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

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

- We can classify processes using two schemes
  - CPU-bound versus I/O-bound
    - We learned about this in previous lectures
  - Interactive versus batch versus real-time
    - We have talked about these concepts in previous lectures, so they should be relatively self-explanatory
      - These classifications are somewhat independent, e.g., a batch process can be either I/O-bound or CPU-bound
- Linux recognizes real-time programs and assigns them high priority, but this is only soft real-time, like streaming audio
- Linux does not recognize batch or interactive processes, instead it implicitly favors I/O-bound processes
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

• Determining the length of the quantum
  – Should be neither too long or too short
  – If too short, the overhead caused by process switching becomes excessively high
  – If too long, processes no longer appear to be executing concurrently
    • For Linux, long quanta do not necessarily degrade response time for interactive processes because their dynamic priority remains high, thus they get the CPU as soon as they need it
    • For long quanta, responsiveness can degrade in instances where the scheduler does not know if a process is interactive or not, such as when a process is newly created
  – The for Linux is the longest possible quantum without affecting responsiveness; this turns out to be about 20 “clock ticks” or 210 milliseconds
Linux Process Scheduling Algorithm

- 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 runnable processes have used all of their quantum
    - The a new epoch starts and all process get a new quantum

Linux Process Scheduling Algorithm

- When does an epoch end?
  Important!
  - An epoch ends when all processes in the ready queue have used their quantum
  - This does not include processes that are blocking on some wait queue, they will still have quantum remaining
  - The end of an epoch is *only* concerned with processes on the ready queue
Linux Process Scheduling Algorithm

• 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

Linux Process Scheduling Algorithm

• Selecting a process to run next
  – The scheduler considers the priority of each process
  – There are two kinds of priorities
    • Static priorities - these are assigned to real-time processes and range from 1 to 99; they never change
    • Dynamic priorities - these apply to all other processes and it is the sum of the base time quantum (also called the base priority) and the number of “clock ticks” left in the current epoch
  – The static priority of real-time process is always higher than the dynamic priority of conventional processes
    • Conventional processes will only execute when there are no real-time processes to execute
Linux Process Scheduling Algorithm

• 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

Linux Process Scheduling Algorithm

• 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
Linux Process Scheduling Algorithm

• 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

Important!

  – Just in case you missed it in the last bullet of the last slide, the priority and counter fields are used to for calculating the dynamic priority of conventional processes
    • These two fields are added together to get the current dynamic priority of a process when searching for a process to schedule
    • These two fields are also used when assigning new quanta at the end of an epoch; if a process has not used its quantum then it is probably an I/O-bound process and will get a bonus added to its quantum for the next epoch
      – This raises the priority of interactive processes over time
Linux Process Scheduling Algorithm

• 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

Linux Process Scheduling Algorithm

• 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
**Linux Process Scheduling Algorithm**

- **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*

**Linux Process Scheduling Algorithm**

- **Actions performed by schedule()**
  - First it runs any kernel control paths that have not completed and other uncompleted house-keeping tasks
    - Remember, the kernel is not preemptive, so it cannot switch to another process if a process is already in the kernel or if the kernel is in the middle of doing something else
  - If the current process is SCHED_RR and has used all of its quantum, then it is given a new quantum and placed at the end of the ready queue
  - If the process is not SCHED_RR, then it is removed from the ready queue
• 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

• How good is a runnable process?
  – Uses goodness() to determine priority
    • (goodness == -1000) - do not select process
    • (goodness == 0) - process has exhausted quantum
    • (0 < goodness < 1000) - conventional process with quantum
    • (goodness >= 1000) - real-time process
  – goodness() is essentially equivalent to
    ```c
    if (p->policy != SCHED_OTHER)
        return 1000 + p->rt_priority;
    if (p->counter == 0)
        return 0;
    if (p->mm == prev->mm)
        return p->counter + p->priority + 1;
    return p->counter + p->priority;
    ```
Linux Process Scheduling Algorithm

- 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

Review of Lectures So Far

- Computer hardware
  - In general, we can think of the CPU as a small, self-contained 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
Review of Lectures So Far

• 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

Review of Lectures So Far

• Computer hardware (con’t)
  – We need to combine the CPU with RAM and a memory bus
    (con’t)
    • By adding memory we must extend the operations that the CPU needs to perform, it needs instructions to read/write to/from memory
    • We can use memory for two purposes now
      – Storing instructions (the program code)
      – Storing data
    • This doesn’t allow us to interact with the program and memory is still pretty expensive for its size
## Review of Lectures So Far

**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)

---

## Review of Lectures So Far

**Computer hardware (con’t)**

- Up until this point we have described what amounts to a simple, but reasonable computer system
  - This system stores programs and data on disks
  - It executes a one program at a time by loading a program’s instructions into memory and sets the program counter to the first instruction of the program
  - A program runs until completion and has complete access to the hardware and I/O devices
    - *There really isn’t much of an operating system and no such thing as a process*
  - This is good, but a lot of the time the CPU is just sitting around with nothing to do because the program is waiting for I/O
Review of Lectures So Far

• 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

Review of Lectures So Far

• 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
Review of Lectures So Far

• 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

Review of Lectures So Far

• Providing an Operating System (con’t)
  – We now have created a multitasking OS
  – Is it a concurrent system?
    • English definition of “concurrent”
      – Happening at the same time as something else
    • Computer science definition of “concurrent”
      – Non-sequential
    • Definition of “parallel”
      – Happening at the same time as something else
      – This is the same as the English meaning of “concurrent”
    • In computer science something that is parallel is also concurrent (i.e., non-sequential), but something that is concurrent is not necessarily parallel