

THREADED SORTING

1 Introduction- Presenting the proble

Our task was to solve a problem of our choosing while using the pthread library in Posix. The problem I chose was sorting a vector of numbers. The algorithm implemented is a variation of the Producer-Consumer design. In the source code there is an array of pseudo-random numbers, the producer thread reads them one by one and puts them into the shared buffer. The consumer thread reads the numbers from the above-mentioned buffer and inserts them in a linked list and performs a sorted insert, ordering the elements from the smallest to biggest number.

1.1 Contents of the Solution

To solve this problem i chose a 3-file approach:

- The main file, where we implement the vast majority of functions and methods
- The linked list header, which defines the list, its constructors and destructors as well as the data type of the list itself
- The linked list source code, which contains the implementations of the sorted insert, as well as the print and destruct functionalities.

2 Analysis of the given problem

As we have established previously, we wish to perform a sort on a series of numbers.

2.1 Example

Let's say we have the following combination of numbers: 32, 21, 90, -1200, 238, 1273, -5.

Having these numbers, we wish to sort them. So of course we are looking for the following outcome: -1200, -5, 21, 32, 90, 238, 1273.

The only limitation is that our buffer is 10 elements. So this means that if our row of numbers consists of more than 10, we shall need the producer thread to load the first 10 elements, and then proceed with the consumed thread. After the consumed thread has finished sorting all available numbers that have been "produced" we can return to the production queue. A solid analogy would be if we owned a small scale shop, we cannot/should not order supplies in advance if we still have some in stock (ideally you would not let yourself run dry, but imagine you don't have infinite money: you need to sell in order to be able to buy more).

3 Defining the structure of the application

As we have mentioned before, our goal is to sort the numbers that we are given. So, the question remains: exactly how do we do that?

3.1 Stages to completion:

- Firstly, we need to take the numbers that have been given to us, and introduce them into the production queue. This operation is symbolised by the "Input data:" output which shows us the numbers we have introduced to be "produced" one by one, until we are done.
- The next step is to begin the production queue itself, we take each number and slowly begin to build our sorted linked list. Each element is compared to each other element, starting from the head of the list, onward until the end. This is also the stage in which our threads are being launched (with the help of `pthread_create`), and ultimately joined.
- After all the numbers have been cycled through, we have to print them out, to show that they have actually been sorted. This stage can be observed on the "Output data:" ping on the terminal. After this task has been successfully completed and the numbers have been printed, we can exit the pthread process and close the log file in which we have been storing all our operations.

4 Defining the Solution

In this section we shall define the essential elements in the code implementation, such as functions, synchronization elements etc.

4.1 Solution:

We have specified that we need to perform a Producer-Consumer activity, so therefore we will need:

- Two threads, one for each of the producer and consumer. These could possibly be supercharged to 2 or more each, in order to speed up either production or consumption.
- We are going to declare a buffer of 10 elements, in order to simulate real usage, and also we shall need a mutex, in order to prevent accessing shared resources at inconvenient times.
- As additional things we are going to use, we will use a sleep timer, in order to put the thread to sleep while the buffer is empty for the consumer, or while the buffer is full for the producer

5 The Implementation

We shall begin by showing a figure which will show how our production and consumer processes work, to be easier to follow up with the code:

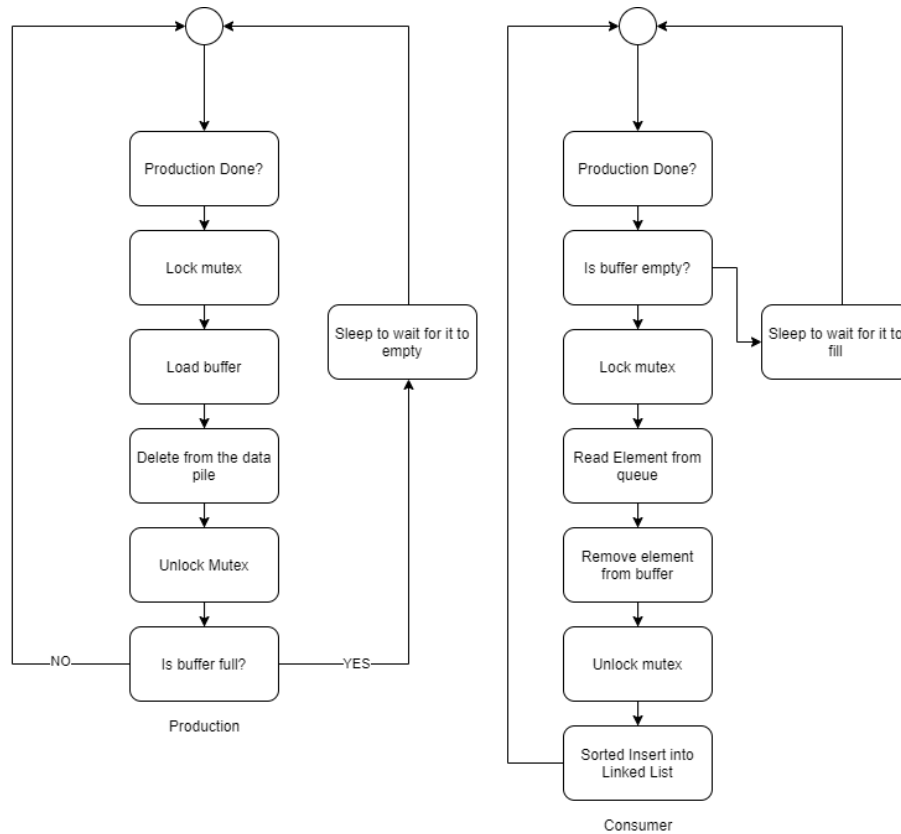


Figure 1: Diagrams for the production and consumption processes

5.1 The actual code

The linked list Header:

```

#include<iostream>

#ifndef LINKED_LIST_HPP
#define LINKED_LIST_HPP

// defining the nodes in the list
typedef struct Node {
int data;
Node *next = nullptr;

Node(int _data) { data = _data; }
} Node;

//defining the linked list and its constructors
class LinkedList{
public:
LinkedList() {};
~LinkedList();

void sortedInsert(int);
void print();
private:

```

```
Node *head = nullptr;
}; //LinkedList

#endif //LINKED_LIST_HPP
```

The linked list source file:

```
#include "linked_list.hpp"

// Free the memory taken by the list(destructor)
LinkedList::~LinkedList() {
    if (head == nullptr) return;

    Node *p, *q;
    for (p = head; p != nullptr; p = q) {
        q = p->next;
        delete p;
    }
}

void LinkedList::sortedInsert(int _value) {
    Node *newNode = new Node(_value); //we declare a new node with the value
    Node *aux = head; // and the auxiliaries, both one that points to the list head
    Node **aux2 = &head; // and another that can modify the list head, if suitable

    //while we have not reached the place of insertion( the data is bigger)
    while (aux != nullptr && aux->data < newNode->data) {
        aux2 = &aux->next; // cycle through
        aux = aux->next;
    }
    *aux2 = newNode; //otherwise insert at the place in which we were
    newNode->next = aux;
}

//print the elements
void LinkedList::print() {
    if (head == nullptr) return;

    Node *p;
    for (p = head; p != nullptr; p = p->next) {
        std::cout << p->data << " ";
    }
}
```

The main file:

```
#include <iostream>
#include <pthread.h>
#include <chrono>
#include <queue>
#include <vector>
#include <fstream>
#include "LinkedList/linked_list.hpp"
#include <unistd.h>
```

```

// ----- Global variables -----
pthread_mutex_t g_mutex;
std::queue<int> g_buffer; // shared buffer between the 2 threads
const int g_buffer_limit = 10; // max number of elements to store in the buffer
unsigned g_sleep = 1; // sleep duration in seconds
std::vector<int> g_input_data = {1, 6, 124, 691, -12, -142, 1242, 1514, 12566, 1205, -2144, 51324, 9
LinkedList g_output_data;
bool g_prod_done = false;
std::ofstream g_log_file;

// ----- Consumer thread function -----
void* consumer(void* ptr) {
while (true) {
if (g_prod_done && g_buffer.empty()) return NULL;

//if the buffer is empty, wait for the producer to add some more data to it
while (g_buffer.empty())
sleep(g_sleep);

// Acquire lock to read from the shared buffer
pthread_mutex_lock(&g_mutex);
int data = g_buffer.front(); // read the first element from the queue
g_buffer.pop(); // and remove it
g_log_file << "Consumed: " << data << std::endl;
pthread_mutex_unlock(&g_mutex); // the lock is no longer needed as we're not accessing shared resour

g_output_data.sortedInsert(data);
}
}

// ----- Producer thread function -----
void* producer(void* ptr) {
while (true) {
if (g_input_data.size() == 0) {
g_prod_done = true;
return NULL;
}

pthread_mutex_lock(&g_mutex);
int data = g_input_data.front();
g_buffer.push(data); // read the first element from the input vector
g_input_data.erase(g_input_data.begin()); // and remove it
g_log_file << "Produced: " << data << std::endl;
pthread_mutex_unlock(&g_mutex);

// if the buffer is full, wait for the consumer to take some data out of it
while (g_buffer.size() >= g_buffer_limit)
sleep(g_sleep);
}
}

// ----- Main thread function -----

```

```

int main() {
g_log_file.open("threads.log");

pthread_t thread1, thread2;

std::cout << "Input data: " << std::endl;
for (auto it = g_input_data.begin(); it != g_input_data.end(); it++) std::cout << *it << " ";
std::cout << std::endl << std::endl;

pthread_create(&thread1, NULL, producer, NULL);
std::cout << "S-a creat firul producer " << std::endl;
pthread_create(&thread2, NULL, consumer, NULL);
std::cout << "S-a creat firul consumer " << std::endl;

pthread_join(thread1, NULL);

pthread_join(thread2, NULL);

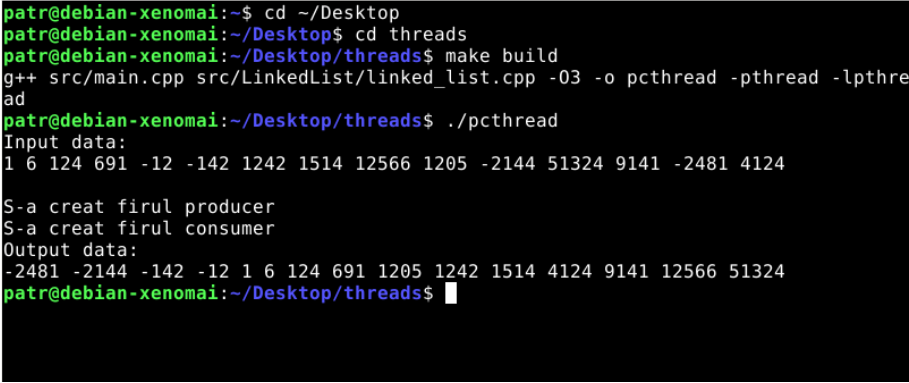
std::cout << "Output data: " << std::endl;
g_output_data.print();
std::cout << std::endl;

pthread_exit(NULL);
g_log_file.close();
return 0;
}

```

6 Testing and bibliography

To run the app, you can download the whole github repo at: <https://github.com/GradCristi/Threaded-Sort>. After that, you will need to navigate in your terminal to the location of these folders, and once you are there input into the terminal the following: `make build`. After that process is over, you are ready to run the binary, which can be done by entering : `./pcthread` into your terminal and hitting enter. After all that is done, you should be greeted with a screen similar to this:



```

patr@debian-xenomai:~$ cd ~/Desktop
patr@debian-xenomai:~/Desktop$ cd threads
patr@debian-xenomai:~/Desktop/threads$ make build
g++ src/main.cpp src/LinkedList/linked_list.cpp -O3 -o pcthread -pthread -lpthread
patr@debian-xenomai:~/Desktop/threads$ ./pcthread
Input data:
1 6 124 691 -12 -142 1242 1514 12566 1205 -2144 51324 9141 -2481 4124

S-a creat firul producer
S-a creat firul consumer
Output data:
-2481 -2144 -142 -12 1 6 124 691 1205 1242 1514 4124 9141 12566 51324
patr@debian-xenomai:~/Desktop/threads$

```

Figure 2: Example of running the program

We can observe that I navigated to the threads folder (which was located on my

desktop), and followed the commands posted above. After all is said and done, we can see the input data, the lines which confirm that the threads were created successfully (which i forgot to translate from romanian, my bad) and the output data, which is a fully sorted vector (not quite a vector) of numbers, from the least big, to the biggest.