Process Scheduling Chapter 2

Multitasking OS comes in two flavours
1. Co-operative
2. Preemptive

All the Linux flavours of Comes in Preemptive Scheduling flavour.

In Preemptive Multitasking Scheduler decides which Process to Start Running and which Process to to Stop Running.

The time a process Runs before it preemption is usually predetermined, and is called the time slice of the ~~process~~. Process's time.

Managing the time Slice enables the scheduler to make global Scheduling decision.

Co-operative Schedulings In this method Process does not stop running until It voluntarely stops. the act of a Process voluntarily suspending itself is called yielding.

Limitation of Cooperative Schedulings

1. Scheduler can not take the global decision.
2. Any process can monopolize the system.

I/o Bound vs Process Bound Processes

I/o Bound Process spend most of the time waiting for some input/output event to occur.

Processor Bound Process spend most of time executing the code. These kind of process run continously without waiting for any I/o event and stopped while preempted by the scheduler.

Process Priority

Linux implements two types of Process priority values

1. Nice Values: A number from -20 to +19 with a default value of 0. Larger Nice value corresponds

to lowest Priority.

② Real-time priority: this value are 0 - 99. higher the Real time value higher the Priority.

⇒ Time Slice: is a time that how long a task can Run until it. is preempted. If the time-slice will be large then System will be less interactive: and when time slice will be low then Content switching will be extra overhead on the System thus Lower the Performance low.

⇒ The Linux Scheduling Algorithm:

Scheduler classes: Linux Scheduler is Modular, enabeling different Algo to schedule different types of process this modularity is called Scheduler classes.

The Complete fair Scheduler (CFS) is the Registered scheduler class for normal Process Called SCHED_NORMAL

in Linux.
CFS is defined in kernel/sched_fair.c

Process Scheduling in Unix Systems

- Need to check.

- Fair Scheduling: It's a scenario in which we assume that our processor is in ideal state and every process gets an equal amount of time. Suppose if we have n process then each process will get 1/nth time of processor.

Perfect multitasking is the situation in which every process gets n number of for the whole time on processor.

CFS will run each process for some amount of time, round-robin, selecting next the process that has run the least. CFS calculate how long a process should run as a function of the total number of runnable processes.

In CFS we calculate the weight of each Process by using the Nice value of that process.
Each Process then Runs for a "time slice" proportional to its weight devided by the total weight of all Runnable thread.

If the numbers of Runnable Process will Reach to infinity then the time slice for each Process will aproach to zero which increase the unnecessary overhead of switching. To avoid CFS impose a flood on timeslice which is the minimum timeslice to be allotted a Process. By default it's 1 millisecond.

as more works to be done on this

⇒ Linux Scheduling Implementations

We will discuss here CFS Actual implementation which lives in kernel/sched/fair.c

(1) Time Accounting
(2) Process Selection
(3) The Scheduler Entry Point
(4) Sleeping and waking up

① Time Accounting:

All Process Schedulers must account for the time that a process Runs. Most Unix System maintains the Account not how much a Process Actual runs on Processor.

② The Scheduler Entity Structure:

CFS does not have the Culture of using the time Slice. But it has to maintain the account for time a Process Actual Runs for that it uses Scheduler entity Structure

Struct sched_entity, defined in <linux/sched.h> to keep track of process accounting.

This structure embed in the Process descriptor structure task_struct.

The VIRTUAL RUNTIME:

The vruntime variable stores the virtual runtime of a process, which is the actual runtime (the amount of time

Spent Running) normalized by the number of runnable processes.

CFS uses vruntime to account for how long a process has run and thus how much longer it ought to run.

update_curr() Calculates the execution time of the current process and stores that value in delta_exec.

update_curr() is invoked periodically by system timer and also whenever a process becomes runnable or blocks, becoming unrunnable.

(2) Process Selection:

In Linux CFS to select the process which has smallest vruntime to next Run.

CFS uses a Red Black tree to manage the list of Runnable process and efficiently finds the process with smallest vruntime.

Picking the next task :→ CFS search the leftmost child in a tree which have the smallest vruntime. It is performed by pick_next_entity(). It has Complexity O(log N) for N Nodes if the tree is balanced.

If the tree Returns NULL It means there is no tasks to Run so schedules the idle task.

Adding Processes to trees

It happens when a Process become Runnable or fork() Call create a process. Adding of Process in tree is Performed by

enqueue_entity( );

It add the Process in the same way as we add a node in BST.

Removing Processes

It happens when a process blocks or terminates.

dequeue_entity()

③ The Schedule Entry Point:-

The main entry point in the Process Schedule is the function schedule() defined in kernel/sched.c. this function is called by Rest of kernel to schedule other Process.

Schedule() is a generic Call it internally Calls each schedular class and check the highest Priority tasks from highest Priority schedular task.

Pick_next_task()

If we know that all tasks are in the fair class we can Call that function directly.

④ Sleeping and Waking Up

Whenever a Process goes into Sleep State for whatever reason it is removed from Runtime and Calls Schedule() to select a new Process to execute. Waking back up is the inverse. The task is set as

runnable, removable from the waitqueue and added back to the kernel - by using the

Wait Queues: It is a simple list of Process waiting for an event to occur. waitqueue are represented in the kernel by wake_queue_head_t. waitqueue are created statically via DECLARE_WAITQUEUE() or dynamically via init_waitqueue_head().

Process put themselves on a waitqueues and mark themselves not runnable. When the event associated with waitqueue occurs, the process on the queue are awakened.

/* q is the wait queue we wish to sleep on */

DEFINE_WAIT (wait);

It will create a wait queue entry.

add_wait_queue (q, &wait);

Add itself to a wait queue via add_wait_queue(). This wait queue

awakens the process when the condition for which it is waiting occurs.

```
while (!Condition)
{
    prepare_to_wait (&q, &wait, TASK_INTERRUPTIBLE);
/* It changes the Process State to either
TASK_INTERRUPTIBLE or * This function also
add the task back to the wait queue
if necessary, which is needed on
subsequent iteration of loop */
    if (signal_pending (current))
        /* handle signal */
        schedule ();
}
finish_wait (&q, &wait);
```

If set the task to TASK_RUNNING and Removes itself from the wait queue.

example notify in filesystem wait queue

Waking up:

Waking is handled by via wake_up(), which wakes up all the tasks waiting on the given wait queue. It calls try_to_wake_up which set the task state to TASK_RUNNING and add the task Red black tree.

For example, when some data arrived in hard disk, the VFS calls wake_up() on the wait queue that holds the process waiting for the data.

Preemption and Context Switching:

Context Switching is switching from one runnable process is handled by context_switch() function defined in kernel/sched.c. It is called by function schedule() when a new process has been selected to run. It does two basic jobs:

(i) switch_mm():→ to switch the virtual memory mapping of that process to new process

(2) Switch_to ():- to switch the Process State from previous process to Current's. It does the saving into of current stack process Register etc.

User Preemption:- It occurs when the kernel is about to return to user-space need_resched is set and therefore the scheduler is invoked.

So user-preemption can occur
① when Returning to user-space from a system call.
② Returning to user-space from an interrupt handler.

Kernel Preemption:

Any task in the kernel which is not holding any locks can be preempted.

To enable the kernel Preemption in kernel kernel embed a variable preempt_count in the thread_info structure.

Everytime when process acquired a locks the value increases by 1 and vice versa. Whenever this value is 0 the kernel is preempted.

Upon Return from the interrupt or returning to kernel Space Value of need_reschedule is set and preempt_Count is 0 then most important task is scheduled