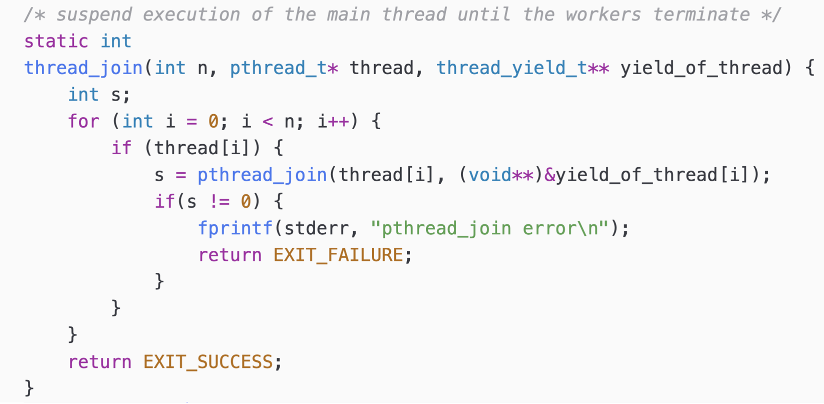
```c
static int create_thread_pool(int n, pthread_t** th, buffer_t *queue)
{
    int err;
    long int x;
    *th = (pthread_t*)malloc(sizeof(pthread_t)*n);
    pthread_t* thread = *th;
    char buffer[128], *end;

    assert(queue && n > 0);

    for (int i = 0; i < n; i++) {
        err = pthread_create(&thread[i], NULL, worker, queue);
        if (err) {
            fprintf(stderr, "%s: %s: unable to create thread %d: %d\n",
                progname, __func__, i, err);
            return EXIT_FAILURE;
        }
    }
    return EXIT_SUCCESS;
}
```

The read_numbers_from_stdin(...) function



```c
/* read numbers from standard input */
static void
read_numbers_from_stdin(buffer_t *queue)
{
    /* read numbers from the standard input */
    char tag[TAG_SIZE];
    unsigned long nat_num;

    while( fscanf(stdin,"%s%lu", tag, &nat_num) != EOF ) {
        enqueue(queue, tag, nat_num);
    }
    // Enter the critical section
    pthread_mutex_lock(&queue->mutex);
    // Critical section
    queue->flags |= BUFFER_CLOSED;
    // Exit the critical section
    pthread_mutex_unlock(&queue->mutex);
    pthread_cond_broadcast(&queue->full);
}
```

The thread_join(...) function



```
/* suspend execution of the main thread until the workers terminate */
static int
thread_join(int n, pthread_t* thread, thread_yield_t** yield_of_thread) {
    int s;
    for (int i = 0; i < n; i++) {
        if (thread[i]) {
            s = pthread_join(thread[i], (void**)&yield_of_thread[i]);
            if(s != 0) {
                fprintf(stderr, "pthread_join error\n");
                return EXIT_FAILURE;
            }
        }
    }
    return EXIT_SUCCESS;
}
```

The write_to_standard_output(...) function

```
/* display the output */
static void
write_to_standard_output(int nw, thread_yield_t** yield_of_thread)
{
    //
    for (int i = 0; i < nw; i++) {
        for(int j=0; j<yield_of_thread[i]->count; j++) {
            fprintf(stderr, "%s ", yield_of_thread[i]->completed_tasks_list[j]->tag);
            for(int k=0; k<yield_of_thread[i]->completed_tasks_list[j]->count; k++) {
                fprintf(stderr, "%lu ", yield_of_thread[i]->
                    completed_tasks_list[j]->factors[k]);
            }
            _NEW_LINE_
        }
    }
}
```

The free_heap_memory(...) function

```c
static void
free_heap_memory(int nw, pthread_t* thread, thread_yield_t** yield_of_thread) {
    /* free tag(s), factors array(s), completed_tasks_list array(s),
       yield_of_thread array(s), and thread array(s) */
    for (int i = 0; i < nw; i++) {
        for(int j=0; j<yield_of_thread[i]->count; j++) {
            free(yield_of_thread[i]->completed_tasks_list[j]->tag);
            yield_of_thread[i]->completed_tasks_list[j]->tag = NULL;
            free(yield_of_thread[i]->completed_tasks_list[j]->factors);
            yield_of_thread[i]->completed_tasks_list[j]->factors = NULL;
            free(yield_of_thread[i]->completed_tasks_list[j]); //
        }
        free(yield_of_thread[i]->completed_tasks_list);
        free(yield_of_thread[i]);
        yield_of_thread[i] = NULL;
    }
    free(thread); thread = NULL;
}
```

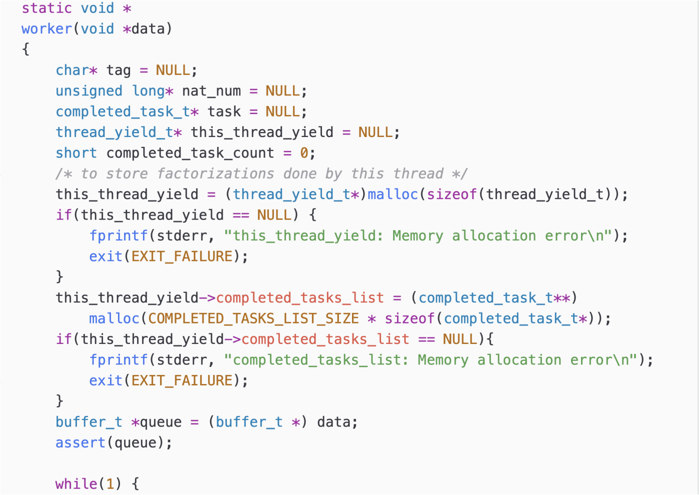
The enqueue(...) function

```c
/* place a (tag, number) pair in queue */
static void
enqueue(buffer_t *queue, char* tag, unsigned long nat_num)
{
    // Entr critical section
    pthread_mutex_lock(&queue->mutex);
    // Critical section
    while (queue->count == BUFFER_SIZE) {
        pthread_cond_wait(&queue->empty, &queue->mutex);
    }

    sprintf(queue->tag[queue->in], "%s", tag);
    queue->data[queue->in] = nat_num;
    queue->in = (queue->in + 1) % BUFFER_SIZE;
    queue->count++;

    pthread_cond_signal(&queue->full);
    // Exit critical section
    pthread_mutex_unlock(&queue->mutex);
}
```

The dequeue(...) function

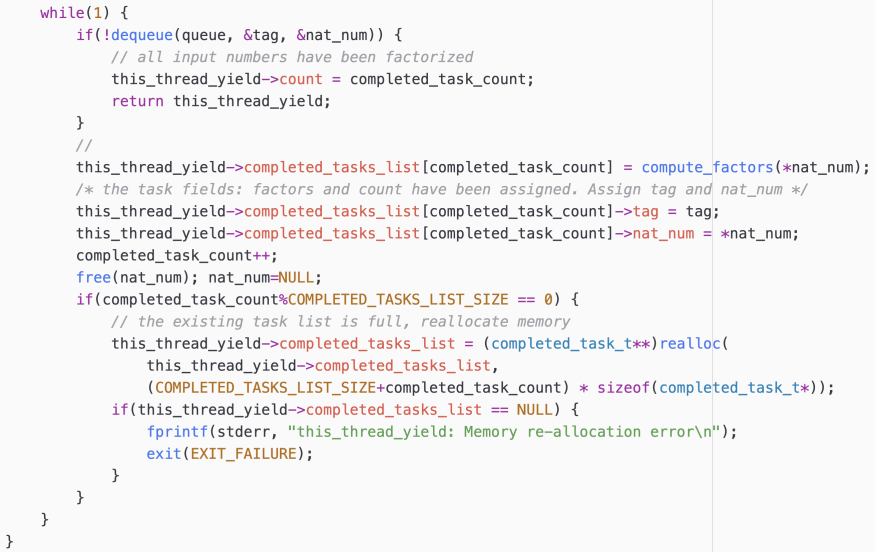
```c
/* pick a number from queue */
static int dequeue(buffer_t *queue, char** tag, unsigned long** nat_num)
{
    short tag_len = strlen(queue->tag[queue->out])+1;
    *tag = (char*)malloc(tag_len*sizeof(char));
    *nat_num = (unsigned long*)malloc(sizeof(unsigned long));
    if(tag==NULL || nat_num==NULL) exit(EXIT_FAILURE);
    /* Enter the critical section */
    pthread_mutex_lock(&queue->mutex);
    while (queue->count == 0) {
        if (queue->flags & BUFFER_CLOSED) {
            pthread_mutex_unlock(&queue->mutex);
            return DEQUEUE_FAILURE;
        }
        pthread_cond_wait(&queue->full, &queue->mutex);
    }
    //
    sprintf(*tag, "%s", queue->tag[queue->out]);
    **nat_num = queue->data[queue->out];
    queue->out = (queue->out + 1) % BUFFER_SIZE;
    queue->count--;

    pthread_cond_signal(&queue->empty);
    // Exit the critical section
    pthread_mutex_unlock(&queue->mutex);
    return DEQUEUE_SUCCESS;
}
```

The worker(...) function – part-I



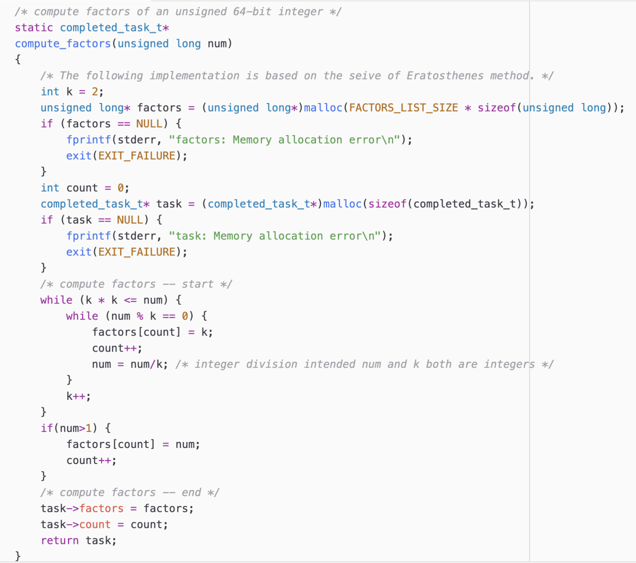
```
static void *
worker(void *data)
{
    char* tag = NULL;
    unsigned long* nat_num = NULL;
    completed_task_t* task = NULL;
    thread_yield_t* this_thread_yield = NULL;
    short completed_task_count = 0;
    /* to store factorizations done by this thread */
    this_thread_yield = (thread_yield_t*)malloc(sizeof(thread_yield_t));
    if(this_thread_yield == NULL) {
        fprintf(stderr, "this_thread_yield: Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    this_thread_yield->completed_tasks_list = (completed_task_t**)
        malloc(COMPLETED_TASKS_LIST_SIZE * sizeof(completed_task_t*));
    if(this_thread_yield->completed_tasks_list == NULL){
        fprintf(stderr, "completed_tasks_list: Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    buffer_t *queue = (buffer_t *) data;
    assert(queue);

    while(1) {
```

The worker(...) function – part-II



```c
    while(1) {
        if(!dequeue(queue, &tag, &nat_num)) {
            // all input numbers have been factorized
            this_thread_yield->count = completed_task_count;
            return this_thread_yield;
        }
        //
        this_thread_yield->completed_tasks_list[completed_task_count] = compute_factors(*nat_num);
        /* the task fields: factors and count have been assigned. Assign tag and nat_num */
        this_thread_yield->completed_tasks_list[completed_task_count]->tag = tag;
        this_thread_yield->completed_tasks_list[completed_task_count]->nat_num = *nat_num;
        completed_task_count++;
        free(nat_num); nat_num=NULL;
        if(completed_task_count%COMPLETED_TASKS_LIST_SIZE == 0) {
            // the existing task list is full, reallocate memory
            this_thread_yield->completed_tasks_list = (completed_task_t**)realloc(
                this_thread_yield->completed_tasks_list,
                (COMPLETED_TASKS_LIST_SIZE+completed_task_count) * sizeof(completed_task_t*));
            if(this_thread_yield->completed_tasks_list == NULL) {
                fprintf(stderr, "this_thread_yield: Memory re-allocation error\n");
                exit(EXIT_FAILURE);
            }
        }
    }
}
```

The compute_factors(...) function



```c
/* compute factors of an unsigned 64-bit integer */
static completed_task_t*
compute_factors(unsigned long num)
{
    /* The following implementation is based on the seive of Eratosthenes method. */
    int k = 2;
    unsigned long* factors = (unsigned long*)malloc(FACTORS_LIST_SIZE * sizeof(unsigned long));
    if (factors == NULL) {
        fprintf(stderr, "factors: Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    int count = 0;
    completed_task_t* task = (completed_task_t*)malloc(sizeof(completed_task_t));
    if (task == NULL) {
        fprintf(stderr, "task: Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    /* compute factors -- start */
    while (k * k <= num) {
        while (num % k == 0) {
            factors[count] = k;
            count++;
            num = num/k; /* integer division intended num and k both are integers */
        }
        k++;
    }
    if(num>1) {
        factors[count] = num;
        count++;
    }
    /* compute factors -- end */
    task->factors = factors;
    task->count = count;
    return task;
}
```

Outline

1 The Problem

2 The Solution Design

3 The Example Code

4 Conclusion

Concluding Remarks

factorize – a multi-threaded program

- The use of thread pool pattern
- Thread synchronization primitives in **C**
  - POSIX Mutex Locks
  - POSIX Condition Variables
- Indicating end of standard input to the threads

A design flaw

- Larger **completed_tasks_list** with larger input (memory inefficient)
- **Solution:** maintain an output queue
- **Simplicity** (for demonstration) vs. Efficiency trade-off

Acknowledgment

The source code has been adapted from the basic **thread pool pattern** implementation available at Jacobs University source code repository for the course: Operating Systems. Although, significant changes have been made to the original source code to solve the given problem, the author of the program **factorize.c** duly acknowledges the contribution of the author(s) of the program **worker.c** available on the given link.