Operating Systems

Tutorial 1

Michael Tänzer

os-tut@nhng.de
http://os-tut.nhng.de

23th November 2010

Outline

True or False

- Interrupts can’t be handled when already executing in kernel mode
- The general purpose don’t necessarily have to be saved by the kernel when a trap occurs
- Sometimes using multiple processes of the same application instead of threads makes sense
- In any thread model the kernel needs to have a notion of threads

How do buffer overflows work?

```
void getPassword(){
    char password[5];
    printf("Please enter your password: ");
    scanf("%s", password);
}
```

How can security critical software make sure stack frames are not accessible after removing them?

```
/* To avoid that a compiler optimizes certain memset calls away
   these macros may be used instead. */
#define wipememory2(_ptr,_set,_len) do { \
              volatile char *_vptr=(volatile char *)(_ptr); \
              size_t _vlen=(_len); \
              while(_vlen) { *_vptr=(_set); _vptr++; _vlen--; } \
                  } while(0)
#define wipememory(_ptr,_len) wipememory2(_ptr,0,_len)

void _gcry_burn_stack(int bytes){
    char buf[64];

    wipememory (buf, sizeof buf);
    bytes -= sizeof buf;
    if (bytes > 0)
        _gcry_burn_stack (bytes);
}
```

“Kernel threads can never be as fast as user threads”

- Higher cost of thread operation due to protection domain crossing (i. e. kernel invocations)
- Overhead caused by general-purpose implementation. User thread libraries can be much more application specific

“Kernel threads are the wrong abstraction for supporting user threads”

- User threads are unaware of the in-kernel scheduling of kernel threads
- Scheduling of kernel threads doesn’t take user-level thread state into account

What are the basic concepts to overcome these problems?

- More interaction between kernel- and user-level scheduler

$\Rightarrow$ Kernel informs user-level scheduler about events that might be of importance. For example:

  - Changes to number of (virtual) processors assigned to the process
  - Blocking syscall performed by one of the user threads

$\Rightarrow$ User-level scheduler notifies the kernel on all thread operations that might influence processor allocation decisions

- Most thread operations however can be executed without kernel invocations $\Rightarrow$ Good performance

Scheduler activation vs. “traditional” kernel thread

Similarities & differences

Similarities

- Serves as execution context for a user thread

$\Rightarrow$ Offers a kernel and user stack

Differences

- Scheduler activation provides a notification mechanism between user- and kernel-level thread management

What happens when performing blocking I/O?

1. User thread blocks in the kernel waiting for the I/O
2. Kernel creates a new activation on the processor the thread was running on
3. Kernel makes an upcall using that activation
4. User-level scheduler can select a new thread to run in that activation (and therefore that processor)
5. I/O completes
6. Kernel needs a processor to perform an upcall ⇒ preempt a process that is assigned to that process
7. Kernel creates another scheduler activation
8. Kernel performs an upcall that notifies the user-level scheduler of I/O completion and preemption of a thread

What is the purpose of scheduling?

- Find a mapping: processes $\rightarrow$ resources so that each process will eventually get the resources it needs
- Try to maximise some quality metrics (goals in policyspeak) e. g. resource utilisation
- We'll focus on CPU scheduling

What metrics can be used to estimate the quality of a scheduling policy?

Utilization percentage of time a resource is not idle

Troughput number of requests (in CPU scheduling processes/threads) completed per unit of time

Turnaround time time from submission of a request to its completion

Response time time from submission of a request until the first response is produced

Waiting time time a request is not being processed (in CPU scheduling time spent in the ready queue)

What are common values for the length of a time slice?

From 10 to 100 ms

Short vs. long time slices

In what kind of systems would you use which?

Long Time Slice

- Fewer context switches (less overhead)

$\Rightarrow$ Higher throughput

$\Rightarrow$ Good for batch systems

Short Time Slice

- Ready processes don't have to wait long until they're executed

$\Rightarrow$ Better responsiveness of the system

$\Rightarrow$ Good for interactive systems

What is Linux' tickless mode?

- The timer interrupt isn't raised at fixed intervals but when it's needed (i. e. the next future event)

$\Rightarrow$ no timer interrupts when the CPU is idle

$\Rightarrow$ deeper sleep states possible (helps to save energy)

Example

Given:

- Three batch processes (which never do blocking syscalls)
  - P_1: execution time $T_1^e = 7$, arrives at $T_1^a = 2$
  - P_2: execution time $T_2^e = 3$, arrives at $T_2^a = 0$
  - P_3: execution time $T_3^e = 1$, arrives at $T_3^a = 7$
- FIFO scheduling

Task:

- Draw Gantt chart
- Calculate average waiting time T^w
- Calculate average turnaround time T^t

How does SJF work?

Preemptive vs. non-preemptive SJF

- Select job with the shortest *remaining* time
  – Most of the time not possible (total time to completion unknown)
  ⇒ use estimated length of next CPU burst
- The preemtive version makes a new decision when a new process arrives in the ready queue

What's the basic idea of priority scheduling?

What's the major problem of priority based algorithms?

- Each process is assigned a priority
- Choose the process with the highest priority from the ready queue
- Need another scheduling policy if there are multiple processes with the same priority (e. g. round robin)
- Starvation may occur if there is a process with a low priority and there are always processes with higher priorities ready to run

Explain the multilevel feedback queue algorithm

What kind of processes can be found in the higher and lower queues?

- Multiple queues
- Processes are taken from highest non-empty queue
- Need another scheduling policy to decide which process to take from the queue
- Higher queues $\Rightarrow$ short time slices, lower queues $\Rightarrow$ long time slices
- Process uses the entire time slice $\Rightarrow$ move it down to the next queue
- Process blocks $\Rightarrow$ when it becomes ready again put it in the queue directly above the one it was in when it blocked
- Lower queues will contain CPU bound and higher ones I/O bound processes

How can starvation be avoided?

- Priority ageing: While a process is waiting its priority is increased
- In the case of multilevel feedback queues – when a process is waiting for a long time without getting the CPU it is moved up to the next queue

Describe the idea of lottery scheduling

- Each process gets a number of lottery tickets
- The scheduler draws a ticket
- The process owning that ticket gets the CPU

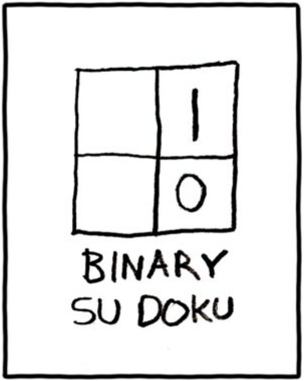
Enumerate possible advantages of lottery scheduling over priority scheduling

- No starvation (if each process gets at least one ticket)
- Possible to grant a process a specific percentage of CPU time (proportional to the number of tickets)
- Possible to have a hierarchical distribution of CPU time (each user gets *n* tickets which he can assign to the applications he wants to run)
- Possible to give CPU time to another process (e. g. the file server) to allow it to process own requests (by donating tickets to it)

Questions & Comments

Any questions or comments?

Binary Sudoku



from xkcd