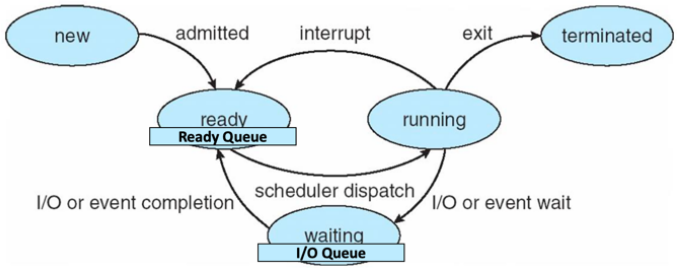
2023-2 학기

운영체제 Term Project #1

2019312014 컴퓨터교육과 박병준

담당 교수: 류은석

1. 주제: Round Robin (RR) Scheduling 구현하기



기본적으로 프로세스를 순서대로 처리하는 FIFO 방식이지만, 정해진 time quantum 마다 running 상태인 process 가 switching 되는 선점형(preemptive) 방식이 추가된다.

2. Shell scripts (commands) to build and execute the source code

```
pbj@LAPTOP-4I8AU5BK:~/codes$ ./RR-scheduling_pbj
[msqid 33] msgget success!
child process created [PID 20907]
child process's init msgsnd complete [PID 20907]
child process created [PID 20908]
child process's init msgsnd complete [PID 20908]
child process created [PID 20909]
child process's init msgsnd complete [PID 20909]
child process created [PID 20910]
child process's init msgsnd complete [PID 20910]
child process created [PID 20911]
child process's init msgsnd complete [PID 20911]
child process created [PID 20912]
child process's init msgsnd complete [PID 20912]
child process created [PID 20913]
child process's init msgsnd complete [PID 20913]
child process created [PID 20914]
child process's init msgsnd complete [PID 20914]
child process created [PID 20915]
parent process executing [PID 20906]
child process's init msgsnd complete [PID 20915]
child process created [PID 20916]
child process's init msgsnd complete [PID 20916]
```

라운드 로빈 스케줄링을 구현한 코드 RR-scheduling_pbj.cpp 의 실행파일인 RR-scheduling_pbj 를 ./RR-scheduling_pbj 명령으로 실행한 화면이다. 터미널 창에 msqid 와 함께 메시지 큐 생성이 성공한 것, pid 와 함께 child process 가 올바르게 생성되고 초기화된 것, 실행 중인 parent process 의 pid 등이 출력되도록 했다. Schedule_dump.txt 파일의 msqid 값과 pid 값들은 터미널 창의 msqid 값과 pid 값들과 동일하다는 것을 확인할 수 있다. 위의 이미지는 time quantum 이 100 일 때의 터미널 창을 캡처한 이미지이다.

3. 구현 내용

Whether the code implementation is complete	O
The setting of time quantum (which executed without errors)	40ms(TICK 의 2 배)
Whether IPC message queue is used	O
How the i/o operation is implemented (one of four options in section 3)	Option 3
Whether the multi-level queue was implemented (extra score in section 3)	X

코드 구현은 완료되었으며, 오류 없이 동작되는 것도 직접 확인했다. TICK 을 20ms 로 설정했는데, TICK 의 배수 중 가장 작은 값인 40ms 을 time quantum 으로 설정했을 때도 오류 없이 올바르게 동작했다. Schedule.dump(q=40).txt 파일을 보면 quantum count 가 40 을 넘어가는 경우는 보이지 않아 올바르게 동작했음을 확인했다. 만약 20ms 을 time quantum 으로 설정하게 되면 running 중간에 io 과정을 삽입할 수 없기 때문에, 40ms 은 이론상 설정할 수 있는 가장 작은 time quantum 으로 볼 수 있다. 코드 구현에 message passing 방식의 IPC 메시지 큐를 사용했다. IPC message 사용처는 다음과 같다.

child 기준: <1. 자신의 init 정보 송신> <2. tick 마다 자신의 정보 수신> <3. terminate 시 parent 에게 알림>

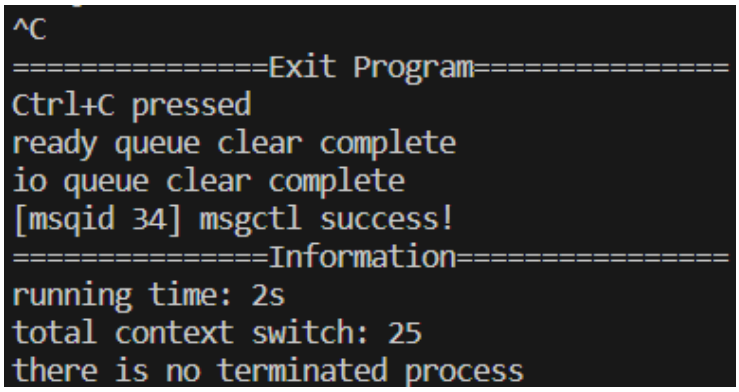
parent 기준: <1. child 의 init 또는 terminate 정보 수신> <2. tick 마다 child 정보 송신>

Option 3 를 선택하여, process 가 생성될 때 io start time(코드에선 io_start 로 선언), io duration(코드에선 io_burst), CPU burst time(코드에선 cpu_burst)가 랜덤하게 주어지도록 했다. Total running time 이 1 분 이상이어야 하므로 cpu_burst 는 TICK(20ms) 간격으로 6 초(6000ms) ~

8 초(8000ms)의 값이 랜덤으로, io_burst 는 TICK(20ms) 간격으로 200ms ~ 320ms 의 값이 랜덤으로, io_start 는 TICK(20ms) 간격으로 TICK(20ms)부터 time quantum 이 끝나기 전까지의 값이 랜덤으로 주어지도록 했다. 이 내용은 init_process 함수에 작성되어 있다.

이제 코드의 구체적인 내용을 설명하고자 한다. 코드 내의 주석을 자세히 작성했기 때문에, 코드의 주석을 통해서도 구체적인 내용을 알 수 있다. 1 개의 message queue 만 사용할 예정이기 때문에 메시지의 식별번호가 될 KEY_NUM 을 미리 선언했다. Child process 의 pid, cpu burst time, state 등의 정보가 포함된 PCB 는 process 구조체를 통해 구현했다. Child 가 생성된 후, init_process 함수를 통해 process(PCB)가 초기화되는데, 그 중 quantum 에 대한 정보는 -1 로 초기화하여 parent 가 ipc 메시지를 전달하기 전까진 time quantum 정보를 모르도록 했다. remainIO 값은 0 으로 초기화해 두었다가 io queue 에 push 되면 랜덤으로 선언된 io_burst 값을 remainIO 에 대입해 remainIO 를 감소시키며 io 가 이루어지도록 했다. Is_io 값은 초기에 -1 로 설정했으며, cpu dispatch 시 정해진 확률(25%)에 따라 io task 가 생성되면 1 로 업데이트, 그렇지 않으면 0 으로 업데이트되도록 했다. State 는 0 이 ready 상태, 1 이 running 상태, 2 가 waiting 상태, 3 이 terminating 상태가 되도록 했다. 이외에도 통계를 분석하기 위해 arrival, response, wait 변수를 두었는데, response 의 경우 running(CPU dispatch) 전까진 response time 을 모르기 때문에 초기에 -1 로 설정했다가 이후 running 시 업데이트하는 방식을 사용했다. 총 context switching 횟수, process 별 response, waiting, turnaround time 의 통계 데이터를 기록하기 위해 analyzeData 구조체를 선언했다. 프로그램 실행 도중 Ctrl+C 입력이 들어오게 되어 발생된 SIGINT 시그널을 이용해 프로그램이 종료되며 ready queue 와 io queue clear, 메시지 큐 삭제 및 최종 결과 출력이 이루어지도록 했다. Ctrl+C 가 입력될 경우 terminated process 가 하나도 존재하지 않을 수 있어, printData 에서 이런 예외 상황에 대한 처리가 이루어지도록 했다.

- Ctrl+C 가 입력된 경우 예시



Schedule 함수는 parent process 에서 setitimer 로 인해 TICK 마다 주기적으로 발생되는 시그널을 핸들링하는 함수이다. 즉, TICK 마다 schedule 함수가 실행된다. Schedule 함수는 종료 조건 상황일 때 프로그램이 종료되도록 하는 역할, ready queue 의 process 들을 총괄하는 역할, io queue 의 process 들을 총괄하는 역할, log 파일에 기록될 정보들을 출력하는 역할 등을 수행한다. 프로그램 종료 조건은 모든 process 가 terminate 된 경우이며, process 각각은 terminate 될 때 자신의 response time, 누적된 waiting time, turnaround time 등을 통계 분석을 위한 analyzeData 구조체형 data 에 누적한다. 이렇게 하면 data 에 누적된 값들을 terminate 된 process 의 개수로 나누어 평균값이 계산될 수 있다. 다음으로, parent 는 child process 의 init 정보를 msgrcv 로 받아 ready queue 에 push 하는데, ready queue 의 process 들 중 front 에 해당하는 process 가 running 되도록 하고, 나머지 process 들은 wait 값을 TICK 만큼 증가시킨다. 그 후, ready queue 에서 io queue 로 가는 경우, terminate 되는 경우, ready queue 의 맨 뒤로 push 되는 경우들을 모두 고려하여 state 값을 바꾸고 pop 하는 등 알맞은 처리를 해준다. 예를 들어 running process 가 time quantum 만큼 실행되어 ready queue 의 맨 뒤로 push 되는 경우를 처리하기 위해, TICK 마다 TICK 만큼 값이 증가하는 quantum_count 변수를 두어 quantum_count 가 지정된 time quantum 까지 도달한 경우 다음 process 처리를 위해 quantum_count 를 0 으로 초기화하고 context switching 이 일어나도록 했다. 모든 처리가 끝나면, parent 는 child 에게 msgsnd 로 메시지를 보낸다. 다음으로, schedule 함수는 io queue 의 process 들도 처리하는데, front 의 remainIO 값만 TICK 만큼 감소시킨다. Front 의 remainIO 값이 0 이하가 되면 관련된 값(state 등)을

적절히 업데이트 후 ready queue 로 보낸다. 마지막으로, running process(ready queue 의 front) 정보, ready queue 리스트, io queue 리스트를 TICK 마다 출력한다.

메인 함수에서는 메시지 큐를 생성하고, 프로그램 시작 시간을 기록하고, fork()로 child process 10 개를 생성하고, child 와 parent 별로 맞는 처리를 해준다. Child process 의 경우 init_process 로 process 를 초기화한 후 msgsnd 로 parent 에게 자신의 초기 정보를 전달하고, 반복문을 통해 TICK 마다 parent 로부터 자신의 정보를 전달받게 된다. 메시지를 받다가 만약 자신의 process state 가 3 이라 terminate 상태인 것이 확인되었다면, parent 에게 메시지를 보내 자신의 terminate 사실을 알린다. 마지막으로 parent process 의 경우, analyzeData 형 data 내의 변수 값을 0 으로 초기화하고, TICK 마다 schedule 함수를 실행하도록, Ctrl+C 입력이 들어오면 exitProgram 함수를 실행하도록 한다.

IPC 메시지를 구현할 때 한가지 고려한 점이 있다. Child 와 parent 가 IPC 메시지를 주고받을 때, child→parent 시 메시지 타입은 child 의 pid 로, parent→child 시 메시지 타입은 child 의 pid*2 로 설정하여, 메시지 타입만으로 어떤 child 가 메시지를 보냈는지 parent 가 알 수 있다. parent 의 schedule 함수에서, 메시지 타입 mtype 을 통해 printf("message from %d₩n",msg.mtype)와 같이 활용할 수 있으나, 이렇게 하면 출력값이 더러워지는 관계로 인해 이를 활용하진 않았다. 그래도 child→parent 시 mtype 을 1 로, parent→child 시 mtype 을 2 로 설정하는 경우보단 활용 가능성이 높다는 점에서 의의가 있다.

4. Extra experiment

analyzeData 구조체를 활용해 각 process 가 기록하고 있던 response time, waiting time, turnaround time 을 process 가 terminate 될 때 모두 더한 후 terminate 된 process 의 개수로 나누어 평균 response, waiting, turnaround time 을 구했다. 총 context switching 횟수도 구했다. 이를 통해, time quantum 에 따른 총 context switching 횟수, 평균 response, waiting, turnaround time 을 비교할 수 있게 되었다.

	Quantum 40	Quantum 100	Quantum 160
Running time	125s	72s	70s
Total context switch	1996	843	501
Average response time	108ms	353ms	602ms
Average waiting time	2620ms	50392ms	52324ms
Average turnaround time	116156ms	67338ms	63792ms

전반적으로 time quantum 이 커질수록 running time 은 감소하고, total context switching 횟수는 감소하고, average response time(first running time – arrival time)은 증가하고, average waiting time(ready queue 에서 기다리는 시간)은 증가하고, average turnaround time(terminate time – arrival time)은 감소하는 경향을 보였다. Time quantum 이 작으면 switching 이 자주 일어나서 response time 과 waiting time 이 작다고 생각한다. time quantum 이 작아 CPU dispatch 가 자주 일어나게 되면 running time 과 turnaround time 이 높게 나오는데, 이유는 CPU dispatch 가 자주 일어나게 되면 CPU dispatch 때마다 랜덤하게 결정되는 io task 발생 여부 때문에 io task 의 빈도가 늘어나기 때문이라고 생각한다.