Assignment 3 (UG)

Video can be found HERE: https://youtu.be/jl8a4uEOfSM

Aidan Carling | a1795819
Elana Parnis | a1831872
Matthew Crawley | a1800227
Xizi Wang | a1824060

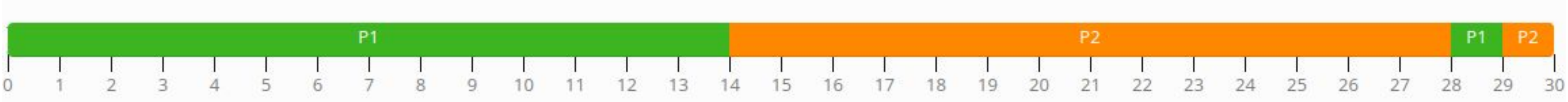
Introduction

Simple Timeout Method

- After x amount of time, process is paused and put back into waiting queue
- Program does not get stuck behind singular long processes
- Decreases waiting time for short processes after long ones
- Allows for more efficient usage of limited resources

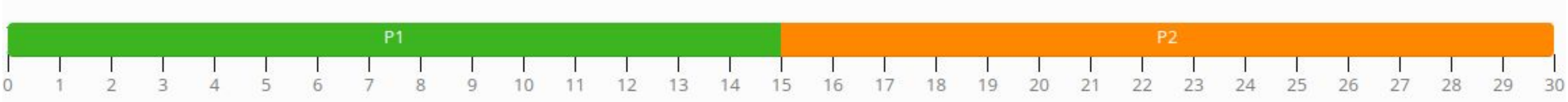
Timeout Flexibility

Without our timeout flexibility



((29-15) + (30-15))/2 = 14.5 average wait time

With our timeout flexibility



(0+15)/2 = 7.5 average wait time

This also helps to reduce the number of context switches that occur!

Priority Queues

```cpp
std::deque<int> lowPriorityQueue; // waiting lowPriorityQueue for regular customers
std::deque<int> highPriorityQueue; // waiting lowPriorityQueue for priority customers
```

```cpp
while (!arrival_events.empty() && (current_time == arrival_events[0].event_time))
{
    if (arrival_events[0].priority == 0)
    {
        highPriorityQueue.push_back(arrival_events[0].customer_id);
    }
    else
    {
        lowPriorityQueue.push_back(arrival_events[0].customer_id);
    }
    arrival_events.pop_front();
}
```

Added to separate queues to be ran based on given ‘membership’ status (AKA it’s priority)

Our defined constant that is compared to the low priority queue.

```cpp
const int SWAP_PRIORITY_THRESHOLD = 300;
```

```cpp
//MOVE LOW PRIORITY TO HIGH PRIORITY QUEUE IF BEEN WAITING TOO LONG
for (int p = 0; p < lowPriorityQueue.size(); p++)
{
    if(current_time - customers.at(lowPriorityQueue.at(p)).arrival_time > SWAP_PRIORITY_THRESHOLD) {
        int swapElArrivalTime = customers.at(lowPriorityQueue.at(p)).arrival_time;

        highPriorityQueue.push_back(lowPriorityQueue.at(p));
        lowPriorityQueue.erase(lowPriorityQueue.begin() + p);
    }
}
```

Priority Queues

```
std::deque<int> lowPriorityQueue; // waiting lowPriorityQueue for regular customers
std::deque<int> highPriorityQueue; // waiting lowPriorityQueue for priority customers
```

```
while (!arrival_events.empty() && (current_time == arrival_events[0].event_time))
{
    if (arrival_events[0].priority == 0)
    {
        highPriorityQueue.push_back(arrival_events[0].customer_id);
    }
    else
    {
        lowPriorityQueue.push_back(arrival_events[0].customer_id);
    }
    arrival_events.pop_front();
}
```

Added to separate queues to be ran based on given ‘membership’ status (AKA it’s priority)

Our defined constant that is compared to the low priority queue.

```
const int SWAP_PRIORITY_THRESHOLD = 300;
```

```
//MOVE LOW PRIORITY TO HIGH PRIORITY QUEUE IF BEEN WAITING TOO LONG
for (int p = 0; p < lowPriorityQueue.size(); p++)
{
    if(current_time - customers.at(lowPriorityQueue.at(p)).arrival_time > SWAP_PRIORITY_THRESHOLD) {
        int swapElArrivalTime = customers.at(lowPriorityQueue.at(p)).arrival_time;

        highPriorityQueue.push_back(lowPriorityQueue.at(p));
        lowPriorityQueue.erase(lowPriorityQueue.begin() + p);
    }
}
```

Priority Queues

```cpp
std::deque<int> lowPriorityQueue; // waiting lowPriorityQueue for regular customers
std::deque<int> highPriorityQueue; // waiting lowPriorityQueue for priority customers
```

```cpp
while (!arrival_events.empty() && (current_time == arrival_events[0].event_time))
{
    if (arrival_events[0].priority == 0)
    {
        highPriorityQueue.push_back(arrival_events[0].customer_id);
    }
    else
    {
        lowPriorityQueue.push_back(arrival_events[0].customer_id);
    }
    arrival_events.pop_front();
}
```

Added to separate queues to be ran based on given ‘membership’ status (AKA it’s priority)

Our defined constant that is compared to the low priority queue.

```cpp
const int SWAP_PRIORITY_THRESHOLD = 300;
```

```cpp
//MOVE LOW PRIORITY TO HIGH PRIORITY QUEUE IF BEEN WAITING TOO LONG
for (int p = 0; p < lowPriorityQueue.size(); p++)
{
    if(current_time - customers.at(lowPriorityQueue.at(p)).arrival_time > SWAP_PRIORITY_THRESHOLD) {
        int swapElArrivalTime = customers.at(lowPriorityQueue.at(p)).arrival_time;

        highPriorityQueue.push_back(lowPriorityQueue.at(p));
        lowPriorityQueue.erase(lowPriorityQueue.begin() + p);
    }
}
```

Priority Queues

```cpp
std::deque<int> lowPriorityQueue; // waiting lowPriorityQueue for regular customers
std::deque<int> highPriorityQueue; // waiting lowPriorityQueue for priority customers
```

```cpp
while (!arrival_events.empty() && (current_time == arrival_events[0].event_time))
{
    if (arrival_events[0].priority == 0)
    {
        highPriorityQueue.push_back(arrival_events[0].customer_id);
    }
    else
    {
        lowPriorityQueue.push_back(arrival_events[0].customer_id);
    }
    arrival_events.pop_front();
}
```

Added to separate queues to be ran based on given ‘membership’ status (AKA it’s priority)

Our defined constant that is compared to the low priority queue.

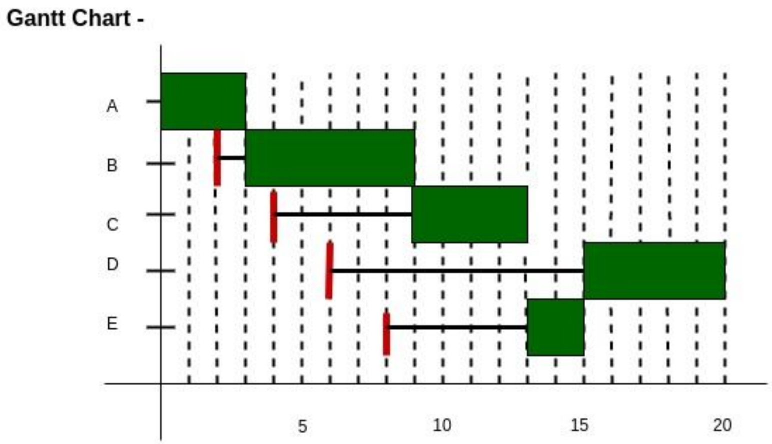
```cpp
const int SWAP_PRIORITY_THRESHOLD = 300;
```

```cpp
//MOVE LOW PRIORITY TO HIGH PRIORITY QUEUE IF BEEN WAITING TOO LONG
for (int p = 0; p < lowPriorityQueue.size(); p++)
{
    if(current_time - customers.at(lowPriorityQueue.at(p)).arrival_time > SWAP_PRIORITY_THRESHOLD) {
        int swapElArrivalTime = customers.at(lowPriorityQueue.at(p)).arrival_time;

        highPriorityQueue.push_back(lowPriorityQueue.at(p));
        lowPriorityQueue.erase(lowPriorityQueue.begin() + p);
    }
}
```

Highest Response Ratio Next

Response Ratio = (Wait Time + Time Remaining) / Time Remaining



Highest Response Ratio Next

```cpp
int getBestCustomer(const std::deque<int> customerIDs, const std::vector<Customer> customers, int current_time) {

    //finds the customer that maximises the ratio of '(waitTime + timeRemaining)/timeRemaining'
    int maxIndex = 0;
    float max = 0;
    for (int i = 0; i < customerIDs.size(); i++) {
        float w = (float)(current_time - customers[customerIDs[i]].arrival_time);
        float s = (float)customers[customerIDs[i]].slots_remaining;
        float responseTimeRatio = -1 * ((w+s)/s + ( customers[customerIDs[i]].priority * LOW_PRIORITY_DESCRIMINATION_CONSTANT));
        if(responseTimeRatio > max) {
            max = responseTimeRatio;
            maxIndex = i;
        }
    }

    return maxIndex;
}
```

Conclusion