

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 (`p fwd` 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:**

```
285 // implement the FIFO scheduling code
286 void sched_FIFO(dlq *const p_fq, int *p_time, metrics *time_data)
287 {
288     *p_time = 1;
289
290     dlq_node *temp_head = p_fq->head;
291     sort_by_arrival_time(p_fq);
292     dlq_ready_queue;
293     ready_queue.head = NULL;
294     ready_queue.tail = NULL;
295
296     while (*p_time <= temp_head->data->ptimearrival)
297     {
298         printf("%d:idle:empty:\n", *p_time);
299         *p_time += 1;
300     }
301
302     dlq_node *nd = p_fq->head;
303     while (nd)
304     {
305         time_data->response_total -= nd->data->ptimearrival;
306         nd = nd->pbck;
307     }
308
309     while (temp_head)
310     {
311         if (temp_head->data->ptimeleft > 0)
312         {
313             get_ready_queue(p_fq, p_time, &ready_queue, temp_head->data->pname);
314             printf("%d:%s:", *p_time, temp_head->data->pname);
315
316             if (is_empty(&ready_queue))
317             {
318                 printf("empty:\n");
319             }
320             else
321             {
322                 print_q(&ready_queue);
323                 printf("\n");
324             }
325             temp_head->data->ptimeleft -= 1;
326             *p_time += 1;
327         }
328         else
329         {
330             time_data->turnaround_total += *p_time - temp_head->data->ptimearrival - 1;
331             time_data->response_total += *p_time;
332             temp_head = temp_head->pbck;
333             if (!temp_head)
334             {
335                 time_data->response_total -= *p_time;
336             }
337         }
338     }
339
340     time_data->throughput_total = *p_time - 1;
341 }
342
343 }
```

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:**

```
423 // implement the SJF scheduling code
424 void sched_SJF(dlq *const p_fq, int *p_time, metrics *time_data)
425 {
426     *p_time = 1;
427
428     dlq_node *temp_head = p_fq->head;
429     sort_by_arrival_time(p_fq);
430     dlq_ready_queue;
431     ready_queue.head = NULL;
432     ready_queue.tail = NULL;
433
434     while (*p_time <= temp_head->data->ptimearrival)
435     {
436         printf("%d:idle:empty:\n", *p_time);
437         *p_time += 1;
438     }
439
440     dlq_node *nd = p_fq->head;
441     while (nd)
442     {
443         time_data->response_total -= nd->data->ptimearrival;
444         nd = nd->pbck;
445     }
446
447     while (temp_head)
448     {
449         if (temp_head->data->ptimeleft > 0)
450         {
451             if (temp_head->data->responded == 0)
452             {
453                 time_data->response_total += *p_time - 1;
454                 temp_head->data->responded = 1;
455             }
456
457             get_ready_queue(p_fq, p_time, &ready_queue, temp_head->data->pname);
458             sort_by_timetocompletion(&ready_queue);
459             printf("%d:%s:", *p_time, temp_head->data->pname);
460
461             if (is_empty(&ready_queue))
462             {
463                 printf("empty:\n");
464             }
465             else
466             {
467                 print_q(&ready_queue);
468                 printf(":\n");
469             }
470
471             temp_head->data->ptimeleft -= 1;
472             *p_time += 1;
473         }
474         else
475         {
476             time_data->turnaround_total += *p_time - temp_head->data->ptimearrival - 1;
477
478             if (!is_empty(&ready_queue))
479             {
480                 temp_head = ready_queue.head;
481             }
482             else
483             {
484                 temp_head = temp_head->pbck;
485             }
486         }
487     }
488
489     time_data->throughput_total = *p_time - 1;
490 }
491
492 }
```

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:**

```
344 // implement the STCF scheduling code
345 void sched_STCF(dlq *const p_fq, int *p_time, metrics *time_data)
347 {
348     *p_time = 1;
349
350     dlq_node *temp_head = p_fq->head;
351     sort_by_arrival_time(p_fq);
352     dlq_ready_queue;
353     ready_queue.head = NULL;
354     ready_queue.tail = NULL;
355
356     while (*p_time <= temp_head->data->ptimearrival)
357     {
358         printf("%d:idle:empty:\n", *p_time);
359         *p_time += 1;
360     }
361
362     dlq_node *nd = p_fq->head;
363     while (nd)
364     {
365         time_data->response_total += nd->data->ptimearrival;
366         nd = nd->pbck;
367     }
368
369     while (temp_head)
370     {
371         if (temp_head->data->ptimelleft > 0)
372         {
373             get_ready_queue(p_fq, p_time, &ready_queue, temp_head->data->pname);
374             sort_by_timetocompletion(&ready_queue);
375
376             if (is_empty(&ready_queue))
377             {
378                 printf("%d:%s:", *p_time, temp_head->data->pname);
379                 printf("empty:\n");
380             }
381             else
382             {
383                 if (temp_head->data->ptimelleft > ready_queue.head->data->ptimelleft)
384                 {
385                     dlq_node *d = remove_from_head(&ready_queue);
386                     add_to_tail(&ready_queue, get_new_node(temp_head->data));
387                     temp_head = d;
388                     sort_by_timetocompletion(&ready_queue);
389                 }
390
391                 printf("%d:%s:", *p_time, temp_head->data->pname);
392                 print_q(&ready_queue);
393                 printf(":\n");
394             }
395
396             temp_head->data->ptimelleft -= 1;
397             if (temp_head->data->responded == 0)
398             {
399                 time_data->response_total += *p_time - 1;
400                 temp_head->data->responded = 1;
401             }
402             *p_time += 1;
403         }
404         else
405         {
406             time_data->turnaround_total += *p_time - temp_head->data->ptimearrival - 1;
407
408             if (!is_empty(&ready_queue))
409             {
410                 temp_head = ready_queue.head;
411             }
412             else
413             {
414                 temp_head = temp_head->pbck;
415             }
416         }
417     }
418
419     time_data->throughput_total = *p_time - 1;
420 }
```

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:**

```
494 void sched_RR(dlq *const p_fq, int *p_time, metrics *time_data)
495 {
496     *p_time = 1;
497     sort_by_arrival_time(p_fq);
498     dlq_node *temp_head = p_fq->head;
499     dlq_ready_queue;
500     ready_queue.head = NULL;
501     ready_queue.tail = NULL;
502
503     while (*p_time <= temp_head->data->ptimearrival)
504     {
505         printf("%d: idle: empty: \n", *p_time);
506         *p_time += 1;
507     }
508
509     dlq_node *nd = p_fq->head;
510     while (nd)
511     {
512         nd->data->responded = 0;
513         time_data->response_total -= nd->data->ptimearrival;
514         nd = nd->pbck;
515     }
516
517     get_ready_queue(p_fq, p_time, &ready_queue, temp_head->data->pname);
518     while (temp_head->data->ptimeleft > 0 || !is_empty(&ready_queue))
519     {
520         temp_head->data->ptimeleft -= 1;
521         if (temp_head->data->ptimeleft <= 0)
522         {
523             time_data->turnaround_total += *p_time - temp_head->data->ptimearrival;
524         }
525         if (temp_head->data->responded == 0)
526         {
527             time_data->response_total += *p_time - 1;
528             temp_head->data->responded = 1;
529         }
530
531         if (is_empty(&ready_queue))
532         {
533             printf("%d: %s: ", *p_time, temp_head->data->pname);
534             printf("empty: \n");
535             *p_time += 1;
536             get_ready_queue(p_fq, p_time, &ready_queue, temp_head->data->pname);
537         }
538         else
539         {
540             printf("%d: %s: ", *p_time, temp_head->data->pname);
541             print_q(&ready_queue);
542             printf("\n");
543
544             dlq_node *d = remove_from_head(&ready_queue);
545             if (temp_head->data->ptimeleft > 0)
546             {
547                 add_to_tail(&ready_queue, get_new_node(temp_head->data));
548             }
549             temp_head->data = d->data;
550             *p_time += 1;
551             get_ready_queue(p_fq, p_time, &ready_queue, temp_head->data->pname);
552         }
553     }
554
555     time_data->throughput_total = *p_time - 1;
556 }
```

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.