

[Lab-13] Deferred work

소프트웨어학과 201720707 나용성

- Describe your implementation

timer.c:

TODO1 : timer handler가 time out이 발생했을 때 호출했음을 알 수 있도록 pr_info api를 사용하고 timer_setup, del_timer_sync api를 사용해 timer를 셋팅하고 제거했다. timer를 scheduling할 때 시간은 jiffies + TIMER_TIMEOUT * HZ로 설정하여 현재 시간에서 TIMER_TIMEOUT초가 지난 뒤에 time out이 발생하도록 했다.

TODO2: timer handler안에서 다시 timer를 scheduling하여 주기적으로 time out이 계속 발생하게 했다.

deferred.c:

deferred모듈은 user의 명령에 따라 timer를 이용해 user가 원하는 작업을 scheduling해준다. 기능들에는 먼저 다른 기능 없이 원하는 시간 뒤에 time out이 발생하게 하는 기능, 설정된 timer를 cancel하는 기능, 원하는 시간 뒤에 time out이 발생해 work_queue에 alloc_io() 함수를 scheduling하게하는 기능, 원하는 프로세스들을 list에 추가하고 주기적으로 time out을 발생시키며 상태를 추적하는 기능들이 있다.

kthread.c:

init함수에서 wait_queue, atomic변수를 초기화 시키고 kernel thread를 mythread라는 이름으로 생성해 my_thread_f 함수를 실행하게 했다. kernel thread는 생성되자마자 자신의 상태를 출력한 다음 wq_stop_thread wait queue를 wait하게 된다. 이 kernel thread는 module이 terminate될 때 module이 kernel thread가 종료됨을 알리는 atomic 변수에 masking을 하고 wq_stop_thread wait queue를 깨워서 kernel thread가 자신이 종료될 때임을 알고 종료하도록 설계됐다. 그러나 종료되기 전에 모듈이 wait queue를 깨운 다음 자신이 wq_thread_terminated wait queue를 wait해서 kernel thread가 정상적으로 종료될 때까지 기다리게 되는데 이는 다시 kernel thread가 다른 atomic 변수에 masking한 다음 wq_thread_terminated wait queue를 깨워서 같이 종료될 수 있도록 설계됐다.

- Explain result each case

```
root@sce394_vm:~/labs/lab13/timer# insmod timer.ko
[ 1705.267969] [timer_init] Init module
root@sce394_vm:~/labs/lab13/timer# [ 1706.322209] [timer_handler] Timer timeout
[ 1707.346224] [timer_handler] Timer timeout
[ 1708.370236] [timer_handler] Timer timeout
[ 1709.394276] [timer_handler] Timer timeout
[ 1710.418212] [timer_handler] Timer timeout
```

그림 1. timer.ko 테스트 결과

module을 설치하자마자 timer_setup이후 1초 뒤에 time out이 발생하도록 schedule하고 time out때 호출되는 timer_handler는 다시 timer를 scheduling하여 약 1초 주기로 time out이 지속적으로 발생하는 결과가 나타나고 있다.

```
root@sce394_vm:~/labs/lab13/deferred/user# ./test s 1
[ 1856.089958] [deferred_open] Device opened
Set timer to 1 seconds
[ 1856.093898] [deferred_ioctl] Command: Set timer
[ 1856.094887] [deferred_release] Device released
root@sce394_vm:~/labs/lab13/deferred/user# [ 1857.106290] [time_handler] pid = 0, comm = swapper/0
root@sce394_vm:~/labs/lab13/deferred/user# ./test s 2
[ 1864.257220] [deferred_open] Device opened
Set timer to 2 seconds
[ 1864.258043] [deferred_ioctl] Command: Set timer
[ 1864.259780] [deferred_release] Device released
root@sce394_vm:~/labs/lab13/deferred/user# [ 1866.322247] [time_handler] pid = 0, comm = swapper/0
```

그림 2. deferred.ko TODO 1 테스트

timer set 기능을 사용하자 argument로 전달된 숫자만큼의 초가 지난 뒤 timeout이 발생해 상태 메시지를 출력하는 모습이 나타나고 있다.

```
root@sce394_vm:~/labs/lab13/deferred/user# ./test a 1
[ 1900.153205] [deferred_open] Device opened
Allocate memory after 1 seconds
[ 1900.154282] [deferred_ioctl] Command: Allocate memory
[ 1900.156068] [deferred_release] Device released
root@sce394_vm:~/labs/lab13/deferred/user# [ 1901.202232] [time_handler] pid = 0, comm = swapper/0
[ 1906.642342] Yawn! I've been sleeping for 5 seconds.
root@sce394_vm:~/labs/lab13/deferred/user# ./test a 2
[ 1912.226777] [deferred_open] Device opened
Allocate memory after 2 seconds
[ 1912.230792] [deferred_ioctl] Command: Allocate memory
[ 1912.231414] [deferred_release] Device released
root@sce394_vm:~/labs/lab13/deferred/user# [ 1914.258212] [time_handler] pid = 0, comm = swapper/0
[ 1919.442284] Yawn! I've been sleeping for 5 seconds.
```

그림 3. deferred.ko TODO 2, 3 테스트1

TODO 2, 3까지 마친 뒤에 timer allocate 기능을 사용하자 argument로 전달된 숫자만큼의 초가 지난 뒤 timeout이 발생해 alloc_io()를 workqueue에 scheduling하고 work queue에서 scheduling되어 실행된 다음 5초 뒤 출력문이 출력되는 모습을 보여주고 있다.

```
root@sce394_vm:~/labs/lab13/deferred/user# ./test a 1
[ 78.023791] [deferred_open] Device opened
Allocate memory after 1 seconds
[ 78.026698] [deferred_ioctl] Command: Allocate memory
[ 78.027898] [deferred_release] Device released
root@sce394_vm:~/labs/lab13/deferred/user# [ 79.035154] [time_handler] pid = 0, comm = swapper/0
root@sce394_vm:~/labs/lab13/deferred/user# ./test a 1
[ 79.681142] [deferred_open] Device opened
Allocate memory after 1 seconds
[ 79.684861] [deferred_ioctl] Command: Allocate memory
[ 79.685393] [deferred_release] Device released
root@sce394_vm:~/labs/lab13/deferred/user# [ 80.699081] [time_handler] pid = 0, comm = swapper/0
[ 84.091646] Yawn! I've been sleeping for 5 seconds.
[ 89.211208] Yawn! I've been sleeping for 5 seconds.
```

그림 4. deferred.ko TODO 2, 3 테스트2

위의 TODO 2, 3에 대한 첫번째 테스트에서 bottom half가 모두 처리되기 전에 새 work가 work queue에 삽입되면 어떠한 결과가 나타날지 궁금해져서 두 번의 time out을 첫 work가 처리되기 전에 연달아 발생시켜 보았다. 그 결과 첫 번째 bottom half에서의 work가 완전히 처리되기 전에 두 번째 time out이 발생하여 bottom half work 하나가 더 work queue에 추가된 다음 첫번째 bottom half work가 모두 처리된 다음 두 번째 work도 정상적으로 처리된 모습을 확인했다.

```
root@sce394_vm:~/labs/lab13/deferred/user# sh sleep.sh &
root@sce394_vm:~/labs/lab13/deferred/user# sleep 1
sleep 1
sleep 1
sleep 1
sleep 1
root@sce394_vm:~/labs/lab13/deferred/user# ps | grep sleep.sh
  585 root      7836 S    sh sleep.sh
  592 root      7836 S    grep sleep.sh
```

```
sleep 1
sleep 1

root@sce394_vm:~/labs/lab13/deferred/user# ./test p 585
[ 2330.540442] [deferred_open] Device opened
sleep 1
Monitor PID 585.
[ 2330.546016] [deferred_ioctl] Command: Monitor pid
[ 2330.546658] [deferred_release] Device released
root@sce394_vm:~/labs/lab13/deferred/user# sleep 1
[ 2331.602275] [time_handler] pid = 0, comm = swapper/0
sleep 1
[ 2332.626217] [time_handler] pid = 0, comm = swapper/0
sleep 1
```

```
[ 2337.746182] [time_handler] pid = 0, comm = swapper/0
sleep 1
[ 2338.770251] [time_handler] pid = 0, comm = swapper/0

root@sce394_vm:~/labs/lab13/deferred/user# kill -SIGINT 585
root@sce394_vm:~/labs/lab13/deferred/user# [ 2339.794210] [time_handler] pid = 0, comm = swapper/0
[ 2339.794844] task sh (585) is dead
[ 2340.818106] [time_handler] pid = 0, comm = swapper/0
[ 2341.842156] [time_handler] pid = 0, comm = swapper/0
```

그림 5. deferred.ko TODO 4 테스트 결과

TODO 4를 테스트 하는 과정은 무한루프를 도는 프로세스를 생성한 다음 deferred module이 해당 프로세스를 time out을 주기적으로 발생시키며 프로세스의 상태를 체크하게 하게 하고 프로세스를 강제종료 시켜서 module이 상태를 제대로 인식하는지를 확인하는 과정으로 테스트 한다. 위의 세 개의 사진에서 그 과정을 확인할 수 있는데 상태를 추적하는 동안 1초 주기로 time out이 발생하여 프로세스와 같이 출력문을 출력하고 프로세스가 종료되자 정상적으로 상태를 인식하고 task가 task dead 상태임을 출력하는 모습을 확인할 수 있었다.

```
root@sce394_vm:~/labs/lab13/kthread# insmod kthread.ko
[ 4013.630414] [kthread_init] Init module
root@sce394_vm:~/labs/lab13/kthread# [ 4013.640419] [my_thread_f] Current process id is 696 (mythread)
root@sce394_vm:~/labs/lab13/kthread# rmmod kthread.ko
[ 4026.089347] [my_thread_f] Exiting
[ 4026.090045] [kthread_exit] Exit module
```

그림 6. kthread.ko 테스트 결과

kthread 모듈은 install됐을 때 kernel thread를 하나 생성하고 terminate 될 때 모듈에서 atomic 변수에 kernel thread가 terminate 되도록 masking을 하고 wait queue를 wake시켜서 wait queue를 waiting 중인 kernel thread를 깨우고 자기 자신이 다른 wait queue를 wait한다. 그러면 깨어난 kernel thread가 making된 atomic 변수를 확인하고 다른 atomic 변수에 module이 종료되야 함을 masking한 다음 module이 waiting중인 wait queue를 깨운 다음 스스로 종료하는 과정으로 동작하며 깨어난 모듈도 atomic 변수를 확인하고 종료되는 과정 거친다. 그러므로 module을 종료시키면 install할 때 생성됐던 kernel thread가 같이 종료될 것이다. 위의 결과를 보았을 때 insmod를 할 때 kernel thread가 정상적으로 생성되고 rmmod 명령으로 모듈을 terminate하려 할 때 kernel thread도 함께 동시에 정상적으로 terminate되는 모습을 출력문을 통해 확인할 수 있었다.