W4118 Operating Systems

Instructor: Junfeng Yang

Outline

- Advanced scheduling issues
    - Multilevel queue scheduling
    - Multiprocessor scheduling issues
    - Real-time scheduling

- Scheduling in Linux
    - Scheduling algorithm
    - Setting priorities and time slices
    - Other implementation issues

Motivation

- No one-size-fits-all scheduler
    - Different workloads
    - Different environment

- Building a general scheduler that works well for all is difficult!

- Real scheduling algorithms are often more complex than the simple scheduling algorithms we've seen

Combining scheduling algorithms

- Multilevel queue scheduling: ready queue is partitioned into multiple queues

- Each queue has its own scheduling algorithm
  - Foreground processes: RR
  - Background processes: FCFS

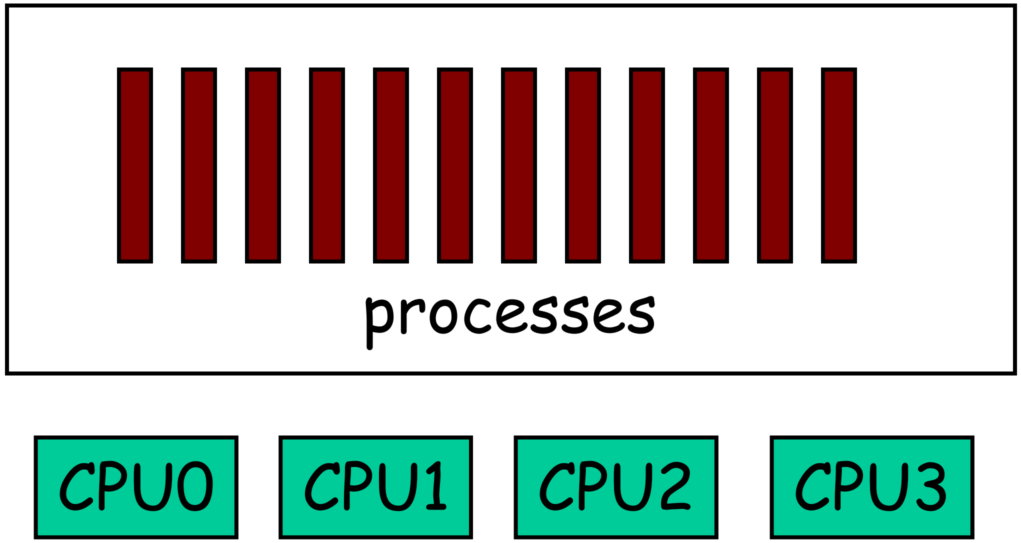
- Must choose scheduling algorithm to schedule between queues. Possible algorithms
  - RR between queues
  - Fixed priority for each queue

Outline

- Advanced scheduling issues
  - Multilevel queue scheduling
  - Multiprocessor scheduling issues
  - Real-time scheduling
- Scheduling in Linux
  - Scheduling algorithm
  - Setting priorities and time slices
  - Other implementation issues

Multiprocessor scheduling issues

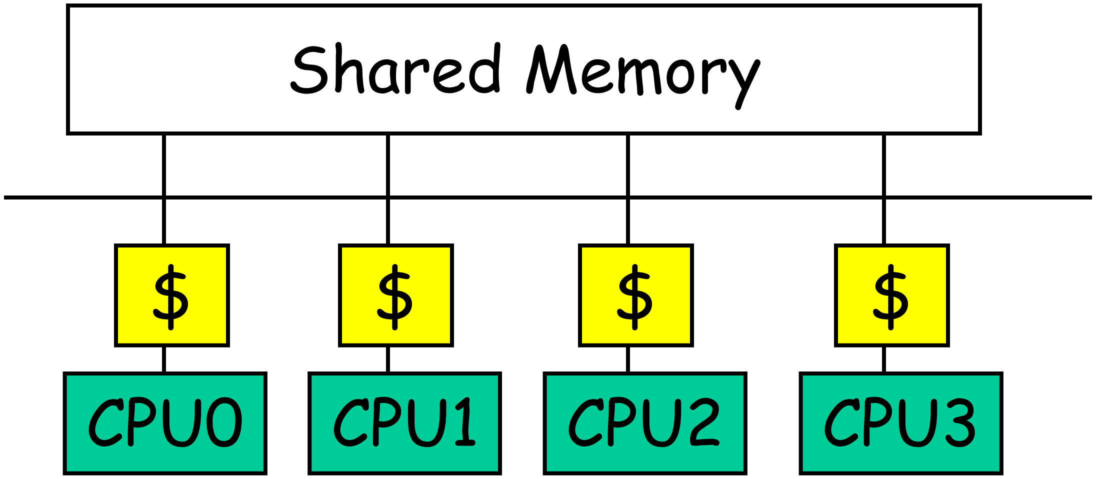
- Shared-memory Multiprocessor



- How to allocate processes to CPU?

Symmetric multiprocessor

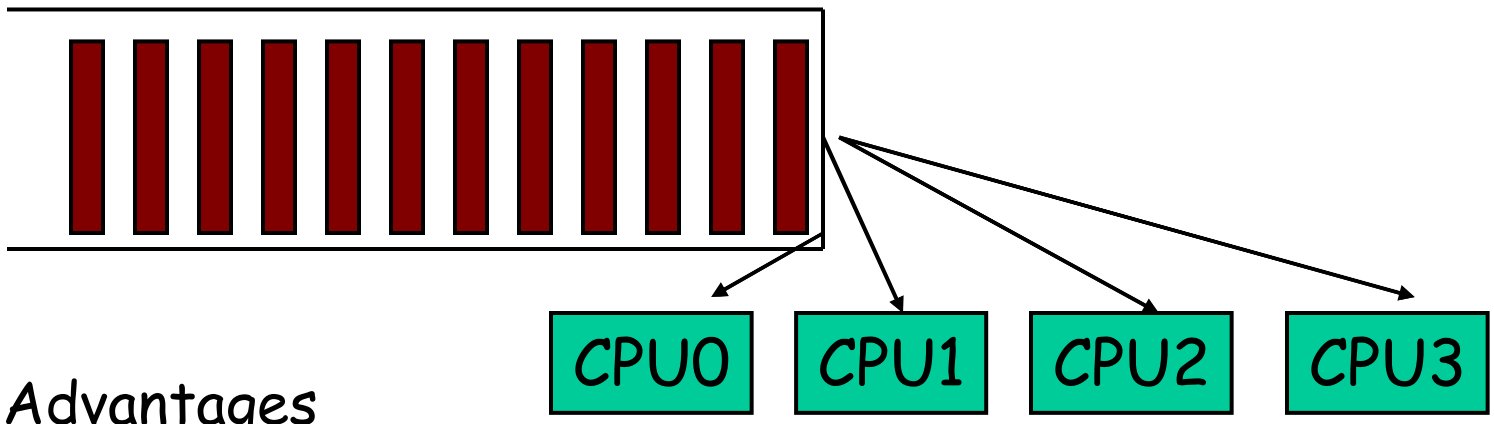
- Architecture



- Small number of CPUs
- Same access time to main memory
- Private cache

Global queue of processes

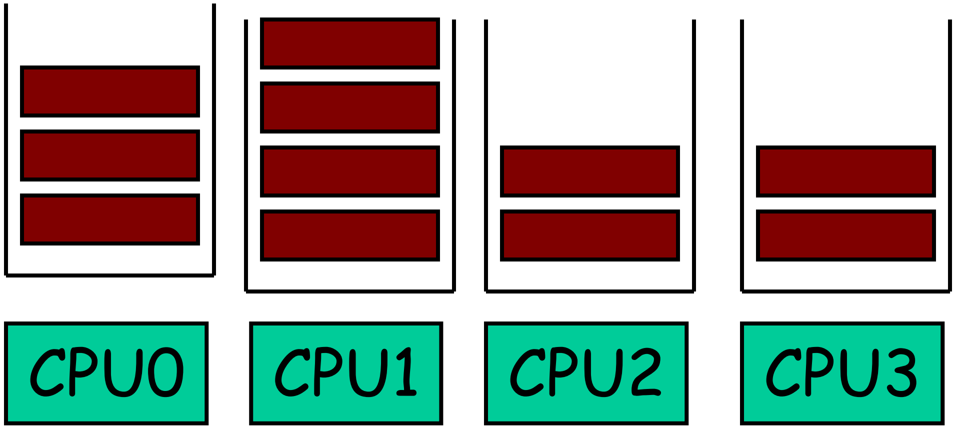
- One ready queue shared across all CPUs



- Advantages
  - Good CPU utilization
  - Fair to all processes
- Disadvantages
  - Not scalable (contention for global queue lock)
  - Poor cache locality
- Linux 2.4 uses global queue

Per-CPU queue of processes

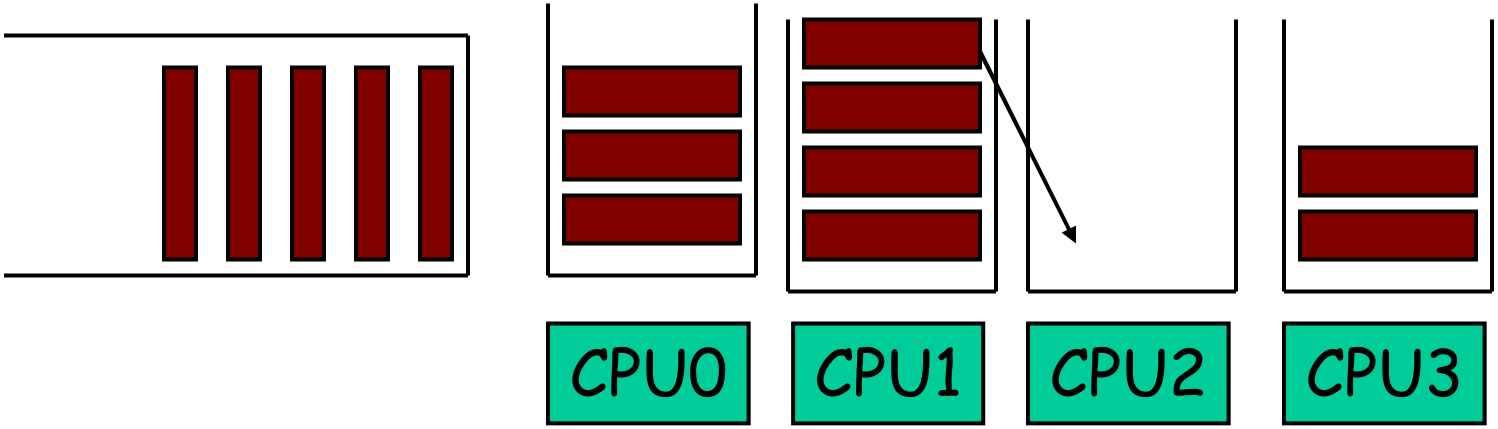
- Static partition of processes to CPUs



- Advantages
    - Easy to implement
    - Scalable (no contention on ready queue)
    - Better cache locality
- Disadvantages
    - Load-imbalance (some CPUs have more processes)
        - Unfair to processes and lower CPU utilization

Hybrid approach

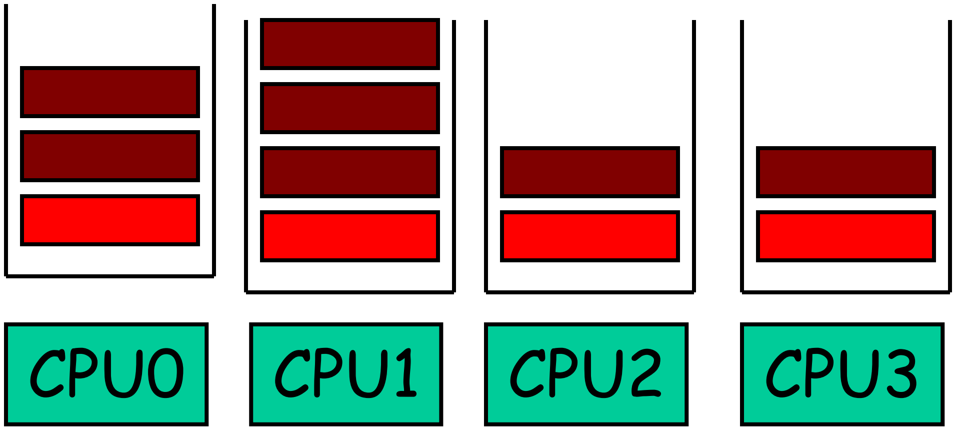
- Use both global and per-CPU queues
- Balance jobs across queues



- Processor Affinity
  - Add process to a CPU's queue if recently run on the CPU
    - Cache state may still present
- Linux 2.6 uses a very similar approach

SMP: “gang“ scheduling

- Multiple processes need coordination
- Should be scheduled simultaneously



- Scheduler on each CPU does not act independently
- Coscheduling (gang scheduling): run a set of processes simultaneously
- Global context-switch across all CPUs

Outline

- Advanced scheduling issues
  - Multilevel queue scheduling
  - Multiprocessor scheduling issues
  - Real-time scheduling

- Scheduling in Linux
  - Scheduling algorithm
  - Setting priorities and time slices
  - Other implementation issues

Real-time scheduling

- Real-time processes have timing constraints
  - Expressed as deadlines or rate requirements
  - E.g. gaming, video/music player, autopilot…

- Hard real-time systems – required to complete a critical task within a guaranteed amount of time
- Soft real-time computing – requires that critical processes receive priority over less fortunate ones

- Linux supports soft real-time

Outline

- Advanced scheduling issues
  - Multilevel queue scheduling
  - Multiprocessor scheduling issues
  - Real-time scheduling

- Scheduling in Linux
  - Scheduling algorithm
  - Setting priorities and time slices
  - Other implementation issues

Linux scheduling goals

- Avoid starvation
- Boost interactivity
  - Fast response to user despite high load
  - Achieved by inferring interactive processes and dynamically increasing their priorities
- Scale well with number of processes
  - O(1) scheduling overhead
- SMP goals
  - Scale well with number of processors
  - Load balance: no CPU should be idle if there is work
  - CPU affinity: no random bouncing of processes
- Reference: Documentation/sched-design.txt

Algorithm overview

- Multilevel Queue Scheduler
  - Each queue associated with a priority
  - A process's priority may be adjusted dynamically

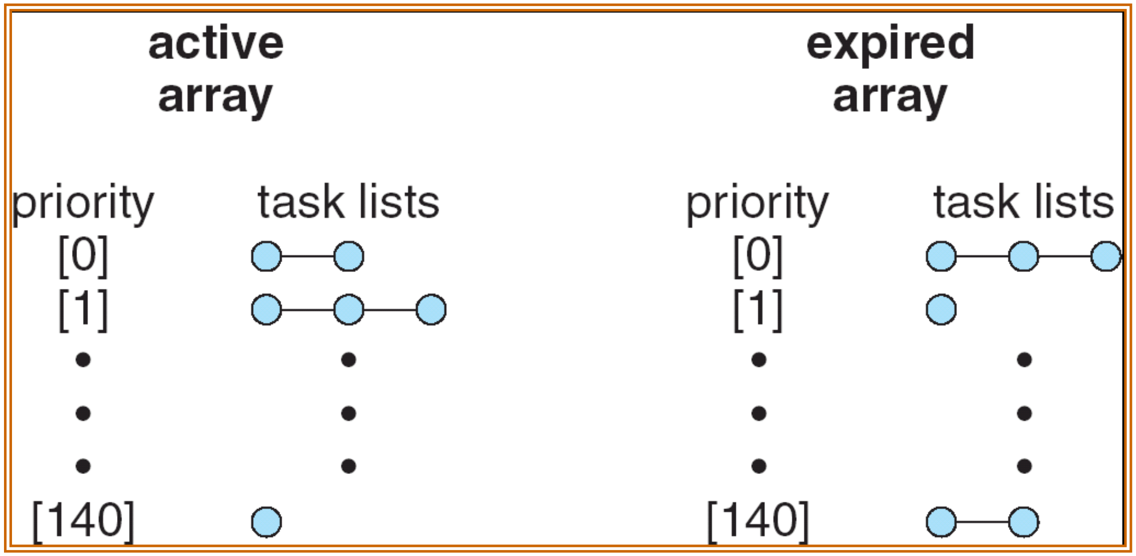
- Two classes of processes
  - Real-time processes: always schedule highest priority processes
    - FCFS (SCHED_FIFO) or RR (SCHED_RR) for processes with same priority
  - Normal processes: priority with aging
    - RR for processes with same priority (SCHED_NORMAL)
    - Aging is implemented efficiently

Priority partition

- Total 140 priorities [0, 140)
    - Smaller integer = higher priority
    - Real-time: [0,100)
    - Normal: [100, 140)
- MAX_PRIO and MAX_RT_PRIO
    - include/linux/sched.h

runqueue data structure

- kernel/sched.c
- *struct prio_array*
  - Array of priority queues
- *struct runqueue*
  - Two arrays, active and expired



Scheduling algorithm

1. Find highest priority non-empty queue in rq->active; if none, simulate aging by swapping active and expired
2. next = first process on that queue
3. Adjust next's priority
4. Context switch to next
5. When next used up its time slice, insert next to the right queue and call schedule again

schedule() in kernel/sched.c

Aging: the traditional algorithm

```
for(pp = proc; pp < proc+NPROC; pp++) {
        if (pp->prio != MAX)
                    pp->prio++;
        if (pp->prio > curproc->prio)
                    reschedule();
}
```

Problem: O(N). Every process is examined on each schedule() call!

This code is taken almost verbatim from 6th Edition Unix, circa 1976.)

Simulate aging

- Swapping active and expired gives low priority processes a chance to run
- Advantage: O(1)
  - Processes are touched only when they start or stop running
- schedule() in kernel/sched.c

Find highest priority non-empty queue

- Use the bitmap field of *struct runqueue*
  - 140 queues → 5 integers
- Time complexity: *O*(1)
  - depends on the number of priority levels, not the number of processes
- Implementation: only a few compares to find the first that is non-zero
  - Hardware instruction to find the first 1-bit
    - bsfl on Intel
- sched_find_first_bit() in include/asm-i386/bitops.h

Outline

- ❑ Advanced scheduling issues
  - Multilevel queue scheduling
  - Multiprocessor scheduling issues
  - Real-time scheduling
- ❑ Scheduling in Linux
  - Scheduling algorithm
  - Setting priorities and time slices
  - Other implementation issues

Priority related fields in *struct task_struct*

- static_prio: static priority set by administrator/users
    - Default: 120 (even for realtime processes)
    - Set use sys_nice() or sys_setpriority()
        - Both call set_user_nice()
- prio: dynamic priority
    - Index to prio_array
- rt_priority: real time priority
    - prio = 99 - rt_priority
- include/linux/sched.h

Adjusting priority

- Goal: dynamically increase priority of interactive process

- How to determine interactive?
  - Sleep ratio
  - Mostly sleeping: I/O bound
  - Mostly running: CPU bound

- Implementation: sleep_avg in *struct task_struct*
  - Before switching out a process, subtract from sleep_avg how many ticks a task ran, in schedule()
  - Before switching in a process, add to sleep_avg how many ticks it was blocked up to MAX_SLEEP_AVG (10 ms), in schedule() ➔ recalc_task_prio() ➔ effective_prio()

Calculating time slices

- Stored in field time_slice in struct task_struct
- Higher priority processes also get bigger time-slice
- task_timeslice() in sched.c
  - If (static_priority < 120) time_slice = (140-static_priority) * 20
  - If (static_priority >= 120) time_slice = (140-static_priority) * 5

Example time slices

Priority:	Static Pri	Niceness	Quantum
Highest	100	-20	800 ms
High	110	-10	600 ms
Normal	120	0	100 ms
Low	130	10	50 ms
Lowest	139	20	5 ms

Outline

- Advanced scheduling issues
  - Multilevel queue scheduling
  - Multiprocessor scheduling issues
  - Real-time scheduling

- Scheduling in Linux
  - Scheduling algorithm
  - Setting priorities and time slices
  - Other implementation issues

Bookkeeping on each timer interrupt

- scheduler_tick()
  - Called on each tick
    - timer_interrupt ➔ do_timer_interrupt ➔ do_timer_interrupt_hook ➔ update_process_times
- If realtime and SCHED_FIFO, do nothing
  - SCHED_FIFO is non-preemptive
- If realtime and SCHED_RR and used up time slice, move to end of rq->active[prio]
- If SCHED_NORMAL and used up time slice
  - If not interactive or starving expired queue, move to end of rq->expired[prio]
  - Otherwise, move to end of rq->active[prio]
    - Boost interactive
- Else // SCHED_NORMAL, and not used up time slice
  - Break large time slice into pieces TIMESLICE_GRANULARITY

Real-time scheduling

- Linux has soft real-time scheduling
  - No hard real-time guarantees
- All real-time processes are higher priority than any conventional processes
- Processes with priorities [0, 99] are real-time
  - saved in rt_priority in the *task_struct*
  - scheduling priority of a real time task is: 99 - rt_priority
- Process can be converted to real-time via sched_setscheduler system call

Real-time policies

- First-in, first-out: SCHED_FIFO
  - Static priority
  - Process is only preempted for a higher-priority process
  - No time quanta; it runs until it blocks or yields voluntarily
  - RR within same priority level
- Round-robin: SCHED_RR
  - As above but with a time quanta
- Normal processes have SCHED_NORMAL scheduling policy

Multiprocessor scheduling

- Per-CPU runqueue
- Possible for one processor to be idle while others have jobs waiting in their run queues
- Periodically, rebalance runqueues
  - Migration threads move processes from one runque to another
- The kernel always locks runqueues in the same order for deadlock prevention

Processor affinity

- Each process has a bitmask saying what CPUs it can run on
  - By default, all CPUs
  - Processes can change the mask
  - Inherited by child processes (and threads), thus tending to keep them on the same CPU
- Rebalancing does not override affinity