

OPERATING SYSTEMS AND SYSTEMS PROGRAMMING 1 CPS1012

Assignment documentation
– TID manager

Andy Debrincat 0135705L

OVERVIEW

For each Task there will be a brief explanation about how each subsection of the task was implemented.

Then the testing for each task is explained and results shown. Any bugs will be reported in the testing section after each task, if there are any.

The testing is done either through a test class or through the CLI by entering varying inputs to the program.

Note also, that each task has its own executable and task 1 also has its own test class.

All of these can be compiled by the MAKEFILE, so long as everything is in the same directory.

Task 1

- a) The data structure chosen for the list of available TID's is an array.

The data structure was implemented through 2 structures, one of them being the structure representing a TID called `tid` and the other representing the array called `tidMap`.

The `tid` structure contains a TID number from 300 to 5000 (included) and a boolean value to indicate whether it is used or not.

The `tidMap` contains the pointer `tidList` which acts as an array and an integer called `free_ptr` that is an offset for the array in which there will be a free `tid`, and hence it acts like a pointer. If there is no free `tid` then it will be set to -1.

Below is the code for these structures.

```
typedef struct{
    int tid;
    bool in_use;
}tid;
typedef struct{
    tid *tidList;
    int free_ptr;
}tidMap;

tidMap * tidMap_ptr = NULL;
```

- b) Note that a global variable called `tidMap_ptr` was created and it is a pointer of type `tidMap`. This variable will be the array of `tid`'s that will be the shared resource throughout the assignment.

The function `allocate_map()` takes in no argument and simply assigns memory on the heap for the array through `malloc()` for $(5000-300 + 1)$ locations of type `tid`. Then sets all the elements in the array to their appropriate number (position + 300) and all their Boolean elements to false since none of them are in use. In addition to this the free pointer is made to point to the first element in the array since it is not in use.

As requested, the function will return 1 if successful and -1 if memory allocation failed.

Below is the code for the function :

```

int allocate_map(){

    tidMap_ptr = (tidMap *)malloc(sizeof(tidMap));

    if(tidMap_ptr == NULL){
        return -1;
    }

    tidMap_ptr->free_ptr = 0;
    tidMap_ptr->tidList = (tid *)malloc((MAXTID - MINTID + 1)*sizeof(tid));

    if(tidMap_ptr->tidList == NULL){
        return -1;
    }

    for(int i = 0; i <= MAXTID - MINTID; i++){
        tidMap_ptr->tidList[i].tid = MINTID + i;
        tidMap_ptr->tidList[i].in_use = false;
    }
    return 1;
}

```

c) The function `allocate_tid()` works as follows :

First it checks the free pointer, if it is not -1 then the `tid` in the location indicated by the `free_ptr` is returned, else the program returns -1 because all `tid`'s are taken.

Then it loops through the `tidMap` in a circular fashion starting from the old free pointer.

Then it sets the free pointer to the first free `tid` it finds.

If no `tid` is found, the free pointer is set to -1.

This was done in this way since most of the time the closest free `tid` is next to the latest free `tid` and will not have to loop through the whole structure to find one.

Note that instead of using the modulo operator, two counters were kept, where one checks the total number of iterations and the other keeps the current location that is being read in the array, if the second counter is 5000 then it is set back to 300 to start again. When the first counter reaches (5000-300) then the whole array has been looped over.

Below is the code for this function:

```
int allocate_tid(){
    int ret = 0;
    if(tidMap_ptr->free_ptr == -1){
        return -1;
    }
    else{
        ret = tidMap_ptr->tidList[tidMap_ptr->free_ptr].tid;
        tidMap_ptr->tidList[tidMap_ptr->free_ptr].in_use = true;
    }

    int ct_1 = tidMap_ptr->free_ptr;
    int ct_2 = 0;
    bool found = false;

    do{
        if(ct_1==MAXTID-MINTID){
            ct_1 = 0;
        }
        else{
            ct_1++;
        }
        ct_2++;

        if(tidMap_ptr->tidList[ct_1].in_use == false){
            tidMap_ptr->free_ptr = ct_1;
            found = true;
        }
    }
    while((ct_2<MAXTID-MINTID) && !found);

    if(!found){
        tidMap_ptr->free_ptr = -1;
    }
    return ret;
}
```

- d) The `release_tid(int tid)` function is very simple, and it just takes in a `tid`, and uses `(tid-300)` as an offset for the array and sets the Boolean value from true to false to show that it is free. In addition to this the free pointer is set to this location, so that it may maximise efficiency since probably more `tid`'s are free around that same location and the loop in part c) will terminate quicker.

Below is the code for part d:

```
void release_tid(int tid){
    if(tid>MAXTID || tid<MINTID){
        printf("Invalid tid\n");
        return;
    }
    tidMap_ptr->tidList[tid-MINTID].in_use = false;
    tidMap_ptr->free_ptr = tid-MINTID;
}
```

Testing:

Task 1 was tested as in [Task1_Test.c](#) as follows:

The [allocate_map\(\)](#) function is first called and checked to be successful.

Then in order to test the [allocate_tid](#) function, its boundary conditions were checked by calling the function for exactly 4702 (5000-300 +1 + 1) times, and all calls should be successful except the last one which should return -1. (Since both 5000 and 300 are included)

Then to check [release_tid\(\)](#), the function calls [release_tid\(\)](#) with values 299 till 5001 and the output should be all valid except for the first and last value. This was done to check that the boundary conditions are correct.

Then the program will again allocate 4701 [tid's](#) and all should be successful, since all the [tid's](#) were freed. This was done to check that the [tid's](#) were freed correctly.

Below is the testing code :

```
printf("This is a test\n");

if(allocate_map() == -1){
    printf( "Could not allocate a map\n");
    return -1;
};

printf("\n\n");
for(int i = 1; i <= 4702; i++){
    int tid = allocate_tid();
    if(tid==-1){
        printf("Could not allocate for iteration number %d\n", i);
    }
}
printf("\n\n");
```

```

for(int i = 299; i <= 5001; i++){
    release_tid(i);

}

printf("\n\n");

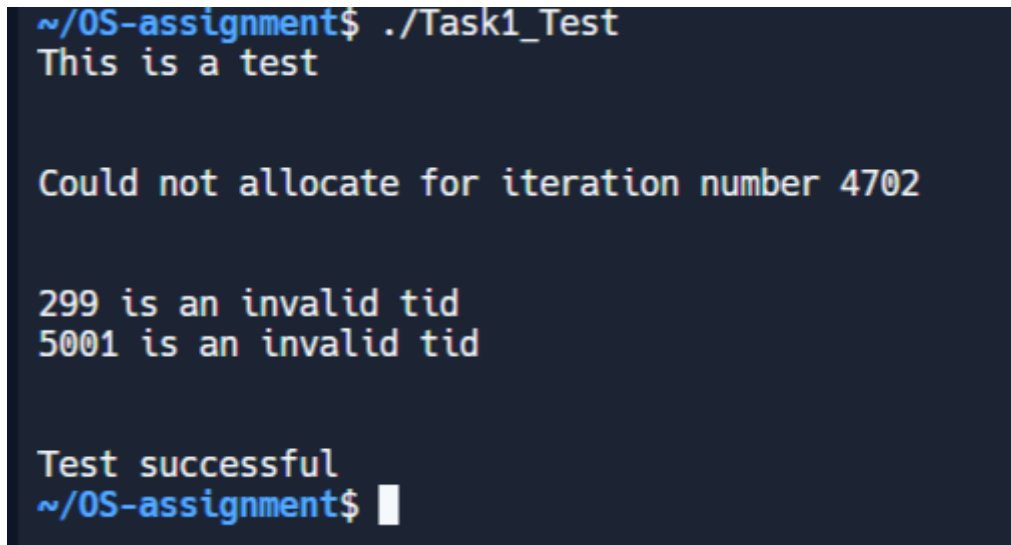
for(int i = 1; i <= 4701; i++){
    int tid = allocate_tid();
    if(tid == -1){
        printf("Could not allocate for iteration number %d\n", i);

    }
}
printf("Test successful\n");

```

The output was exactly as expected, and hence the code is functional.

Below is the output of the test class :



```

~/OS-assignment$ ./Task1_Test
This is a test

Could not allocate for iteration number 4702

299 is an invalid tid
5001 is an invalid tid

Test successful
~/OS-assignment$

```

Note that the code for task1 has no bugs that have been found.

TASK 2

- a) A simple main class was made that takes the requested command line arguments.

The program expects to receive the number of threads followed by the **min** sleep time followed by the **max**.

It receives any string but then through error checking, makes certain that the values are correct, if the values are not correct or within certain bounds then the program will terminate.

If no command line arguments are given, the default values will be applied.

If n = number of threads = 200, l = min sleep time = 1 and h = max sleep time = 5, then the program should be executed as follows :



```
~/OS-assignment$ ./Task2 200 1 5
```

If default values are desired, then simply enter no arguments.

The arguments were converted to integers using a function called **str_to_int** that returns -1 if the string is not a positive integer.

```
if (argc > 4) {
    printf("Too many arguments\n");
    return 0;
}
else if (argc == 4) {
    number_of_threads = str_to_int(argv[1]);
    min_sleep_time = str_to_int(argv[2]);
    max_sleep_time = str_to_int(argv[3]);
} else if (argc == 3) {
    number_of_threads = str_to_int(argv[1]);
    min_sleep_time = str_to_int(argv[2]);
} else if (argc == 2) {
    number_of_threads = str_to_int(argv[1]);
}
```

There was ample checking and validation of the command line arguments, but an example is that the min sleep time should be smaller than the max :

```
if (min_sleep_time >= max_sleep_time || max_sleep_time >= MAXSLEEPTIME) {
    printf("Invalid min or max sleep time\n");
    return 0;
}
```


- b) Using the `allocate_map` in task 1 was very simple, and it was done by putting task1 in a header file called `tid_map.h` and including it in task 2. Then the `allocate_map()` function was simply called.

```
if (allocate_map() == -1) {  
    printf("Error allocating memory to tid_map\n");  
    return 0;  
}
```

- c) The thread function was created in the same file and initialised before the main class.

Since the parameter is a void pointer, then it must first be typecast and dereferenced accordingly to extract the min and max sleep time values.

Then `srand()` was used to seed the random function before using it, because when it was not being used, very similar values were being produced, and the output was not random.

It was seeded based on the id the kernel allocated using `pthread_self()`.

Then, the random number was fit into the range specified by `min` and `max` and sleep was called using this random number.

The thread then terminates.

```
void *thread_function(void *arg) {  
    int *args = (int *)arg;  
  
    int min = args[0];  
    int max = args[1];  
  
    srand(pthread_self());  
    int rand_int = rand() % (max - min + 1) + min;  
    sleep(rand_int);  
    pthread_exit(NULL);  
}
```

- d) As requested, the function was adjusted so that it calls the `allocate_tid()` and `release_tid()` functions.

```
void *thread_function(void *arg) {  
    int tid = allocate_tid();  
    if (tid == -1) {  
        printf("Error allocating tid\n");  
        pthread_exit(NULL);  
    }
```

```
}
```

```
//All other code in previous part
```

```
release_tid(tid);  
pthread_exit(NULL);  
}
```

- e) Back in the main class, an array of type `pthread_t` was created of size `'number_of_threads'`.

Then the argument to be given to the thread function was prepared by typecasting an array filled with the min and max sleep values, to a `null` pointer.

Then a for loop called `pthread_create` with the appropriate arguments for the specified number of times.

Then another `for loop` waited for all the threads to join using `pthread_join` and the array of `kernel thread id's` as parameter to indicate which thread to join. The result was simply stored in a dummy variable.

```
pthread_t *kernel_thread_id_list =  
    (pthread_t *)malloc(number_of_threads * sizeof(pthread_t));  
  
if (kernel_thread_id_list == NULL) {  
    printf("Error allocating memory\n");  
}  
  
int thread_args[2] = {min_sleep_time, max_sleep_time};  
void *thread_arg = (void *)thread_args;  
  
for (int i = 0; i < number_of_threads; i++) {  
    pthread_create(&kernel_thread_id_list[i], NULL, &thread_function,  
        thread_arg);  
}  
  
for (int i = 0; i < number_of_threads; i++) {  
    void *result;  
    pthread_join(kernel_thread_id_list[i], &result);  
}
```

Testing :

Note that the program will not run for more than 950 threads on [replit](#), the reason being memory since the program always crashes with a memory error message. It is likely that [replit](#) has a limited amount of memory and CPU, and this is the source of the problem.

First various bounds of input to the command line were tested, or incorrect values given, and the output always displayed a message displaying invalid input, as expected.

```
~/OS-assignment$ ./Task2 955 1 200000000
Invalid min or max sleep time
~/OS-assignment$ ./Task2 -955 1 200000000
Invalid input
~/OS-assignment$ ./Task2 -955 1 2
Invalid input
~/OS-assignment$ ./Task2 -955 1 -2
Invalid input
~/OS-assignment$ ./Task2 0 0 0
Invalid min or max sleep time
```

Then no input was given, and the default values were used.

```
~/OS-assignment$ ./Task2
Number of threads: 100
Min sleep time : 1
Max sleep time : 10
Successful
~/OS-assignment$ █
```

Then normal input was given, and the run was successful.

```
~/OS-assignment$ ./Task2 900 1 2
Number of threads: 900
Min sleep time : 1
Max sleep time : 2
Successful
~/OS-assignment$ █
```

Since all runs were successful up till 950 threads, which is a memory problem, the program seems to be working.

The testing for Task 2 has been done through the CLI and not through a test class, and so no test executable is available.

TASK 3

- a) For this part, the code in task 2 was copied to a new file and adjusted as required since a header file could not really be made to include task 2.

Then a mutex lock was initialised as a global variable in the outer scope using the default initialiser.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

The critical sections of code that must be protected in the thread function are when the functions `allocate_tid()` and `release_tid()` are called since throughout the code for these functions, they are accessing the shared data structure `tidMap`, and if two threads are allowed to manipulate the `tidMap` simultaneously then a `tid` that is not available might be allocated, or two processes might get the same `tid`.

Hence the mutex lock is locked before the two function calls and released directly after, to minimise the amount of code in the critical section.

```
pthread_mutex_lock(&mutex);  
int tid = allocate_tid();  
pthread_mutex_unlock(&mutex);
```

```
//rest of the code
```

```
pthread_mutex_lock(&mutex);  
release_tid(tid);  
pthread_mutex_unlock(&mutex);
```

At the end of the main class the mutex lock was destroyed.

```
pthread_mutex_destroy(&mutex);
```

- b) The number of seconds to sleep if a lock cannot be acquired is achieved by taking the `kernal thread id` using `pthread_self` and taking the modulus over the range of the `min` and `max`.

This is done since the `kernal id's` are usually very large and even in micro seconds, the wait time could mean an hour.

Then the more tries that a thread makes, the larger the range becomes making it more likely that the thread will wait for longer before asking again. This was simply done by multiplying the range by the number of tries.

A maximum number of tries was checked so that if some deadlock had occurred, the program would terminate

The newly adjusted mutex lock acquisition is as follows :

```
tries = 0;
while (tries < MAXTRIES) {
    if (pthread_mutex_trylock(&mutex) == 0) {
        break;
    }
    usleep((kernel_thread_id%((max - min + 1 )*(tries+1)))+1);
    tries++;
}

if (tries == MAXTRIES) {
    printf("No mutex lock could be acquired\n");
    exit(1);
}

//allocate_tid() or release_tid() function call

pthread_mutex_unlock(&mutex);
```

The formula for getting the number of seconds to sleep is protected from division by zero by adding 1 at the end.

This backoff scheme ensures that no errors occur, and while the thread is waiting for a mutex lock, some other threads could continue to execute their code since it is asleep.

Testing:

The executables for 3a and 3b were run with 950 threads with 1 second min and 2 second max for 5 times.

Then with 1 second min and 5 second max for another 5 times.

Each run terminated successfully, showing no immediate errors.

To test the mutual exclusion property, the program needs to check that no race conditions occurred.

For example, two processes acquiring the same `tid` at once.

This however could be very rare and is hard to test for in an empirical manner, with a max of 950 threads.

Instead, a shared integer was added as a global variable and set to 2.

Then each thread would increment the integer if it were odd, and decrement it if were even.

```
if(shared_counter%2 == 0){
    shared_counter--;
}
else {
    shared_counter++;
}
```

In this way if n is the number of threads and $n = 1$, then the integer should be 1 by the end of the program, since one thread has run and the integer is even, so it would have decremented it by 1.

If $n = 2$ then the integer should be 2 by the end of the program

Inductively, if n is even then the integer should be 2 and if n is odd it should be 1.

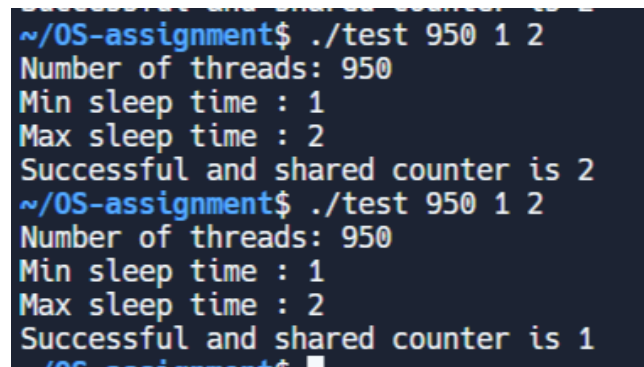
This chunk of code was simply put inside the thread function outside any critical sections at first to test whether any race conditions were active.

The program was then run in a sample size of 20, with an even number of threads, signifying that the answer should always be 2.

In the 20 times it was run the output was 2, 18 times out of 20 and it was 1 twice.

The fact that the output was incorrect once means there is a race condition that has around a 10% chance of occurring.

The output on the 7th and 8th tries was as follows:



```
~/OS-assignment$ ./test 950 1 2
Number of threads: 950
Min sleep time : 1
Max sleep time : 2
Successful and shared counter is 2
~/OS-assignment$ ./test 950 1 2
Number of threads: 950
Min sleep time : 1
Max sleep time : 2
Successful and shared counter is 1
~/OS-assignment$
```

Next the updating of the counter was done inside the critical section between the mutex lock acquisition and release.

```
pthread_mutex_lock(&mutex);
if(shared_counter%2 == 0){
    shared_counter--;
}
else {
    shared_counter++;
}
release_tid(tid);
pthread_mutex_unlock(&mutex);
```

Now the program was run with 950 threads for 50 times, and an output of 2 should always be the case.

The output was 2 every time.

Note that if the probability of a race condition is 10%, the probability that the output is 2 every time for 50 times, and the program still has race conditions that are 10% likely, is $((1-0.1)^{50})$ which is approximately a 0.5 % chance of happening.

Hence the mutex locks are probably functional.

This experiment was repeated for the code in Task 3 part b.

Out of a sample of 20, 3 were incorrect when the updating of the counter was done outside the critical section.

When the updating of the counter was done in the critical section and run for 50 times, the output was always 2. Hence 3b is also very likely to be correct.

A test executable for Task 3 was not made since there were for different main classes which would have made the project a bit too large.

Instead, one test executable was made, it was run for as many times as needed and then it was adjusted for the next case.

Hence why no test executable will be provided for task 3.

TASK 4

- a) The **MAKEFILE** file was made is in the given format requiring only that all files be in the same directory to work.

It works by compiling all the tasks into different executables, where task 1 also has its own test executable.

All that needs to be done to compile all the programs, is to run the command **make** in the replit shell.

The code for the file is as follows:

```
CC = g++
CFLAGS = -Wall -Wextra -pthread
SRC_DIR = .

# Executable targets for each task
# Note that a different executable file was made for each task since
# each of them has a different main class and could no be compiled into
# one executable.

TARGET1 = Task1
TARGET2 = Task2
TARGET3 = Task3a
TARGET4 = Task3b
TARGET5 = Task1_Test

# Source and header files for each target
SRC_FILES1 = $(SRC_DIR)/Task1.c
SRC_FILES2 = $(SRC_DIR)/Task2.c
SRC_FILES3 = $(SRC_DIR)/Task3a.c
SRC_FILES4 = $(SRC_DIR)/Task3b.c
SRC_FILES5 = $(SRC_DIR)/Task1_Test.c
INC_FILES = $(SRC_DIR)/tid_map.h

all: $(TARGET1) $(TARGET2) $(TARGET3) $(TARGET4) $(TARGET5)

$(TARGET1): $(SRC_FILES1) $(INC_FILES)
    $(CC) $(CFLAGS) $(SRC_FILES1) -o $(TARGET1)

$(TARGET2): $(SRC_FILES2) $(INC_FILES)
    $(CC) $(CFLAGS) $(SRC_FILES2) -o $(TARGET2)
```

```
$(TARGET3): $(SRC_FILES3) $(INC_FILES)
$(CC) $(CFLAGS) $(SRC_FILES3) -o $(TARGET3)
```

```
$(TARGET4): $(SRC_FILES4) $(INC_FILES)
$(CC) $(CFLAGS) $(SRC_FILES4) -o $(TARGET4)
```

```
$(TARGET5): $(SRC_FILES5) $(INC_FILES)
$(CC) $(CFLAGS) $(SRC_FILES5) -o $(TARGET5)
```

```
clean:
rm -f $(TARGET1) $(TARGET2) $(TARGET3) $(TARGET4) $(TARGET5)
```

PLAGERISM DECLARATION:

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / We* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Andy Debrincat
Student Name

andy debrincat
Signature

Student Name

Signature

Student Name

Signature

Student Name

Signature

CPS1012
Course Code

T10 Manager
Title of work submitted

26/05/2024
Date