OS-L5

HW 2 Report

M. Taimoor Azim Ansari / ma07595 12/10/23

Files:

- **main.c**

  1. Data Structures:
  metrics: Structure to hold various metrics like throughput, response time, and turnaround time.
  pcb: Process Control Block structure representing a process with attributes such as process ID (`pid`), process name (`pname`), remaining time (`ptimeleft`), arrival time (`ptimearrival`), and a flag for whether the process has responded (`responded`).
  dlq_node: Node structure for a doubly-linked queue, containing a pointer to the next and previous nodes (`pfwd` and `pbck`) and a pointer to the associated process control block (`data`).
  dlq: Doubly Linked Queue structure with pointers to the head and tail nodes.

  2. Queue Operations:
  get_new_node: Allocates memory for a new node and initializes it with the given process control block.
  add_to_tail: Adds a node to the tail of the doubly linked queue.
  remove_from_head: Removes and returns the node from the head of the doubly linked queue.
  print_q: Prints the contents of the queue.
  is_empty: Checks if the queue is empty.

  3. Sorting Functions:
  sort_by_timetocompletion: Implements a bubble sort on the queue based on the remaining time for completion of processes.
  sort_by_arrival_time: Implements a bubble sort on the queue based on arrival times.

  4. Tokenization:
  tokenize_pdata: Tokenizes a string containing process information and returns a pointer to a process control block.

5. Scheduler Functions:
get_ready_queue: Populates a ready queue with processes that have arrived at the current system time.
sched_FIFO: Implements First-Come-First-Serve scheduling algorithm.
sched_STCF: Implements Shortest Time to Completion First scheduling algorithm.
sched_SJF: Implements Shortest Job First scheduling algorithm.
sched_RR: Implements Round Robin scheduling algorithm.

6. Main Function:
- Reads the number of processes and the scheduling policy from the standard input.
- Reads process information, tokenizes it, and adds it to the process queue.
- Initializes system time and sorts the process queue by arrival time.
- Calls the appropriate scheduler function based on the chosen policy.
- Prints various metrics such as throughput, turnaround time, and response time.

7. Output:
- The program outputs scheduling information for each time unit, and at the end, it prints metrics such as throughput, turnaround time, and response time.

- **Makefile**

This Makefile includes four targets:

build: Compiles the main.c file using GCC with the -Wall flag to enable warnings. The output executable is named out.

run: Executes the compiled program.

rebuild: Cleans the project using the clean target and then builds it again using the build target.

clean: Removes the compiled executable (out).

To use this Makefile, you can open your terminal, navigate to the directory containing the source code and the Makefile, and then run the following commands:

- make build: Compiles the code.
- make run: Runs the compiled program.
- make rebuild: Cleans and rebuilds the project.
- make clean: Removes the compiled executable.

Implemented Scheduling Processes

First-Come-First-Serve (FIFO):

- **Implementation:** The FIFO scheduling algorithm is implemented in the sched_FIFO function. It prioritizes processes based on their arrival time, executing the earliest arriving process first.
- **Logic:** The algorithm maintains a ready queue and processes each task in the order of their arrival. It does not consider the remaining time needed for completion but focuses solely on the order of arrival. This approach is simple and ensures fairness, but it may lead to longer waiting times for processes that arrive later.
- **Code:**

```c
// implement the FIFO scheduling code
void sched_FIFO(dlq *const p_fq, int *p_time, metrics *time_data)
{
    *p_time = 1;

    dlq_node *temp_head = p_fq->head;
    sort_by_arrival_time(p_fq);
    dlq ready_queue;
    ready_queue.head = NULL;
    ready_queue.tail = NULL;

    while (*p_time <= temp_head->data->ptimearrival)
    {
        printf("%d:idle:empty:\n", *p_time);
        *p_time += 1;
    }

    dlq_node *nd = p_fq->head;
    while (nd)
    {
        time_data->response_total -= nd->data->ptimearrival;
        nd = nd->pbck;
    }

    while (temp_head)
    {

        if (temp_head->data->ptimeleft > 0)
        {
            get_ready_queue(p_fq, p_time, &ready_queue, temp_head->data->pname);
            printf("%d:%s:", *p_time, temp_head->data->pname);

            if (is_empty(&ready_queue))
            {
                printf("empty:\n");
            }
            else
            {
                print_q(&ready_queue);
                printf(":\n");
            }
            temp_head->data->ptimeleft -= 1;
            *p_time += 1;
        }
        else
        {
            time_data->turnaround_total += *p_time - temp_head->data->ptimearrival - 1;
            time_data->response_total += *p_time;
            temp_head = temp_head->pbck;
            if (!temp_head)
            {
                time_data->response_total -= *p_time;
            }
        }
    }

    time_data->throughput_total = *p_time - 1;
}
```

Shortest Job First (SJF):

- **Implementation:** The SJF scheduling algorithm is implemented in the sched_SJF function. It selects the process with the shortest remaining time for execution.
- **Logic:** The algorithm maintains a ready queue, and at each time unit, it selects the process with the shortest remaining time. This approach minimizes the total time taken for all processes to complete, leading to optimal turnaround times. However, predicting the time needed for completion accurately can be challenging.
- **Code:**

```c

// implement the SJF scheduling code
void sched_SJF(dlq *const p_fq, int *p_time, metrics *time_data)
{
    *p_time = 1;

    dlq_node *temp_head = p_fq->head;
    sort_by_arrival_time(p_fq);
    dlq ready_queue;
    ready_queue.head = NULL;
    ready_queue.tail = NULL;

    while (*p_time <= temp_head->data->ptimearrival)
    {
        printf("%d:idle:empty:\n", *p_time);
        *p_time += 1;
    }

    dlq_node *nd = p_fq->head;
    while (nd)
    {
        time_data->response_total -= nd->data->ptimearrival;
        nd = nd->pbck;
    }

    while (temp_head)
    {

        if (temp_head->data->ptimeleft > 0)
        {
            if (temp_head->data->responded == 0)
            {
                time_data->response_total += *p_time - 1;
                temp_head->data->responded = 1;
            }

            get_ready_queue(p_fq, p_time, &ready_queue, temp_head->data->pname);
            sort_by_timetocompletion(&ready_queue);
            printf("%d:%s:", *p_time, temp_head->data->pname);

            if (is_empty(&ready_queue))
            {
                printf("empty:\n");
            }
            else
            {
                print_q(&ready_queue);
                printf(":\n");
            }

            temp_head->data->ptimeleft -= 1;
            *p_time += 1;
        }
        else
        {
            time_data->turnaround_total += *p_time - temp_head->data->ptimearrival - 1;

            if (!is_empty(&ready_queue))
            {
                temp_head = ready_queue.head;
            }
            else
            {
                temp_head = temp_head->pbck;
            }
        }
    }

    time_data->throughput_total = *p_time - 1;
}
```

Shortest Time to Completion First (STCF):

- **Implementation:** The STCF scheduling algorithm is implemented in the sched_STCF function. It selects the process with the shortest remaining time but allows preemption.
- **Logic:** Similar to SJF, STCF selects the process with the shortest remaining time. However, if a new process arrives with a shorter time to completion, it can preempt the currently executing process. This approach aims to minimize waiting times and improve overall system efficiency.
- **Code:**

```c
// implement the STCF scheduling code
void sched_STCF(dlq *const p_fq, int *p_time, metrics *time_data)
{
    *p_time = 1;

    dlq_node *temp_head = p_fq->head;
    sort_by_arrival_time(p_fq);
    dlq ready_queue;
    ready_queue.head = NULL;
    ready_queue.tail = NULL;

    while (*p_time <= temp_head->data->ptimearrival)
    {
        printf("%d:idle:empty:\n", *p_time);
        *p_time += 1;
    }

    dlq_node *nd = p_fq->head;
    while (nd)
    {
        time_data->response_total -= nd->data->ptimearrival;
        nd = nd->pbck;
    }

    while (temp_head)
    {

        if (temp_head->data->ptimeleft > 0)
        {
            get_ready_queue(p_fq, p_time, &ready_queue, temp_head->data->pname);
            sort_by_timetocompletion(&ready_queue);

            if (is_empty(&ready_queue))
            {
                printf("%d:%s:", *p_time, temp_head->data->pname);
                printf("empty:\n");
            }
            else
            {
                if (temp_head->data->ptimeleft > ready_queue.head->data->ptimeleft)
                {
                    dlq_node *d = remove_from_head(&ready_queue);
                    add_to_tail(&ready_queue, get_new_node(temp_head->data));
                    temp_head = d;
                    sort_by_timetocompletion(&ready_queue);
                }

                printf("%d:%s:", *p_time, temp_head->data->pname);
                print_q(&ready_queue);
                printf(":\n");
            }

            temp_head->data->ptimeleft -= 1;
            if (temp_head->data->responded == 0)
            {
                time_data->response_total += *p_time - 1;
                temp_head->data->responded = 1;
            }
            *p_time += 1;
        }
        else
        {
            time_data->turnaround_total += *p_time - temp_head->data->ptimearrival - 1;

            if (!is_empty(&ready_queue))
            {

                temp_head = ready_queue.head;
            }
            else
            {

                temp_head = temp_head->pbck;
            }
        }
    }
    time_data->throughput_total = *p_time - 1;
}
```

Round Robin (RR):

- **Implementation:** The Round Robin scheduling algorithm is implemented in the sched_RR function. It allocates a fixed time slice to each process in a cyclic manner.
- **Logic:** The algorithm uses a time-slicing approach, where each process gets a turn to execute for a fixed time quantum. If a process doesn't complete within its time quantum, it is moved to the back of the queue, and the next process in line gets a chance. This approach ensures fairness but may lead to higher turnaround times for processes with longer execution times.
- **Code:**

```c
void sched_RR(dlq *const p_fq, int *p_time, metrics *time_data)
{

    *p_time = 1;

    sort_by_arrival_time(p_fq);

    dlq_node *temp_head = p_fq->head;
    dlq ready_queue;
    ready_queue.head = NULL;
    ready_queue.tail = NULL;

    while (*p_time <= temp_head->data->ptimearrival)
    {
        printf("%d:idle:empty:\n", *p_time);
        *p_time += 1;
    }

    dlq_node *nd = p_fq->head;
    while (nd)
    {
        nd->data->responded = 0;
        time_data->response_total -= nd->data->ptimearrival;
        nd = nd->pbck;
    }

    get_ready_queue(p_fq, p_time, &ready_queue, temp_head->data->pname);
    while (temp_head->data->ptimeleft > 0 || !is_empty(&ready_queue))
    {

        temp_head->data->ptimeleft -= 1;
        if (temp_head->data->ptimeleft <= 0)
        {
            time_data->turnaround_total += *p_time - temp_head->data->ptimearrival;
        }
        if (temp_head->data->responded == 0)
        {
            time_data->response_total += *p_time - 1;
            temp_head->data->responded = 1;
        }

        if (is_empty(&ready_queue))
        {
            printf("%d:%s:", *p_time, temp_head->data->pname);
            printf("empty:\n");
            *p_time += 1;
            get_ready_queue(p_fq, p_time, &ready_queue, temp_head->data->pname);
        }
        else
        {

            printf("%d:%s:", *p_time, temp_head->data->pname);
            print_q(&ready_queue);
            printf(":\n");

            dlq_node *d = remove_from_head(&ready_queue);
            if (temp_head->data->ptimeleft > 0)
            {
                add_to_tail(&ready_queue, get_new_node(temp_head->data));
            }
            temp_head->data = d->data;
            *p_time += 1;
            get_ready_queue(p_fq, p_time, &ready_queue, temp_head->data->pname);
        }
    }

    time_data->throughput_total = *p_time - 1;
}
```

Helper Function:

get_ready_queue:

- The function takes as parameters a pointer to the process queue (p_fq), a pointer to the system time (p_time), a pointer to the ready queue (ready_queue), and the name of the currently executing process (pname).
- It traverses the entire process queue (p_fq) to check if any new processes have arrived at the current system time.
- If a process has arrived, and it is not the same as the currently executing process (pname), it is added to the ready queue (ready_queue).
- If the ready queue is not empty and the process at the head is the same as the currently executing process (pname), it is removed from the ready queue. This ensures that the process doesn't get added to the ready queue and immediately removed, avoiding unnecessary context switches

Performance Metrics

- **Throughput:** Throughput is the number of processes completed or executed within a given time period.
- **Turnaround Time:** Turnaround time is the total time taken for a process to complete from the moment it arrives in the system until it finishes execution.
- **Response Time:** Response time is the time elapsed between the arrival of a process in the system and the time it gets its first response (starts execution).

	FIFO			SJF		
	Turnaround Time	Throughput	Response Time	Turnaround Time	Throughput	Response Time
Test Case 0	10.333	0.167	5	9	0.167	3.67
Test Case 2	10.333	0.167	5	9	0.167	3.67
Test Case 5	19.167	0.154	12.67	16.83	0.154	10.3
Test Case 10	27.67	0.107	18.33	23	0.107	13.67
Test Case 13	36.5	0.113	27.63	27.38	0.113	18.5
Averages	20.8006	0.1416	13.726	17.042	0.1416	9.962

	STCF			RR		
	Turnaround Time	Throughput	Response Time	Turnaround Time	Throughput	Response Time
Test Case 0	9	0.167	3.67	12.33	0.167	1
Test Case 2	9	0.167	3.67	12.33	0.167	1
Test Case 5	16.83	0.154	10.33	26	0.154	2.5
Test Case 10	22.67	0.107	12.5	36.33	0.107	2.5
Test Case 13	27.25	0.113	17.88	49.63	0.113	3.5
Averages	16.95	0.1416	9.61	27.324	0.1416	2.1

These calculations are based on the following terms:
- Completion Time: Time at which a process completes its execution.
- Arrival Time: Time at which a process arrives in the system.
- Response Time: Time elapsed between a process's arrival and its first response.

Results/Conclusions:

- The type of algorithm doesn’t affect the throughput.
- STCF has the best performance in terms of turnaround time.
- RR has the best performance in terms of the response time.