Contents of Lecture 4

- Process scheduling

Process Scheduling

- UNIX is a time-sharing system where a process is allowed to use the CPU a short time called a **time slice** or **time quantum**.
- Then another process is allowed to run etc.
- It's the **process scheduler** which decides:
  - which process should run next
  - for how long
- A process **context switch** saves the registers in the **process control block** (PCB) of the previously running process and loads them from the PCB of the next process.
- The PCB is architecture dependent and is usually a part of the u area.

Clock Interrupt Handling

- A hardware clock makes an interrupt at fixed intervals.
- The time period is called a **CPU tick**, **clock tick** or simply **tick**.
- The kernel usually measures time in ticks rather than seconds.
- The clock interrupt has the second highest priority — only power failure has a higher priority.
- A typical tick is 10 ms, i.e. 100 interrupts per second.
- The clock interupt must therefore be fast.

Operation of the Clock Interrupt

- Restart the hardware clock if needed.
- Update CPU usage statistics of the running process.
- Recompute priority and handle time-slice expiration.
- Send the signal SIGXCPU if the running process has exceeded its CPU quota.
- Update the time-of-day clock
- Check for **callouts** — see below.
- Check for **alarms** — and send **SIGALRM** to processes.
- Wake up system processes if needed.

Callouts

- The kernel often needs to perform operations at a future time, eg:
  - Retransmit a packet,
  - Scheduler or memory management functions, or
  - Polling devices which do not support interrupts
- Such operations are scheduled using the function

```
int timeout(void (*fun)(caddr_t), caddr_t arg, long delta);
```

- which returns a time-out-id that can be cancelled with

```
void untimeout(int id);
```

More About Callouts

- A registered callout function does **not** run as part of the clock interrupt — they are run after the interrupt priority level has been restored to a normal level, i.e. after completion of the clock and any other interrupts.
- Adding a timeout is not so time-critical.
- Checking whether a timeout is due is performed every tick and so must be done quickly.
- In 4.3BSD a sorted list is used where the difference between the new and the previous entry's time is stored.

Traditional UNIX Scheduling

- Each process has a priority that changes with time.
- It is always the highest priority process that runs.
- With multiple processes having highest priority, pre-emptive time slicing is used.
- When a higher priority process becomes ready to run, the currently running is preempted — despite not having used up all of its time slice.
- Recall that the traditional UNIX kernel itself is non-preemptible.

4.3BSD Process Priorities

- The priority is a number in the range 0..127 with 0 the highest priority.
- Kernel processes have the range 0..49 and user processes 50..127.
- The proc structure contains the fields:

```
p_pri           /* current priority     */
p_usrpri        /* user mode priority   */
p_cpu           /* recent CPU usage     */
p_nice          /* nice factor          */
```

- The scheduler uses `p_pri` as the priority.
- For processes in user mode `p_pri` and `p_usrpri` are equal.
- A process running in the kernel is temporarily given a higher priority (i.e. in `p_pri`) to get it out from the kernel faster.
- The user mode priority value of `p_pri` is saved in `p_usrpri`.

Sleep Priorities

- A process blocked (i.e. sleeping) in the kernel is given a new priority when it wakes up
- The value of `p_pri` depends on why it was blocked. For example:
  - Waiting on a hard disk results in `p_pri` being set to 28.
  - Waiting on a terminal results in `p_pri` being set to 20.
- Since these priorities are lower than 50, such processes can leave the kernel before user mode processes are scheduled.
- When a process has completed a system call and is about to return to user mode, its saved `p_usrpri` is copied to `p_pri` and if it is no longer the highest priority process a context switch occurs.
- The command `nice` can change the nice-value and decrease the priority of a process (only root can increase it).

Computing the Priority

- The clock interrupt increments the `p_cpu` of the running process up to a maximum of 127.
- Every second a callout, `schedcpu`, reduces the `p_cpu` by a **decay factor** so that recent CPU usage is accounted for the most.
- The decay factor is: `2 * load_avg / (2 * load_avg + 1)`.
- The new priority: `p_usrpri = 50 + p_cpu/4 + 2 * p_nice`.
- I/O bound processes are given a high priority (e.g. editors)
- Compute bound have lower priority.
- No process starves.

Scheduler Implementation

- There is an array `qs` with 32 double linked lists.
- 32 and not 128 due to special instructions made that better on the VAX.
- A 32-bit variable `whichqs` represents which lists are nonempty.
- When `schedcpu` recomputes a priority the process is removed from a list and then inserted into a list (possibly the same list).
- The clock interrupt in 4.3BSD recomputes the priority of the running process every 4th tick.

Analysis of the 4.3BSD Scheduler

- With a large number of processes it is inefficient to recompute their priorities every second
- No way to guarantee a certain amount of CPU time for a process
- Since the 4.3BSD kernel was non-preemtible a runnable process may have had to wait a long time to be scheduled — i.e. 4.3BSD potentially had a long dispatch latency.
- There is no support for soft real-time processes such as multimedia applications.

Some Improvements in UNIX System V Release 4, SVR4

- Support for soft real-time applications.
- More control to processes over their priorities.
- Makes it possible to dynamically load a new scheduler.
- Scheduling classes: time-sharing and real-time.
- The SVR4 kernel was still non-preemptible but to reduce dispatch latency of real-time processes certain preemption points were introduced such as:
    - before creating a file
    - before freeing pages of a process
    - before parsing pathname components

Object-Oriented Approach to Scheduling Classes

- A number of virtual functions (i.e. methods in subclasses) are implemented for use by various parts of the kernel through macros:
  - `CL_TICK` — called by the clock interrupt handler
  - `CL_FORK` — called when a process is created
  - `CL_FORKRET` — may make the child run before the parent
  - `CL_ENTERCLASS` — allocates class data structures
  - `CL_EXITCLASS` — deallocs them
  - `CL_SLEEP` — may recompute process priority
  - `CL_WAKEUP` puts the process on a suitable run queue

```
/* the dots are not C99 preprocessor syntax. */

#define CL_SLEEP(proc, clproc, ...) \
((*(proc)->p_clfuncs->cl_sleep)(clproc, ...))
```

The SVR4 Time-Sharing Class

- To support thousands of processes, their priorities are not recomputed every second.
- Instead, the priority of a process changes as specified in a static table.
- Processes which use up all of their time slice get a lower priority.
- Processes which block get a higher priority when they wake up.
- A **dispatcher parameter table** has a row for each priority level which specifies among other things
  - the time slice
  - a new priority if the process uses all of its time slice

The SVR4 Real-Time Class

- Real-time processes have the highest priority — even higher than processes in the kernel.
- Recall that the SVR4 kernel is non-preemptible except at preemption points.
- Only superuser processes can enter the real-time class.
- Fixed priorities and time slices are used for real-time processes.
- The code path between preemption points is often too long.
- There are no facilities for deadlines.
- Experiments with an X-server, a video program, a batch job, and an editor showed that both the X-server and the video program (which uses the X-server) must be in the real-time class but that it was very difficult to find useful scheduling parameters.

Solaris Scheduling

- Solaris is a modern kernel for multiprocessors based on SVR4 but ***very*** much improved.
- Solaris has a fully preemptible kernel.
- It is threads and not processes which are scheduled.
- There is a single queue of runnable threads for all CPUs.
- When a thread has become runnable, the CPU with lowest priority thread is selected.
- This means either currently running thread or thread chosen to be dispatched (but not yet running) on the CPU.

The Priority Inversion Problem

- Basically it means that a lower priority thread T_1 holds a resource needed by a higher priority thread T_2. T_2 is not allowed to just take the resource from T_1 (a bullet in T_1 — this is possible using transactions as we will see later today) but must wait until T_1 is ready.
- A different scenario is with an additional thread T_3 with priority between the other two. Since it has higher priority than T_1 it will preempt T_1 so now T_2 must wait even longer.
- To reduce waiting time of T_2 the priority of T_1 can be temporarily increased to the level of T_2 which prevents T_3 from preempting T_1.
- This is called priority inheritance.

Solaris' Implementation of Priority Inheritance

- Each thread has two priorities: the normal global and an inherited priority.
- The inherited is normally zero.
- The scheduling priority is the higher of the two.
- When a thread is about to block waiting for a resource, it traverses the chain of other threads blocked to let them inherit its priority if higher.
- When a resource is returned a thread recalculates its inherited priority as the maximum from the resources it still owns and other threads are waiting for.
- A limitation is that the owner of the resource must be known which it is e.g. for a mutex but not so easily for a reader-writer lock, where only one reader is kept track of (better than nothing).

Saving Memory Technique: Solaris Turnstile

- There are hundred of synchronization objects in the kernel but only a few of them are used at any given time.
- To save memory an abstraction called a **turnstile** is used (a turnstile is a gate with revolving arms such as those in some undergrounds).
- All data for managing the object is stored there including the queue of blocked threads and a pointer to the owning thread.
- Solaris uses a two-byte integer (instead of a pointer) to map from a resource to a turnstile to further save memory.

Linux Scheduling

- The $O(1)$-scheduler
- The Completely Fair Scheduler since 2.6.23

The $O(1)$-scheduler

- The scheduler in the 2.4 kernel was optimized for smaller systems
- For large scale multiprocessors a new scheduler was needed
- It was called the $O(1)$-scheduler.
- It had a drawback of not so good performance for interactive processes.

Linux Scheduler Goals

- Low latency for interactive (I/O bound) processes
- High throughput for compute bound processes
- Compute bound processes are best served by large time slices while I/O bound are best serve by short time slices.
- What benefits are there with large time slices?
  - Reduced overhead due to running the scheduler and doing context switches
  - Probably better use of the memory hierarchy — few cache missses.
- How can we strike a good balance for Linux?

Mapping the Nice Value to a Timeslice

- If we use the nice value to select a timeslice we can run into problems of too much scheduler overhead.
- Assume one high prioriy and one low priority processes are runnable.
- Let the high priority have a time slice of 100 ms and the low priority a time slice of 5 ms.
- This works out well since the high priority processes is preempted after 100 ms.
- However, if we instead would have two low priority processes, we would let each run only 5 ms before context switching, which leads to a large relative overhead.
- This is not what is wanted for Linux.

CFS Principles

- Assign not timeslices but fractions of the CPU time to processes.
- This adjusts the actual time before the next context switch to depend on the current load.
- In CFS, the nice value is used to compute a **weight**.
- Higher priority processes (i.e. lower nice value) receive a higher weight.
- A process/thread gets a timeslice which proportional to its weight divided by the sum of all threads' weights.
- CFS uses a **target latency** which is a portion of time the threads will share.
- Suppose the target latency is 20 ms. Then a thread gets a timeslice which is a fraction of those 20 ms calculated as its weight divided by the sum of weights.
- There is also a minimum timeslice, called the **minimum granularity**, since otherwise it would become too small to be efficient.

CFS Process Selection

- The variable `vruntime` represents the process runtime so far.
- It's not an absolute time but weighted by the number of runnable processes.
- The highest priority process is the process with smallest value of `vruntime`.
- The CFS Scheduler maintains a red-black tree, sorted by the value of `vruntime`.
- So, the process the CFS scheduler selects is the leftmost node in the tree.
- To avoid having to traverse the tree too often, this node is cached as `cfs_rq->rb_leftmost`.

Calling `schedule()`

- There is a variable `need_resched` which the clock interrupt can set, or when a higher priority process has just woke up.
- It was a global variable but is moved to the task to increase chances of a cache hit.
- When the kernel is going to return to user mode for a process from a system call or an interrupt, the `need_resched` flag is checked.
- Since Linux version 2.6 the kernel can be preemted.
- To preempt a kernel thread, that thread must not hold a lock, and for this a counter `preempt_count` is used.