PROCESS SCHEDULING

ANIRUDH JAYAKUMAR

LAST TIME

- Build a customized Linux Kernel from source
- System call implementation
- Interrupts and Interrupt Handlers

TODAY'S SESSION

- Process Management
- Process Scheduling

PROCESSES

- “ a program in execution”
  - An active program with related resources (instructions and data)
  - Short lived (“pwd” executed from terminal) or long-lived (SSH service running as a background process)
  - A.K.A **tasks** – the kernel’s point of view
  - Fundamental abstraction in Unix

THREADS

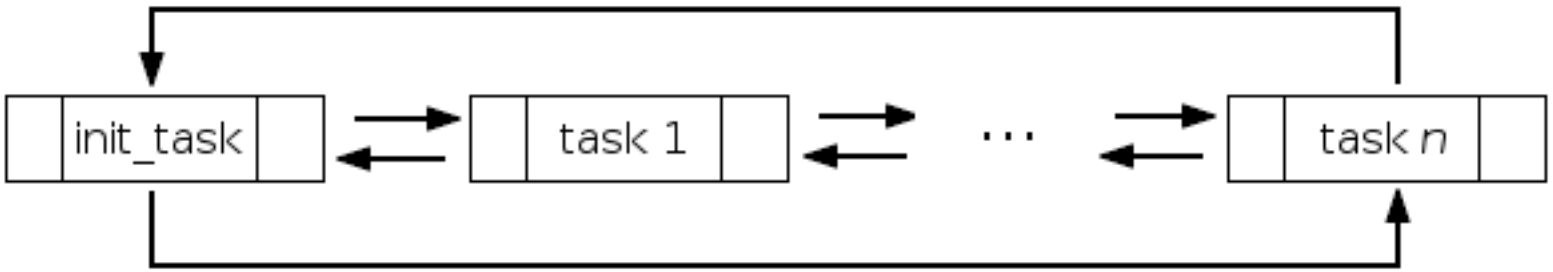
- Objects of activity within the process
  - One or more threads within a process
  - Asynchronous execution
- Each thread includes a unique PC, process stack, and set of processor registers
- Kernel schedules individual threads, not processes
  - **tasks** are Linux threads (a.k.a kernel threads)

TASK REPRESENTATION

- The kernel maintains info about each process in a process descriptor, of type task_struct
  - See include/linux/sched.h
- Each task descriptor contains info such as run-state of process, address space, list of open files, process priority etc
- The kernel stores the list of processes in a circular doubly linked list called the task list.

TASK LIST

- `struct list_head tasks;`
- init the "mother of all processes" – statically allocated
  - `extern struct task_struct init_task;`

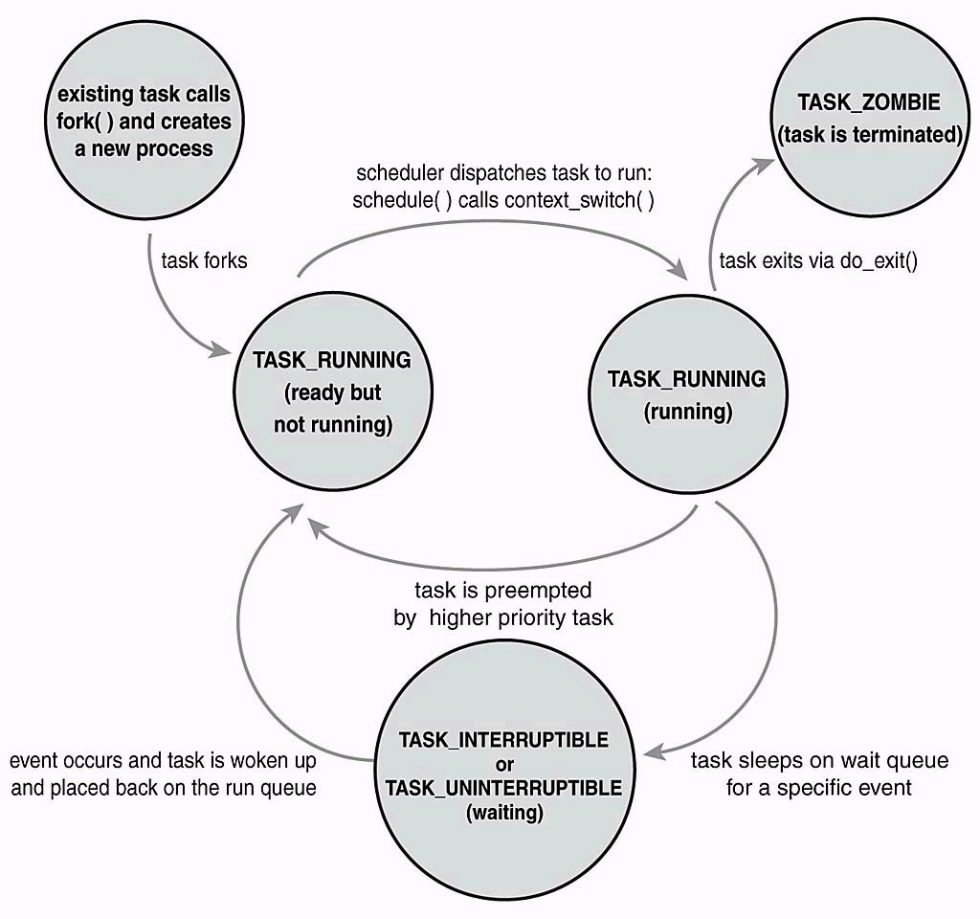


  - for_each_process() - iterates over the entire task list
  - next_task() - returns the next task in the list

PROCESS STATE

- TASK_RUNNING: running or on a run-queue waiting to run
- TASK_INTERRUPTIBLE: sleeping, waiting for some event to happen; awakes prematurely if it receives a signal
- TASK_UNINTERRUPTIBLE: identical to TASK_INTERRUPTIBLE except it ignores signals
- TASK_ZOMBIE: The task has terminated, but its parent has not yet issued a wait4(). The task's process descriptor must remain in case the parent wants to access it.
- TASK_STOPPED: Process execution has stopped; the task is not running nor is it eligible to run. (SIGSTOP, SIGTSTP). Can run again on receiving SIGCONT
- EXIT_DEAD: The final state (just like it sounds)

FLOW CHART OF PROCESS STATES



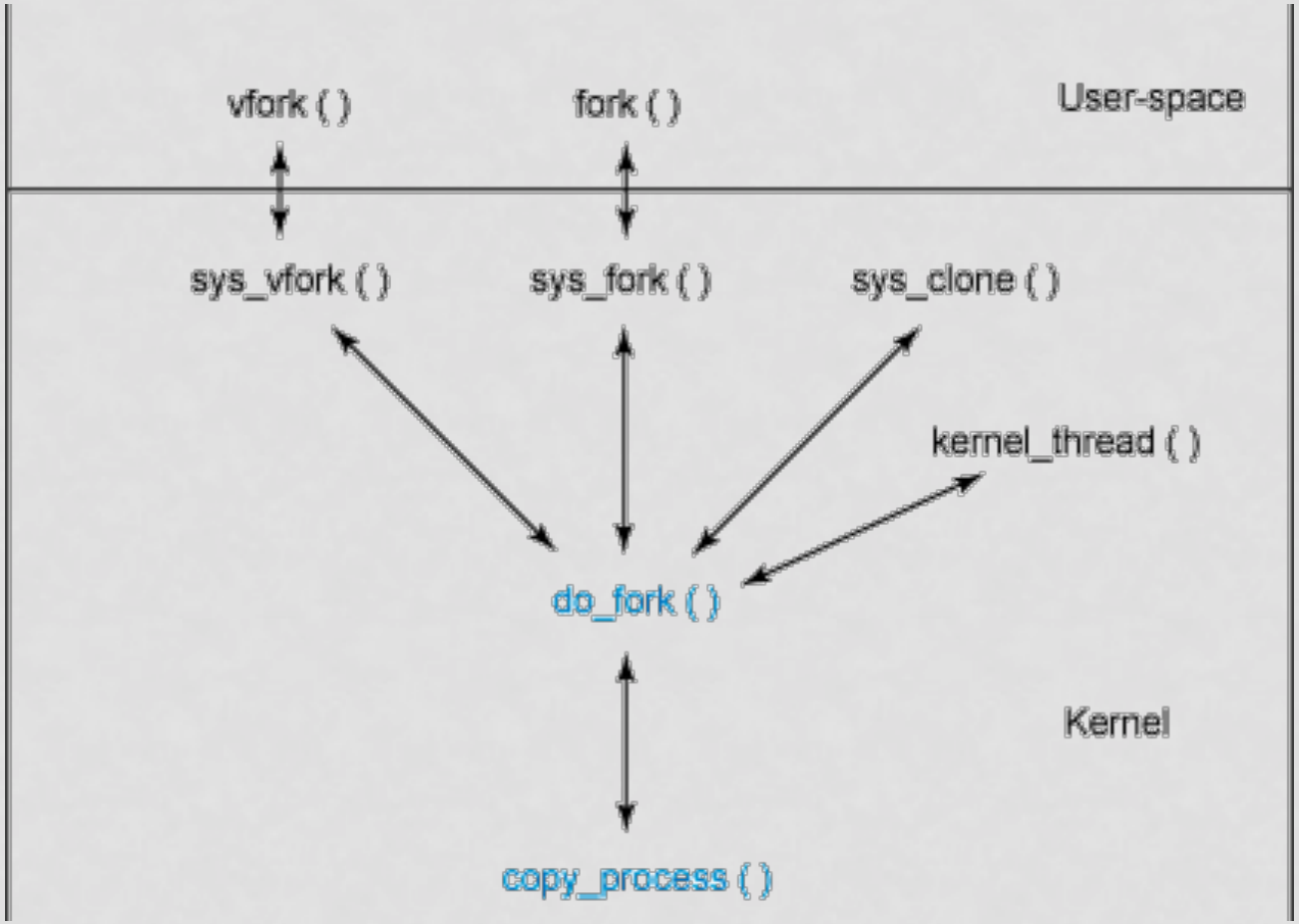
MANIPULATING PROCESS STATE

- set_task_state(task, state);
  - /* set task 'task' to state 'state' */
  - Better than task->state = state;
  - provides a memory barrier to force ordering on other processors
- set_current_state(state)
  - synonymous to set_task_state(current, state)

PROCESS CREATION

- user-space tasks and kernel tasks rely on a function called do_fork to create the new process
- Creating Kernel thread
  - the kernel calls a function called kernel_thread - performs some initialization
  - calls do_fork (linux/kernel/fork.c)
- user-space process
  - a program calls fork (C library)
  - system call to the kernel function called sys_fork (linux/arch/i386/kernel/process.c)
  - calls do_fork

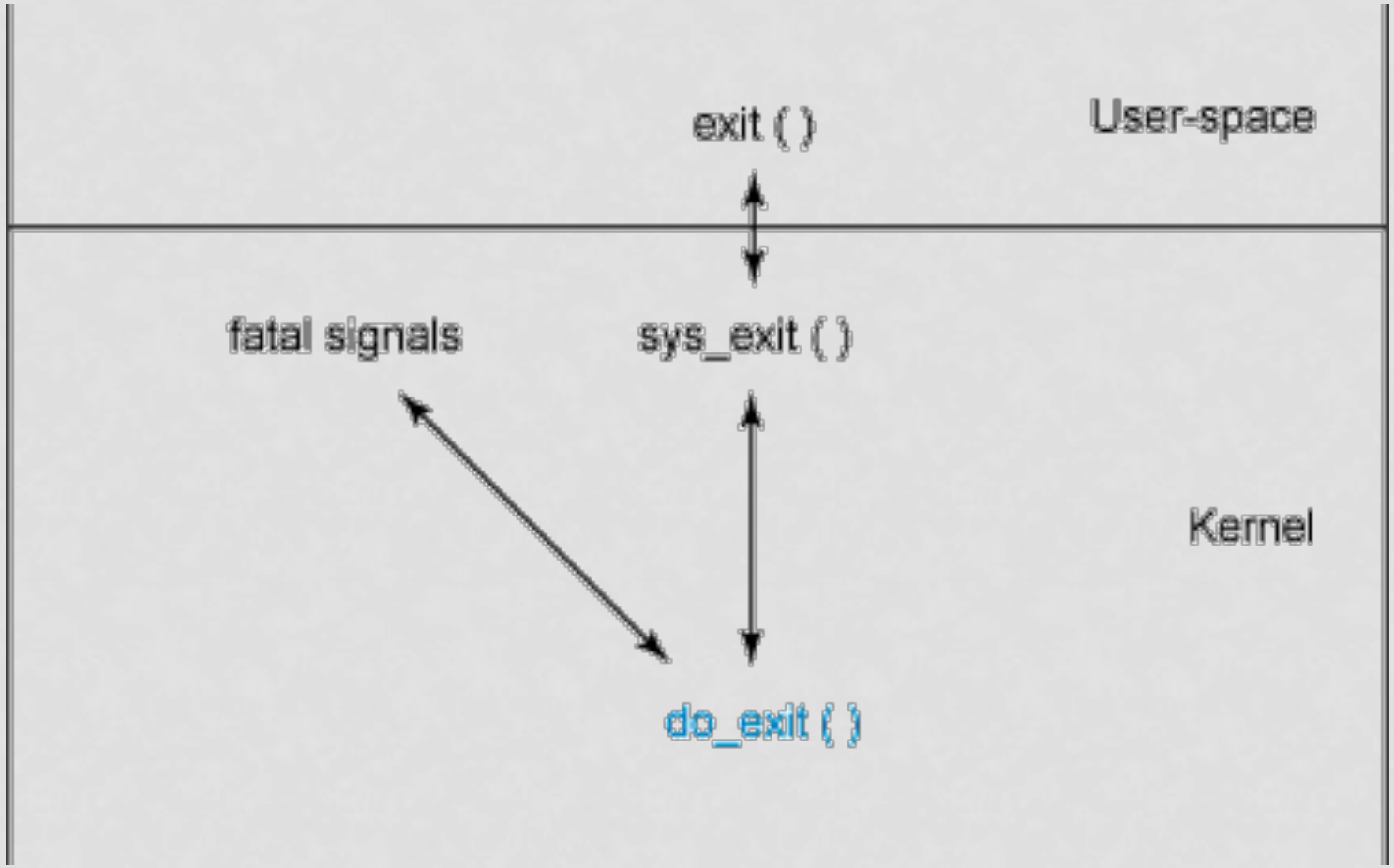
PROCESS CREATION



PROCESS CREATION

- do_fork()
  - call alloc_pidmap() //allocates a new PID
  - call copy_process(flags, stack, registers, parent process, new PID)
    - Validates flag
    - Consults Linux security module
    - call dup_task_struct()
      - allocates a new task_struct and copies the current process's descriptors into it
      - new thread stack is set up, some state information is initialized
    - sequence of copy functions is then invoked that copy individual aspects of the process
      - copy_files, copy_signal, copy_mm
    - new task is then assigned to a processor
  - wake_up_new_task – places task into runqueue

PROCESS TERMINATION



PROCESS TERMINATION

- do_exit()
  - exit_mm - to remove memory pages
  - exit_keys - which disposes of per-thread session and process security keys
  - exit_notify - series of notifications (for example, to signal the parent that the child is exiting)

PROCESS SCHEDULER

- the component of the kernel that selects which process to run next
- provides preemptive multitasking - the scheduler decides when a process is to cease running and a new process is to resume running
- preemption - the act of involuntarily suspending a running process
- timeslice - a predetermined time a process runs before it is preempted

I/O-BOUND VERSUS PROCESSOR-BOUND PROCESSES

- I/O-Bound
  - spends much of its time submitting and waiting on I/O requests
  - often runnable, but only for short periods, because it will eventually block waiting on more I/O
- Processor-Bound
  - processor-bound processes spend much of their time executing code
  - run until they are preempted because they do not block
- Linux favors I/O-bound processes - optimizes for process response
- Processor-Bound processes run less frequently but for longer periods.

PROCESS PRIORITY

- Linux uses (dynamic) priority-based scheduling
  - Higher priority process runs before lower priority process
  - Same priority – round robin
  - Process starts with a base priority and the scheduler adjusts this value
- Two separate priority ranges
  - 0 – 99 : RT priority
  - 100 – 139 : user task priority (-20 to 19 nice values)

HISTORY

- 1.2 kernel: scheduler used a circular queue with a round-robin scheduling policy
- 2.4 kernel included a relatively simple scheduler that operated in O(N) time
  - iterated over every task during a scheduling event
  - divided time into epochs, and within each epoch, every task was allowed to execute up to its time slice
- 2.6 kernel: O(1) scheduler
  - kept track of runnable tasks in two run queues for each priority level—one for active and one for expired tasks
  - the scheduler simply needed to dequeue the next task off the a higher-priority queue
  - was much more scalable and incorporated interactivity metrics with numerous heuristics
  - large mass of code needed to calculate heuristics was fundamentally difficult to manage

CFS: COMPLETELY FAIR SCHEDULER

- Introduced in Linux 2.6.23
- Free of heuristics
- Fairness algorithm is simple math
- Extendible framework, which makes it easy to introd uce new scheduler algorithms or even a pluggable scheduler implementation

CFS SCHEDULER CLASSES

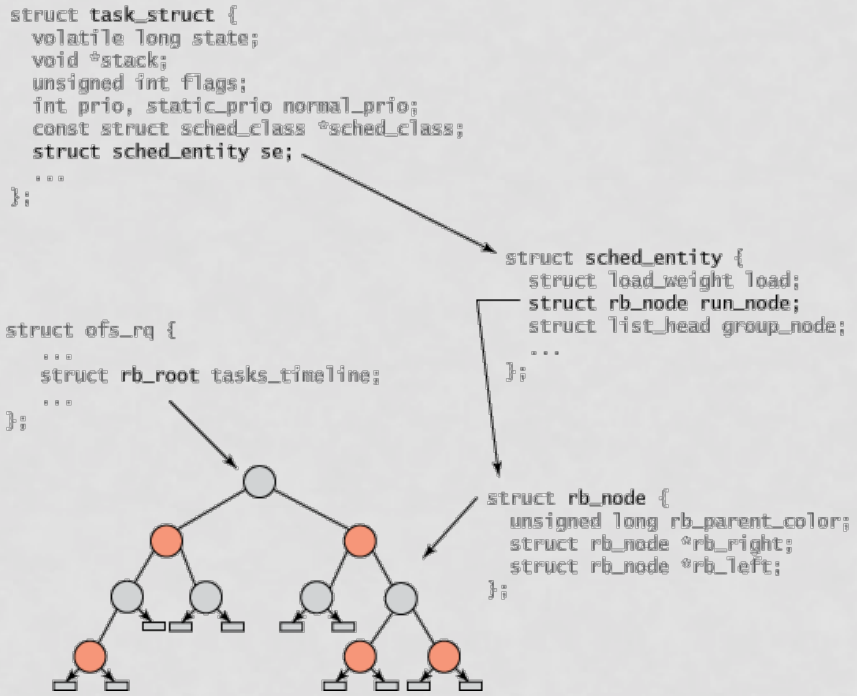
- Extensible hierarchical set of scheduler classes
- rt_sched_class
  - Handles SCHED_FIFO/RR tasks
  - O(1) priority array
- fair_sched_class
  - Handles SCHED_OTHER tasks
  - O(log(N)) red black tree

OVERVIEW OF CFS

- main idea:
  - behind the CFS is to maintain balance (fairness) in providing processor time to tasks
  - when the time for tasks is out of balance then those out-of-balance tasks should be given time to execute.
- maintains the amount of time provided to a given task in what's called the virtual runtime
- includes the concept of sleeper fairness - sleep time is honored by allocating comparable share of the processor when they eventually need it

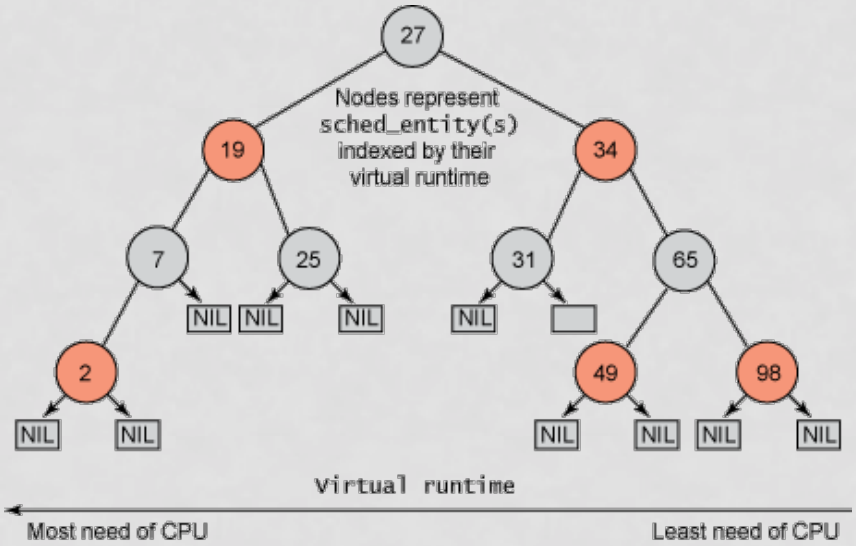
CFS INTERNALS

- new structure called sched_entity tracks scheduling information
- root of the tree is referenced via the rb_root element from the cfs_rq structure (kernel/sched.c)
- Each node in the red-black tree is represented by an rb_node
- rb_node is contained within the sched_entity structure
- the sched_entity contains the vruntime (64-bit field)



THE ALGORITHM

- tasks stored in the time-ordered red-black tree
- tasks with the gravest need for the processor are stored toward the left side of the tree
- scheduler picks the left-most node of the red-black tree to schedule next to maintain fairness
- cpu time of the task is added to the virtual runtime and is then inserted back into the tree
- contents of the tree migrate from the right to the left to maintain fairness



PRIORITY

- CFS doesn't use priorities directly
- uses them as a decay factor for the time a task is permitted to execute
- Lower-priority tasks have higher factors of decay
- time allowed for a lower-priority task to execute is less as compared to higher-priority task

SCHEDULING FUNCTION

- schedule() (./kernel/sched.c)
  - put_prev_task() // currently running task (now preempted) is returned to the red-black tree
  - pick_next_task() // function simply picks the left-most task from the red-black tree and returns the associated sched_entity (./kernel/sched_fair.c)
  - task_of() //identifies the task_struct reference
- The above functions have a specific sched_class implementation

CFS SCHEDULER TUNING

- sched_latency_ns – epoch duration (length) in nanoseconds (20 ms by default)
- sched_min_granularity_ns – granularity of the epoch in nanoseconds (4 ms by default);

$$\texttt{nr_running} > \frac{\texttt{sched_latency_ns}}{\texttt{sched_min_granularity_ns}},$$

$$\texttt{period} = \texttt{sched_min_granularity_ns} \cdot \texttt{nr_running},$$

- sched_wakeup_granularity_ns – this parameter describes the ability of tasks being waken up to preempt the current task (5ms default)
- sched_rt_runtime_us – the maximum CPU time that can be used by all the real-time tasks (1 second by default).
- sched_rt_period_us – the CFS scheduler waits this amount of time (0.95 s by default) before scheduling any of the real-time tasks again

REFERENCE

- http://www.ibm.com/developerworks/library/l-linux-process-management/
- http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/
- https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt
- https://www.fizyka.umk.pl/~jkob/prace-mag/cfs-tuning.pdf