

## Lecture 8: Scheduling in modern OS

### Previous lecture review

- † Out of basic scheduling techniques none is a clear winner:
  - † FCFS - simple but unfair
  - † RR - more overhead than FCFS may not be fair
  - † SJF - optimal, but high overhead, starvation possible
  - † SRF - optimal, even higher overhead, starvation possible
- † combined techniques:
  - † priority scheduling
  - † multiple feedback scheduling
- † all are incorporated in modern OS scheduler design
  - † non-preemptive kernel to avoid kernel data corruption
  - † classical Unix (SVR3, 4.3BSD) scheduler design
  - † real-time requirements
  - † SVR4 scheduling improvements
  - † Solaris scheduling improvements
  - † multiprocessor scheduling

1

2

### Non-Preemptive Kernel

- † To prevent data structures (especially kernel structures) corruption by simultaneous by different processes/threads we need to ensure *mutual exclusion* of access
- † classical approach
  - † make kernel non-preemptive - process in kernel mode cannot be suspended when it is in the middle of a shared structure modification
  - † disable interrupts when vital structures are modified (see Nachos lecture) - interrupt handler cannot corrupt shared structures modified by kernel
- † problems:
  - † can be unfair/does not scale
  - † cannot be used for real-time scheduling
  - † cannot be used for multiprocessor systems

3

### Classical Unix CPU Scheduling (System V release 3(SVR3), 4.3BSD)

- † Policy:
  - † Multiple queues (32), each with a priority value - 0-127 (low value = high priority):
    - † Kernel processes (or user processes in kernel mode) the lower values (0-49) - kernel processes are not preemptive!
    - † User processes have higher value (50-127)
  - † Choose the process from the occupied queue with the highest priority, and run that process preemptively, using a timer (time slice typically around 100ms)
    - † Round-robin scheduling in each queue
  - † Move processes between queues
    - † Keep track of clock ticks (60/second)
    - † add the number of clock ticks to processes in the ready queue
    - † Also change priority based on whether or not process has used more than it's "fair share" of CPU time (compared to others)
  - † users can decrease (but not increase!) priority

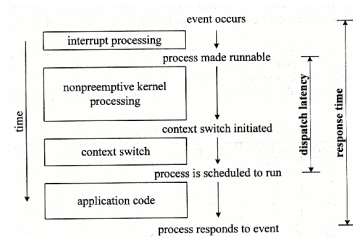
4

### Analysis of classical UNIX CPU scheduling

- † advantages:
  - † simple (relatively) and effective
  - † ok for general purpose, single processor, small sized systems
- † disadvantages:
  - † recomputing priorities every second is inefficient in large systems
  - † no response time guarantee
  - † *priority inversion* - a high priority process has to wait for a lower priority process which is in kernel space (non-preemptive kernel) - some kernel code paths take several ms
  - † applications do not have adequate control over priority (only superuser can increase)

5

### Real-time scheduling



Reproduced from "Unix Internals" by Uresh Vahalia

- † Soft real-time capabilities are needed for quality of service sensitive applications - video, audio, multimedia, virtual reality
- † require bounded dispatch latency, and response time

- † *dispatch latency* - time from the moment the process becomes runnable to the moment it begins to run
- † *response time* = interrupt processing + dispatch latency + real-time process execution

6

## System V release 4 (SVR4) scheduling

- † Scheduling classes (in the order of priority):
  - † real-time - fixed priority and time slices
  - † system - kernel
  - † time-sharing (default) - RR scheduling, dynamic priorities, lower priority processes are given larger time slices (to offset overall I/O favoring)
- † possibility of adding other classes (dynamic loading of scheduler implementations)
- † on-event priority recompilation - priority changes on specific events
  - † priority reduced when process uses up time slice
  - † priority upped if process blocks
- † kernel preemptive in preemption points - points defined where it is safe to preempt the kernel

7

## Analysis of SVR4 scheduling

- † Advantages
  - † flexible, allows real-time, scalable
  - † modifiable (allows to add classes)
  - † efficient priority computation
  - † more balanced scheduling between I/O and CPU bound processes
- † Problems
  - † switching time sharing -> real-time is not allowed - hand tuning required and not always possible
  - † kernel is not completely preemptible
  - † no multiprocessor support

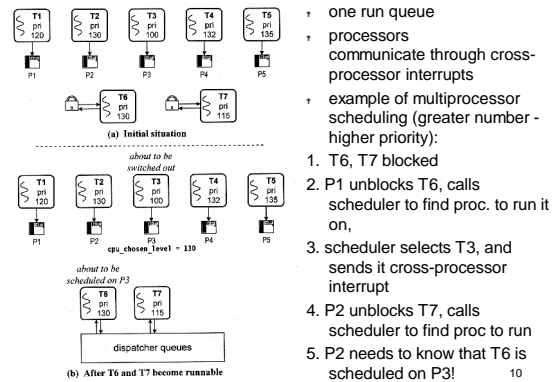
8

## Solaris Scheduling

- † fully preemptive kernel, shared kernel structures are protected by explicit synchronization mechanisms
- † kernel is multithreaded, interrupts are implemented as threads - no need to change interrupt level
- † symmetric multiprocessor scheduling
- † *priority inheritance* or *priority lending* (solves priority inversion problem) - when a higher priority thread is needs a resources used by a lower priority thread - the higher priority thread *lends* its priority to the lower priority thread; must be transitive!
- † Does not have hard real-time capabilities

9

## Multiprocessor scheduling in Solaris



- † one run queue
  - † processors communicate through cross-processor interrupts
  - † example of multiprocessor scheduling (greater number - higher priority):
1. T6, T7 blocked
  2. P1 unblocks T6, calls scheduler to find proc. to run it on,
  3. scheduler selects T3, and sends it cross-processor interrupt
  4. P2 unblocks T7, calls scheduler to find proc to run
  5. P2 needs to know that T6 is scheduled on P3!

10