# Individual Programming Assignment
# User Mode Scheduling in MINIX 3

Björn Patrick Swift
bst360@few.vu.nl

Supervisors:
Tomas Hruby
thruby@few.vu.nl

Andrew S. Tanenbaum
ast@cs.vu.nl

October 27, 2010

**Abstract**

This Individual Programming Assignment focused on moving scheduling in MINIX 3 from kernel mode to user mode. There are several motives, most importantly decoupling scheduling policy and kernel.

We will discuss design decisions and present a working implementation of user mode scheduling. This implementation is simple, but can support a wide variety of policies and has been part of the MINIX 3 since the 3.1.7 release. We also present scheduling feedback, where the kernel supplies the scheduler with process and system statistics. This information can be used by the scheduler to make better informed scheduling decisions. Finally, we analyze message flow and discuss performance overhead introduced, which we find to be acceptable.

# Contents

**1 Introduction** **3**

**2 User Mode Scheduler** **4**
2.1 Event-driven vs. Pro-active scheduling . . . . . 4
2.2 Interface . . . . . 4

**3 Kernel implementation** **5**
3.1 Process table entries . . . . . 5
3.2 Out of quantum messages . . . . . 5
3.3 System calls . . . . . 5
3.4 Default kernel policy . . . . . 6

**4 User Mode implementation** **7**
4.1 Process Manager (PM) . . . . . 7
4.2 Scheduler (SCHED) . . . . . 7

**5 Multiple schedulers** **8**

**6 Message flow** **9**
6.1 Starting the system . . . . . 9
6.2 Spawn a new process . . . . . 9
6.3 Process termination . . . . . 11
6.4 Out of quantum message . . . . . 11
6.5 Change nice level . . . . . 12
6.6 Multiple schedulers . . . . . 12

**7 Process feedback** **14**
7.1 Proof-of-concept Scheduler . . . . . 15

**8 Performance overhead** **17**
8.1 IPC in MINIX 3 . . . . . 17
8.2 Communication overhead . . . . . 17
8.3 Process feedback accounting . . . . . 18

**9 Future work** **19**

**10 Conclusion** **20**

# 1 Introduction

MINIX 3 is a micro-kernel operating system structured in four layers [4]. Device drivers, networking and memory management are examples of services running in layers above the kernel, but until recently scheduling was still handled in-kernel (Figure 1).

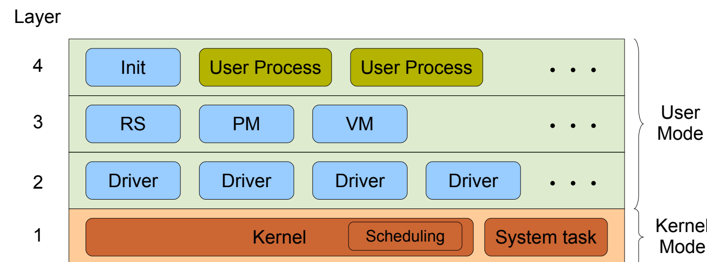


Figure 1: MINIX is structured in four layers. Only the bottom layer may use privileged (kernel mode) instructions. Device drivers reside in layer 2, system server processes in layer 3 and user processes in layer 4.

This IPA explored possibilities of moving scheduling to user mode (Figure 2). There are several motives, most importantly decoupling scheduling policy from kernel. With a *user mode scheduler*, it is possible to change the scheduling policy without modifying the kernel. Furthermore, it is possible to run multiple user mode schedulers each with their own policy. A user mode scheduler in a multi-core environment can also afford to spend more time evaluating scheduling decisions than the kernel can, since the system (and kernel) keep running on other cores.

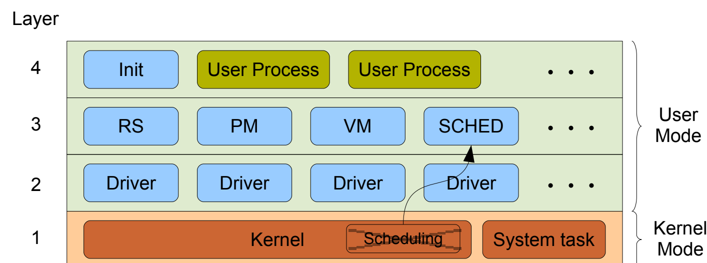


Figure 2: Scheduling moved from kernel to a server process in user mode.

This project posed several interesting questions: How much of the scheduler can be moved to user mode? What should the interface between kernel and scheduler look like? What are the performance implications? Who should make scheduling decisions for the scheduler itself? Should the scheduler be event driven and react to messages from kernel, or periodically step in to modify current behavior? This report will try to answer these questions and present an implementation that has been merged to MINIX 3 (as of release 3.1.7).

# 2 User Mode Scheduler

The first question we asked ourselves was: when should the scheduler make its scheduling decisions? This section will focus on this question and discuss the interface implemented between kernel and user mode.

## 2.1 Event-driven vs. Pro-active scheduling

Two approaches were discussed in terms of scheduling decision time: event-driven and pro-active scheduling. In the former case, the scheduler would largely sit idle and only react to event messages sent from kernel. For example, when a process runs out of quantum. In the latter case, the scheduler would periodically query the kernel for process status and make adjustments as required.

An event-driven scheduler is likely respond quicker to changes in process behavior. On the other hand, an event-driven scheduler does involve a kernel up-call and a dependency on a running scheduler in user mode. In contrast, with a pro-active scheduler, in case of failure the kernel could keep on scheduling the process as last instructed by the scheduler. It would simply interpret the scheduler's lack of action as if no adjustments were required.

We decided to implement an event-driven scheduler. This user mode dependency would not be a first, other examples include VM, PM and FS. The event-driven model allows for greater flexibility and a very minimal in-kernel implementation. Had we implemented the pro-active model, where the scheduler would tweak a few setting and then leave the kernel to it - we would be further limiting which scheduling decisions could be implemented. In fact, that would be more like a "kernel policy with user tweakable knowbs." Finally, the cost of a pro-active scheduler periodically reading the process table from kernel, to make possible scheduling changes, was thought to outweigh the availability benefits. In future releases, a crashed scheduler may be resurrected by RS at a later time.

## 2.2 Interface

We explored several popular scheduling policies, ranging from policies in the MINIX 3 book to recent policies implemented in Linux. For each policy, we identified required input and events, and which system calls were needed to implement the policy in user mode.

It was found that many popular policies can be implemented through a very simple interface. Two system library routines are exposed to user mode, `sys_schedctl` and `sys_schedule`. These send messages to kernel requesting to start scheduling a particular process and to give a particular process quanta and priority, respectively. Once a process runs out of its quantum, the user mode scheduler is notified with a message. This message contains system and process feedback that the scheduler may use for making scheduling decisions (Section 7).

Round Robin, Priority Scheduling, [4] Staircase Scheduler [1] and Rotating Staircase Deadline [2] are among the policies that can be implemented using this interface. The previous MINIX 3 policy was ported and we also implemented a new proof-of-concept feedback scheduler (Section 7).

# 3 Kernel implementation

The kernel defines $n$ priority queues and implements a simple preemptive round robin scheduler, always choosing the process at the head of the highest priority queue. When a process runs out of quantum, the kernel sets the process's `RTS_NO_QUANTUM` runtime flag, effectively dequeuing the process and marking it as not runnable.

## 3.1 Process table entries

The kernel holds minimal scheduling information on each process in its process table. This includes the process' priority queue, remaining quantum and its scheduler's endpoint. The kernel also keeps record of the quantum originally given to the process, something only relevant when the process has no scheduler and the kernel is forced to use its built-in policy (Section 3.4)

Listing 1: Process table entries.

```
1  char p_priority;               /* current process priority */
2  u64_t p_cpu_time_left;         /* time left to use the cpu */
3  unsigned p_quantum_size_ms;    /* assigned time quantum in ms*/
4  struct proc *p_scheduler;      /* who gets out of quantum msg */
```

## 3.2 Out of quantum messages

The scheduler is notified when a process runs out of quantum by sending a message to the scheduler on the process' behalf. The main advantage of sending the message on the process' behalf is to offload the buffering of messages to MINIX's IPC mechanism. Let us revise how messages are sent in MINIX. If a destination process is blocked waiting for a message, an incoming message will be written to the process' `p_delivermsg` struct in the kernel's process table. However, if the receiving process is not blocked receiving, the message will be buffered in the sender's `p_sendmsg` struct and the sender's endpoint added to the destinations call chain. By having the kernel send the message on the process' behalf we make use of this build-in IPC buffering, and avoid having to do our own buffer management in kernel mode—with the associated headaches of handling full buffers etc.

Note that an out of quantum message sent on behalf of a process will never overlap with a message being sent by the process itself. This is because sending a messages is a system call and a process does not run out of quantum while in the middle of a system call.

## 3.3 System calls

Two system calls are exposed to user mode. `sys_schedctl` is called by a user mode scheduler to take over scheduling of a particular process. The kernel will make note of the scheduler's endpoint in its process table and send an out of quantum messages to that scheduler.

The scheduler will make its scheduling decision and reschedule the process using the `sys_schedule` system call. The scheduler may choose to place the process in a different priority queue or give it a different quantum, all depending on its policy. The `sys_schedule` call can also be used to modify the priority and quantum of a currently runnable process.

Listing 2: System calls.

```
1 _PROTOTYPE( int sys_schedctl, (unsigned flags, endpoint_t proc_ep,
2                 unsigned priority, unsigned quantum));
3 _PROTOTYPE( int sys_schedule, (endpoint_t proc_ep,
4                 unsigned priority, unsigned quantum));
```

## 3.4 Default kernel policy

Having moved the scheduling policy to user mode, we still found that a minimal built-in kernel policy was required. First, when starting the system, processes may run out of quantum before the scheduler has been started. Second, in case the scheduler crashes it may be desirable to have the kernel take over scheduling for a short period of time until a new scheduler can be spawned and continue scheduling. Third, to avoid deadlocks, it may be desirable to have the kernel schedule some servers, such as the scheduler itself and RS.

The default kernel policy is relatively simple and is applied only when the process' `p_scheduler` pointer is set to `NULL`. When a process runs out of quantum, it is preempted but immediately placed at the end of its current priority queue with a new quantum (based on `p_quantum_size_ms`). This is a very naive policy and is neither starvation free nor tries to be fair. As mentioned, it is mainly meant to schedule servers during system startup.

# 4 User Mode implementation

## 4.1 Process Manager (PM)

PM has its own process table which is the user mode counterpart of the in-kernel process table. PM's process table has two attributes related to scheduling: `mp_nice` and `mp_scheduler`.

PM is the only user mode process that knows who is scheduled by whom. This information is stored in `mp_scheduler` which points to the scheduler's endpoint, or to `KERNEL` if the process is scheduled by the kernel's default policy.

Besides knowing the process' scheduler, it also keeps track of its niceness. How this value is interpreted in respect to prioritizing processes is entirely up to the scheduler to decide. While it is likely that a scheduler respects process niceness, at least relative to other processes it schedules, it is not a requirement. Different schedulers with different policies may choose to interpret niceness in very different ways. That is generally not a concern, unless multiple user mode schedulers are scheduling processes on the same core. More on multiple schedulers in Section 5.

As PM is the only user mode process that knows who is scheduled by whom, operations affecting scheduling are routed through PM. An example of this would be changing a process' niceness (Section 6.5).

## 4.2 Scheduler (SCHED)

The user mode scheduler implementation is embarrassingly simple. In essence, the scheduler blocks receiving either control messages from PM or out of quantum messages from processes. When receiving a control message the scheduler will either take over or give up scheduling a particular process, for example in conjunction with a process fork or exit.

When the scheduler receives an out of quantum message it needs to reschedule the process using `sys_schedule`. This is where the user mode scheduling policy kicks in. The policy will dictate in which priority queue the process should be scheduled and given what quantum.

The current SCHED implementation mimics the policy that was previously implemented in kernel. Each time a process runs out of quantum, it will be bumped down in priority by one. Then, periodically, the scheduler will run through all processes that have been bumped down and push them up, one queue at a time. This way, a CPU bound process will quickly be pushed down to the lowest priority queue, but gradually make its way back up once it stops hawking the CPU. See [4] for further discussion.

This policy was ported directly from kernel. For a more interesting approach, see the proof-of-concept feedback scheduler in Section 7.

## 5 Multiple schedulers

User mode scheduling opens up the possibility of running multiple schedulers at once. We can think this as multiple scheduling domains (Figure 3). A scheduling domain consists of a set of user processes, system servers, drivers and a single scheduler. One scheduler could run a classic multi queue round robin algorithm, another could prioritize based on user privileges and the third pin processes to cores. It would even be possible to write a new scheduling algorithm, compile and run a new scheduling server and have it take over scheduling a set processes. All without affecting the rest of the system.

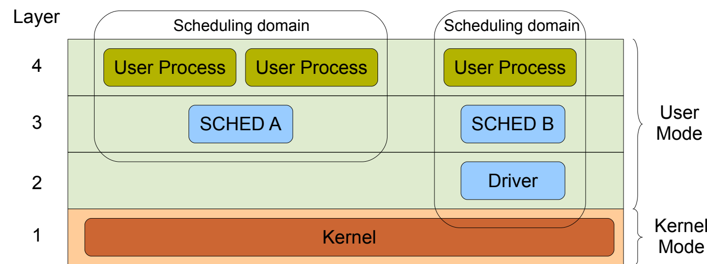


Figure 3: User mode scheduling allows for having multiple running schedulers.

We imagine this becoming more relevant with the increasing number of cores per machine. The system could dynamically allocate schedulers a set of cores, providing isolation between different policies, priority classes, users, etc. In case two schedulers schedule processes on the same core, they would share the in-kernel priority queues. Therefore, a policy that only used priorities 1-4 would starve a policy that only used priorities 5-8.

Both kernel and PM have basic support for multiple schedulers, as they associate a scheduler endpoint to each process individually. However, running multiple schedulers in parallel is not trivial, especially when it comes to migrating processes between schedulers. The current implementation does provide the foundation for multiple schedulers, including mechanism allowing a scheduler to redirect a schedule request from PM when the process is forked.

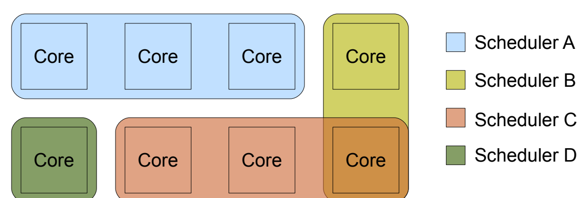


Figure 4: Schedulers could either have dedicated cores or share cores with other schedulers.

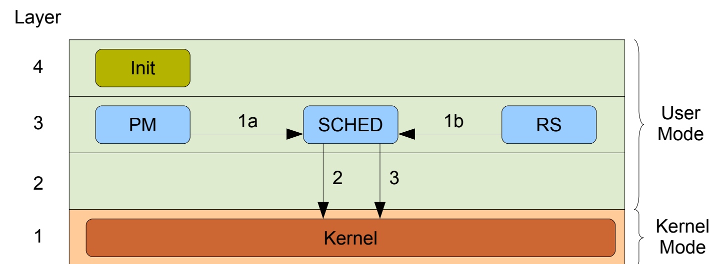


Figure 5: Messages sent when PM asks SCHED to start scheduling Init.

# 6 Message flow

In this section we will describe the message flow during different requests.

## 6.1 Starting the system

When the system is booted the kernel's `main()` function will populate its process table from the `boot_image` struct. Processes are given a fixed quantum but no scheduler, meaning that scheduling is handled by the kernel. Once the the kernel switches to user mode the servers included in the boot image start initializing.

The user mode scheduler does not know which processes are currently running in the system, and therefore does not pro-actively take over scheduling of any processes. It will go straight to receive incoming messages and wait for requests to start scheduling.

PM populates its local process table from the same `boot_image` struct. At the end of its `main()` function it will send the user mode scheduler a request to start scheduling processes. At this time, `init` will be the the only non-privileged processes in PM's process table, and without quantum. This is the only process that PM requests the scheduler to schedule, but as all user processes are forked from `init` (or its children), they inherit `init`'s assigned scheduler.

RS will act similarly to PM, requesting the user mode scheduler to schedule device drivers and certain system servers. In theory the user mode scheduler can schedule any process, but care must be taken to avoid deadlocks. The scheduler should not schedule servers that itself depends on. The scheduler should not schedule itself, nor for example VM if it uses dynamic memory allocation.

### 6.1.1 Messages

1. PM and RS send a `SCHEDULING_START` start message to SCHED through the `sched_start` library routine.
2. SCHED populates its local process table and send a message to kernel, notifying that it will take over scheduling the process.
3. SCHED will make a scheduling decision based on its local policy and schedule the process for the first time, allocating it quantum and placing it in a given priority queue.

## 6.2 Spawn a new process

Previously, processes inherited their parent's priority and half their parent's quantum. This was revisited when moving scheduling to user mode. When a process is forked it is now spawned without quantum, is therefore not runnable and has no real priority

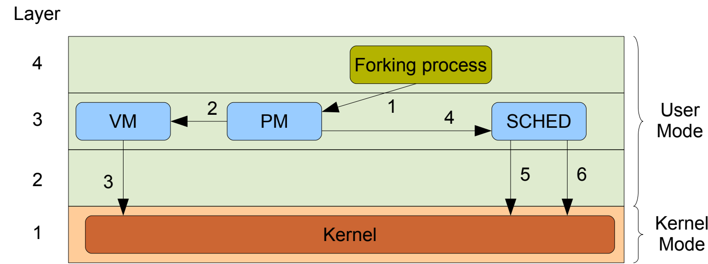


Figure 6: Messages sent when spawning a new process. Note that communication with VFS between steps 3 and 4 are left out for clarity.

yet[1]. The process will not be placed in a run queue before a user mode scheduler has taken over scheduling the process and given it a priority and queue.

In which priority the process is placed is entirely up to the user mode scheduler to decide. However, which ever policy gets implemented, it is often desirable for child processes to inherit not only their parent's *base priority* but also their *current priority*. The base priority denotes the general priority, essentially the highest priority this process can have at any given time. This value is local to the scheduler, is never exposed to kernel, and is typically affected by the nice level. The current priority denotes in which priority queue the process is currently scheduled. These two values may differ, for example a common approach is to lower the priority of processes that start CPU bound workloads. Were the current priority not inherited with the base priority, a CPU bound process could keep spawning short lived processes to stay at its base priority.

Inheriting scheduling attributes from parent processes is trivial in case of a single user mode scheduler. With multiple schedulers this requires some inter process communication. This was not addressed in the IPA.

### 6.2.1 Messages

1. POSIX program issues `fork()`. This sends a message to PM, which prepares its process table for the new process and forwards the message on to VM.
2. VM prepares memory for the new process and forwards the message to kernel.
3. The kernel forks the process, basing its process table entry on its parent entry. The parent process' quantum remains unchanged but the child is spawned without quantum (blocked with `RTS_NO_QUANTUM` flag).
4. At this point PM will asynchronously notify VFS of the new process and wait for its reply before moving on. This is left out of Figure 6 for the sake of clarity. Once VFS replies, PM will notify SCHED that a new process has been spawned and it needs to be scheduled.
5. SCHED will populate its local process table and send a message to kernel, taking over the scheduling of the process
6. SCHED will make a scheduling decision based on its local policy and schedule the process for the first time, allocating it quantum and placing it in a given priority queue.

[1]Actually, the process does inherit its parent priority in kernel but that priority gets overridden when the process is first scheduled. It is therefore never used.

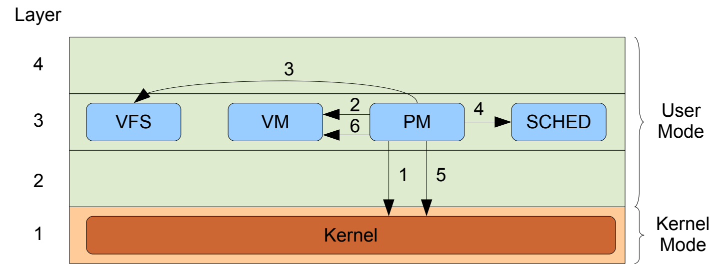


Figure 7: Messages sent when terminating a process.

## 6.3 Process termination

Terminating a process involves a similar communication pattern as when spawning a process. Multiple servers keep application state and they need to be notified. The scheduler is one of those servers.

### 6.3.1 Messages

1. PM gets a request to terminate a process and sends a `sys_stop` call to kernel, setting the `RTS_PROC_STOP` flag.
2. PM notifies VM that the process is about to exit.
3. VFS gets notified, taking a core dump if requested.
4. SCHED will stop schedule the process and clean up its process table.
5. The kernel will clean up its process table (mark the slot as empty).
6. VM is notified that this process has been terminated and that memory can be released.

## 6.4 Out of quantum message

When a process runs out of quantum, the kernel sends a message to the user mode scheduler on the process' behalf. For the more basic scheduling algorithms, the scheduler only checks the message sender and takes action based solely on the knowledge that this process ran out of quantum. More advanced algorithms may make use of the kernel feedback embedded in the message (Section 7).

When messages are sent from kernel on behalf of other processes, a privileged flag is set in the message. A user mode scheduler must verify that this flag is set to prevent a process sending an out of quantum itself. Without such a sanity check, a process could for example give the scheduler false accounting information resulting in the process getting a higher priority.

### 6.4.1 Messages

1. When the process runs out of quantum, the kernel sends a message to the user mode scheduler on the process' behalf. This message includes aggregate process and system accounting information.
2. The scheduler makes a scheduling decision and schedules the process using `sys_schedule`. This results in the kernel placing the process back on the run queue with a new quantum.

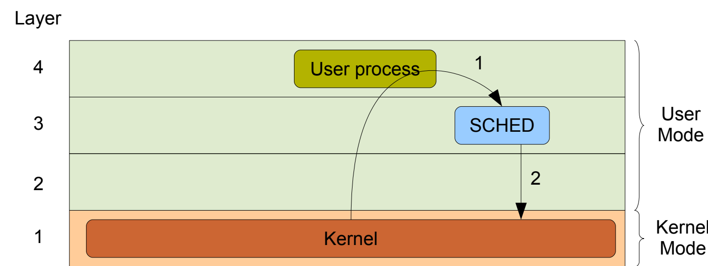


Figure 8: Out of quantum message sent from kernel on behalf of a user process.

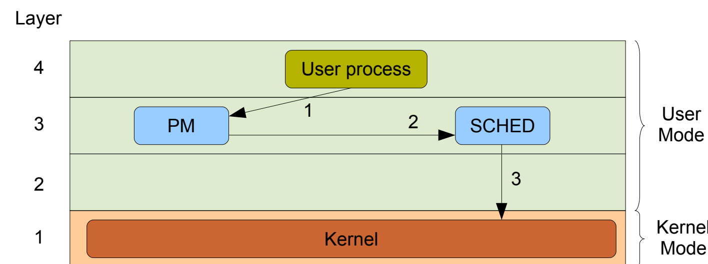


Figure 9: Changing niceness.

## 6.5 Change nice level

User processes do generally not know who schedules them. The only user mode process that keeps account of who schedules whom is PM. As a result, all requests relating to scheduling from user mode processes get routed through PM. Changing niceness (`getpriority` and `setpriority`) is an example of this.

### 6.5.1 Messages

1. A user process invokes the `setpriority` system routine, which sends a message to PM
2. PM will look up the scheduler for this particular process and send a message to SCHED with the new value.
3. SCHED will validate this new nice level and may choose to alter this process' priority or quantum. In that case, the kernel will be notified with `sys_schedule`. Once SCHED replies to PM, PM will store the new nice value locally and can serve `getpriority` requests without contacting SCHED.

## 6.6 Multiple schedulers

Section 5 discusses the potential of multiple schedulers. We did not manage to fully research corner cases related to with migrating processes between schedulers, but we do have basic support for multiple schedulers. When forking a process, PM will always ask the forking process' scheduler to schedule the child as well. However, the parent's scheduler may choose to forward the scheduling request to another user mode scheduler. The user mode schedulers could jump through several hoops negotiating who should schedule the process for all PM cares, it just waits for its original scheduling request to complete as the reply will contain the endpoint of the scheduler who is responsible for the process.

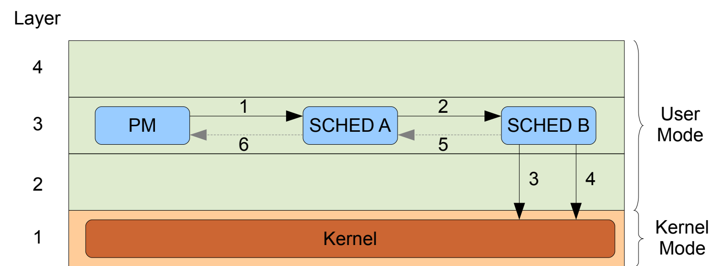


Figure 10: Support for multiple schedulers when forking. Note that two replies are included in this Figure.

Alternative approaches were discussed, such as adding options to the `fork` system call. This way, the parent could give PM a hint as to who should schedule the child. This approach has two drawbacks. First and most importantly, it would break the POSIX interface. Second, since the child's scheduler is likely to want information on how the parent was scheduled, messaging between user mode schedulers is likely to happen even if PM is given explicit information on who should schedule the process.

### 6.6.1 Messages

In Figure 10 we include two replies, noted in gray.

1. PM sends a `SCHEDULING_START` message to SCHED A.
2. SCHED A decides that another scheduler should handle scheduling this process and forwards the scheduling request to SCHED B.
3. SCHED B takes over scheduling with `sys_schedctl`.
4. SCHED B schedules the process for the first time, giving the process a priority and quantum.
5. SCHED B sends a confirmation reply to SCHED A, stating that it is now scheduling the process.
6. SCHED A sends a confirmation reply to PM, but includes SCHED B's endpoint instead of its own. PM now knows that the process has been scheduled by a scheduler, even though it is not the scheduler that PM initially thought would schedule the process. PM stores SCHED B's endpoint as the scheduler of the process, as it would have stored SCHED A in the case of a single scheduler.

# 7 Process feedback

After the initial implementation we realized that the out of quantum message sent from kernel was essentially an empty message. We were only making use of the message's sender address, not the body. As we had already discussed ways of exposing system and process status to the user mode scheduler, we decided to take advantage of this unused message space and push metrics to the scheduler.

The more metrics a user mode scheduler has, the more advanced scheduling decisions it can take. Of course, there is a practical limit of how much information you want to expose and how much time to spend on scheduling. Having gone though several existing policies, and looking towards potential benefits to multi-core scheduling, we decided to expose the attributed listed in Table 1. Other metrics of interest might include the L2 cache misses for the core running the process or the time a process spends waiting for incoming messages.

| Attribute | Description |
|---|---|
| SCHEDULING_ACNT_QUEUE | The process' CPU sojourn time (ms)[2]. A value close or equal to the process' quantum indicates that the process got access to the CPU when requested (didn't spend a lot of time in the run queue waiting for its chance to run). |
| SCHEDULING_ACNT_IPC_SYNC | Number of times this process blocked on synchronous IPC calls. The kernel does not know whether the process is blocking on IO or doing some other inter process communication. All we know is that the process is blocked waiting on another process. |
| SCHEDULING_ACNT_IPC_ASYN | The number of asynchronous messages sent. |
| SCHEDULING_ACNT_DEQS | The number of times this process got dequeued. |
| SCHEDULING_ACNT_PREEMPT | The number of times this process got preempted due to a higher priority process being switched in. Continuously preempting a process, even if only for a moment each time, may severely impact its performance (for instance if its working set gets pushed out of cache). |
| SCHEDULING_ACNT_CPU_LOAD | The CPU utilization (0-100) on the core where the process is scheduled. This value is mainly useful when balancing processes between cores. |

Table 1: System and process feedback provided in out of quantum messages.

These metrics allow for informed policing. For example, a process that sends many IPC messages will release the CPU more frequently than a process that never sends messages. It is likely to be IO bound, while the process that never communicates is CPU bound. IO bound processes as often given a high priority but a short quantum, while CPU bound processes may get a lower priority but larger quantum. A policy similar to this was implemented, but not checked in (Section 7.1).

Schedulers managing multiple cores can also make use of this information. Obviously, the CPU utilization on the two processors gives an indication of load balance. But other metrics may also prove useful. The scheduler may try to balance CPU and IO bound processes, it may find that two low latency, high priority processes running on the same core are better served on different cores etc. Future work might include collecting information on which processes communicate with which so that processes could be clustered based on communication patterns.

[2]This metric is fairly accurate on real hardware, but should not be relied on through virtualization. The qemu emulator on Linux seem close to the real value, but VMWare on Windows is way off.

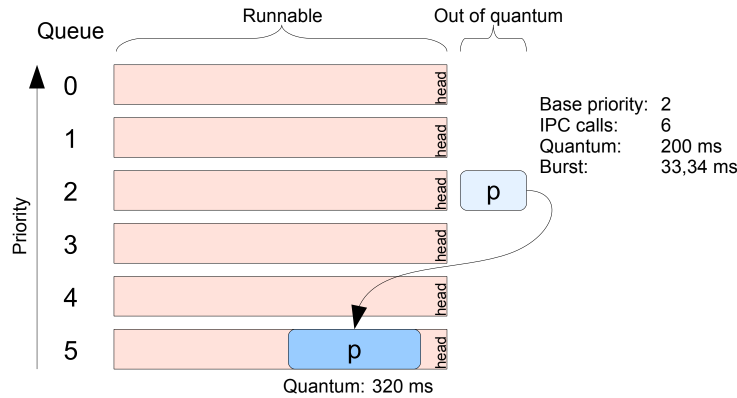


Figure 11: The feedback scheduler adjusts priority and quantum based on the number of IPC calls performed during the last run.

Currently, information is only pushed to the scheduler when processes run out of quantum. The may be a problem if the processes scheduled are either sitting idle or have very long quantum. To address this, a periodic out of quantum message may be send on behalf of the idle process with a regular interval, if no process has run out of quantum.

## 7.1 Proof-of-concept Scheduler

We implemented a proof-of-concept scheduler that makes use of the metrics included in the out of quantum message. This scheduler was not checked into the MINIX 3 mainline, but is available as a patch on the MINIX 3 wiki.

### 7.1.1 Policy

This policy introduces *bursts*, the average amount of work in milliseconds between IPC calls (Equation 1). The burst is used to estimate whether the process is IO or CPU bound.

$$burst = \frac{process\ quantum}{number\ of\ IPC\ calls} \quad (1)$$

We calculate a moving average, based on the last 10 samples, and prioritize the process based on this value. Each process has both a base priority and base quantum. These define the process' highest priority and the quantum it should have at that priority. For each 10 ms of burst[3]we penalize the process by lowering its priority by one. When lowering priority, we also increase the priority by 20%.

**Example:** In Figure 11 we show a process $p$ with a base priority of 2 and base quantum of 200 ms. When $p$ runs out of quantum the scheduler will inspect the number of IPC calls and calculate its burst.

$$burst = \frac{process\ quantum}{number\ of\ IPC\ calls} = \frac{200\,\text{ms}}{6} = 33,34\,\text{ms} \quad (2)$$

Given the burst, we determine whether this process needs to be pushed down in priority.

[3]Configured by the INC_PER_QUEUE constant

$$queue\ bump = \left\lfloor \frac{burst}{INC_PER_QUEUE} \right\rfloor = \left\lfloor \frac{33,34\,\mathrm{ms}}{10\,\mathrm{ms}} \right\rfloor = 3 \tag{3}$$

$$current\ priority = base\ priority + queue\ bump = 2 + 3 = 5 \tag{4}$$

Finally, for each priority penalty (queue bump) we increase the quantum by 20% of the base quantum.

$$current\ quantum = 200\,\mathrm{ms} + 3 \cdot 200\,\mathrm{ms} \cdot 20\% = 320\,\mathrm{ms} \tag{5}$$

Having calculated these estimations, the scheduler will schedule $p$ in priority queue 5 with a quantum of 320 ms.

#### 7.1.2 Limitations

The goal of this algorithm was to show that it would be possible to implement a policy based on kernel feedback. That said, this policy has several limitations. Most important, it is not starvation free. If a process enters a period of heavy CPU workload it may be pushed down to the lowest priority. Given that there are enough processes in the system with a higher priority, the process bumped down will never be run. Therefore, it would be advisable for the scheduler to optimistically increase the priority of low priority processes, similar to the current implementation's `balance_queues` approach.

# 8 Performance overhead

Moving scheduling to user mode introduces some overhead in the form of messages and context switches. Accurately measuring this overhead is hard with current tools available, but this section will discuss the overhead involved.

## 8.1 IPC in MINIX 3

A microkernel approach comes at a cost: kernel, servers and drivers have to do inter process communication. The MINIX 3 team has been open about the fact that MINIX 3 has been focused on stability and security, not performance.

Sending a message from kernel to a receiving process (with a free message slot) is not expensive. The delivery itself is mostly a matter of sanity checks and a memcpy. For the process to then consume the message, it has to be switched in. Sending a message between servers is more expensive, as the the kernel needs to be switched in to deliver messages between processes. At best, a blocking request from $p$ to $q$ requires 1 message, 2 mode switches[4] and 2 context switches.[5]

$$p \quad \underset{1}{\overset{mode}{\longrightarrow}} \quad kernel \quad \underset{1}{\overset{cxt}{\longrightarrow}} \quad q \quad \underset{2}{\overset{mode}{\longrightarrow}} \quad kernel \quad \underset{2}{\overset{cxt}{\longrightarrow}} \quad p \tag{6}$$

Intuitively we find that a large portion of communication overhead can be attributed to context switching between communicating processes. Exact measurements are not readily available, but it is estimated that sending a message between processes takes about 500 nsec [3]. This is considered acceptable.

## 8.2 Communication overhead

Communication overhead can be split in several categories. First, several messages are sent when spawning and terminating processes. Second, two messages are sent when a process runs out of quantum, one from kernel and another from the scheduler. Third, depending on the user mode scheduler, additional messages may be sent either to kernel or other user mode schedulers. Lets look at these three cases.

### 8.2.1 Forking and terminating

Forking a process is an expensive operation. It involves multiple servers and thus multiple messages. Before user mode scheduling, messages were sent between the forking process, PM, VM, VFS and kernel. With user mode scheduling, PM will additionally send one message to SCHED, which will send two messages to kernel, before returning to the parent process (Section 6.2). At minimum, this requires 6 mode switches and 2 context switches.

$$\begin{array}{cccccccc} \cdots & \longrightarrow & PM & \underset{1}{\overset{mode}{\longrightarrow}} & kernel & \underset{1}{\overset{cxt}{\longrightarrow}} & SCHED & \\ & \underset{2}{\overset{mode}{\longrightarrow}} & kernel & \underset{3}{\overset{mode}{\longrightarrow}} & SCHED & \underset{4}{\overset{mode}{\longrightarrow}} & kernel & \\ & \underset{5}{\overset{mode}{\longrightarrow}} & SCHED & \underset{6}{\overset{mode}{\longrightarrow}} & kernel & \underset{2}{\overset{cxt}{\longrightarrow}} & PM & \longrightarrow \cdots \end{array} \tag{7}$$

The two messages sent by SCHED to kernel could be combined into one. This would save 2 mode switches. This optimization was not pursued as it would add complexity for limited benefit.

[4] When a process communicates with kernel we don't switch context, only privilege mode.

[5] A context switch is required when the kernel switches in a process that was previously not running (includes a mode switch).

The overhead of user mode scheduling when terminating a process is 1 message, 2 mode switches and 2 context switches.

$$\begin{array}{ccccccc} \cdots & \longrightarrow & PM & \xrightarrow[1]{mode} & kernel & \xrightarrow[1]{cxt} & SCHED \\ & \xrightarrow[2]{mode} & kernel & \xrightarrow[2]{cxt} & PM & \longrightarrow & \cdots \end{array} \tag{8}$$

### 8.2.2 Out of quantum messages

When a process runs out of quantum its scheduler is sent a message. The message is sent by the kernel, so sending the message does not require any additional mode switching. However, the process that ran out of quantum is currently not runnable, and will not be scheduled until the user mode scheduler has been woken up and rescheduled the process. This involves switching the scheduler into context. Therefore the overhead is 1 message, 1 mode switch and 1 context switch.

$$kernel \xrightarrow[1]{cxt} SCHED \xrightarrow[1]{mode} kernel \tag{9}$$

Having to send a message and perform a context switches each time a process runs out of quantum may not sound ideal. Considering however that the default quantum in MINIX 3 is 200 ms and sending a message takes 500 nsec, comparing this single message to the number of messages required to read a block from a disk drive makes this overhead seem limited.

A potential cause for concern is the user mode scheduler's response time. Time spent making a scheduling decision for process $p$, is time that the $p$ spends waiting for quantum.

### 8.2.3 Additional messages

Depending on the scheduling policy, additional messages may be required. This could for example involve pro-actively modifying process priorities, as per the default policy's queue balancing. In case of multiple schedulers, some inter-scheduler communication could take place, for example when spawning a process (as described in Section 5).

## 8.3 Process feedback accounting

For providing the scheduler with system and process feedback, the kernel keeps track of several metrics. These include counters for the number of times the process has been queued and preempted, the number of messages spend, the amount of time spent in queue and keeping track of the average CPU load since the last out of quantum message. Keeping track of 32 bit counters is cheap but calculating queue time is more expensive as it involves 64 bit arithmetic.

# 9 Future work

There is lots of room for improvement, and now that scheduling is in its own user mode server we hope that developing new policies is accessible to more than before. An interesting project would be to implement more policies, especially policies that make use of system and process feedback. More feedback can be added, such as the L2 cache misses on a particular core.

Another interesting project would be implementing multiple schedulers, each allocated a set of cores. Finally, the IPC code itself would use a little attention. In particular, there is no notion of priority inheritance—which may be limiting when implementing policies.

# 10 Conclusion

We have successfully moved schedule policing in MINIX 3 from kernel mode to user mode. The kernel implements a raw priority queue scheduler while scheduling decisions are taken by a scheduling server in user mode. A simple interface was introduced where the kernel notifies the scheduler when a process runs out of quantum and the scheduler make system calls to schedule processes.

Feedback on system and process statistics is piggybacked to the out of quantum message, providing the scheduler with metrics that can be used to make more intelligent scheduling decisions. A proof-of-concept scheduling policy was implemented to demonstrate this.

The first phase of this IPA was checked into MINIX 3 mainline on March 29th and was released in MINIX 3.1.7. This included full kernel mode/user mode separation. Following phases have since been committed and released as of MINIX 3.1.8.

# References

[1] Con Kolivas. staircase scheduler 2.6.7-rc1, May 2004.

[2] Con Kolivas. Rsdl completely fair starvation free interactive cpu scheduler, March 2007.

[3] Andrew S. Tanenbaum. Minix 3, a modular, self-healing posix-compatible operating system. In *Room Janson @ FOSDEM 2010*, Brussels, Belgium, February 2010.

[4] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series)*. Prentice Hall, January 2006.