Producer Consumer Problem with Dynamic Thread Adjustment

CSC-204 OS Project

The Team

Name	Enrollment No.
T. V Pavan Kumar	23114099
T. Charan Kumar	23114098
G. V. S Charan	23114033
N. Shashank	23114070
P. Balaji	23114078

Overview

The Producer Consumer Problem is a classic example of synchronization problem, where producers and consumer share a buffer. Producers generate data and place it in the shared buffer, and consumers retrieve data from the buffer. To ensure proper synchronization in this problem, we have to use synchronization mechanisms like semaphores, monitors/condition variables. Also there are variants of buffers: it can be finite, it can be circular, and it can be infinite. In this project, we only use a circular buffer.

Dynamic thread adjustment refers to the ability of a system or program to change the number of threads it uses for executing tasks at runtime, adapting to changing workloads or resource availability. This contrasts with static thread assignment, where the number of threads is determined beforehand and remains fixed.

Advantages of dynamic thread adjustment:

- Improved Resource Utilization: Dynamic thread adjustment allows systems to optimize resource utilization. It can scale up the number of threads when more processing power is needed (say due to a surge in workload) and scale down when less power is needed (say during periods of low activity).
- Enhanced Performance: By adapting to the workload, dynamic thread adjustment can prevent over-using of threads, which can lead to wasted resources and performance bottlenecks.
- Adaptability to Dynamic Workloads: Many applications, especially those involving real-time data processing or handling varying input rates, benefit from the ability to dynamically adjust the number of threads to match the workload.

In this project, we implement a dynamic thread adjustment mechanism that adapts the number of active producer and consumer threads based on buffer status. This approach improves resource utilization and reduces bottlenecks by scaling thread activity.

Design and Implementation

Circular Buffer

Implemented a thread-safe circular buffer class so that producer and consumer threads can interact with the buffer without race conditions. Each producer produces an item which consists of a value and a timestamp (the time when the producer produced this item), and consumer later consumes these items. Ensured safety by using mutexes and semaphores to prevent race conditions. Code for inserting and removing items in buffer:

```
void insertItem(int id, int item) {
    sem_wait(&empty);
    if(!running) return;
    pthread_mutex_lock(&mutex);

    buffer[in] = {item, high_resolution_clock::now()};
    total_produced++;
    string msg = "Producer " + to_string(id) + " produced: " + to_string(item) + " at index " + to_string(in);
    log.log(msg);
    in = (in + 1) % SIZE;
    count++;

    pthread_mutex_unlock(&mutex);
    sem_post(&full);
}

void removeItem(int id) {
    sem_wait(&full);
    if(!running) return;
    pthread_mutex_lock(&mutex);

    int item = buffer[out].value;
    auto consumed_time = high_resolution_clock::now();
    total_wait_time += duration_cast<milliseconds>(consumed_time - buffer[out].produced_time).count();
    total_consumed++;
    string msg = "Consumer " + to_string(id) + " consumed: " + to_string(item) + " at index " + to_string(out);
    log.log(msg);
    out = (out + 1) % SIZE;
    count--;

    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
}
```

Logger

Implemented a thread-safe logger class so that threads can post into the log file in a safe manner. Ensured safety by using mutexes and conditional variables. Also implemented a thread-safe FIFO queue inside the logger class so that the messages posted by the threads are written into the log file in the same order.

Also implemented a background thread inside the logger class so that logging is asynchronous: the thread may post the message and then returns almost immediately to continue its work. Then the background thread will post the message in the log file later. Code for background thread creation and initialization of logger class:

```
struct Logger {
    ofstream ofs;
    pthread_mutex_t mtx;
    pthread_cond_t cv;
    queue<string> _queue;
    pthread_t _thread;
    bool stopping = false;

    Logger(const string &file) {
        ofs.open(file, ios::out | ios::trunc);
        if (!ofs.is_open())
            throw runtime_error("Cannot open log file: " + file);

        pthread_mutex_init(&mtx, nullptr);
        pthread_cond_init(&cv,  nullptr);

        pthread_create(&_thread, nullptr, threadEntry, this);
    }
```

Code for logging and queue operations:

```
void log(const string &msg) {
    pthread_mutex_lock(&mtx);
    string timestamp = getCurrentTime();
    string log_msg = "[" + timestamp + "] " + msg;
    _queue.push(log_msg);
    pthread_mutex_unlock(&mtx);
    pthread_cond_signal(&cv);
}

static void* threadEntry(void* arg) {
    return ((Logger*)arg)->process();
}

void* process() {
    while (true) {
        pthread_mutex_lock(&mtx);
        while (_queue.empty() && !stopping) {
            pthread_cond_wait(&cv, &mtx);
        }
        if (_queue.empty() && stopping) {
            pthread_mutex_unlock(&mtx);
            break;
        }

        string msg = _queue.front();
        _queue.pop();
        pthread_mutex_unlock(&mtx);

        ofs << msg << "\n";
        ofs.flush();
    }
    return nullptr;
}
```

We now turn to the dynamic thread adjustment strategies.

Strategy 1

The system dynamically adjusts the number of producer and consumer threads based on buffer's count. In other words:

- If the buffer is more than half full, it adds a new consumer thread to increase consumption rate.
- If the buffer is less than half full, it adds a new producer thread to increase production rate.

Buffer monitoring: Every 3 seconds, the system checks the buffer's count against its size. Then according to the above decision logic, it adjusts the producer and consumer threads. Code for this strategy:

```
for (int i = 0; i < iterations; ++i) {
    int count = buffer->count;
    int size = buffer->SIZE;

    string msg = "Buffer size: " + to_string(count) + " / " + to_string(size);
    buffer->log.log(msg);

    if (count > size / 2) {
        msg = "Buffer more than half full. Adding consumer";
        buffer->log.log(msg);
        add_consumer();
    } else {
        msg = "Buffer less than or equal to half full. Adding producer.";
        buffer->log.log(msg);
        add_producer();
    }
    sleep(3);
}
```

Strategy 2

In this strategy, the system keeps a record of previous buffer count and current buffer count. Then:

- If the previous count is less than the current count, consumer is added.
- If the previous count is more than the current count, producer is added.

Buffer monitoring: Every 3 seconds, the system checks the buffer's count against its size. Then according to the above decision logic, it adjusts the producer and consumer threads. Code for this strategy:

```
for (int i = 0; i < iterations; ++i) {
    int prevCount = buffer->count;
    sleep(3);
    int currCount = buffer->count;

    string msg = "Buffer previous count: " + to_string(prevCount) + " and current count: " + to_string(currCount);
    buffer->log.log(msg);

    if (currCount > prevCount) {
        msg = "Buffer count is increasing, so adding consumer";
        buffer->log.log(msg);
        add_consumer();
    }
    else if (currCount < prevCount) {
        msg = "Buffer count is decreasing, so adding producer";
        buffer->log.log(msg);
        add_producer();
    }
    else {
        msg = "Buffer size is same, so nothing added";
        buffer->log.log(msg);
    }
}
```

Log File

Here is an example of log file and messages in it. Here for each message, we provide the time stamp:

```
[2025-04-26 23:24:40.315] Buffer size: 6 / 10
[2025-04-26 23:24:40.315] Buffer more than half full. Adding consumer
[2025-04-26 23:24:40.315] Added a consumer. Total consumers: 2
[2025-04-26 23:24:40.315] Consumer 1 consumed: 50 at index 6
[2025-04-26 23:24:40.316] Consumer 0 consumed: 80 at index 7
[2025-04-26 23:24:40.316] Producer 0 produced: 62 at index 2
[2025-04-26 23:24:40.316] Producer 1 produced: 80 at index 3
[2025-04-26 23:24:40.816] Consumer 1 consumed: 38 at index 8
[2025-04-26 23:24:40.816] Consumer 0 consumed: 92 at index 9
[2025-04-26 23:24:40.816] Producer 0 produced: 48 at index 4
[2025-04-26 23:24:40.816] Producer 1 produced: 2 at index 5
[2025-04-26 23:24:41.316] Consumer 1 consumed: 91 at index 0
[2025-04-26 23:24:41.316] Consumer 0 consumed: 63 at index 1
[2025-04-26 23:24:41.316] Producer 0 produced: 91 at index 6
[2025-04-26 23:24:41.316] Producer 1 produced: 85 at index 7
[2025-04-26 23:24:41.816] Consumer 0 consumed: 62 at index 2
[2025-04-26 23:24:41.816] Consumer 1 consumed: 80 at index 3
[2025-04-26 23:24:41.816] Producer 0 produced: 61 at index 8
[2025-04-26 23:24:41.816] Producer 1 produced: 78 at index 9
[2025-04-26 23:24:42.316] Consumer 0 consumed: 48 at index 4
```

Performance Analysis

The performance metrics we used are producer/consumer throughput and average waiting time of items produced in buffer. Producer/consumer throughput is the number of items produced/consumed over an interval of time. Waiting time of an item is the time elapsed between its production by some producer and consumption by some consumer. Average waiting time is the average of all the waiting times of consumed items.

To measure these metrics, we varied the buffer size and number of iterations (the number of times the system monitors the state and changes accordingly). For strategy 1, the results are as follows:

Buffer size	Iterations	Avg. waiting time	Producer throughput	Consumer throughput
5	5	0.513 ms	5.065 items/s	4.732 items/s
15	15	0.833 ms	9.598 items/s	9.598 items/s
30	20	1.006 ms	12.698 items/s	12.298 items/s
50	30	1.500 ms	17.664 items/s	17.108 items/s
85	50	1.508 ms	27.336 items/s	27.216 items/s
100	50	1.688 ms	27.556 items/s	27.436 items/s
160	80	1.765 ms	42.494 items/s	42.494 items/s
200	100	1.855 ms	52.380 items/s	52.180 items/s
250	125	1.617 ms	65.111 items/s	64.871 items/s

For strategy 2, the results are as follows:

Buffer size	Iterations	Avg. waiting time	Producer throughput	Consumer throughput
5	5	1.051 ms	3.533 items/s	3.199 items/s
15	15	1.419 ms	3.866 items/s	3.733 items/s
30	20	1.441 ms	3.899 items/s	3.799 items/s
50	30	1.461 ms	3.932 items/s	3.866 items/s
85	50	1.477 ms	3.959 items/s	3.919 items/s
100	50	1.480 ms	3.959 items/s	3.921 items/s
160	80	1.485 ms	3.974 items/s	3.949 items/s
200	100	1.488 ms	3.979 items/s	3.959 items/s
250	125	1.491 ms	3.983 items/s	3.967 items/s

Results plotted:

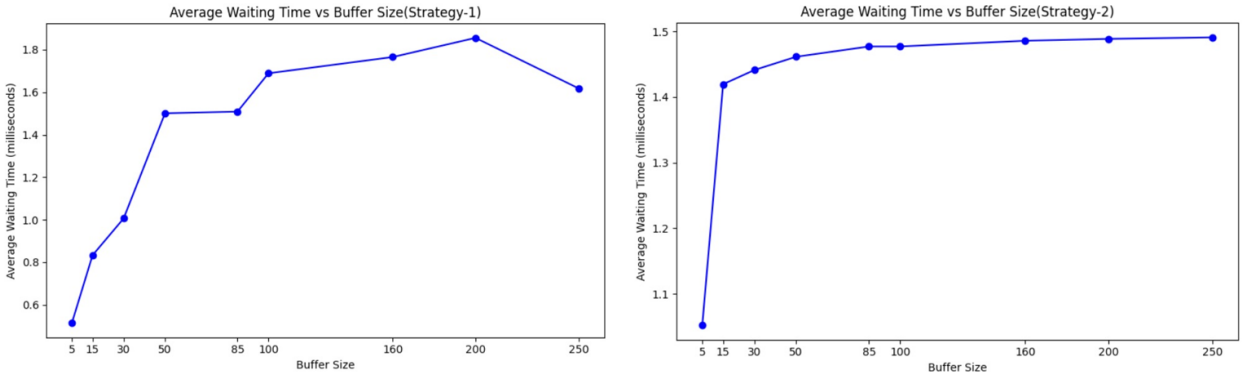

Figure 1: Average waiting time vs. buffer size for both strategies

From these results, we infer two things:

- As the buffer size increases, average waiting time and producer/consumer throughput increase in both strategies, since for large buffer sizes, producers keep pumping items into the queue and consumers keep draining without waiting or stalling, hence higher throughput. When you increase the buffer size, there are more items in the buffer, hence each new item has more predecessors ahead of it, so it waits longer on average.
- We see that producer/consumer throughput increases drastically in strategy 1 than in strategy 2. This is because in strategy 1, we are using a simple threshold rule (if the buffer is more than half full, add a

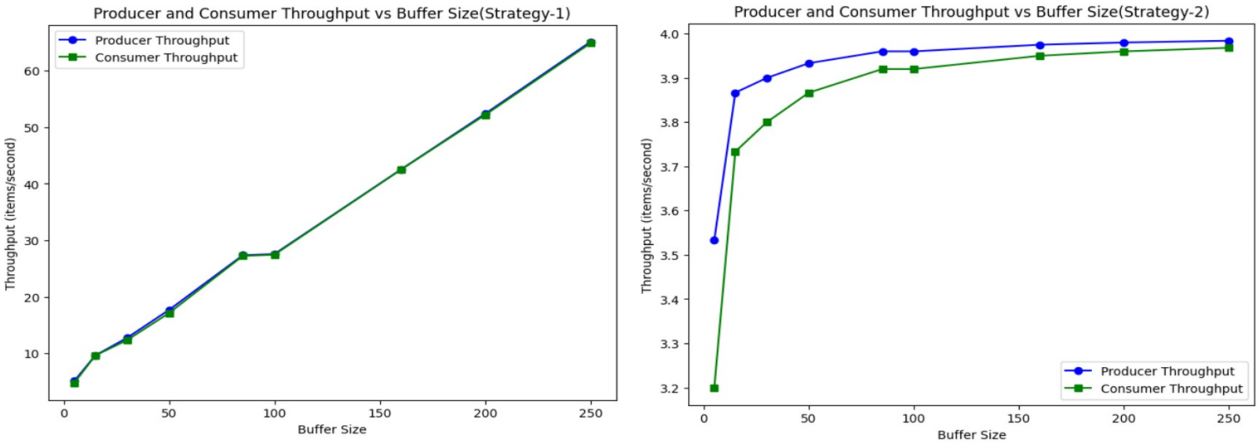

Figure 2: Throughput vs. buffer size for both strategies

producer, otherwise add a consumer) which means we quickly ramp up whichever side is falling behind, so we keep both producers and consumers busy in parallel. In contrast, strategy 2 only looks at whether the previous buffer count was larger than the current. This means we lag behind adjustments (we wait for the consumer to over-drain before we add another producer, and vice versa) leading to long gaps with few threads active. Also in strategy 2, we don't add more threads when previous count is same as current count, hence if there are many such instances, we don't add more threads, which doesn't increase throughput by a large factor.

Challenges Faced

- Implementing a thread-safe logger class was a big challenge. Also in order for messages to be posted in the same order, we also had to implement a thread-safe FIFO queue.
- Choosing the right metrics. Here we chose producer/consumer throughput and average waiting time.
- Design of dynamic thread adjustment strategies.