Linux

CS 256/456

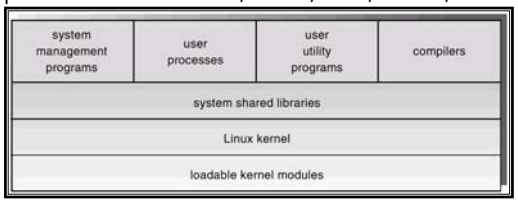
Dept. of Computer Science, University of Rochester

History

- Linux is a modem, free operating system based on UNIX standards.
- First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility.
- The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code.
- Its history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet.

Linux Kernel/System/Distribution

- Kernel
  - the OS code that runs on privileged mode
- System
  - essential system components, but runs in user mode
  - compilers & system libraries
- Linux distribution
  - extra system-installation and management utilities
  - precompiled and ready-to-install tools & packages
  - popular distributions: Redhat, Debian, SuSE, Caldera, …



Kernel Modules

- Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel.
- A kernel module may typically implement a device driver, a file system, or a networking protocol.
- The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems (that could not be distributed under the GPL).
- Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in.

Processes and Threads

- Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent.
- A distinction is only made when a new thread is created by the **clone** system call.
  - a process is a task with its own entirely new context (including address space)
  - a thread is a task with its own identity, but not an dedicated address space
- The **clone** system call allows fine-grained control over exactly what is shared between two threads.
  - open files, memory space, page tables

Linux Task Scheduling

- Linux uses two task-scheduling classes:
  - A time-sharing class for fair preemptive scheduling between multiple tasks.
  - A real-time class that conforms to POSIX real-time standard. (FIFO/RR)
- A prioritized, epoch-based algorithm for time-sharing
  - Each task has a static priority (default=20) and an dynamic quantum (initially set to priority)
  - Scheduling is prioritized based on "quantum+priority"
  - Quantum of the running task decrements by one at every clock tick
  - Recrediting when no runnable tasks have any quantum (end of an epoch):

$$\text{initial quantum in new epoch} = \frac{\text{remaining quantum}}{2} + \text{priority}$$

  - This quantum crediting system automatically prioritizes interactive or I/O-bound tasks.

Linux Task Scheduling: O(1) Scheduler

- Linux O(1) scheduler
  - the scheduling overhead is constant, which is independent of the number of processes in the system
- Main operations in Linux scheduler
  - schedule(), quantum recalculation, wakeup()
- Using two priority arrays
  - one for active array, one for those whole quantum has been used up (called "expired")
  - O(1) operation to find the highest priority in a particular array: use array index bitmap and BSFL
  - problem: have to use static priority since array index indicates the priority and it is too costly to move tasks in the array

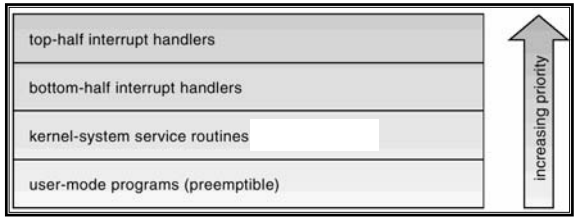
Kernel Synchronization

- When kernel-mode execution occurs?
  - system call or software trap.
  - hardware interrupts.
- Kernel synchronization: allow the kernel's critical sections to run without interruption by conflicting critical section.
  - disabling the interrupts

Kernel Synchronization (Cont.)

- To avoid performance penalties, Linux's kernel uses a synchronization architecture that allows long service routines to run without having interrupts disabled for the critical section's entire duration.
- Kernel routines (often interrupt service routines) are separated into a *top half* (urgent) and a *bottom half* (not so urgent).
  - The top half runs with interrupts disabled.
  - The bottom half is run, with all interrupts enabled.
  - Bottom halves do not interrupt themselves.

Interrupt Protection Levels



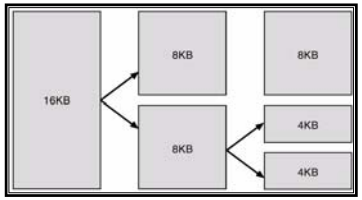
- Each level may be interrupted by code running at a higher level, but will never be interrupted by code running at the same or a lower level.

Multiprocessor

- Linux 2.0 was the first Linux kernel to support SMP hardware; separate processes or threads can execute in parallel on separate processors.
- To preserve the kernel's nonpreemptible synchronization requirements, SMP imposes the restriction, via a single kernel spinlock, that only one processor at a time may execute kernel-mode code.
- The nonpreemptible restriction is removed in recent kernels.

Managing Physical Memory

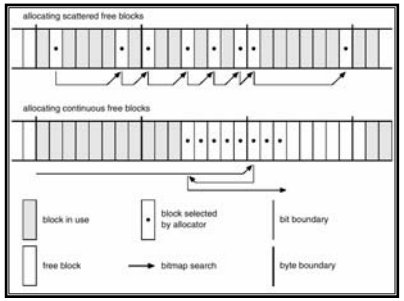
- Keep track of free memory?
- Linux page allocator can allocate ranges of physically-contiguous pages on request.
- The allocator uses a *buddy-heap* algorithm to keep track of available physically-contiguous memory regions.



- A free region list is maintained for each region size

Ext2fs Block-Allocation Policies

- Ext2fs uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation.



The Linux /proc File System

- The **proc** file system does not store data, rather, its contents are computed on demand according to user file I/O requests.
  - When data is read from one of these files, **proc** collects the appropriate information, formats it into text form and places it into the requesting process's read buffer.

Prefetching and I/O Scheduling

- File prefetching/read-ahead
  - Linux prefetching sequences
- Disk I/O scheduling
  - seek-reduction scheduling: SSTF, LOOK, SCAN
  - non-work conserving scheduling: anticipatory scheduling with deadline to prevent starvation

Disclaimer

- Parts of the lecture slides contain original work of Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, and Gary Nutt. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).