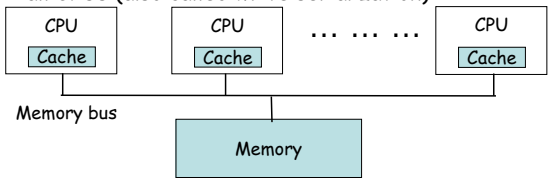
Multiprocessor Operating Systems

CS 256/456

Dept. of Computer Science, University of Rochester

Multiprocessor Hardware

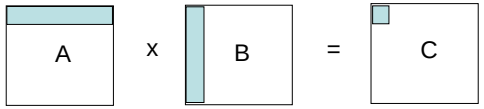
- A computer system in which two or more CPUs share full access to the main memory
- Each CPU might have its own cache and the coherence among multiple caches is maintained
  - write operation by a CPU is visible to all other CPUs
  - writes to the same location is seen in the same order by all CPUs (also called write serialization)



  - bus snooping and cache invalidation

Multiprocessor Applications

- Multiprogramming
  - Multiple regular applications running concurrently
- Concurrent servers
  - Web servers, … …
- Parallel programs
  - Utilizing multiple processors to complete one task (parallel matrix multiplication, Gaussian elimination)

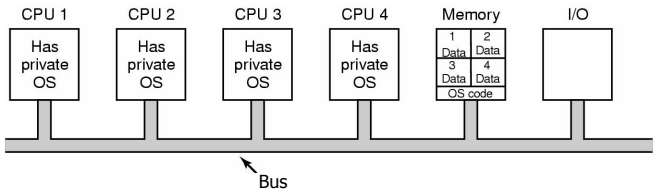


  - Strong synchronization

Single processor OS vs. Multi-processor OS

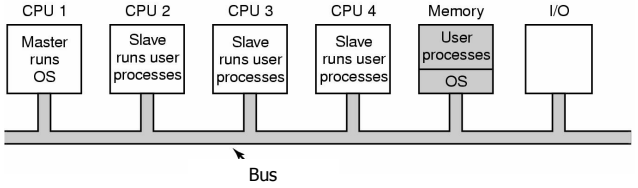
- Single-processor OS
  - easier to support kernel synchronization
    - fine-grained locking vs. coarse-grained locking
    - disabling interrupts to prevent concurrent executions
  - easier to perform scheduling
    - which to run, not where to run
- Multi-processor OS
  - evolution of OS structure
  - synchronization
  - scheduling

Multiprocessor OS



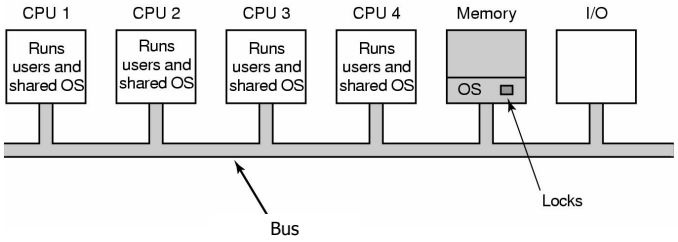
- Each CPU has its own operating system
  - quick to port from a single-processor OS
- Disadvantages
  - difficult to share things (processing cycles, memory, buffer cache)

Multiprocessor OS – Master/Slave



- All operating system functionality goes to one CPU
  - no multiprocessor concurrency in the kernel
- Disadvantage
  - OS CPU consumption may be large so the OS CPU becomes the bottleneck (especially in a machine with many CPUs)

Multiprocessor OS – Shared OS



- A single OS instance may run on all CPUs
- The OS itself must handle multiprocessor synchronization
  - multiple OS instances from multiple CPUs may access shared data structure

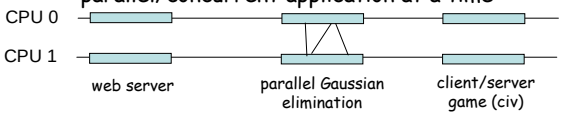
Synchronization (Fine/Coarse Grain Locking)

- Fine-grain locking – only locking necessary for critical section
- Coarse-grain locking – locking large piece of code, much of which is unnecessary
  - simplicity, robustness
  - prevent simultaneous execution

Simultaneous execution is not possible on uniprocessor anyway

Multiprocessor Scheduling

- Timesharing
  - similar to uni-processor scheduling – one queue of ready tasks (protected by synchronization), a task is dequeued and executed when a processor is available
- Space sharing
- cache affinity
  - affinity-based scheduling – try to run each process on the processor that it last ran on
- caching sharing and synchronization of parallel/concurrent applications
  - gang/cohort scheduling – utilize all CPUs for one parallel/concurrent application at a time

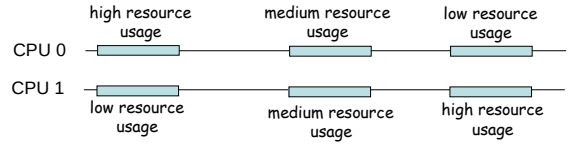


Resource Contention-Aware Scheduling I

- Hardware resource sharing/contention in multi-processors
  - SMP processors share memory bus bandwidths
  - Multi-core processors share L2 cache
  - SMT processors share a lot more stuff
- An example: on an SMP machine
  - a web server benchmark delivers around 6300 reqs/sec on one processor, but only around 9500 reqs/sec on an SMP with 4 processors
- Contention-reduction scheduling
  - co-scheduling tasks with complementary resource needs (a computation-heavy task and a memory access-heavy task)
  - In [Fedorova et al. USENIX2005], IPC is used to distinguish computation-heavy tasks from memory access-heavy tasks

Resource Contention-Aware Scheduling II

- What if contention on a resource is unavoidable?
- Two evils of contention
  - high contention ⇒ performance slowdown
  - fluctuating contention ⇒ uneven application progress over the same amount of time ⇒ poor fairness
- [Zhang et al. HotOS2007] Scheduling so that:
  - very high contention is avoided
  - the resource contention is kept stable



Anderson et al. 1989 (IEEE TOCS)

- Raises issues of
  - Locality (per-processor data structures)
  - Granularity of scheduling tasks
  - Lock overhead
  - Tradeoff between throughput and latency
    - Large critical sections are good for best-case latency (low locking overhead) but bad for throughput (low parallelism)

Performance Measures

- Latency
  - Cost of thread management under the best case assumption of no contention for locks
- Throughput
  - Rate at which threads can be created, started, and finished when there is contention

Optimizations

- Allocate stacks lazily
- Store deallocated control blocks and stacks in free lists
- Create per-processor ready lists
- Create local free lists for locality
- Queue of idle processors (in addition to queue of waiting threads)

Ready List Management

- Single lock for all data structures
- Multiple locks, one per data structure
- Local freelists for control blocks and stacks, single shared locked ready list
- Queue of idle processors with preallocated control block and stack waiting for work
- Local ready list per processor, each with its own lock

Multiprocessor Scheduling in Linux 2.6

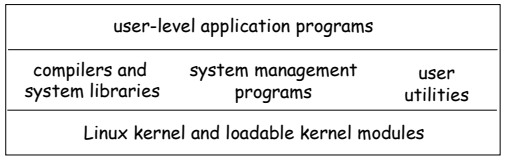
- One ready task queue per processor
  - scheduling within a processor and its ready task queue is similar to single-processor scheduling
- One task tends to stay in one queue
  - for cache affinity
- Tasks move around when load is unbalanced
  - e.g., when the length of one queue is less than one quarter of the other
  - which one to pick?
- No native support for gang/cohort scheduling or resource-contention-aware scheduling

History of Linux

- Linux is a modern, open-source operating system that is mostly POSIX-compliant.
- First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility.
- Collaboration by many users all around the world, corresponding almost exclusively over the Internet.

Linux Kernel/System/Distribution

- Kernel
  - the OS code that runs in 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/Fedora, Debian, SuSE, Caldera, …



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 a dedicated address space

Linux Task Scheduling

- Linux uses two task-scheduling classes:
  - time-sharing and real-time
- A prioritized, epoch-based algorithm for time-sharing
  - Each task has a static credit (default=20) and a dynamic quantum
  - Scheduling is prioritized based on quantum at the beginning of each epoch; each tasks runs its quantum length of time
  - The initial process quantum at its first epoch is credit
  - An epoch ends when no runnable tasks have any quantum; new quantum is calculated for new 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.

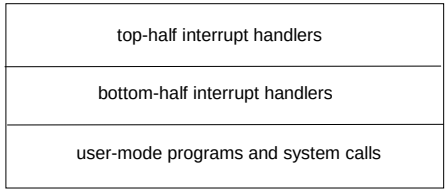
Linux Task Scheduling: O(1) Scheduler

- 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(), epoch transition
- Using two priority arrays
  - one for active array, one for those who have used up their entire quantum (called "expired")
  - array index indicates the priority (multiple tasks with the same priority chained in a link list pointed to from the array index)
- O(1) scheduling
  - fixed number of priorities (bit search instruction like for speed)
- O(1) epoch transition
  - swap active and expired arrays.

Interrupt Handling

- Interrupt handling is usually atomic
  - new interrupts are disabled during the handling of an interrupt
- Linux's kernel allows long interrupt service routines to run without having interrupts disabled for too long
- 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 later, with interrupts enabled
  - Bottom halves run one by one (they do not interrupt each other)

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.

Synchronization in Linux

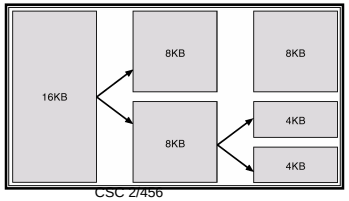
- Per-CPU variables to avoid synchronization
- Atomic variables (non-blocking)
- Read-copy-update (non-blocking)
- Spin-locks – basic, r/w (blocking)
- Semaphores (sleeping)
- Local interrupt disabling

Goal: Maximize concurrency

Managing Physical Memory

Keeping 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: 4KB, 8KB, 16KB, 32KB, … …
  - A large region can be split into multiple smaller regions if necessary



Memory Page Replacement

- All memory pages are managed together
  - stack/heap/code, …
  - file system buffer cache
- Memory pages are managed in two LRU lists: active list and inactive list
  - each LRU list is managed using a CLOCK (second-chance) LRU approximation
  - pages evicted from the active list go to the inactive list; pages evicted from the inactive list are out of the system
  - pages in the inactive list may be promoted to the active list under certain circumstances

Ext2fs File System

- Disks are divided into contiguous block groups
  - the hope is that there is not much seeking within each block group
  - there is a section for inodes in each block group
  - the FS tries to keep inodes and corresponding file blocks in the same block group
- Ext2fs tries to place logically adjacent blocks of a file into physically adjacent blocks on disk
  - with the help of the free block bitmap
- Ext3fs supporting file system journaling

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
  - prefetching sequentially when the I/O access is considered as sequential
  - how to detect sequential pattern?
- Disk I/O scheduling
  - an elevator-style seek-reduction scheduling
  - non-work conserving scheduling: anticipatory scheduling
  - deadline to prevent starvation

Robustness and Dependability

- Modern operating systems are complex and potentially contain bugs
  - Linux is no exception – including memory errors, synchronization errors (races, deadlocks, …), etc.
- A study [Chou et al. sosp2001] finds that:
  - device drivers are 3-7 times more error-prone
  - average bugs live for 1.8 years
  - errors cluster significantly

Disclaimer

- Parts of the lecture slides contain original work by Andrew S. Tanenbaum. 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).