

[Lab-13] Deferred work

소프트웨어학과 201921085 곽수정

- Describe your implementation

- timer.c

먼저 `timer_init`에서 타이머를 초기화하기 위해 선언된 `timer`와 `timer`가 `expire`될 때 불릴 `callback` 함수를 `timer_handler`로 지정한 `timer_setup`을 사용하였습니다. 타이머를 등록하기 위해 `mod_timer`를 사용하였으며 현재 시스템의 시간은 `jiffies`에 저장되어 있으므로 현재 시간에서 `TIMER_TIMEOUT`초 뒤에 `timeout`이 발생하도록 `jiffies + TIMER_TIMEOUT * HZ`로 지정하였습니다. `timer_handler`에서는 `timer`를 `mod_timer`로 다시 스케줄링 해주어 `time_out`이 계속 발생하도록 하였습니다. 마지막으로 `timer_exit`에서 `timer`를 제거하기 위해 `del_timer_sync`를 사용하였습니다.

- deffered.c

`ioctl`을 이용하여 `timer`를 `control`하기 위해 TODO 1을 구현하였습니다. `my_device_data`에 타이머를 추가해주고, `deffrred_init`에 타이머를 초기화하였습니다. 초기화시에 `callback` 함수로 지정한 `timer_handler`에서는 `global variable`인 `my_device_data` 데이터를 얻기 위해 `from_timer`를 사용하였습니다. `timer handler`의 동작을 확인하기 위한 플래그인 `MY_IOCTL_TIMER_SET`에 해당하면 `mod_timer`로 타이머를 스케줄하도록 하였습니다. `timer`를 `delete`하기 위한 플래그인 `MY_IOCTL_TIMER_CANCEL`에서는 타이머를 제거해주었습니다. 마지막으로 `deferred_exit`에서 타이머를 제거해주었습니다.

TODO 2에서는 `blocking operation(error check)`를 구현하였습니다. `flag`를 선언하고 초기화해주었으며 `MY_IOCTL_TIMER_SET`에 해당하면 `flag`를 세팅해주었습니다. 그리고 `MY_IOCTL_TIMER_ALLOC`에 해당하는 경우에는 `flag`를 세팅하고 `mod_timer`로 타이머를 스케줄링 해주었습니다.

`alloc_io()` 사용을 위한 `workqueue` 구현하기 위해 TODO 3을 작성하였습니다. `deferred_init`에서 `work`를 초기화해주었고, `TIMER_TYPE_ALLOC`에 해당하는 경우 `work`을 스케줄링 해주었습니다.

TODO 4에서는 리스트에 프로세스 정보를 저장하고 `flag`를 변경한 뒤 타이머를 스케줄링 하였습니다. `flag`가 `TIMER_TYPE_MON`일 때 타이머를 다시 스케줄링해서 `timer_handler`가 주기적으로 호출될 수 있도록 하였고 기능으로는 리스트에 추가된 프로세스를 다 확인하여 `TASK_DEAD` 상태이면 어떤 `task`가 죽었는지 출력하고 리스트에서 프로세스 정보를 삭제하도록 하였습니다.

- kthread.c

kthread_init에서 waitqueue와 flag들을 초기화해주었습니다. 이후에 kernel thread를 만들고 my_thread_f를 실행시키도록 하였습니다. my_thread_f에서는 kernel thread가 먼저 자신의 정보를 출력하고 wq_stop_thread wait queue를 wait하게 됩니다. 이 kernel thread는 이 module이 exit(terminate) 될 때 atomic 변수에 1로 세팅되고 wake_up_interruptible에 의해 wq_stop_thread wait queue를 깨워서 종료하도록 하였습니다. kernel thread가 종료되기 전에 모듈에서는 wq_thread_terminated wait queue를 wait해서 kernel thread가 정상적으로 종료될 때까지 기다리게 됩니다. 그럼 kernel thread가 flag_thread_terminated 를 세팅하고 wq_thread_terminated wait queue를 깨워서 종료하도록 구현하였습니다.

- Explain result each case

- TODO 1 test

```
root@sce394_vm:~/labs/lab13/deferred/user# ./test s 1
[ 49.278108] [deferred_open] Device opened
Set timer to 1 seconds
[ 49.281873] [deferred_ioctl] Command: Set timer
[ 49.282091] [deferred_release] Device released
root@sce394_vm:~/labs/lab13/deferred/user# [ 50.325886] [timer_handler] pid = 0, comm = swapper/0
[ 50.326894] flag = TIMER_TYPE_SET

root@sce394_vm:~/labs/lab13/deferred/user# ./test s 2
[ 62.046746] [deferred_open] Device opened
Set timer to 2 seconds
[ 62.047340] [deferred_ioctl] Command: Set timer
[ 62.047497] [deferred_release] Device released
root@sce394_vm:~/labs/lab13/deferred/user# [ 64.085764] [timer_handler] pid = 0, comm = swapper/0
[ 64.086279] flag = TIMER_TYPE_SET
```

timer set 기능을 사용하였을 때 argument로 전달된 숫자만큼 초가 지나고 나서 time out이 발생하게 됩니다. 따라서 의도한대로 상태메세지를 출력하는 모습을 확인할 수 있었습니다.

- TODO 2 & 3 test

```
root@sce394_vm:~/labs/lab13/deferred/user# ./test a 1
[ 121.989259] [deferred_open] Device opened
Allocate memory after 1 seconds
[ 121.992092] [deferred_ioctl] Command: Allocate memory
[ 121.992296] [deferred_release] Device released
root@sce394_vm:~/labs/lab13/deferred/user# [ 123.030177] [timer_handler] pid = 0, comm = swapper/0
[ 123.030624] flag = TIMER_TYPE_ALLOC
[ 128.086400] Yawn! I've been sleeping for 5 seconds.

root@sce394_vm:~/labs/lab13/deferred/user# ./test a 2
[ 135.778827] [deferred_open] Device opened
Allocate memory after 2 seconds
[ 135.779341] [deferred_ioctl] Command: Allocate memory
[ 135.779490] [deferred_release] Device released
root@sce394_vm:~/labs/lab13/deferred/user# [ 137.813972] [timer_handler] pid = 0, comm = swapper/0
[ 137.814478] flag = TIMER_TYPE_ALLOC
[ 142.934121] Yawn! I've been sleeping for 5 seconds.
```

TODO 2, 3 까지 작성하고 `time allocate` 기능을 사용하여 테스트한 결과입니다. `argument`로 전달된 초만큼 시간이 지난 뒤에 `time out`이 발생하게 됩니다. 이에 따라서 `alloc_io()`를 스케줄링하고, `workqueue`에서 스케줄링되어 실행됩니다. 그 다음에 5초 뒤에 확인하는 출력문이 출력되어 결과를 확인하였습니다.

- TODO 4 test

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

./test p 260sleep 1
[ 71.035052] [deferred_open] Device opened
Monitor PID 260.
[ 71.038994] [deferred_ioctl] Command: Monitor pid
[ 71.039875] [deferred_release] Device released
[ 71.041935] test (281) used greatest stack depth: 13336 bytes left
root@sce394_vm:~/labs/lab13/deferred/user# sleep 1
[ 72.085619] [timer_handler] pid = 0, comm = swapper/0
sleep 1
[ 73.109846] [timer_handler] pid = 0, comm = swapper/0
sleep 1
[ 74.133840] [timer_handler] pid = 0, comm = swapper/0
sleep 1
[ 75.157275] [timer_handler] pid = 0, comm = swapper/0
sleep 1
[ 76.181326] [timer_handler] pid = 0, comm = swapper/0
sleep 1
kill -SIGINT 260
root@sce394_vm:~/labs/lab13/deferred/user# [ 77.205269] [timer_handler] pid =
0, comm = swapper/0
[ 78.229301] [timer_handler] pid = 0, comm = swapper/0
[ 78.229854] task sh (260) is dead
[ 79.253299] [timer_handler] pid = 0, comm = swapper/0
[ 80.277318] [timer_handler] pid = 0, comm = swapper/0
```

TODO 4 테스트에서는 무한루프를 도는 프로세스를 생성하였습니다. 그 다음 `deferred` 모듈이 `time out`을 주기적으로 발생시킵니다. 모듈이 프로세스의 상태를 확인하고 강제 종료 시켜서 모듈이 상태를 인식하도록 하였습니다. 그 결과로는 PID 260인 프로세스를 `monitor`할 때 1초 주기로 `time out`이 발생하는 모습을 볼 수 있었으며 프로세스가 종료되었을 때 `task`가 `dead` 상태임을 알려주는 출력을 확인할 수 있었습니다.

- timer.ko test

```
root@sce394_vm:~/labs/labl3/timer# insmod timer.ko
[ 21.847536] timer: loading out-of-tree module taints kernel.
[ 21.853596] [timer_init] Init module
root@sce394_vm:~/labs/labl3/timer# [ 22.872319] [timer_handler] Call timer handler jiffies : 429469004
[ 23.896344] [timer_handler] Call timer handler jiffies : 4294691072
[ 24.920311] [timer_handler] Call timer handler jiffies : 4294692096
[ 25.944441] [timer_handler] Call timer handler jiffies : 4294693120
```

timer module은 `install`될 때 1초 뒤 `timeout0`이 되도록 timer를 스케줄링 해줍니다. 그 다음 `timeout0`이 발생해서 `timer_handler`가 호출되면 현재 `jiffies`를 출력하고 다시 1초 뒤에 `timeout0`이 발생하도록 스케줄링 합니다. 그 결과 1초 주기로 `timeout0`이 지속적으로 발생하며 `timer handler`가 호출되어 `jiffies`를 지속적으로 출력하는 모듈이 된 것을 출력된 결과를 통해 확인할 수 있었습니다.

- kthread.ko test

```
[ 1698.029711] [kthread_init] Init module
root@sce394_vm:~/labs/labl3/kthread# [ 1698.036458] [my_thread_f] Current process id is 377 (my_thread)

root@sce394_vm:~/labs/labl3/kthread# rmmod kthread.ko
[ 1718.621394] [my_thread_f] Exiting
[ 1718.621758] [kthread_exit] Exit module
```

kthread module은 `init`함수에서 kernel thread 하나를 생성합니다. kernel thread는 자신의 상태를 출력한 다음 `wq_stop_thread` wait queue에 진입합니다.

그 다음 module이 종료될 때 `exit` 함수는 kernel thread도 종료될 수 있도록 `flag_stop_thread`를 1로 변경한 다음 `wq_stop_thread` wait queue를 향해 `wake_up` api를 호출하고 `wq_thread_terminated` wait queue에 진입합니다. 이 때 깨어난 kernel thread는 `flag_stop_thread`의 값을 읽고 `wait queue`를 탈출한 다음 종료될 것임을 module에게 알리기 위해 `flag_thread_terminated`를 1로 변경한 다음 `wq_thread_terminated` wait queue를 향해 `wake_up` api를 사용하고 종료 문구를 출력한 다음 종료됩니다.

다시 깨어난 module은 `flag_thread_terminated`의 값을 읽은 다음 `kernel_thread`가 종료됨을 확인하고 자신도 종료문구를 출력한 다음 종료되어 module과 kernel thread가 동시에 종료됨이 구현됩니다. 그리고 그 결과를 출력문에서 module을 `terminate`할 때 kernel thread와 module이 같이 종료 문구를 출력하는 것을 통해 확인할 수 있었습니다.