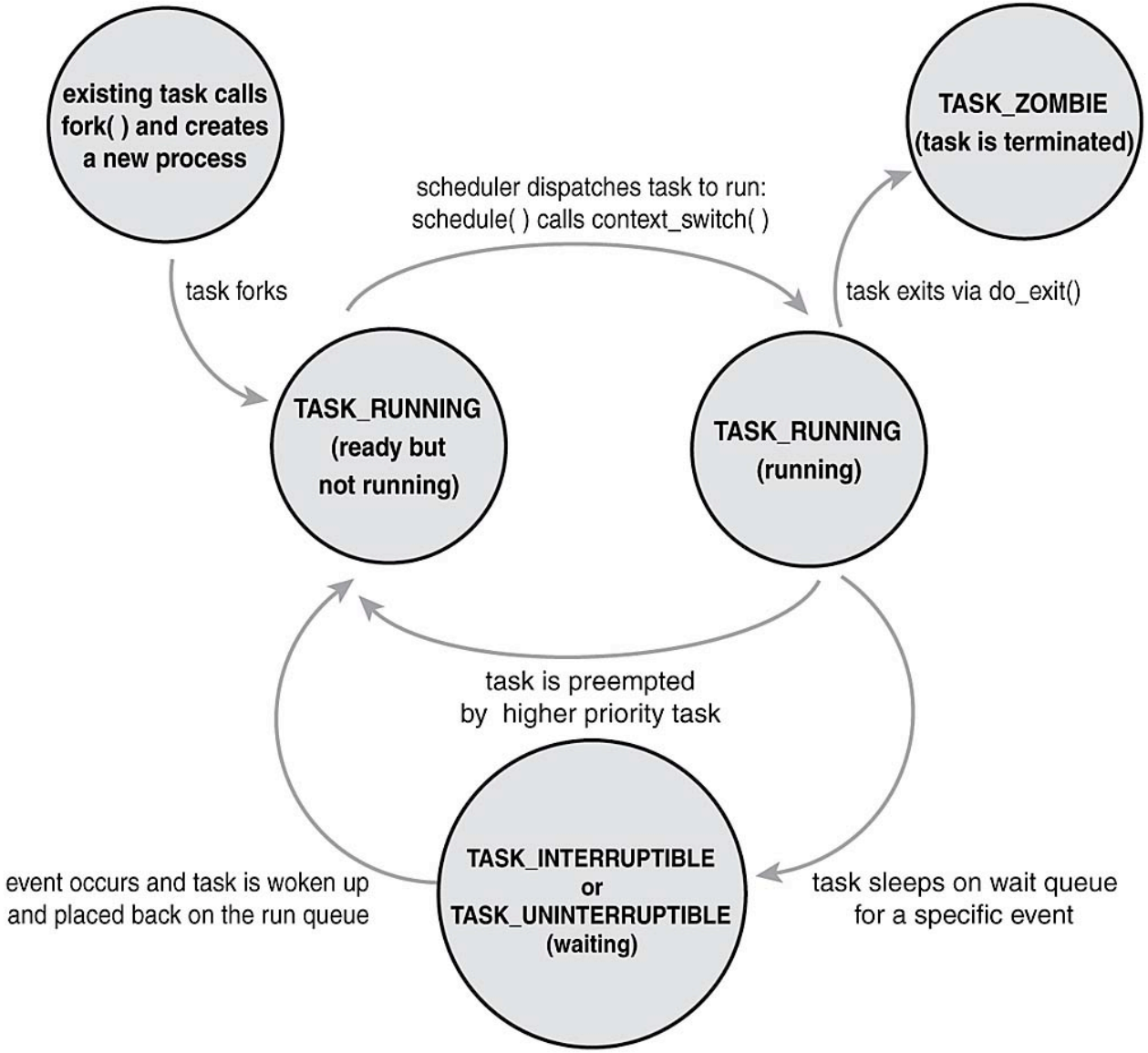
Linux kernel: Processes, threads and scheduling

- Process management: Linux calls its PCB as *struct task_struct* (<kernel source>/include/linux/sched.h)
  - Large structure, about 1.7 KB on a 32 bit machine
  - Each task's thread_info structure it allocated at the end of its kernel stack, its points to the task_struct
- Process IDs are 0-32767 (like most UNIX). Only possible to have 32768 concurrent tasks
- Processes can be in TASK_RUNNING, TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_ZOMBIE and TASK_STOPPED state.


existing task calls fork( ) and creates a new process
TASK_ZOMBIE (task is terminated)
scheduler dispatches task to run: schedule( ) calls context_switch( )
task forks
task exits via do_exit()
TASK_RUNNING (ready but not running)
TASK_RUNNING (running)
task is preempted by higher priority task
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)
event occurs and task is woken up and placed back on the run queue
task sleeps on wait queue for a specific event

Process creation

- Fork(), vfork() or thread()
  - Linux threads are basically tasks which share data
  - Fork optimizations:
    - Copy-on-write (COW): Child data not cloned till modified
    - Schedule child first in order to prevent parent from tripping over COW. More on COW later in memory management
  - Vfork: optimized fork where child is not allowed to modify any data
- All implemented using clone()
- Kernel threads are created by other kernel threads and run in kernel-space (does not context switch to user space at all)

Process creation

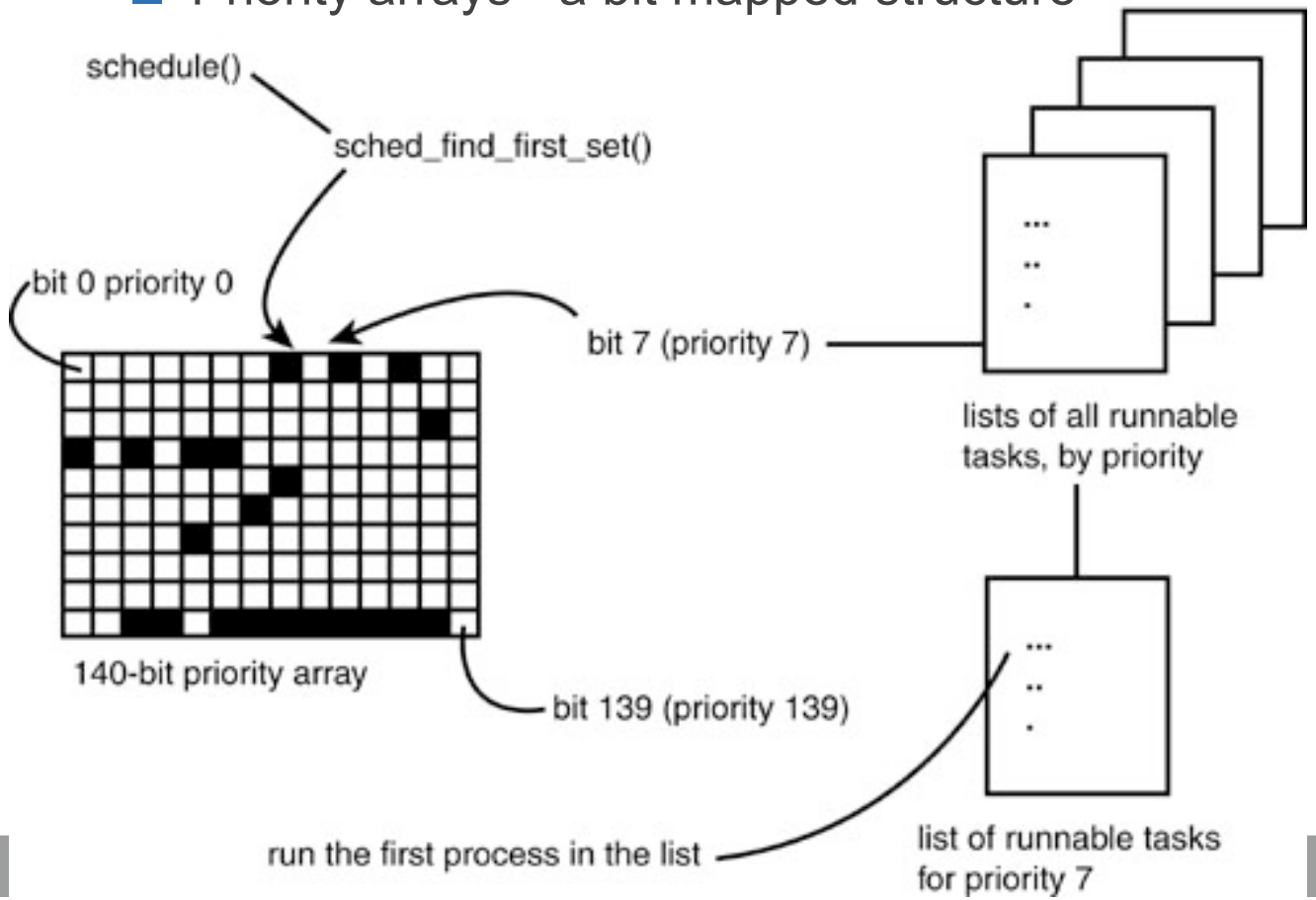
- Processes are created using clone() system call.
  - `CLONE_FILES` Parent and child share open files.
  - `CLONE_FS` Parent and child share filesystem information.
  - `CLONE_IDLETASK` Set PID to zero (used only by the idle tasks)
  - `CLONE_NEWNS` Create a new namespace for the child
  - `CLONE_PARENT` Child is to have same parent as its parent
  - `CLONE_PTRACE` Continue tracing child
  - `CLONE_SETTID` Write the TID back to user-space
  - `CLONE_SETTLS` Create a new TLS for the child
  - `CLONE_SIGHAND` Parent and child share signal handlers and blocked signals
  - `CLONE_SYSVSEM` Parent and child share System V `SEM_UNDO` semantics
  - `CLONE_THREAD` Parent and child are in the same thread group
  - `CLONE_VFORK` `vfork` was used and the parent will sleep until the child wakes it
  - `CLONE_UNTRACED` Do not let the tracing process force `CLONE_PTRACE` on the child
  - `CLONE_STOP` Start process in the `TASK_STOPPED` state
  - `CLONE_SETTLS` Create a new TLS (thread-local storage) for the child
  - `CLONE_CHILD_CLEARTID` Clear the TID in the child
  - `CLONE_CHILD_SETTID` Set the TID in the child
  - `CLONE_PARENT_SETTID` Set the TID in the parent
  - `CLONE_VM` Parent and child share address space.

Process scheduling

- Process priority
  - POSIX define nice values of -20 to +19. Larger value means lower priority
  - Real time priorities are in the range 0 to 99. Higher than for normal processes
- Time slice: allocated to each process based on its niceness. Minimum is the time slice. Maximum is 800 ms.
  - Once time slice runs out, moved to exhausted queue. After all processes exhaust their time slice, everyone gets new time slice.
    - Instead of waiting to reassign time slices, assign time slice before moving task to exhausted list
    - Rotate pointers to move between exhausted and active
    - O(1) operation

Scheduling

- O(1) scheduling - scheduling duration takes constant time and does not depend on the number of tasks running (compare with O(n), $O(n^2)$)
  - Priority arrays - a bit mapped structure



Scheduling

- Full SMP. Each processor runs its own queue except to load balance. Idle processors may take processes from other processors (take the exhausted pool in order to avoid affinity issues with cache)
- Improved SMP affinity
- Good interactive performance and fair: Kernel keeps track of CPU use and the rate of usage
  - How much time a process spent sleeping vs how much time the process spends in a runnable state
  - Takes care of processes which sleep for a while and then hog CPU
  - Interactive jobs are put in active list even if they exhaust their time slice
- Kernel is fully pre-emptible

Real-time

- Supports SCHED_FIFO and SCHED_RR. Normal processes, SCHED_NORMAL always yield to SCHED_FIFO and SCHED_RR.
  - Linux implements soft realtime
  - Performance is quite good