Implement my own shell report – 과제3

201521033 조경선

- Work description

본 과제는 User-level threading을 구현하고 라이브러리 형태로 1:n thread model을 구현하는 것이 목적이다. Scheduling으로는 First-in First-out과 Round robin을 구현한다.

일단 main 함수에서 가장 먼저 나오는 함수인 mythread_init함수는 scheduling 방식을 입력 받아서 tick에 대한 timer signal이 구현된다. 이는 sigaction 함수와 itimerval 함수를 이용하여 구현했다.

```
76  void mythread_init(enum mythread_scheduling_policy policy)
77  {
78    sched_policy = policy;
79
80    ticker.sa_handler = &tick; //when signal being called tick is exec
81
82    sigemptyset(&ticker.sa_mask);
83    ticker.sa_flags = 0;
84    ticker.sa_handler = &tick;
85
86    sigaction(SIGALRM, &ticker, NULL);
87
88    time_quantum.it_interval.tv_sec = 0;
89    time_quantum.it_value.tv_sec = 0;
90    time_quantum.it_interval.tv_usec = 40000;
91    time_quantum.it_value.tv_usec = 40000; // expire time
92
93
94    if(setitimer(ITIMER_REAL, &time_quantum, NULL)==-1){
95      printf("ERROR: setitimer\n");
96    }
97
98    //while(1);
99    //for tick
```

그림 1 mythread_init 함수

그림 1에서 보는 것처럼 먼저 sigaction ticker에 대한 설정을 해주고 handler에서 알람이 울릴 때마다 tick의 함수가 불려 지게 설정한다. 그리고 time_quantnum의 구조체를 이용하여 시간을 설정해준다. 이렇게 설정된 timer는 setitimer를 통해 설정해준다. 여기서 본인의 역량 부족으로 tick의 함수는 timer에 따라 불렀지만, while(1)을 주석 처리하면 tick 함수에서 빠져나오지 못하는 상황이 발생한다.

Tick 함수의 경우 nextTcb()의 함수를 불러 Tcb의 tid를 불러서 scheduling 방식마다 scheduling을 해주는 영역이다. 이에 더해 nextTcb()도 scheduling방식에 따라 time quantum이 지났을 때 다음으로 불러야 하는 thread가 무엇인지 알려주는 함수이다. 이 두 함수는 본인의 역량 부족으로 구현해내지 못했다.

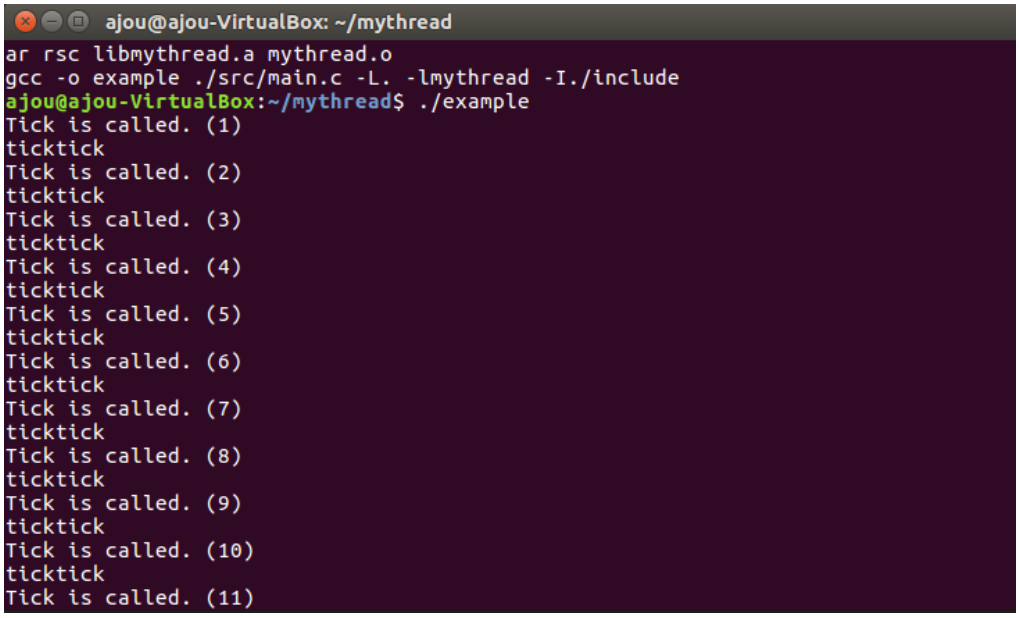

그림 2 timer 작동

그림 2은 timer가 작동하고 있다는 것을 보여준다. 하지만 while(1)을 없애는 순간 작동하지 않고 다음 라인인 mythread_create로 넘어간다.

```
121  int mythread_create(void (*stub)(void*), void* args)
122  {
123    int tid = -1; // Thread ID
124
125    printf("Thread create\n");
126    tcbs[n_tcbs] = (struct tcb*)malloc(sizeof(struct tcb*));
127    current_thread_id = n_tcbs;
128
129    printf("current id: %d\n",current_thread_id);
130
131    if(getcontext(&tcbs[current_thread_id]->ucp)==-1){
132      printf("ERROR: getcontext\n");
133    }
134    tcbs[n_tcbs]->status = ready;
135
136    tcbs[n_tcbs]->thread_id = rand() % 100;
137    tid = tcbs[n_tcbs]->thread_id;
138    printf("Tid: %d\n",tid);
139
140    tcbs[n_tcbs]->finished = 0;
141
142    tcbs[current_thread_id]->ucp.uc_stack.ss_sp = tcbs[n_tcbs]->stack;
143    tcbs[current_thread_id]->ucp.uc_stack.ss_size = sizeof(tcbs[n_tcbs]->stack);
144    tcbs[current_thread_id]->ucp.uc_link = &ucp_main;
145    makecontext(&tcbs[current_thread_id]->ucp, new_stub, 2, stub, args);
146
147    printf("swap main and tcbs[%d]\n",current_thread_id);
148    if (swapcontext(&ucp_main, &tcbs[current_thread_id]->ucp) == -1){
149      printf("ERROR: swapcontext\n");
150    }
151
152
153    n_tcbs++;
154
155
156    // stub -> the body of thread
157    // put info in the tcb
158    // TODO: Implement your own code
159
160    return tid;
161  }
```

그림 3 mythread_create 함수

그림 3에서 보는 바와 같이 tcbs의 list에 메모리 할당을 해준 후 getcontext를 사용하여 현재 상태를 저장하고 tcb 안에 정보들을 저장하고 makecontext를 하기 위한 정보들을 입력한다. 그 후 makecontext를 통해 stub에 대한 새로운 함수를 저장한다. New_stub 함수는 그림 5와 같다.

```
111  void new_stub(void (*stub)(void*), void* args){
112    printf("Do stub\n");
113    stub(args);
114    tcbs[current_thread_id]->finished =1;
115    if (swapcontext(&tcbs[current_thread_id]->ucp, &ucp_main) == -1){
116      printf("ERROR: swapcontext\n");
117    }
118
119  }
```

그림 4 new_stub 함수

New_stub함수는 stub을 실행시키고 종료가 되었을 때 finished 변수를 바꾸기 위함으로 만들었다. 여기서 함수의 실행이 끝나면 swapcontext를 통해 main함수로 다시 가도록 한다.

```
ajou@ajou-VirtualBox:~/mythread$ ./example
Thread create
current id: 0
Tid: 83
swap main and tcbs[0]
Do stub
f
83
Thread create
current id: 1
Tid: 86
swap main and tcbs[1]
Do stub
g
g
g
g
g
```

그림 5 ucontext를 통해 실행되는 f와 g함수들을 담은 thread들

그림 5은 thread들이 생성되고 thread들이 swap되었다는 것을 보여준 실행결과이다. 이 결과는 timer가 없는 FIFO의 결과와 같을 것이다. 하지만 오직 mythread_create 함수에서 실행된 결과이기 때문에 과제에서 원하는 실행결과는 아닐 것이다.

마지막으로 mythread_join 함수는 각 thread가 모두 끝날 때까지 main 함수가 기다려주는 함수이다. 이 또한 본인의 역량 부족으로 구현하지 못하였다.

- Lessons

이번 과제를 하면서 구현하는데 많은 어려움을 겪었다. 하지만 ucontext에 대한 매뉴얼을 보고 thread들이 context들이 어떻게 swap하는지 볼 수 있었고 user-level threading에 대한 개념을 좀

더 확실히 알 수 있었다. 또한 sigaction과 itimerval 구조체에 대해 더 공부할 수 있었다.

- Feedback

개인적으로 이번 과제를 하면서 저번 과제보다 훨씬 어렵고, 쉽지 않다는 것을 깨달았다. 구현을 하기 전에 공부해야 하는 양이 많았고, 알고리즘이 쉽게 머릿속에 나오지 않았다. Ucontext와 timer에 대한 개념을 잡는데도 튼 어려움을 겪었고, 결국에는 이번 과제를 완성하지 못했다. 하지만 저번 과제와는 다른 점은 이번 과제는 코드를 어떻게 짜느냐에 더 많은 머리를 굴리기보다, system call들을 이해하고 사용법을 익히는데 큰 시간을 할애했다. 그 안에서 어떻게 시스템콜들이 흘러가는지 파악하기 어려웠다. 후에 좀 더 생각을 해보아서 이번 과제를 마무리 짓고 싶다는 욕심이 있다.