### **Lecture Overview**

- Introduction to Linux process scheduling
  - Policy versus algorithm
  - Linux' overall process scheduling objectives
    - Timesharing
    - Dynamic priority
    - Favor I/O-bound process
  - Linux' scheduling algorithm
    - Dividing time into epochs
    - Remaining quantum as process priority
    - When scheduling occurs

Operating Systems - May 17, 2001

# **Linux Process Scheduling Policy**

- First we examine Linux' scheduling policy
  - A scheduling policy is the set of decisions you make regarding scheduling priorities, goals, and objectives
  - A scheduling algorithm is the instructions or code that implements a given scheduling policy
- Linux has several, conflicting objectives
  - Fast process response time
  - Good throughput for background jobs
  - Avoidance of process starvation
  - etc.

# **Linux Process Scheduling Policy**

- Linux uses a timesharing technique
  - We know that this means that each process is assigned a small quantum or time slice that it is allowed to execute
    - This relies on hardware timer interrupts and is completely transparent to the processes
- Linux schedule process according to a priority ranking, this is a "goodness" ranking
  - Linux uses *dynamic priorities*, i.e., priorities are adjusted over time to eliminate starvation
    - Processes that have not received the CPU for a long time get their priorities increased, processes that have received the CPU often get their priorities decreased



## **Linux Process Scheduling Policy**

- Linux uses process preemption, a process is preempted when
  - Its time quantum has expired
  - A new process enters TASK\_RUNNING state and its priority is greater than the priority of the currently running process
    - The preempted process is not suspended, it is still in the ready queue, it simply no longer has the CPU
- Consider a text editor and a compiler
  - Since the text editor is an interactive program, its dynamic priority is higher than the compiler
  - The text editor will be block often since it is waiting for I/O
  - When the I/O interrupt receives a key-press for the editor, the editor is put on the ready queue and the scheduler is called since the editor's priority is higher than the compiler
  - The editor gets the input and quickly blocks for more I/O



- The Linux scheduling algorithm is not based on a continuous CPU time axis, instead it divides the CPU time into *epochs* 
  - An epoch is a division of time or a period of time
  - In a single epoch, every process has a specified time quantum that is computed at the beginning of each epoch
    - This is the maximum CPU time that the process can use during the current epoch
  - A process only uses its quantum when it is executing on the CPU, when the process is waiting for I/O its quantum is not used
    - As a result, a process can get the CPU many times in one epoch, until its quantum is fully used
  - An epoch ends when all <u>runnable</u> processes have used all of their quantum
    - The a new epoch starts and all process get a new quantum



- Calculating process quanta for an epoch
  - Each process is initially assigned a *base time quantum*, as mentioned previously it is about 20 "clock ticks"
  - If a process uses its entire quantum in the current epoch, then in the next epoch it will get the base time quantum again
  - If a process does not use its entire quantum, then the unused quantum carries over into the next epoch (the unused quantum is not directly used, but a "bonus" is calculated)
    - Why? Process that block often will not use their quantum; this is used to favor I/O-bound processes because this value is used to calculate priority
  - When forking a new child process, the parent process' remaining quantum divided in half; half for the parent and half for the child



- Scheduling data in the process descriptor
  - The process descriptor (task\_struct in Linux) holds essentially of the information for a process, including scheduling information
  - Recall that Linux keeps a list of all process task\_structs and a list of all ready process task\_structs
  - The next two slides describe the relevant scheduling fields in the process descriptor

- Each process descriptor (task\_struct) contains the following fields
  - need\_resched this flag is checked every time an interrupt handler completes to decide if rescheduling is necessary
  - policy the scheduling class for the process
    - For real-time processes this can have the value of
      - SCHED\_FIFO first-in, first-out with unlimited time quantum
      - SCHED\_RR round-robin with time quantum, fair CPU usage
    - For all other processes the value is - SCHED OTHER
    - For processes that have yielded the CPU, the value is - SCHED\_YIELD

- Process descriptor fields (con't)
  - rt\_priority the static priority of a real-time process, not used for other processes
  - priority the base time quantum (or base priority) of the process
  - counter the number of CPU ticks left in its quantum for the current epoch
    - This field is updated every clock tick by update\_process\_times()
  - The priority and counter fields are used to for *time-sharing and dynamic priorities* in conventional processes, for *only time-sharing* in SCHED\_RR real-time processes, and are *not used at all* for SCHED\_FIFO real-time processes



- Scheduling actually occurs in schedule()
  - Its objective is to find a process in the ready queue then assign the CPU to it
  - It is invoked in two ways
    - Direct invocation
    - Lazy invocation

- Direct invocation of schedule()
  - Occurs when the current process is going to block because it needs to wait for a necessary resource
    - The current process is taken off of the ready queue and is placed on the appropriate wait queue; its state is changed to TASK\_INTERRUPTIBLE or TASK\_UNINTERRUPTIBLE
  - Once the needed resource becomes available, the process is immediately woken up and remove from the wait queue

- Lazy invocation of schedule()
  - Occurs when
    - The current process has used up its quantum; this is checked in update\_process\_times()
    - A process is added to the ready queue and its priority is higher than the currently executing process; this check occurs in wake\_up\_process()
    - A process calls sched\_yield()
  - Lazy invocation used the need\_resched flag of the process descriptor and will cause schedule() to be called later



- Actions performed by schedule() (con't)
  - It scans the ready queue for the highest priority process
    - It calculates the priority using the goodness() function
    - It may not find any processes that are "good" when all processes on the ready queue have used up their quantum (i.e., all have a zero counter field)
      - In this case it must start a new epoch by assigned a new quantum to all processes as described on a previous slide (both running and blocked processes this allows us to favor I/O-bound processes)
    - If a higher priority process was found, then the scheduler performs a process switch



- Linux scheduler issues
  - Does not scale very well as the number of process grows because it has to recompute dynamic priorities
    - Tries to minimize this by computing at end of epoch only
    - Large numbers of runnable processes can slow response time
  - Predefined quantum is too long for high system loads
  - I/O-bound process boosting is not optimal
    - Some I/O-bound processes are not interactive (e.g., database search or network transfer)
  - Support for real-time processes is weak

- Computer hardware
  - In general, we can think of the CPU as a small, selfcontained computer
    - It has instructions for performing mathematical operations
    - It has a small amount of storage space (its registers)
    - We can feed instructions to the CPU one at a time and use it to perform complex calculations
      - This is the ultimate in "interactive" operation; the user does everything
      - It would be better if there was some way to give the CPU a lot of instructions all at once, rather than one at a time

- Computer hardware (con't)
  - We need to combine the CPU with RAM and a memory bus
    - The bus connects the CPU to the RAM and allows the CPU to access address location contents
    - Since we are going to load many instructions (i.e., a program) into memory, the CPU must have a special register to keep track of the current instruction, the *program counter* 
      - The program counter is incremented after each instruction
      - Some instructions directly set the value of the program counter, like JUMP or GOTO instruction



- Computer hardware (con't)
  - Now we add I/O devices to the communication bus
    - The CPU communicates with I/O devices via the bus
    - This allows user interaction with the program (e.g., via a terminal)
    - This also allows more data and bigger programs (e.g., stored on a disk)
    - Since the CPU is much faster than the I/O devices it has three options when performing I/O
      - It can simply wait (not very efficient)
      - It can poll the device and try to do other work at the same time (complicated to implement and not necessarily timely)
      - It can allow the I/O devices to notify it when they are done via interrupts (still a bit complicated, but efficient and timely)



- Providing an Operating System
  - We could get a bigger benefit if we could run more than one program at once (i.e., time-sharing)
    - With time-sharing the CPU can execute other program when the current program blocks for I/O
      - This introduces the notion of a *process* (i.e., an executing program)
  - Currently we have no way of interrupting the current process and starting a new process, there are two options
    - Implement all I/O calls to give up CPU when they might block; this is *cooperative multitasking*
    - Add a *hardware timer interrupt* to our CPU so that we can automatically interrupt processes after some amount of time; this is called *preemptive multitasking*



- Providing an Operating System (con't)
  - On a uniprocessor system, a process can only make progress when it has the CPU and only one process can have the CPU at a time
  - How does the OS share the CPU among multiple processes?
    - It preempts the current process (or the current process cooperatively blocks) and the OS chooses another process for the CPU

- Providing an Operating System (con't)
  - What happens when a process is preempted?
    - The OS must save the CPU registers for the current process since they contain unfinished work; the CPU registers are saved in the *process descriptor* in RAM
      - The process descriptor keeps track of all process information for a specific process
    - The OS must also save the program counter in the process descriptor so it knows where to resume the current process
    - For the new process, the OS must restore its CPU registers from the saved values in the process descriptor and restore the program counter to the next instruction for the new process

