Run/Wait Queues

W4118 Operating Systems I

https://cs4118.github.io/www/2024-1/

Credits to Jae

Logistics

- HW3 is done
  - Please, please, please follow the instructions. We were lenient this time but not for the next homework.

- Get your VMWare license

- HW4 probably released today, due 2/21

Process States

```
/* Used in tsk->state: */
#define TASK_RUNNING                0x0000
#define TASK_INTERRUPTIBLE          0x0001
#define TASK_UNINTERRUPTIBLE        0x0002
```

TASK_RUNNING: the task is runnable – either currently running or on a run queue waiting to run

TASK_INTERRUPTIBLE: the task is sleeping waiting for some condition to exist - can be awakened prematurely if it receives a signal

TASK_UNINTERRUPTIBLE: the task is sleeping waiting for some condition to exist - cannot be awakened prematurely if it receives a signal

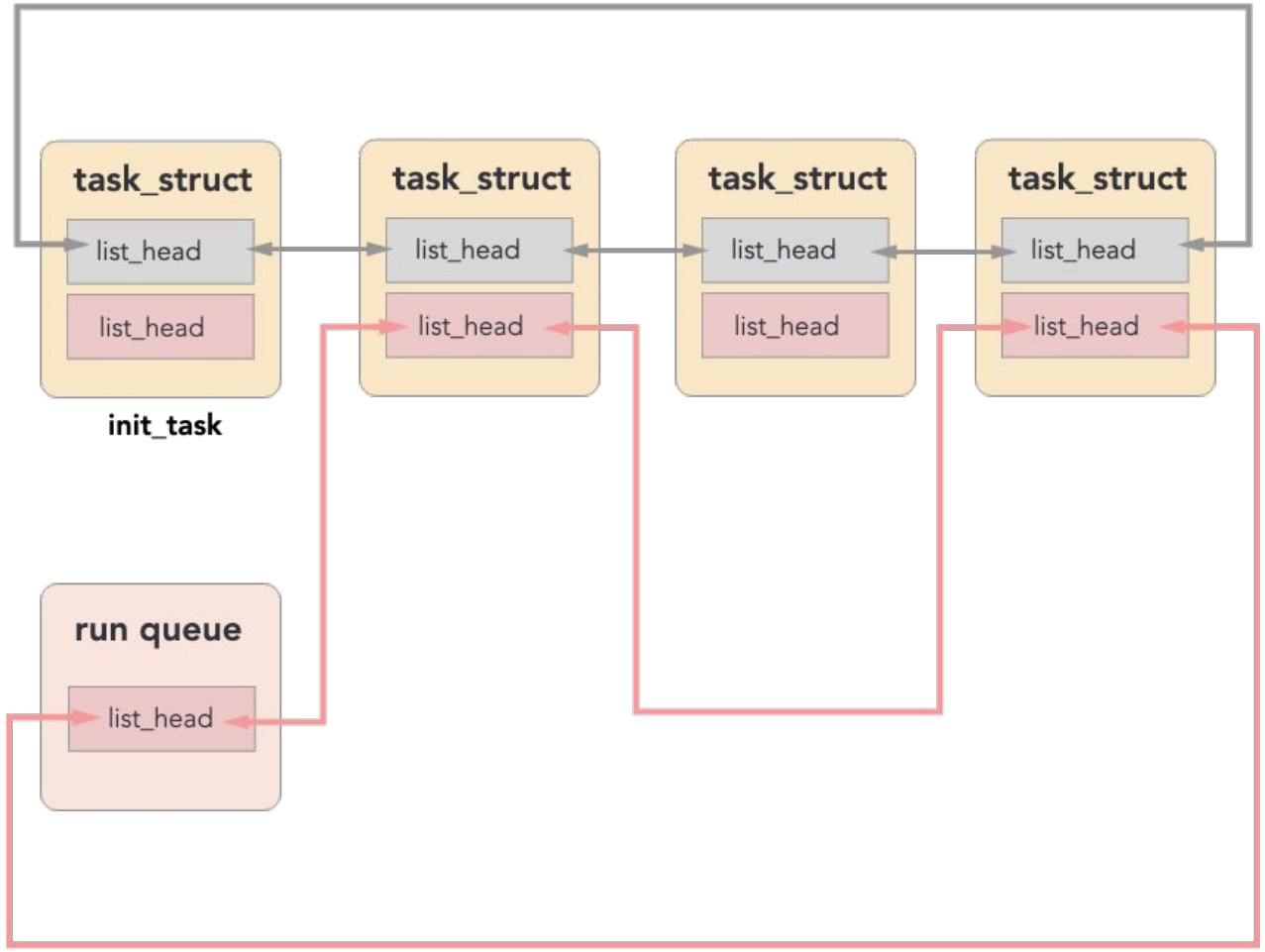
Run Queue

`task_structs` are linked via `children/sibling` list_heads

Per-CPU `run_queue` links tasks with state `TASK_RUNNING`

Why need a separate `list_head` for the run queue?

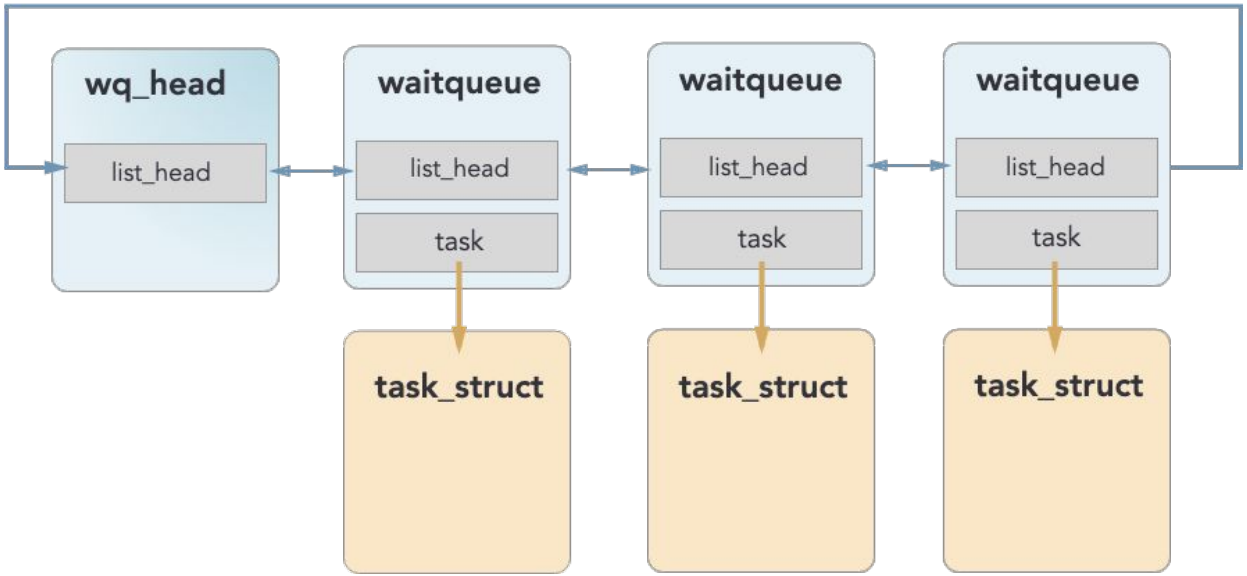
`include/linux/sched.h`



Wait Queue

Per-event `wait_queue`

Wait queue entry is NOT embedded in `task_struct`



Wait Queue Data Structures

***pseudocode

```
struct wait_queue_head {
        spin_lock_t lock;
        struct list_head task_list;
};

struct waitqueue {
        struct task_struct *task;
        wait_queue_func_t func; // callback function, e.g. try_to_wake_up()
        struct list_head entry;
};
```

How to wait

include/linux/wait.h – (kernel 3.12.74 for simplicity)

1. **`prepare to wait()`:** add yourself to wait queue, change state to **`TASK INTERRUPTIBLE`**
2. **`signal_pending()`:** check for “spurious wakeup”, i.e. signal interrupted sleep before condition was met
   - break out of loop instead of sleeping
3. **`schedule()`**: put yourself to sleep
4. **`finish_wait()`**: change state to **`TASK_RUNNING`**, remove yourself from the wait queue

Perform 1-3 in a loop to handle spurious wakeups

Notes:

1. LKD page 59 is outdated and incorrect, use **`wait_event_interruptible()`**
2. **`wait_event_interruptible()`** is a generic macro, probably not appropriate to use directly
   a. Doesn’t account for synchronization
   b. You may want to handle **`signal_pending()`** differently

Scheduling Basics

`kernel/sched/core.c`

1. `pick_next_task()`: choose a new task to run from the run queue
2. `context_switch()`: put current task to sleep, start running new task

Wait Queue Walkthrough

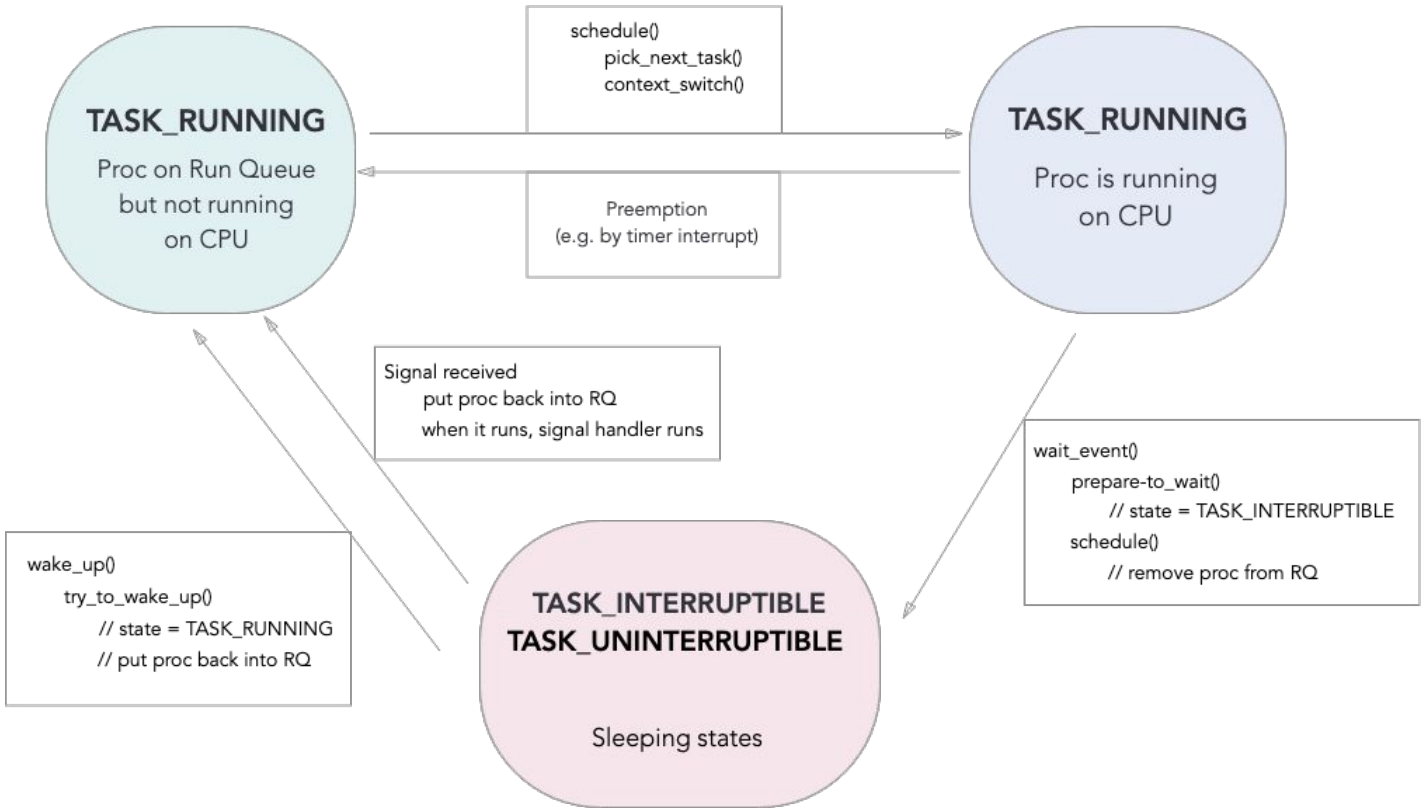
Sleeping:

1. `wait_event()`
2. Enqueued on wait queue
3. Remove from run queue
4. `schedule()`
   - `pick_next_task()`
   - `context_switch`
5. Other task runs

Waking up:

1. Task signals event: `wake_up()`
2. Call `try_to_wake_up()` on each task
3. Enqueue each task on run queue
4. Eventually other tasks calls `schedule()` and previously sleeping task gets chosen*
5. Previously sleeping task checks condition
   - If true, `finish_wake()`
   - Else, repeat 3-6 from “sleeping”

Process State Transition



Example: `read()`

1. Trap into kernel
   - save registers into per-proc kernel stack
2. Device driver issues an I/O request to the device
3. Put the calling process to sleep
   - **`wait_event()`** → **`schedule()`** → **`pick_next_task()`** → **`context_switch()`**
4. Another process starts running
5. The device completes the I/O request and raised a hardware interrupt
6. Trap into kernel and jump to the interrupt handler:
   - **`wake_up():`** enqueue blocked tasks back on run queue
   - Current task eventually calls `schedule()` → `pick_next_task()` → `context_switch()`
7. Another process starts running
   - This process may or may not be the one that called **`read()`**