Kernel Internals – Wait Queues

AGENDA

- Recapitulation of previous lecture
- Introduction
- Concept of sleep
- Wait queue creation
- Sleeping and waking up of process

RECAPITULATION

- In last module we learnt to control our device from user space using ioctl

INTRODUCTION

- It's may not be possible for the driver to provide data/resources at all instances when it is requested
  - When the user process calls the read function, the driver may not have data to provide to the user
  - When the user process tries to write data, the driver may not have space in the device buffers
- Driver has to put the process into WAIT STATE, until its request can be served
  - Wait queues comes into picture at this moment
  - Wait queues helps kernel to keep the track of waiting or sleeping processes

How does a Process Sleep?

- A process sleeps when it is waiting for some event to take place
  - A sleeping process is put-off from the scheduler's ready/running queue
  - It is marked as being in the special state called waiting state
  - This process will not wake up till something comes along to change the state of the process
  - Even after something has woken your process up, process checks if the event, for which it is waiting, is taken place or not
  - If the event, for which it is waiting, has taken place, process wakes up and its state changes from waiting to running
  - However, if event has not taken place, it will again go back to sleep

Precaution Before Your Driver Sleeps

- **Never sleep** when your driver is running in an atomic context
  - Atomic context is a state when multiple steps are done without any sort of concurrent access
  - Therefore, the driver should avoid sleeping while holding a busy waiting synchronization mechanism like Spinlock, Seqlock, or RCU lock
- Driver **should not sleep** after interrupts are disabled by it
  - It will increase the interrupt latency
- Your driver **can sleep** with holding the semaphore
  - But care should be taken, because it will delay other processes waiting on the semaphore
  - This could lead to a deadlock situation if the process that is supposed to wake you up is also pending on the semaphore.
  - **Bottom Line:** Keep the sleep, as small as possible, ensure that there is no deadlock situation.
- Your driver **cannot sleep** unless it is assured that some event will wake it up
  - There should be a mechanism for the event to be able to find the sleeping process, to wake it up.
- When you **wake up**, the code is not aware of its state
  - Did it wake up after the condition for which it slept is successfully completed?
  - Are there other processes that have been woken up, similar to it?
  - Bottom Line: Always check and confirm the condition for waking up has been fulfilled.

Wait Queues

- Mechanism to put a user process into sleep/wait state before resources are available.
- Implementation is a data structure that allows queuing of multiple processes, waiting for an event to occur
  - Therefore, wait queue is a list of processes all waiting for a specific event
- A wait queue is managed by means of a wait queue head, which is a structure of type **wait_queue_head_t**
- It is defined in **“linux/wait.h”**
- It can be defined and initialized by following functions:
  - **Statically:** DECLARE_WAIT_HEAD(name of the queue);
  - **Dynamically:** wait_queue_head_t driver_queue;
    init_ waitqueue_head(&driver_queue) ;

Process Going for Sleep

- The Great Expectation
  - When a process goes to sleep, it expects some condition in the future to become true before it can wake up
  - When it wakes up therefore, the condition has to be checked before it can decide to continue execution or return to sleep
- Macros ***wait_event*** and its variants can be used to put process in sleep state
  - wait_event(queue, condition)
  - wait_event_interruptible(queue, condition)
  - wait_event_timeout(queue, condition, timeout)
  - wait_event_interruptible_timeout(queue, condition, timeout)
  - ***queue*** is the wait queue head and ***condition*** is an arbitrary boolean expression that is evaluated by the macro before and after sleeping
  - Until that ***condition*** gets true, process continues to sleep

Process Going for Sleep contd..

- ***wait_event*** put the process in uninterruptible sleep. So, nothing can wake up the process, unless the condition is satisfied.
- ***wait_event_interruptible*** is the preferred alternative which can be interrupted by signals.
  - It provides an escape hatch by allowing user signals to remove the process from its sleep.
  - **Return value** is non-zero when it receives a signal other than the condition it is waiting for
  - At this time driver should return –ERESTARTSYS to the calling process
- On success, that means if *condition* became true, return value is zero
- ***Timeout variants:*** The two variants with timeout are similar with the above two except they wait till time expires mentioned as third argument
  - Returns 0 on wake up from timeouts. So, make sure you check the return value to take appropriate action.

Time to Wake Up

- Waking up is done by some other process that can be a different process or interrupt handler
- The functions that can be used to wake up the sleeping process are:
  - ***void wake_up(wait_queue_head_t *queue);***
  - ***void wake_up_interruptible(wait_queue_head_t *queue);***
- ***wake_up*** wakes up all the processes that are waiting in the given queue whether interruptible or non-interruptible
- ***wake_up_interruptible*** wakes up only interruptible sleeping processes
- ***wake_up*** wakes up all the processes waiting in the given queue,
  - Only those processes whose condition becomes true will change its state to RUNNABLE
  - Otherwise process whose condition is not true, will again go to sleep state

Thank you

Alternate Way to Sleep

- To sleep you need to flow the following steps:
- Declare the wait queue head using,

  DECLARE_WAIT_QUEUE_HEAD(driver_queue)
- Allocate and initialise the *wait_queue_t* structure, followed by its addition to the wait queue using

  DEFINE_WAIT(driver_wait)
- Changing its state to TASK_INTERRUPTIBLE or TASK_ UNINTERRUPTIBLE that can be done using

  void prepare_to_wait(wait_queue_head_t *queue, wait_queue_t *wait, in t state)

  -- 1st argument is the driver_queue, 2nd argument will be driver_wait and 3rd argument will be state
- Testing for the external event and calling *schedule()*, if event has not occurred yet.
- After the external event occurs, setting itself to the TASK_RUNNING state.
- To finish wait call

  void finish_wait(wait_queue_head_t *queue, wait_queue_t *wait)

Exclusive Wait

- It is very much similar to normal sleep, with two important difference:
  - When a wait queue entry has the WQ_FLAG_EXCLUSIVE flag set, it is added at the end of the waiting queue. Entries without this are, instead, added to the beginning
  - When a *wake_up* is called on a wait_queue, it stops after waking the first process that has the WQ_FLAG_EXCLUSIVE flag set
- So the process performing exclusive waits are awakened one at a time, in orderly manner
- But kernel still wakes up all nonexclusive waiters every time
- To implement exclusive wait:
  - void prepare_to_wait_exclusive (wait_queue_head_t *queue,
    wait_queue_t *wait, int state);

More Info on Wait Queue

- Wait queues are actually implemented using a linked list and spinlock
- Head of the list is *wait_queue_head_t* and nodes consist of list of type *wait_queue_t*