Systems Programming

Process Management

H. Turgut Uyar   Şima Uyar

2009

Topics

Processes

Definition
A process is an instance of a program in execution.

- are created
- may create other processes
- perform a series of actions
- may be suspended
- are terminated

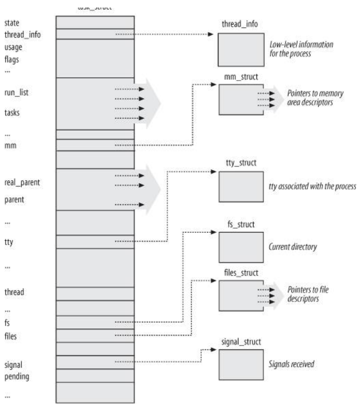
Process Descriptor

Definition
For each process, the kernel keeps a process descriptor in the form of a *task_struct* structure.

The process descriptor,a.k.a. task descriptor, contains information

- showing the process state
- showing process identification (pid, uid, euid, ...)

Process Descriptor Structure



The Process List

- doubly linked list
- *tasks* field of *task_struct*
  - prev points to previous process descriptor
  - next points to next process descriptor
- composed of all process descriptors
- *current* macro gives process descriptor of the running process
  - e.g. used as *current->pid*

Process 0

- a.k.a. the *idle process* or the *swapper*
- is the first entry in the process list
- created during the initialization stage of the kernel
- is the only process created without using the *fork* system call
- is the ancestor of all processes
- uses a statically allocated data structure
  - process descriptor stored in *init_task* variable
  - initialized by INIT_TASK macro
- executes *start_kernel()* function
  - initializes all data structures needed by kernel
  - enables interrupts
  - creates process 1 (commonly known as the *init process*)
- executes *cpu_idle()* function

Creating Processes

- traditional *fork* system call implemented as *clone* system call in Linux
- the *do_fork()* function
  - handles the *clone* system call
  - allocates a pid for the child process
  - uses *copy_process()* function to set up the process descriptor and other kernel data structures for new process
    - uses *dup_task_struct()* to allocate a new process descriptor and to copy parent process' process descriptor info
  - adjusts some parameters of parent and child processes
  - returns pid of child process

Destroying Processes

- through the *_exit()* system call
- uses *do_exit()* function

Introduction

- critical sections and race conditions also exist for kernel code
- must use synchronization
- Linux provides several kernel level synchronization primitives
- primitive must be chosen based on requirements of operation

Primitives

- may use *atomic* read-modify-write operations
- may use spin locks (locks with busy waiting)
- may use memory barriers (to avoid instruction reordering)
- may use kernel semaphores (lock with blocking wait)
- may use interrupt disabling (local CPU)
- ...

Memory Barriers

- kernel may reorder assembly instructions for optimization
- reordering must be avoided when synchronization is needed
- barrier ensures that instructions before the primitive are completed before those after the primitive
- read memory barriers *rmb()*
- write memory barriers *wmb()*
- memory barrier *barrier()* same as *wmb()*

Spin Locks

- for locking access to shared data (critical sections)
- for multiprocessor environments
- uses busy waiting
  - kernel resources usually locked for very short periods
  - more time consuming to release and reacquire cpu
- represented by a *spinlock_t* structure
- macros used for working with spin locks
- read and write spin locks to increase concurrency in kernel (*rwlock_t* structure)

Introduction

Definition
The scheduler divides the finite resource of processor time between the runnable processes on a system.

Two types of processes:

- processor bound processes
- I/O bound processes

Scheduling policy must satisfy two conflicting goals:

- fast process response time (low latency)
- maximal system utilization (high throughput)

Linux scheduler favors I/O bound processes, i.e. optimizes for low latency

O(1) Scheduler

- previous scheduler in Linux
- constant-time algorithm for timeslice calculation and per processor runquees
- scalable
- ideal for large server workloads
- has problems for interactive processes

CFS Scheduler

- the Completely Fair Scheduler
- current scheduler in Linux
- aims at improving scheduling for interactive processes

Linux Scheduler

- enables different algorithms to schedule different types of processes
- scheduler classes with priorities
- scheduler code iterates over each scheduler class in order of priority (`kernel/sched.c`)
- `CFS for normal processes - SCHED_NORMAL (kernel/sched_fair.c)`
- `scheduler for real time processes (kernel/sched_rt.c) has two policies`
  - `SCHED_FIFO`
  - `SCHED_RR`

Completely Fair Scheduler 1

- assigns processes a *proportion* of processor
- nice value (priority) acts as weight to determine proportion of processor time
- preemptive (based on proportions of processor time consumed)

Completely Fair Scheduler 2

- *timeslice* proportional to process' weight over sum of weights of all runnable processes
- targeted latency
- minimum granularity

Implementation 1

- *scheduler entity structure* (`struct sched_entity`) used for process accounting
- `struct sched_entity is a member of struct task_struct`
- *virtual runtime* (`vruntime`) is the actual runtime (nanoseconds) of a process normalized by the number of runnable processes
- in a perfectly multitasking system all processes should have same virtual runtime
- this is updated periodically by system timer and also whenever a process becomes runnable or blocks

Implementation 2

- the runnable process with the smallest `vruntime` is selected to run
- CFS uses a red-black tree to manage list of runnable processes (search O(log n))
  - leftmost node has lowest vruntime
  - leftmost node is cached
- scheduler entry point: `schedule()` in `kernel/sched.c`

Reading Material

- Linux Kernel Development, 3rd Edition
  - Author: Robert Love
  - Publisher: Addison-Wesley Professional
  - Year: 2010
  - Chapters: **3**, **4**, 5, 9 and 10
  - Availability: accessible from the ITU Library through Safari e-books