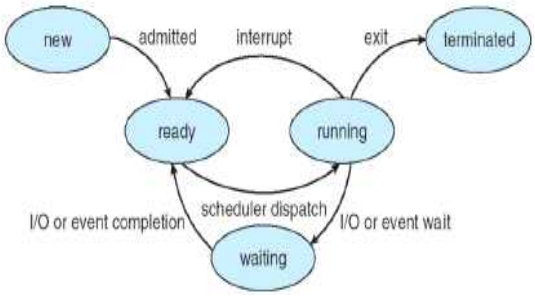
Os Scheduling Simulation

32143153 이대훈
32140301 권태완

Ⅰ. Introduction

프로그램이 실행되면 CPU가 일을 수행하고 I/O 작업요청에 대한 응답을 기다리는 일을 반복합니다. CPU가 일을 수행하는 시간을 CPU burst time이라 하고, I/O 작업요청에 대한 응답을 기다리는 시간을 I/O burst time이라고 합니다.

Scheduling 이란 커널 프로세스가 여러 개의 user 프로세스에 대하여 처리해야 할 일의 순서와 처리 시간을 정해주는 것으로 Round-Robin, FIFO, SJF 등 여러 알고리즘이 있습니다. 커널 프로세스가 user 프로세스를 고르는 것은 현재 실행 중인 프로세스가 어떤 상태들에 진입할 때에 결정이 되며 다음의 경우가 있습니다.



1. Running 상태에서 I/O 요청에 의해 block 되어 Waiting 상태로 변경될 때
2. Running 상태에서 인터럽트에 의해 Ready 상태로 변경될 때
3. Waiting 상태에서 I/O 작업을 마치고 Ready로 변경될 때
4. 프로세스가 작업을 마치고 제거될 때

이런 경우들에 한하여 I/O 요청을 고려하지 않았을 때는 2, 4번에 대한 scheduling을 해주면 되고 I/O 요청을 고려하면 4가지 모두에 대하여 scheduling 해줄 필요가 있습니다. 그리고 이 작업을 커널 프로세스를 parent로 user 프로세스를 child로 설정하여 simulation 하였습니다.

Ⅱ. Motivation

앞서 공부한 스레드는 데이터를 공유하며 이 데이터를 동기화에 신경 써서 사용 하였지만 프로세스끼리는 데이터를 공유할 수가 없습니다. 이에 프로세스들은 메시지 큐, 파이프, 소켓 등을 이용하여 데이터를 전달하게 되는데 이 중에서 메시지 큐를 사용하여 user 프로세스에서 커널 프로세스에 데이터를 전달할 수 있도록 하였습니다. 이 외에도 프로그램이 실행하게 되면 CPU와 I/O가 순차적으로 일을 수행하게 되는데 I/O의 요청 없이 CPU의 일만 고려할 때와 I/O의 작업을 같이 수행했을 때의 차이를 생각하며 두 가지 경우의 simulation을 만들었습니다.

Scheduling 할 때 Round-Robin, FIFO, SJF 등의 알고리즘이 있습니다.

Round-Robin : CPU에서 실행시킬 시간(Time quantum)을 정해놓고 각 user 프로세스에 이 시간 동안 번갈아 실행하도록 하는 것

FIFO(First In First Out) : 먼저 ready 상태로 들어온 user 프로세스에게 기회를 주어 먼저 실행하도록 하는 것

SJF(Shortest Job First) : ready 상태에 있는 user 프로세스 중에 실행시간이 가장 적은 프로세스에 기회를 주어 먼저 실행하도록 하는 것

이 3가지 알고리즘에 따른 Waiting Time(ready 상태에서 기다리는 시간)을 확인해 경우에 따른 효율성을 비교하였습니다.

Ⅲ. Concept

1. 3가지 알고리즘 비교(FIFO, SJF, Round-Robin)

1) FIFO(First In First Out)

=> ready에 먼저 들어온 프로세스 순서대로 실행순서를 부여하는 알고리즘입니다.

+) 구현이 간단합니다.

-) 먼저 들어온 순서대로 burst time이 적으면 좋겠지만 다음 프로세스에 대해 burst time은 알 수 없습니다. 앞의 burst time이 길면 그 뒤에 프로세스들의 waiting time이 길어져 비효율적입니다.

프로세스	p1	p2	p3	p4	p5
burst time	8	12	4	1	2
waiting time	0	8	20	24	21
average waiting time	(0+8+20+24+21) / 5 = 14.6				
burst time	10	10	10	10	10
waiting time	0	10	20	30	40
average waiting time	(0+10+20+30+40) / 5 = 20				

2) SJF(Shortest Job First)

=> ready에 들어온 프로세스들의 실행시간을 비교해서 실행시간이 작은 순서대로 실행순서를 부여하는 알고리즘입니다.

+) burst time이 적은 것이 우선적으로 실행되므로 총 실행시간 면으로 봤을 때 제일 효과적일 수 있습니다.

-) 항상 burst time이 큰 프로세스가 작은 프로세스에게 양보를 해야 하므로 기아 상태가 발생할 수 있습니다.

-) ready 상태에 있는 프로세스들의 요구시간에 대한 데이터를 받기 어렵습니다.

프로세스	p1	p2	p3	p4	p5
burst time	1	2	4	8	12
waiting time	0	1	3	7	15
average waiting time	(0+1+3+7+15) / 5 = 5.2				
burst time	10	10	10	10	10
waiting time	0	10	20	30	40
average waiting time	(0+10+20+30+40) / 5 = 20				

3) Round-Robin
=> Ready에 있는 프로세스들에 균등하게 실행시간(time quantum)을 부여하고 정해진 실행시간 동안 번갈아 가며 실행을 합니다.
+) 모든 프로세스들에 균등한 실행기회를 주어 공정합니다, 짧은 일을 할 때 효율이 더 좋습니다.
-) Context switching이 자주 일어나면 그만큼의 성능저하가 발생합니다.
-) CPU burst time이 모두 동일 할 때, 차이가 별로 없을 때 효율 down.

(time quantum = 4일 때)

프로세스	p1	p2	p3	p4	p5
burst time	8	12	4	1	2
waiting time	11	15	8	12	13
average waiting time	(11+15+8+12+13) / 5 = 11.8				
burst time	10	10	10	10	10
waiting time	32	34	36	38	40
average waiting time	(32+34+36+38+40) / 5 = 36				

2. Only CPU burst

fork -> child : user 프로세스, parent : 커널 프로세스의 역할
fork를 이용하여 child를 생성하고 이 child들의 정보(pid, time quantum, execution time)를 저장시키고 ready 상태로 대기하도록 합니다.
setitimer : Timer tick interval을 설정하여 parent나 child에게 매 tick 마다 SIGALRM을 보내줍니다.
sigaction : 특정 signal을 받을 때마다 행동하는 핸들러를 선언
msgsend : Child 프로세스가 parent 프로세스에 데이터를 전송할 때 사용
msgrcv : Parent 프로세스가 child 프로세스에서 보낸 데이터를 받을 때 사용

* 3가지 알고리즘에 따른 실행

1) Round-Robin
Timer tick interval을 설정하여 매 tick이 지날 때마다 커널은 signal(SIGALRM)을 받고 이에 해당하는 signal handler를 호출합니다. 이 handler가 실행되면 ready의 맨 앞(front)에 있는 user 프로세스에 정해진 time quantum 동안 실행시간을 부여합니다. 이때 정해진 time quantum 동안에 실행이 완료되었다면 ready 상태에서 나와 종료시키고 실행시간이 남아있다면 ready의 맨 뒤로 다시 들어가 실행을 기다립니다. 그리고 이 실행하는 동안에 실행시간이 줄 때마다 child에게 알려 본인의 시간을 계속 확인하도록 하고 실행을 모두 했다면 종료합니다. Ready에 모든 child 들이 terminate 때까지 위 과정을 반복하며 ready에 모든 child 들의 실행이 완료되었다면 parent도 무사히 종료시킵니다.

==> fire 함수로 setitimer를 설정하여 매 tick이 지날 때마다 handler를 호출합니다. 이 handler는 tick을 count하고 ready가 빌 때까지 Round-Robin 함수를 실행합니다.

RR 함수는 ready의 맨 앞에 있는 user 프로세스의 time quantum이 0보다 클 때와 0일 때 두 가지 경우로 나눠서 0보다 클 때는 child에게 신호를 보내 exec_time의 실행을 kill로 알리고 time quantum과 exec_time을 감소시키고 exec_time이 0이 되면 ready에서 빼줍니다. 그리고 time quantum이 0이 되면 다시 초기값으로 설정해주고 ready의 맨 뒤로 보내줍니다. Exec_time의 실행을 child에게 알린다고 했는데 child가 이 signal을 받으면 본인의 exec_time을 감소시키고 0이 되면 스스로 exit 하도록 하였습니다. 이 과정을 ready가 모두 나갈 때까지 반복하고 모두 나간 것이 emptyQueue로 확인이 되면 parent도 exit 합니다.

2) FIFO(First In First Out)

매 tick이 지날 때마다 parent는 signal(SIGALRM)을 받고 이에 해당하는 signal handler를 호출합니다. 이 handler가 실행되면 ready의 맨 앞(head)에 있는 user 프로세스(제일 먼저 들어온 프로세스)에게 실행시간을 부여합니다. 본인의 실행시간이 모두 끝날 때까지 실행하며 실행을 모두 완료하면 ready에서 나가게 됩니다. 그리고 이 실행하는 동안에 실행시간이 줄 때마다 child에게 알려 본인의 시간을 계속 확인하도록 하고 실행을 모두 했다면 종료합니다. 마지막에 들어온 user 프로세스가 실행을 종료할 때까지 위 과정을 반복하며 모두 실행이 완료되었다면 parent도 무사히 종료시킵니다.

=> fire 함수로 setitimer를 설정하여 매 tick이 지날 때마다 handler를 호출합니다. 이 handler는 tick을 count하고 ready가 빌 때까지 FIFO 함수를 실행합니다. FIFO 함수는 ready의 맨 앞에 있는 user 프로세스의 exec_time이 0보다 큰지 우선 확인한 후에 exec_time을 0이 될 때까지 감소시키고 0이 되면 ready에서 빼줍니다. 이때 해당 pid의 child에게 확인 signal을 보내 exec_time의 실행을 kill로 알리고 child는 이 signal을 받으면 본인의 exec_time을 감소시키고 0이 되면 exit 합니다. 이 과정을 ready가 모두 나갈 때까지 반복하고 모두 나간 것이 emptyQueue로 확인이 되면 parent도 exit 합니다.

3) SJF(Shortest Job First)

매 tick이 지날 때마다 parent는 signal(SIGALRM)을 받고 이에 해당하는 signal handler를 호출합니다. 이 handler가 실행되면 ready의 user 프로세스들의 실행시간을 보고 가장 적게 걸리는 user 프로세스에 실행시간을 부여합니다. 본인의 실행시간이 모두 끝날 때까지 실행하며 실행을 모두 완료하면 ready에서 나가게 됩니다. 그리고 이 실행하는 동안에 실행시간이 줄 때마다 child에게 알려 본인의 시간을 계속 확인하도록 하고 실행을 다했다면 종료합니다. 실행시간이 가장 큰 user 프로세스가 실행을 종료할 때까지 위 과정을 반복하며 모두 실행이 완료되었다면 parent도 무사히 종료시킵니다.

=> 우선 SJF는 fork를 할 때 랜덤 값을 받고 오름차순으로 정렬을 한 뒤에 qready에 집어넣어 줍니다. fire 함수로 setitimer를 설정하여 매 tick이 지날 때마다 handler를 호출합니다. 이 handler는 tick을 count하고 ready가 빌 때까지 SJF 함수를 실행합니다. SJF 함수는 ready의 맨 앞에 있는 user 프로세스의 exec_time이 0보다 큰지 우선 확인한 후에 exec_time을 0이 될 때까지 감소시키고 0이 되면 ready에서 terminate 시킵니다. 이때 해당 pid의 child에게 확인 시그널을 보내 exec_time의 실행을 kill로 알리고 child는 이 시그널을 받으면 본인의 exec_time을 감소시키고 0이 되면 exit 합니다. 이 과정을 ready가 모두 나갈 때까지 반복하고 모두 나간 것을 emptyQueue로 확인이 되면 parent도 exit 합니다.

3. CPU burst + I/O burst

child - 실행시간, I/O실행시간 parent의 pid
parent - 실행시간, child의 pid, time quantum, CPU burst time, I/O burst time

*Round-Robin
timer tick interval을 설정하여 매 tick이 지날 때마다 parent는 signal(SIGALRM)을 받고 이에 해당되는 signal handler를 호출합니다. 이 핸들러가 실행되면 ready의 맨 앞(head)에 있는 user 프로세스에게 정해진 time quantum 동안의 CPU burst 실행시간을 부여합니다. 이때 정해진 time quantum 동안에 실행이 다 완료되지 못하면 ready의 맨 뒤로 다시 들어가 실행을 기다리고 time quantum 동안에 실행이 다 완료되면 ready에서 wait 상태로 이동시킵니다. child는 이를 확인하고 있다가 자신의 CPU burst time을 다하면 I/O burst를 시작하기 위해 parent에게 메시지 큐를 이용하여 자신의 데이터를 전송합니다. 그리고 parent에서 이 데이터를 전송받으면 이 데이터를 가지고 I/O burst를 시작하게 되고 I/O burst를 모두 완료하면 wait에서 다시 ready의 맨 뒤로 보내주면서 CPU burst time과 I/O burst time을 새롭게 설정해줍니다. 매 tick 마다 카운트를 해줘 원하는 tick이 될 때까지 이 과정을 무한으로 반복하며 원하는 tick까지 실행을 시켰을 때 실행 중이던 모든 wait과 ready 상태의 child를 종료시키고 parent도 종료시킵니다.

=> RR 함수를 사용하는 부분에 대해서는 CPU burst만 있는 것과 큰 차이가 없었습니다. I/O burst를 하는 과정에서 특별히 메시지 큐를 사용하여 user process의 I/O time을 받아옵니다. 이때 msgsend 와 msgrcv라는 함수를 사용합니다. 여기서 중요한 점은 msgrcv를 언제 하는것인지입니다. msgrcv를 계속적으로 수행할 경우 받아올 메시지가 없는 경우에 대해 bounded하게 되며 이때 기다리는 상태에 빠지게 됩니다. 이런 부분을 해결하기 위해서 msgsend를 하고 이후에 signal을 보내주어 메시지가 보내질 때에만 실행하게 하였습니다.

4. Program Structure

* Only Cpu burst

--FIFO
1. Structure & Global value initialize

2. fork
    2-1. parent
        2-1-1. get data(pid, timequantum, execution time)
        2-1-2. enqueue qready
    2-2. child
        2-2-1. child exit handler
        2-2-2. exit

3. kernel
    3-1 SIGALRM handler
    3-2 check ready queue empty?
    3-3 FIFO
    3-4 check (exec_time ==0)?

--SJF
1. Structure & Global value initialize

2. fork
    2-1. ascending order set
    2-2. parent
        2-1-1. get data(pid, timequantum, execution time)
        2-1-2. enqueue qready
    2-3. child
        2-2-1. child exit handler
        2-2-2. exit
3. kernel
    3-1 SIGALRM handler
    3-2 check ready queue empty?
    3-3 SJF
    3-4 check (exec_time ==0)?

--Round-Robin
1. Structure & Global value initialize

2. fork
    2-1. parent
        2-1-1. get data(pid, timequantum, execution time)
        2-1-2. enqueue qready
    2-2. child
        2-2-1. child exit handler
        2-2-2. exit
3. kernel
    3-1 SIGALRM handler
    3-2 check ready queue empty?
    3-3 RR
    3-4 check (time quantum == 0)?
    3-4 check (exec_time == 0)?

* Cpu burst + I/O burst
1. Structure & Global value initialize

2. fork
    2-1. parent
        2-1-1. get data(pid, timequantum, execution time)
        2-1-2. enqueue qready
    2-2. child
        2-2-1. child exit handler
        2-2-2. check(exec_time == 0)?
        2-2-2. msgsend
3. kernel
    3-1 SIGALRM handler
    3-2 check ready queue empty?
    3-3 do_wait for I/O burst
    3-4 check (I/O burst time == 0)?
    3-5 set new CPU, I/O burst time
    3-6 check (exec_time == 0)?
    3-7 RR
    3-8 check (time quantum == 0)?
    3-9 msgrcv

Ⅳ. Build environment

Compilation : Linux Assam, with GCC
Header file : msg.h, sched.h, queue.h
To compile : gcc basic_FIFO.c, gcc basic_SJF.c, gcc basic_RR.c, gcc Final.c
To run : ./a.out
Output : schedule_dump.txt

Ⅳ. Problem & Solution

* Only CPU burst

1. problem) tick이 지날 때마다 parent(커널)에서 SIGALRM을 받고 이 SIGALRM을 받으면 다시 child(user 프로세서)한테 signal을 보내 child에서 CPU burst time을 줄이는 일을 수행합니다. 하지만 parent(커널)에서는 user process의 CPU burst time을 알 수 없습니다. 커널에서 해당 user의 CPU burst time을 알지 못한다면 user가 끝났을 때 ready queue에서 내보내 terminate을 시켜야 하는 순간을 알 수 없습니다.
☞ sol) 커널이 tick을 받을 때마다 child에게 신호를 보내주고 그에 해당하는 child는 CPU burst time을 줄이는 일을 수행하도록 하였습니다. 이때 parent가 처음 fork 할 때 child의 CPU burst time을 가지고 들어가 같은 tick에 대해 커널에서도 같이 줄입니다. 이를 통해 서로 같은 값을 유지하며 커널에서도 user process의 CPU burst를 알 수 있습니다. 이와 다른 방법으로는 메시지 큐를 사용하는 방법이 있습니다. 메시지 큐를 사용하여 커널에 user process의 CPU busrt time을 알려 줄 수도 있습니다. 또는 user process의 CPU burst time이 0이 되는 때에 커널에 signal을

보내주어 ready queue에 있는 user process를 terminate 시킬 수도 있습니다.

2. problem) parent에서 CPU burst time이 0이 되면 child에 signal을 보내 해당 child를 exit 합니다.
☞ sol) Child의 종료에 관한 것은 child 스스로가 자신의 burst time을 알고 burst time이 0이 될 때 종료하도록 합니다. 이때 parent에서 tick을 받을 때마다 child에게 신호를 주어 CPU burst time을 줄이도록 합니다.

3. problem) Parent가 끝나는 조건을 처음에는 tick이 정해진 횟수만큼 시행하면 종료하도록 하였습니다. 이런 경우 user process가 종료하지 못하고 커널이 종료되는 경우가 생깁니다.
☞ sol) Ready queue가 empty일 때 커널을 exit 합니다. Ready queue가 empty라는 것은 모든 user process가 자신이 가진 CPU burst time을 모두 마치고 종료했다는 것을 의미합니다. 이런 경우에 대해 커널을 종료하였습니다.

4. problem) Round-Robin 함수에서 if문 조건을 time quantum이 0보다 클 때와 0과 같을 때 두 가지로 나누었습니다. if와 else if를 사용하여 나누었을 때 time quantum은 0이 되었지만 CPU burst time이 아직 남아있는 경우에 대한 처리에 오류가 났습니다.
☞ sol) else if의 경우를 if 문으로 바꾸어 주고 위의 if 문을 지나서 다시 한번 조건을 time quantum에 대한 조건을 검사하게 하였습니다. 하지만 위의 조건에 대해 CPU burst time이 0일 경우에 ready queue에서 user process를 terminate 하고 return 하여 아래의 if 문을 실행하지 않았습니다.

5. problem) CPU burst time을 임의로 받아주기 위해 rand() 함수를 사용하였습니다. 이때 임의의 값이 너무 크지 않도록 rand() % n을 하면 난수는 0부터 n-1까지의 수가 나왔습니다.
☞ sol) 처음에는 user process의 CPU burst time이 0인 것에 대한 예외처리를 해주었습니다. 하지만 CPU burst time이 처음부터 0이라는 부분은 하나의 process 자체가 의미가 없는 작업이므로 난수를 1부터 n까지 나올 수 있도록 (rand() % n) + 1을 해주었습니다.

6. problem) Queue에 PCB 데이터를 집어넣기 위해서 structure 자체를 queue에 넣는 방법으로 Queue ADT를 바꿨습니다. 이때 queue에 들어간 값을 나타내기 위해 포인터 자체를 structure로 받아야 하며 그 안에서의 값을 빼기 위해 다시 간접적으로 나타냅니다.
☞ sol) PCB structure 자체를 queue에 넣는 대신 PCB의 index를 queue에 넣는 방식으로 바꿔 queue adt를 그대로 사용하였습니다. 이런 경우 queue에서 값을 가지고 올 때 int 변수를 선언하여 ptr = queue->front->data라고 선언하였습니다.

7. problem) Child에서 exit을 하는 경우 htop에 parent 프로세스 하나만 생성되어 실행하는 것을 확인하였습니다. 또는 child가 exit을 하지 않고 parent가 exit을 하면 htop에 좀비 프로세스가 남습니다.
☞ sol) Child에 while을 사용하여 user process가 계속 실행하게 하면 htop에 parent process까지 포함한 11개 프로세스가 동작하는 것을 볼 수 있습니다. 또한, htop를 통해 커널이 종료되었을 때 모든 user process 또한 정상적으로 종료하였는지를 확인하였습니다.

8. problem) User process의 CPU burst time을 배열로 만들어 fork() 할 때 child에서 배열을 차례로 넣어주었습니다. 이런 경우 각각의 child에 대해 출력을 하면 모두 배열의 처음 값을 출력합니다.

☞ sol) 함수 fork()를 사용하면 child process는 parent process의 모든 값을 가지고 생겨납니다. 배열로 집어넣는 것은 해당 child마다 배열이 있고 그중에 한 index에만 execution time이 들어 있다는 뜻으로 잘못된 방법입니다. 글로벌 변수를 설정하여 fork() 할 때 CPU burst time을 설정해 주고 parent에서 집어 넣어주는 방식으로 했습니다.

9. problem) Ready queue에 처음 실행하는 값에 대해 Dequeue를 하고 이후 작업을 할 때 tick이 매우 빠른 속도로 신호를 보낸다면 작업이 마치기 전에 Dequeue가 다시 이뤄지고 이에 대해 작업하던 PCB가 바뀌는 현상이 발생합니다.
☞ sol) 이러한 부분을 해결하기 위해 작업을 진행하는 과정에서 ready queue에서 값을 Dequeue 시키지 않고 다른 변수로 값을 받아온 후 마지막에 Enqueue를 진행하기 바로 전에 Dequeue를 수행해 줍니다.

10. problem) Kill로 sigaction 함수에 신호를 주는 과정에서 kill에 들어가는 변수에 대해서 많은 어려움을 경험하였습니다. 어떤 process id가 들어가야 하며 뒤에 주어지는 신호는 어떤 것을 사용해야 하는지에 대해 이해하는 부분에 많은 시간을 투자하였습니다.
☞ sol) Kill 함수에 들어가는 process id는 sigaction을 수행하고자 하는 process의 id입니다. 또한 SIGUSR를 사용하여 사용자가 기능을 지정할 수 있도록 하였습니다.

* CPU burst + I/O burst

1. problem) CPU와 I/O가 무한으로 돌기 때문에 원하는 tick에서 종료하도록 tick을 세주고 해당 tick이 되면 종료 커널을 종료합니다.
☞ sol) User process가 terminate 하지 않고 반복적으로 사용되기 때문에 원하는 tick에 종료하도록 하였습니다. 하지만 user process들은 terminate 되지 않기 때문에 좀비 프로세스로 남게 됩니다. 이를 방지하기 위해 커널이 종료하기 전 모든 user process에 종료 신호인 SIGINT를 보내줍니다.

2. problem) CPU burst time이 종료되지 않는 경우 이때 주어진 time quantum이 종료되면 해당 user process는 다시 ready queue로 돌아갑니다. 이와 동시에 I/O burst time이 종료된 user process도 wait queue에서 ready queue로 들어갑니다.
☞ sol) 둘 중 어떠한 user process를 먼저 ready queue에 넣어주는 것은 개발자의 몫입니다. 하지만 Round-Robin algorithm은 모든 user process에 대해 균등한 기회를 주기 위해 만들어진 알고리즘입니다. 이러한 부분을 생각해서 ready queue에서 기회를 받고 다시 들어가는 user process보다 I/O 작업이 끝나 돌아온 user process에 대해 먼저 CPU 작업을 할 수 있도록 우선권을 주었습니다.

3. problem) Sigaction의 사용에 대해 for 문 안에서 지속적으로 선언을 하였습니다. 이런 경우 문제가 생기진 않지만 선언을 불필요하게 많이 해 오버헤드가 커집니다.
☞ sol) Sigaction에 대한 선언은 한 번만 하면 됩니다. Sigaction은 signal이 어떠한 작업을 수행하는지에 대한 선언이지 신호를 받았을 때 여기서 실행하겠다는 함수가 아닙니다. 따라서 kill 함수를 사용하는 위치가 중요합니다.

4. problem) I/O burst가 진행 중인 경우에는 wait queue에 있는 모든 PCB에 대해서 tick마다 작

업을 해야합니다. 이때 for 문을 사용하여 wait queue에 있는 모든 PCB의 I/O time에 대해 작업을 하였으며 segmentation error가 발생했습니다.
☞ sol) for 문을 돌릴 때 wait queue에 들어 있는 값의 수가 중요합니다. Wait queue에 7개만 존재하고 있을 때 for 문을 10번 돌리면 segmentation error가 발생합니다. 이런 점을 해결하기 위해 for 문이 wait queue의 count보다 적게 수행하도록 하였습니다.

5. problem) tick이 들어오는 시간이 짧아지면 scheduling의 안정성을 확보해주는 부분에서 어려움이 발생합니다. 함수가 실행 중에 함수에 대한 값이 변하거나 오류가 날수도 있기 때문입니다. msgrcv를 할 때 이러한 문제점이 발생하였습니다. Tick이 매우 짧은 시간 단위로 들어올 때 ipcs를 해보면 message가 남아있었습니다. 이런 경우 msgrcv가 실패하여 -1을 return하고 다시 수행하지 못하였습니다.
☞ sol) 이러한 문제를 해결하기 위해 처음 선언에 대해 ret를 -1로 선언하여 msgrcv의 return 값이 -1과 같으면 while 문을 계속 진행하도록 하였습니다. msgrcv 성공하여 0을 return 하면 while 문을 빠져나오도록 하였습니다.

6. problem) Loop이 없는 부분에 대해서 출력할 때 같은 값이 다시 출력되는 경우가 발생하였습니다.
☞ sol) Kill 위치를 printf 보다 아래로 내려 줌으로써 문제를 해결하였습니다. 하지만 이 문제에 대해 정확한 이유는 알 수 없었습니다.

7. problem) 메시지를 받아 와서 받아온 메시지에 대해 PCB에 저장할 때 ready queue의 front에 있는 PCB에 저장하도록 하였습니다. 이런 경우 PCB가 가지고 있는 pid와 msg에서 가지고 있는 pid를 먼저 비교하고 같은 경우에만 값을 받아오게 하였습니다.
☞ sol) 이런 경우 tick이 짧은 시간에 들어오면 비교하는 과정에서 PCB가 다음 PCB로 넘어갈 수 있기 때문에 msg로부터 값을 받아오지 못하는 경우가 생깁니다. 이런 점을 보완하기 위해 for 문을 사용하여 모든 PCB와 확인을 하였습니다.

8. problem) 메시지 큐를 사용하기 위해 예제 코드를 사용하여 실험했을 때 넣은 값과는 다른 값이 받아졌습니다.
☞ sol) key 값을 통해 메시지 큐를 구분하는데 이 부분에 대하여 같은 key 값으로 연습하는 경우 다른 사람이 넣은 값을 받아 올 수 있습니다. key 값을 개인이 사용하는 값으로 입력 후에 문제가 해결되었습니다.

Ⅴ. Personal Feelings

1 player - 지난 homework에서 스레드에 관한 주제로 공부를 했다면 이번 프로젝트에서는 좀 더 큰 실행 단위인 프로세스를 주제로 하여 연구를 해보는 시간을 가졌습니다. 스레드는 서로 데이터를 공유할 수 있기 때문에 일어날 수 있는 문제점들을 생각해 볼 필요가 있었고, 프로세스는 서로 데이터를 공유할 수 없기 때문에 매개체를 통해 데이터를 전달할 필요가 있었습니다. 이러한 전달 방법들로 파이프, 소켓, 메시지큐 등이 있는데 그 중에 메시지큐에 관해서 공부하고 사용하게 되었습니다. 운영체제의 실행 단위들인 스레드와 프로세스가 처음에는 그저 비슷하게 느껴졌지만 엄연히 다른 실행방식을 가지고 있는 단위체들임을 알 수 있었습니다. 연속된 과제로 인하여 집에 갈 타이밍도 놓치고 피로가 계속 쌓여 힘들었지만 이번 프로젝트가 끝나면 잠깐이나마 쉴 수 있다는 생각에 설레고 행복해지는 것 같습니다. 몇 일 후면 또다시 다음 프로젝트와 마주하고 있을 것이지만 언제나 그랬듯이 힘들지만 힘내서 해보겠습니다.

2 player - 처음 시그널에 대한 공부를 할 때 많은 시간이 필요로 하였습니다. 또한, 안정성에 대해서 signal을 사용하는 것이 좋은 방법은 아니라고 배웠습니다. 이러한 문제점 때문에 signal을 짧게 주면 먼저 수행하던 부분이 끝나지 않는 점 때문에 error가 발생하였습니다. 이러한 부분을 해결하고 안정성을 높이며 시간을 줄여보기 위해 많은 시간을 들여 코드에서 필요 없는 부분을 삭제하고 여러 번의 실험 끝에 Final.c를 30Microsecond까지 줄여보았습니다.

Ⅶ. Code

<Header file>

1. msg.h

```
struct msgbuf {
        int mtype;

        // pid will sleep for io_time
        int pid;
        int io_time;
        int cpu_time;
};
```

2. sched.h

```
#include <signal.h>
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAXPROC 10
#define DEFAULT_TQ 3

void signal_handler(int signo);
void signal_handler2(int signo);
void signal_handler3(int signo);
void order();
void FIFO();
void SJF();
void RR();
void do_wait();
int tick;
void fire(int interval_sec);
int msgsend();
int msgrecieve();

// only for parent!
typedef struct pcb_{
        int pid;
        int remaining_tq;
        int exec_time;
        int cpu_time;
        int IO_time;
}pcb_t;


// only for child!
typedef struct child{
        int exec_time;
        int IO;
        int ppid;
}user;
```

3. queue.h

```
#include <signal.h>
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/wait.h>
```

```
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAXPROC 10
#define DEFAULT_TQ 3

void signal_handler(int signo);
void signal_handler2(int signo);
void signal_handler3(int signo);
void order();
void FIFO();
void SJF();
void RR();
void do_wait();
int tick;
void fire(int interval_sec);
int msgsend();
int msgrecieve();

// only for parent!
typedef struct pcb_{
        int pid;
        int remaining_tq;
        int exec_time;
        int cpu_time;
        int IO_time;
}pcb_t;


// only for child!
typedef struct child{
        int exec_time;
        int IO;
        int ppid;
}user;
daehoon14@assam:~/os_2018/2018_os_proj1$ cat queue.h
typedef struct node{
        int data;
        struct node* next;
} QUEUE_NODE;

typedef struct
{
```

```
        QUEUE_NODE* front;
        QUEUE_NODE* rear;
        int count;
} QUEUE;

QUEUE* CreateQueue(void){

        QUEUE* pNewQueue=(QUEUE*)malloc(sizeof(QUEUE));
        if(pNewQueue==NULL)
                return NULL;

        pNewQueue->count=0;
        pNewQueue->front=pNewQueue->rear=NULL;

        return pNewQueue;
};

void Enqueue(QUEUE* pQueue, int item){

        QUEUE_NODE* Newptr=(QUEUE_NODE*)malloc(sizeof(QUEUE_NODE));
        if (Newptr==NULL)
                return ;

        Newptr->data = item;
        Newptr->next = NULL;

        if(pQueue->count == 0){
                pQueue->front=pQueue->rear=Newptr;
        }
        else{
                pQueue->rear->next=Newptr;
                pQueue->rear=Newptr;
        }

        pQueue->count++;
};

int Dequeue(QUEUE* pQueue){

        QUEUE_NODE* delLoc=NULL;
        int item;

        if(pQueue->count==0)
                return 0;
```

```
        delLoc=pQueue->front;
        item=delLoc->data;

        if(pQueue->count==1){
                pQueue->front=pQueue->rear=NULL;
        }
        else{
                pQueue->front=delLoc->next;
        }

        free(delLoc);
        pQueue->count--;

        return item;
};

int emptyQueue(QUEUE *queue)
{
        return(queue->count == 0);
}

QUEUE *DestroyQueue(QUEUE *queue)
{
        QUEUE_NODE *deletePtr;

        if(queue)
        {
                while(queue->front != NULL)
                {
                queue->front->data = 0;
                deletePtr = queue->front;
                queue->front = queue->front->next;
                free(deletePtr);
                }
        free(queue);
        }
return NULL;
}
```

<Source file>

1. basic_FIFO.c

```
#include "sched.h"
#include "queue.h" // queue fuction
```

```
#include "msg.h" // msg structure

pcb_t PCB[MAXPROC]; // only for parent
int exec_time; // only for child

QUEUE *qwait;
QUEUE *qready;
int ptr;
int wait_time[MAXPROC];
float aver_time;

int main()
{
	struct sigaction old_sa;
	struct sigaction new_sa;
	memset(&new_sa, 0, sizeof(new_sa));
	qwait = CreateQueue(); // create wait queue
	qready = CreateQueue(); // create ready queue
	srand((unsigned)time(NULL));

	pid_t pid;
	for(int k=0; k<MAXPROC; k++)
	{
		exec_time = (rand() % 15) + 1; // get the exec_time for process randomly
		pid=fork();
		if(pid<0){
			perror("fork fail");
		}
		else if(pid == 0){//child
			new_sa.sa_handler = &signal_handler2;
			sigaction(SIGUSR1, &new_sa, &old_sa); // set SIGUSR1 signal as new_sa function
			while(1);
			exit(0);
		}
		else{//parent
			PCB[k].pid = pid;
			PCB[k].remaining_tq = DEFAULT_TQ;
			PCB[k].cpu_time = exec_time;
			PCB[k].exec_time = exec_time;
			Enqueue(qready, k); // put the index of PCB in the ready queue
		}
	}
	new_sa.sa_handler = &signal_handler;
```

```
	sigaction(SIGALRM, &new_sa, &old_sa); // set SIGALRM signal as new_sa function
	fire(1); // give siganl to SIGALRM to work
	while (1);
}

void fire(int interval_sec)
{
	struct itimerval new_itimer, old_itimer;
	new_itimer.it_interval.tv_sec = 0;
	new_itimer.it_interval.tv_usec = 10; // give 10Microsecond for signal
	new_itimer.it_value.tv_sec = 0; //interval_sec;
	new_itimer.it_value.tv_usec = 10;
	setitimer(ITIMER_REAL, &new_itimer, &old_itimer);
}

void FIFO() // First In First Out algorithm
{
	ptr = qready->front->data; // get the first data of ready queue
	if(PCB[ptr].exec_time > 0)
	{
		PCB[ptr].exec_time--;
		printf("Tick : %d Proc[%d].exe : %d\n", tick, ptr, PCB[ptr].exec_time);
		kill(PCB[ptr].pid, SIGUSR1);
		if(PCB[ptr].exec_time == 0)
		{
			Dequeue(qready); // user process works done then terminate
			wait_time[ptr] = tick - PCB[ptr].cpu_time; // how long does it
take for waiting
		}
	}
}

void signal_handler(int signo)
{
	tick++; // count the tick
	if(emptyQueue(qready)) // ready queue is empty then kernal has nothing to work
so exit
	{
		DestroyQueue(qready); // destroy ready queue
		DestroyQueue(qwait); // destroy wait queue
		for(int k=0; k<MAXPROC; k++)
		{
			aver_time += wait_time[k]; // get the average waiting time for
process
		}
```

```
                aver_time = aver_time/MAXPROC;
                printf("Average waiting time : %f microseconds\n", aver_time);
                exit(0);
        }
        FIFO();
}

void signal_handler2(int signo)
{
        exec_time--;
        if(exec_time == 0) // user process works done then exit
        {
                exit(0);
        }
}
```

2. basic_SJF.c

```
#include "sched.h"
#include "queue.h" // queue fuction
#include "msg.h" // msg structure

pcb_t PCB[MAXPROC]; // only for parent
int exec_time = 0; // only for child
int set[MAXPROC]; // get the exec_time

QUEUE *qwait;
QUEUE *qready;
int ptr;
int wait_time[MAXPROC];
float aver_time;

int main()
{
        struct sigaction old_sa;
        struct sigaction new_sa;
        memset(&new_sa, 0, sizeof(new_sa));
        qwait = CreateQueue(); // create wait queue
        qready = CreateQueue(); // create ready queue
```

```
	srand((unsigned)time(NULL));
	for(int k=0; k<MAXPROC; k++)
	{
		set[k] = (rand() % 15) + 1; // get the set randomly
	}
	order(); // ascending order the set for the exec_time
	pid_t pid;
	for(int k=0; k<MAXPROC; k++)
	{
		exec_time = set[k]; // get the exec_time for set that already ascending order
		pid=fork();
		if(pid<0){
			perror("fork fail");
		}
		else if(pid == 0){//child
			new_sa.sa_handler = &signal_handler2;
			sigaction(SIGUSR1, &new_sa, &old_sa); // set SIGUSR1 signal as new_sa function
			while(1);
			exit(0);
		}
		else{//parent
			PCB[k].pid = pid;
			PCB[k].remaining_tq = DEFAULT_TQ;
			PCB[k].cpu_time = exec_time;
			PCB[k].exec_time = exec_time;
			Enqueue(qready, k); // put the index of PCB in the ready queue
		}
	}
	new_sa.sa_handler = &signal_handler;
	sigaction(SIGALRM, &new_sa, &old_sa); // set SIGALRM signal as new_sa function
	fire(1); // give siganl to SIGALRM to work
	while (1);
}

void order()
{
	int temp;
	int j,k;

	for(j=0; j<MAXPROC; j++)
	{
		for(k=0; k<MAXPROC-1; k++)
		{
```

```
                        if(set[k] > set[k+1])
                        {
                                temp = set[k];
                                set[k] = set[k+1];
                                set[k+1] = temp;
                        }
                }
        }
}

void fire(int interval_sec)
{
        struct itimerval new_itimer, old_itimer;
        new_itimer.it_interval.tv_sec = 0;
        new_itimer.it_interval.tv_usec = 10; // give 10Microsecond for signal
        new_itimer.it_value.tv_sec = 0;//interval_sec;
        new_itimer.it_value.tv_usec = 10;
        setitimer(ITIMER_REAL, &new_itimer, &old_itimer);
}

void SJF() // Short Job First algorithm
{
        ptr = qready->front->data; // get the first data of ready queue
        if(PCB[ptr].exec_time > 0)
        {
                PCB[ptr].exec_time--;
                printf("Tick : %d Proc[%d].exe : %d\n", tick, ptr, PCB[ptr].exec_time);
                kill(PCB[ptr].pid, SIGUSR1);
                if(PCB[ptr].exec_time == 0)
                {
                        Dequeue(qready); // user process works done then terminate
                        wait_time[ptr] = tick - PCB[ptr].cpu_time; // how long does it
take for waiting
                }
        }
}

void signal_handler(int signo)
{
        tick++; // count the tick
        if(emptyQueue(qready)) // ready queue is empty then kernal has nothing to work
so exit
        {
                DestroyQueue(qready); // destroy ready queue
                DestroyQueue(qwait); // destroy wait queue
```

```
                for(int k=0; k<MAXPROC; k++)
                {
                        aver_time += wait_time[k]; // get the average waiting time for
process
                }
                aver_time = aver_time/MAXPROC;
                printf("Average waiting time : %f microseconds\n", aver_time);
                exit(0);
        }
        SJF();
}

void signal_handler2(int signo)
{
        exec_time--;
        //printf("child exe : %d\n", exec_time);
        if(exec_time == 0)
        {
                exit(0);
        }
}
```

3. basic_RR.c

```
#include "sched.h"
#include "queue.h" // queue fuction
#include "msg.h" // msg structure

pcb_t PCB[MAXPROC]; // only for parent
int exec_time; // only for child

QUEUE *qwait;
QUEUE *qready;
int ptr;
int wait_time[MAXPROC];
float aver_time;

int main()
{
```

```
	struct sigaction old_sa;
	struct sigaction new_sa;
	memset(&new_sa, 0, sizeof(new_sa));
	qwait = CreateQueue(); // create wait queue
	qready = CreateQueue(); // create ready queue
	srand((unsigned)time(NULL));

	pid_t pid;
	for(int k=0; k<MAXPROC; k++)
	{
		exec_time = (rand() % 15) + 1; // get the exec_time for process randomly
		pid=fork();
		if(pid<0){
			perror("fork fail");
		}
		else if(pid == 0){//child
			new_sa.sa_handler = &signal_handler2;
			sigaction(SIGUSR1, &new_sa, &old_sa); // set SIGUSR1 signal as new_sa function
			while(1);
			exit(0);
		}
		else{//parent
			PCB[k].pid = pid;
			PCB[k].remaining_tq = DEFAULT_TQ;
			PCB[k].cpu_time = exec_time;
			PCB[k].exec_time = exec_time;
			Enqueue(qready, k); // put the index of PCB in the ready queue
		}
	}
	new_sa.sa_handler = &signal_handler;
	sigaction(SIGALRM, &new_sa, &old_sa); // set SIGALRM signal as new_sa function
	fire(1); // give siganl to SIGALRM to work
	while (1);
}

void fire(int interval_sec)
{
	struct itimerval new_itimer, old_itimer;
	new_itimer.it_interval.tv_sec = 0;
	new_itimer.it_interval.tv_usec = 10; // give 10Microsecond for signal
	new_itimer.it_value.tv_sec = 0; //interval_sec;
	new_itimer.it_value.tv_usec = 10;
	setitimer(ITIMER_REAL, &new_itimer, &old_itimer);
```

```
}

void RR() // Round Robin algorithm
{
	ptr = qready->front->data; // get the first data of ready queue
	if(PCB[ptr].remaining_tq > 0)
	{
		kill(PCB[ptr].pid, SIGUSR1); // give signal to SIGUSR1 to work
		PCB[ptr].remaining_tq--;
		PCB[ptr].exec_time--;
		printf("Tick : %d TQ : %d PCB[%d].exec_time : %d\n", tick,
PCB[ptr].remaining_tq, ptr, PCB[ptr].exec_time);
		if(PCB[ptr].exec_time == 0)
		{
			Dequeue(qready); // user process works done then terminate
			wait_time[ptr] = tick - PCB[ptr].cpu_time; // how long does it
take for waiting
			return;
		}
	}
	if(PCB[ptr].remaining_tq == 0) // user process need to work more
	{
			PCB[ptr].remaining_tq = DEFAULT_TQ; // give another time to do
work
			Enqueue(qready, Dequeue(qready)); // put it back of ready queue
	}
}

void signal_handler(int signo)
{
	tick++; // count the tick
	if(emptyQueue(qready)) // ready queue is empty then kernal has nothing to work
so exit
	{
		DestroyQueue(qready); // destroy ready queue
		DestroyQueue(qwait); // destroy wait queue
		for(int k=0; k<MAXPROC; k++)
		{
			aver_time += wait_time[k]; // get the average waiting time for
process
		}
		aver_time = aver_time/MAXPROC;
		printf("Average waiting time : %f microseconds\n", aver_time);
		exit(0);
	}
```

```
        RR();
}

void signal_handler2(int signo)
{
        exec_time--;
        if(exec_time == 0) // user process works done then exit
        {
                exit(0);
        }
}
```

4. Final.c

```
#include "sched.h"
#include "queue.h"
#include "msg.h"

QUEUE *qrun;
QUEUE *qwait;
QUEUE *qready;
int ptr;
int wptr;
int msgq;
int key = 0x32143153;

struct sigaction old_sa;
struct sigaction new_sa;
struct msgbuf msg;
pcb_t PCB[MAXPROC];
user uProc;

int main()
{
        memset(&new_sa, 0, sizeof(new_sa));
        msgq = msgget( key, IPC_CREAT | 0666);
        memset(&msg, 0, sizeof(msg));
        qwait = CreateQueue();
        qready = CreateQueue();
        srand((unsigned)time(NULL));
        pid_t pid;

        for(int k=0; k<MAXPROC; k++)
        {
                uProc.exec_time = (rand() % 10) + 1;
```

```
                uProc.ppid = getpid();
                uProc.IO = (rand() % 10) + 1;
                pid=fork();
                if(pid<0){
                        perror("fork fail");
                }
                else if(pid == 0){//child
                        new_sa.sa_handler = &signal_handler2;
                        sigaction(SIGUSR1, &new_sa, &old_sa);
                        while(1);
                        exit(0);
                }
                else{//parent
                        PCB[k].pid = pid;
                        PCB[k].remaining_tq = DEFAULT_TQ;
                        PCB[k].exec_time = uProc.exec_time;
                }
                Enqueue(qready, k);
        }

        new_sa.sa_handler = &signal_handler;
        sigaction(SIGALRM, &new_sa, &old_sa);
        new_sa.sa_handler = &signal_handler3;
        sigaction(SIGUSR2, &new_sa, &old_sa);
        fire(1);

        while(1);
}

void RR()
{
        if(emptyQueue(qready))return;
        else{
                ptr = qready->front->data;
                if(PCB[ptr].remaining_tq > 0)
                {
                        PCB[ptr].remaining_tq--;
                        PCB[ptr].exec_time--;
                        printf("Tick : %d, PID : %d\tPCB[%d].exec_time : %d\n", tick,
PCB[ptr].pid, ptr, PCB[ptr].exec_time);
                        kill(PCB[ptr].pid, SIGUSR1);
                        if(PCB[ptr].exec_time == 0)
                        {
                                PCB[ptr].remaining_tq = DEFAULT_TQ;
                                Enqueue(qwait, Dequeue(qready));
```

```
                                return;
                        }
                }
                if(PCB[ptr].remaining_tq == 0)
                {
                        PCB[ptr].remaining_tq = DEFAULT_TQ;
                        Enqueue(qready, Dequeue(qready));
                        return;
                }
        }
        return;
}

void do_wait()
{
        if(emptyQueue(qwait))
        {
                return;
        }
        else
        {
                for(int k=0; k<qwait->count; k++)
                {
                        wptr = qwait->front->data;
                        if(PCB[wptr].IO_time > 0)
                        {
                                PCB[wptr].IO_time--;
                                //printf("PCB[%d].IOtime : %d\n", wptr, PCB[wptr].IO_time);
                                if(PCB[wptr].IO_time == 0)
                                {
                                        Enqueue(qready, Dequeue(qwait));
                                }
                                else if(PCB[wptr].IO_time > 0)
                                {
                                        Enqueue(qwait, Dequeue(qwait));
                                }
                        }
                }
        }
}

void signal_handler(int signo)
{
        tick++;
        if(tick == 10000)
```

```
		{
			for(int k=0; k<MAXPROC; k++)
			{
				kill(PCB[k].pid, SIGINT);
			}
			DestroyQueue(qready);
			DestroyQueue(qwait);
			printf("Tick : %d, OS exit\n", tick);
			exit(0);
		}
		do_wait();
		RR();
}

void signal_handler2(int signo)
{
		uProc.exec_time--;
		if(uProc.exec_time == 0)
		{
			uProc.exec_time = (rand() % 10) + 1;
			msgsend();
			kill(uProc.ppid, SIGUSR2);
		}
}

void signal_handler3(int signo)
{
		msgrecieve();
		for(int k=0; k<MAXPROC; k++)
		{
			if(PCB[k].pid == msg.pid)
			{
				PCB[k].IO_time = msg.io_time;
				PCB[k].exec_time = msg.cpu_time;
				//printf("mother CPU : %d IO : %d, index : %d\n",
PCB[k].exec_time, PCB[k].IO_time, k);
			}
		}
}

int msgsend()
{
		int ret = -1;
		//struct msgbuf msg;
		msg.mtype = 0;
```

```
        msg.pid = getpid();
        msg.io_time = uProc.IO;
        msg.cpu_time = uProc.exec_time;
        while(ret == -1){
        ret = msgsnd(msgq, &msg, sizeof(msg), 0);}
        return 0;
}

int msgrecieve()
{
        int ret = -1;
        //struct msgbuf msg;
        while(ret == -1)
        {
                ret = msgrcv(msgq, &msg, sizeof(msg), 0, MSG_NOERROR);
        }
        return 0;
}

void fire(int interval_sec)
{
        struct itimerval new_itimer, old_itimer;
        new_itimer.it_interval.tv_sec = 0;
        new_itimer.it_interval.tv_usec = 100;
        new_itimer.it_value.tv_sec = 0;//interval_sec;
        new_itimer.it_value.tv_usec = 100;
        setitimer(ITIMER_REAL, &new_itimer, &old_itimer);
}
```