VERIFICATION OF THE L4 TASK SCHEDULER

VASILY TSYBA
September 2003
ABSTRACT

The document contains 85 pages in A4 format, including 7 figures, 1 table, 20 references and 1 appendix.

L4 OPERATING SYSTEMS, MICROKERNEL, SCHEDULING, VERIFICATION, FORMAL METHODS, PVS VERIFICATION SYSTEM

This work aims to establish correctness of implemented in L4 operating system scheduling mechanism. In this case the correctness is the satisfaction of the formal model of the scheduler to some criteria established through the analysing of the source code.

Ultimately, through formulating correctness criteria at first in natural language then formalizing the necessary data types, functions and the above criteria in PVS specification language it has been done the proving of correctness of the part of the L4 task scheduler with respect to the formalized criteria.
4.2.1 Thread control block ..................................................................................29
4.2.2 Organization of system time in L4...............................................................32
4.2.3 Implementation of the idle thread ...............................................................33
4.2.4 Implementation of the ready queue ...........................................................33
4.2.5 Implementation of the wakeup queue .........................................................34
4.3 Functions used by the scheduler .....................................................................35
4.3.1 Auxiliary functions used by the scheduler ................................................35
4.3.1.1 void switch_to_idle(tcb_t* current) ..................................................35
4.3.1.2 void switch_to_thread(tcb_t* next, tcb_t* current)..........................36
4.3.1.3 void thread_enqueue_ready(tcb_t* t) ............................................36
4.3.1.4 void thread_dequeue_ready(tcb_t* tcb) ..................................36
4.3.1.5 void dispatch_thread(tcb_t* t) ......................................................36
4.3.2 Basic functions ..........................................................................................37
4.3.2.1 tcb_t* find_next_thread() ................................................................37
4.3.2.2 tcb_t* parse_wakeup_queue(dword_t current_prio, qword_t current_time)..........................................................................................................38
4.3.2.3 void handle_timer_interrupt().........................................................39
4.4.1 void sys_schedule(schedule_param_t param, l4_threadid_t tid)...........39
4.4.2 void sys_thread_switch(l4_threadid_t tid)............................................40
4.5. Order of scheduling .....................................................................................40
4.6 Architecture of user’s scheduler .....................................................................41
5 Verification .......................................................................................................42
5.1 The correctness criteria ................................................................................42
5.2 PVS ..............................................................................................................44
5.3 Specification of double linked list ..................................................................45
5.3.1 Properties of the ring data type ...............................................................47
5.3.2 Operations over ring data type ...............................................................48
5.3.3 Lemmas stating correctness of specification of the double linked list. .................49
5.4 Formal specification of the struct tcb_t .......................................................50
5.5 Formal specification of the ready queue .................................................... 53
  5.5.1 Formal description of the initial state of ready queue ......................... 54
  5.5.2 Operations over the formal model of ready queue ............................. 55
  5.5.3 Formal specification of properties of ready queue ............................ 56
  5.5.4 Ready queue correctness proof sketch ............................................. 59
  5.5.5 Lemmas about operations over ready queue .................................... 60
5.6 Formal specification of \textit{find\_next\_thread} function .......................... 62
  5.6.1 Lemmas concerning \textit{find\_next\_thread} function ....................... 64

Conclusion ....................................................................................................... 66
References ....................................................................................................... 67
Appendix A PVS files...................................................................................... 69
### 1 CLASSIFICATION OF OPERATING SYSTEMS

An operating system is a program which acts as intermediary between user and computer hardware. An operating system has two main objectives - the first one is to make using of computer's hardware convenient. The second is to manage hardware resources as effectively as possible. The kernel is that part of OS, which executes in system mode. Some parts of OS can be implemented as user mode programs.

Modern OSs offer two encapsulations of program:

- **task or process** is a program in execution. Each task has individual address space protected from other tasks, unique identifier and associated set of threads.
- **thread** is a part of a task. It encapsulates an execution flow. Each thread has an unique identifier; a state (it can be: running, waiting for some event etc.); an individual set of registers (thus, its own virtual CPU). Threads which belong to the same task share its address space.

Operating systems can be arranged according to the structure of their kernel. The most widely used modern OSs have monolithic or microkernel structure.

#### 1.1 OSs without Kernel

Not all OS's have a kernel which is protected from user programs and which manages the hardware and the user programs. Some early operating systems only provided an interface to the hardware to be used by programs. However, they do not protect themselves from user programs nor provide the protection of the programs from each other.

#### 1.2 OS with a Kernel

This architecture evolved to an OS design with two process classes: one running in system mode, and another running in user mode. The kernel has full control of the hardware and provides abstractions for the processes running in user mode. A process running in user mode cannot access the hardware but must use the abstractions provided by the kernel. It can call certain services of the kernel by making „system calls“ or kernel calls. The kernel only offers the basic services. All others are provided by programs running in user mode.
1.3 Monolithic Kernels

The older monolithic kernels were written as a mixture of everything the OS needed without much of an organisation. A monolithic kernel offers: processes, memory management, interprocess communication (IPC), device access, file systems, network protocols and whatever the OS should implement. Newer monolithic kernels have a modular design which offers run-time adding and removing of services. The whole kernel runs in „kernel mode“. The processes running on top of the kernel run in „user mode“. Although modern OSs have a big modularity, they are still monolithic. Examples for such OSs are Unix, Solaris.

Monolithic kernels have high performance because the kernel interacts directly with the hardware. The monolithic kernels can be optimised for a particular hardware architecture, but services cannot be changed without need to restart the whole system. The hardware-specific part of a monolithic kernel is relatively large, therefore the kernel is not easy portable.

1.4 Microkernel Architecture

Microkernel designs run as many OS services as possible at user level. This makes the kernel a lot smaller and offers a far greater flexibility. File systems, device drivers, process management and even parts of the memory management can be put in processes running on top of the microkernel. This architecture is actually a client-server model: processes (clients) can call OS services by sending requests through IPC to server processes e.g., a process that wants to read from a certain file sends a request to the file system process. The central processes that provide the process management, file system etc. are frequently called the servers. Microkernels are often also highly multithreaded, putting every different service in a different thread, offering greater speed and stability. The main difficulty of microkernels is to make IPC as fast as possible. This was a design problem in early microkernel design, because IPC, while being intended to be the power of the architecture, often proved to be the bottleneck. Now, however, microkernels do offer fast IPC.
When talking about microkernels, one must clearly make a difference between the first and the second generation. The first-generation microkernels, like Mach [3], are fat and provide lots of services, or multiple ways to do the same thing. The second-generation microkernels follow more the „pure“ microkernel idea: very small kernels, only offering the abstractions really needed, with a clean and unambiguous Application Program Interface (API). Examples of the second generation are L4 [4] and QNX [5]. Microkernels move many of the OS services into „user space“ that on other operating systems are kept in the kernel. This has significant effects in the following areas:

- **Robustness**: if there is a problem with a particular service, it normally can be reconfigured and restarted without having to restart the complete OS. This should be helpful for situations requiring high availability. Moreover, since services would now run in completely independent memory spaces (which is not the case for kernel-level services), bugs and misconfiguration can not easily corrupt the kernel. The „true kernel“ winds up being smaller in scope, and thus ought to be easier to understand and verify. The L4-microkernel occupies about 32K [6] of memory, which severely limits how complex it can realistically be.

- **Security**: services that run within the kernel effectively have kernel class privileges, which is to say that they can do anything, anywhere, at anytime. Unix processes that run as root do not have that much control over the system. A problem at present with Linux is that any program that has graphics access must be set uid root because that is the only way of having permission to access the graphics hardware. This may have the unfortunate effect that the program gets „root“ access to everything on the system and not just the screen. By running the services as lower level user processes, their access to system resources (e.g. the ability to mess things up) is far more restricted. Moreover, „security“ becomes less monolithic, which allows even system services, and indeed security services for that matter, to themselves be forced to comply with security requirements.

- **Configurability**: services can be changed without need to restart the whole system. Work has been ongoing (for instance) to make Linux more dynamically reconfigurable, loadable kernel modules being a good example.
• Makes coding easier: kernel code usually requires the use of special memory allocation and output routines since the kernel cannot depend on a lower level to manage these things for it. User-mode code is thus simpler to write than „kernel“ code, because it does not need to worry about hardware-specific restrictions.

• Performance: communication between components of the extended OS requires that formalised message-passing mechanisms are used. As a result, code must be written to use the formal mechanisms, rather than processes being able to informally use system memory. This may reduce performance. New kinds of deadlocks and other error conditions are possible between system components that would not be possible with a monolithic kernel.

1.5 ExoKernel

Exokernels [7] are a further extension of the microkernel approach where the „kernel“ is almost devoid of functionality; it merely passes requests for resources to user „space“ libraries. The exokernel's architecture implements nothing in kernel space. The exokernel's sole purpose is to securely multiplex hardware resources among user-space processes. Device drivers, virtual memory, even CPU multiplexing and process management are implemented in user space. Supervisor-mode hardware events, like timer ticks, page faults, etc., activate stub handlers in the kernel that simply pass the event to a user-level process that implements the relevant facility’s policy. The same system can simultaneously implement forward and inverted page tables, compute-job-friendly or interactive-job-friendly process scheduling, and an application can pick and choose which ever policies will provide it with the best performance.

1.7 OS Summary

The different types of OS’s allow to cover all needs in different areas. The popular OSs with monolithic kernel provide a high performance for specific hardware, but they have a relative low reliability. In the last time microkernel ideas have a growing tendency and good performance with high level safety and security. An example of a successful microkernel OS is Mach OS [3], but it has a rather big kernel.
The L4 and QNX[5] are the last popular ideas in that direction. L4 has a small kernel (about 32k) and has only 7 system calls, that it is enough for building a complete OS with memory management, message passing, security polices etc. As perspective ideas to improve performances of OS’s we observed Exo-kernel OS, as logical successors of the mikrokernel idea.

2 THE L4 OPERATING SYSTEM

L4 by itself is not an OS, but rather constitutes a minimal base on which a variety of complete operating systems can be built. The basic idea of a microkernel goes back to Brinch Hansen's Nucleus and Hydra and has been popularised by Mach [3]. The argument goes that by reducing the size of the kernel (the part of the OS executing in privileged mode), it becomes possible to build a system which is more secure and reliable (because the trusted computing base is smaller) and easy to extend. A further advantage is that a microkernel-based system can easily implement a number of different APIs (also called OS personalities) without having to emulate one within the other.

There was also hope of improved efficiency, as operating systems tend to grow as new features are added, resulting in an increase of the number of layers of software that need to be traversed when asking for service. (An example is the addition of the VFS layer in UNIX for supporting NFS.) A microkernel based system, in contrast, would grow horizontally rather than vertically: Adding new services means adding additional servers, without lengthening the critical path of the most frequently used operations. However, performance of these first-generation microkernels proved disappointing, with applications generally experiencing a significant slowdown compared to a traditional („Monolithic“) operating system. Liedtke [10,11], however, has shown that these performance problems are not inherent in the microkernel concept and can be overcome by good design and implementation. L4 is the constructive proof of this theorem.
2.1 L4’s design principles

The most fundamental task of an operating system is to provide secure sharing of resources, in essence this is the only reason why there needs to be an operating system. A microkernel has to be as small as possible. Hence, the main design criterion of the microkernel is minimality with respect to security: A service (feature) is to be included in the kernel if and only if it is impossible to provide that service outside the kernel without loss of security. The idea is that once we make things small, performance will look after itself. A strict application of this rule has some surprising consequence. For example device drivers: Some device drivers access physical memory and can therefore break security. They need to be trusted. This, however, does not mean that they need to be in the kernel. If they need not execute privileged instructions, and if the kernel can provide sufficient protection to run them at user level, then this is what should be done, and, consequently, this is what L4 does. Another important principle is that it should be possible to implement arbitrary systems on top of the microkernel. Together with the minimality principle this leads to requirements for a small number of powerful and orthogonal abstractions, and for a strictly policy-free kernel.

2.2 Abstraction in L4

Based on the described ideas Liedtke [11] concludes that the microkernel needs to provide:

- address spaces - because they are the base of protection.
- threads - because there needs to be an abstraction of program execution.
- inter-process communication (IPC) - as there needs to be a way to transfer data between address spaces.
- unique identifiers (UIDs) - for context-free addressing in IPC operations.

2.2.1 Threads

A thread is the basic execution abstraction. A thread has an address space (shared with the other threads belonging to the same task), a UID, a register set (including an instruction pointer and a stack pointer), a page fault handler (pager), and an exception handler. IPC operations are addressed to threads (via their UIDs). Threads
are extremely light-weight and cheap to create, destroy, start and stop. The lightweight thread concept, together with very fast IPC, is the key to the efficiency of L4 and OS personalities built on top.

2.2.2 Address Space

An address-space contains all the data (other than hardware registers) which are directly accessible by a thread. An address space is a set of mappings (For more explanations see [9],[21]) from virtual to physical memory(which is partial in the sense that many mappings are undefined, making the corresponding virtual memory inaccessible). Address spaces in L4 can be recursively constructed: A thread can map parts of its address space into another thread’s address space (provided the receiver cooperates) and thereby share the data mapped by that region of the address space. Mappings can be revoked at any time, so the mapper retains full control. Alternatively, virtual address space can be granted to a different address space. In this case the granter relinquishes all control over the data mapped by that part of the address space and no longer has a valid mapping for that address space region. The granter can not revoke a grant. The grantee, in contrast, inherits full control over the granted address space (unless the grant was read-only, in which case write access is lost.) Note that while a grant is irreversible, the granter has, in general, received the address space (directly or indirectly) via mapping, and an address space at the beginning of the mapping chain can still revoke the mapping. Mapping and granting are implemented as operations on page tables, without copying any actual data. Mapping and granting is achieved as a side effect of IPC operations and specified by the means of flex pages. This is not accidental: For security reasons mapping requires an agreement between sender and receiver, and thus requires IPC anyway. The concept of a task is essentially equivalent to that of an address space.

In L4, a task is the set of threads sharing an address space. Creating a new task creates an address space with one running thread. Strictly speaking the number of tasks is a constant (255 in implementation for x86 architecture). There are two kinds of tasks: active and inactive ones. When we say that a task is created we mean that an inactive task is activated. Inactive tasks are essentially capabilities (task creation rights). This is important, as a thread can only create a task if it already owns the task ID to use.
Inactive tasks can be donated to other tasks. There is a hierarchy of tasks, with parents having some limited control over their children. The main purpose of this is to be able to control (IPC-based) information flow between address spaces. It has nothing to do with process hierarchies a particular L4-based personality OS may implement. Such a hierarchy is under full control of the particular OS personality.

2.2.3 IPC

Message-passing IPC is the heart of L4. The microkernel provides a total of seven system calls (IPC being one of them), which provide some very rudimentary OS functionality. Everything else must be built on top, implemented by server threads, which communicate with their clients via IPC. IPC is used to pass data by value (i.e., the microkernel copies the data between two address spaces) or by reference (using mapping or granting). Data by value can be transferred in two ways: in-line, when a limited amount of such data is passed directly in registers (3 words) with any remainder in a message buffer. And out-of-line, an arbitrary out-of-line buffers are copied to the receiver. IPC is also used for synchronisation (as it is blocking, so each successful IPC operation results in a rendez-vous), wakeup-calls (as timeouts can be specified), pager invocation (the microkernel converts a page fault into an IPC to a user-level pager), exception handling (the microkernel converts an exception fault into an IPC to a user-level exception handler), and interrupt handling (the microkernel converts an interrupt fault into an IPC to a user-level interrupt-handler from a pseudo-thread). Device control is registered via IPC (although actual device access is memory mapped).

2.2.4 Clans & Chiefs

Clans and chiefs are L4’s basic mechanism enabling the implementation of arbitrary security policies. They allow controlling IPC and thus information flow. The basic idea is simple: A task’s creator is that task’s chief, all tasks (directly) created by a particular chief constitute that chief’s clan. Threads can directly send IPC only to other threads in the same clan, or to their chief. If a message is sent to a thread outside the clan containing the sender, that message is instead delivered to the sender's chief (who may or may not forward the message). If a message is sent to a member of a subclan of
the clan containing the sender, that message is delivered to the task in the clan whose clan (directly or indirectly) contains the addressee.

2.2.5 UIDs

A microkernel supplies unique identifiers (UIDs) for something, either for threads or tasks or communication cannels. UIDs are required for reliable and efficient local communication. If $S_1$ wants to send a message to $S_2$ it needs to specify the destination $S_2$ (or some channel leading to $S_2$). Therefore, the microkernel must know which UID relates $S_2$. On the other hand, the receiver $S_2$ wants to be sure that the message comes from $S_1$. Therefore the identifier must be unique, both in space and time. A UID of a thread is that of its task plus the number of the thread within the task. The UID of a task consists of the task number, some fields describing its place in the task hierarchy, and a version number. Both, tasks and threads are limited (there are 255 tasks and within each task 64 threads). This means that tasks (or address spaces) and threads must be recycled. The microkernel ensures uniqueness of task IDs by incrementing the version number whenever a task number is reused. The version number has valid values from 0 to 1023. Developers of the system consider this amount to be enough. As far as threads are concerned, the L4 view is that threads do not die as long as their task exists, they can only be blocked (waiting for IPC which will never arrive). This avoids the issue of lthread uniqueness. Obviously, both thread and task numbers are insufficient for a real multi-user operating system. This means that an OS personality will in general need to map its own task and thread abstraction onto L4’s. How this is done is up to the OS personality, L4 only provides the tools.
3 PROCESS SCHEDULING

Since in all modern operating systems more than one process in system is runnable, the operating system must decide which one to run first. The part of the operating system that makes this decision is called the scheduler; the algorithm it uses is called the scheduling algorithm.

Before looking at specific scheduling algorithms, we should think about what the scheduler is trying to achieve. After all, the scheduler is concerned with deciding on policy, not providing a mechanism. Various criteria come to mind as to what constitutes a good scheduling algorithm. Some of the possibilities include:

1. Fairness—make sure each process gets its fair share of the CPU.
2. Efficiency—keep the CPU busy 100 percent of the time.
3. Response time—minimise response time for interactive users.
4. Turnaround—minimise the time batch users must wait for output.
5. Throughput—maximise the number of jobs processed per hour.

A little thought will show that some of these goals are contradictory. To minimise response time for interactive users, the scheduler should not run any batch jobs at all. It can be shown that any scheduling algorithm that favors some class of jobs hurts another class of jobs. The amount of CPU time available is finite, after all.

A complication that schedulers have to deal with is that every process is unique and unpredictable. Some spend a lot of time waiting for file I/O, while others would use the CPU for hours at a time if given the chance. When the scheduler starts running some process, it never knows for sure how long it will be until that process blocks, either for I/O, or on a semaphore, or for some other reason. To make sure that no process runs too long, nearly all computers have an electronic timer or clock built in, which causes an interrupt periodically. At each clock interrupt, the operating system gets to run and decide whether the currently running process should be allowed to continue, or whether it has had enough CPU time for the moment and should be suspended to give another process the CPU.

The strategy of allowing processes that are logically runnable to be temporarily suspended is called preemptive scheduling, and is in contrast to the run to completion

---

1 Parts of this chapter are literally taken from [20].
method. Run to completion is also called nonpreemptive scheduling. Using preemptive scheduling a process can be suspended at an arbitrary instant, without warning, so another process can be run. This leads to race conditions and necessitates semaphores, monitors, messages, or some other sophisticated method for preventing them. On the other hand, a policy of letting a process run as long as it wanted to would mean that some process computing π to a billion places could deny service to all other processes indefinitely.

Thus although nonpreemptive scheduling algorithms are simple and easy to implement, they are usually not suitable for general-purpose systems with multiple competing users. On the other hand, for a dedicated system, such as a data base server, it may well be reasonable for the master process to start a child process working on a request and let it run until it completes or blocks. The difference from the general-purpose system is that all processes in the data base system are under the control of a single master, which knows what each child is going to do and about how long it will take.

3.1 Round Robin Scheduling

Now let us look at some specific scheduling algorithms. One of the oldest, simplest, fairest, and most widely used algorithms is round robin. Each process is assigned a time interval, called its time slice or quantum, which it is allowed to run. If the process is still running at the end of the quantum, the CPU is preempted and given to another process. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks, of course. Round robin is easy to implement. All the scheduler needs to do is maintain a list of runnable processes. When the process uses up its quantum, it is put on the end of the list.

The only interesting issue with round robin is the length of the quantum. Switching from one process to another requires a certain amount of time for doing the administration—saving and loading registers and memory maps, updating various tables and lists, etc. Setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests.
3.2 Priority Scheduling

Round robin scheduling makes the implicit assumption that all processes are equally important. But even on a PC with a single owner, there may be multiple processes, some more important than others. The basic idea is straightforward: each process is assigned a priority, and the runnable process with the highest priority is allowed to run.

To prevent high-priority processes from running indefinitely, the scheduler may decrease the priority of the currently running process at each clock tick (i.e., at each clock interrupt). If this action causes its priority to drop below that of the next highest process, a process switch occurs. Alternatively, each process may be assigned a maximum quantum that it is allowed to hold the CPU continuously. When this quantum is used up, the next highest priority process is given a chance to run.

Priorities can be assigned to processes statically or dynamically. Statically assigned priorities are often used to provide an advantage for some user, or to ensure running of more important processes before less important. Priorities can also be assigned dynamically by the system to achieve certain system goals. For example, some processes are highly I/O bound and spend most of their time waiting for I/O to complete. Whenever such a process wants the CPU, it should be given the CPU immediately, to let it start its next I/O request, which can then proceed in parallel with another process actually computing. Making the I/O bound process wait a long time for the CPU will just mean having it around occupying memory for an unnecessarily long time. A simple algorithm for giving good service to I/O bound processes is to set the priority to 1/f, where f is the fraction of the last quantum that a process used.

It is often convenient to group processes into priority classes and use priority scheduling among the classes but round-robin scheduling within each class. Figure 3.1 shows a system with four priority classes. The scheduling algorithm is as follows: as long as there are runnable processes in priority class 4, just run each one for one quantum, round-robin fashion, and never bother with lower priority classes. If priority class 4 is empty, then run the class 3 processes round robin. If classes 4 and 3 are both empty, then run class 2 round robin, and so on. If priorities are not adjusted occasionally, lower priority classes may all starve to death.
3.3 Multiple Queues

The earliest priority schedulers have the problem that process switching was very slow because the earliest computers could hold only one process in memory. Each switch meant swapping the current process to disk and reading in a new one from disk. The obvious way to improve efficiency is to give CPU-bound processes a large quantum once in a while, rather than giving them small quanta frequently (to reduce swapping). On the other hand, giving all processes a large quantum would mean poor response time. One of possible solutions is to set up priority classes. Processes in the highest class are run for one quantum. Processes in the next highest class are run for two quanta. Processes in the next class are run for four quanta, and so on. Whenever a process used up all the quanta allocated to it, it is moved down one class.

As an example, consider a process that needed to compute continuously for 100 quanta. It would initially be given one quantum, then swapped out. Next time it would get two quanta before being swapped out. On succeeding runs it would get 4, 8, 16, 32, and 64 quanta, although it would have used only 37 of the final 64 quanta to complete its work. Only 7 swaps would be needed (including the initial load) instead of 100 with a pure round-robin algorithm. Furthermore, as the process sank deeper and deeper into the priority queues, it would be run less and less frequently, saving the CPU for short, interactive processes.

3.4 Shortest Job First

Most of the above algorithms were designed for interactive systems. Now let us look at one that is especially appropriate for batch jobs for which the run times are
known in advance. When several equally important jobs are sitting in the input queue waiting to be started, the scheduler should use shortest job first. At Figure 3.2 depicted four jobs $A$, $B$, $C$, and $D$ with run times of 8, 4, 4, and 4 minutes, respectively. By running them in that order, the turnaround time for $A$ is 8 minutes, for $B$ is 12 minutes, for $C$ is 16 minutes, and for $D$ is 20 minutes for an average of 14 minutes.

![Figure 3.2 – An example of shortest job first scheduling.](image)

Now let us consider running these four jobs using shortest job first, as shown in Figure 3.2(b). The turnaround times are now 4, 8, 12, and 20 minutes for an average of 11 minutes. Shortest job first is provably optimal. Consider the case of four jobs, with run times of $a$, $b$, $c$, and $d$, respectively. The first job finishes at time $a$, the second finishes at time $a + b$, and so on. The mean turnaround time is $(4a + 3b + 2c + d)/4$. It is clear that $a$ contributes more to the average than the other times, so it should be the shortest job, with $b$ next, then $c$ and finally $d$ as the longest as it affects only its own turnaround time. The same argument applies equally well to any number of jobs.

Because shortest job first always produces the minimum average response time, it would be nice if it could be used for interactive processes as well. To a certain extent, it can be. Interactive processes generally follow the pattern of wait for command, execute command, wait for command, execute command, and so on. If we regard the execution of each command as a separate "job," then we could minimise overall response time by running the shortest one first. The only problem is figuring out which of the currently runnable processes is the shortest one.

It is worth pointing out that the shortest job first algorithm is only optimal when all the jobs are available simultaneously. As a counterexample, consider five jobs, $A$ through $E$, with run times of 2, 4, 1, 1, and 1, respectively. Their arrival times are 0, 0, 3, 3, and 3.

Initially, only $A$ or $B$ can be chosen, since the other three jobs have not arrived yet. Using shortest job first we will run the jobs in the order $A$, $B$, $C$, $D$, $E$, for an
average wait of 4.6. However, running them in the order B, C, D, E, A has an average wait of 4.4.

3.5 Guaranteed Scheduling

A completely different approach to scheduling is to make real promises to the user about performance and then live up to them. One promise that is realistic to make and easy to live up to is this: If there are \( n \) users logged in while you are working, you will receive about \( \frac{1}{n} \) of the CPU power. Similarly, on a single-user system with \( n \) processes running, all things being equal, each one should get \( \frac{1}{n} \) of the CPU cycles.

To make good on this promise, the system must keep track of how much CPU each process has had since its creation. It then computes the amount of CPU each one is entitled to, namely the time since creation divided by \( n \). Since the amount of CPU time each process has actually had is also known, it is straightforward to compute the ratio of actual CPU had to CPU time entitled. A ratio of 0.5 means that a process has only had half of what it should have had, and a ratio of 2.0 means that a process has had twice as much as it was entitled to. The algorithm is then to run the process with the lowest ratio until its ratio has moved above its closest competitor.

3.6 Lottery Scheduling

While making promises to the users and then living up to them is a fine idea, it is difficult to implement. However, another algorithm can be used to give similarly predictable results with a much simpler implementation. It is called lottery scheduling.

The basic idea is to give processes lottery tickets for various system resources, such as CPU time. Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource. When applied to CPU scheduling, the system might hold a lottery 50 times a second, with each winner getting 20 msec of CPU time as a prize.

More important processes can be given extra tickets, to increase their odds of winning. If there are 100 tickets outstanding, and one process holds 20 of them, it will have a 20 percent chance of winning each lottery. In the long run, it will get about 20 percent of the CPU. In contrast to a priority scheduler, where it is very hard to state
what having a priority of 40 actually means, here the rule is clear: a process holding a fraction \( f \) of the tickets will get about a fraction \( f \) of the resource in question.

Lottery scheduling has several interesting properties. For example, if a new process shows up and is granted some tickets, at the very next lottery it will have a chance of winning in proportion to the number of tickets it holds. In other words, lottery scheduling is highly responsive.

Cooperating processes may exchange tickets if they wish. For example, when a client process sends a message to a server process and then blocks, it may give all of its tickets to the server, to increase the chance of the server running next. When the server is finished, it returns the tickets so the client can run again. In fact, in the absence of clients, servers need no tickets at all.

Lottery scheduling can be used to solve problems that are difficult to handle with other methods. One example is a video server in which several processes are feeding video streams to their clients, but at different frame rates. Suppose that the processes need frames at 10, 20, and 25 frames/sec. By allocating these processes 10, 20, and 25 tickets, respectively, they will automatically divide the CPU in the correct proportion.

3.7 Real-Time Scheduling

A real-time system is one in which time plays an essential role. Typically, one or more physical devices external to the computer generate stimuli, and the computer must react appropriately to them within a fixed amount of time. For example, the computer in a compact disc player gets the bits as they come off the drive and must convert them into music within a very tight time interval. If the calculation takes too long, the music will sound peculiar. Other real-time systems are patient monitoring in a hospital intensive-care unit, the autopilot in an aircraft, and safety control in a nuclear reactor. In all these cases, having the right answer but having it too late is often just as bad as not having it at all.

Real-time systems are generally categorised as hard real time, meaning there are absolute deadlines that must be met, or else, and soft real time, meaning that missing an occasional deadline is tolerable. In both cases, real-time behaviour is achieved by dividing the program into a number of processes, each of whose behaviour is
predictable and known in advance. These processes are generally short lived and can run to completion in under a second. When an external event is detected, it is the job of the scheduler to schedule the processes in such a way as that all deadlines are met.

The events that a real-time system may have to respond to can be further categorised as periodic (occurring at regular intervals) or aperiodic (occurring unpredictably). A system may have to respond to multiple periodic event streams. Depending on how much time each event requires for processing, it may not even be possible to handle them all. For example, if there are $m$ periodic events and event $i$ occurs with period $P_i$ and requires $C$ seconds of CPU time to handle each event, then the load can only be handled if

$$\sum_{i=1}^{m} \frac{C}{P_i} \leq 1 \quad (3.1)$$

A real-time system that meets this criteria is said to be schedulable.

As an example, consider a soft real-time system with three periodic events, with periods of 100, 200, and 500 msec, respectively. If these events require 50, 30, and 100 msec of CPU time per event, respectively, the system is schedulable because $0.5 + 0.15 + 0.2 < 1$. If a fourth event with a period of 1 sec is added, the system will remain schedulable as long as this event does not need more than 150 msec of CPU time per event. Implicit in this calculation is the assumption that the context-switching overhead is so small that it can be ignored.

Real-time scheduling algorithms can be dynamic or static. The former make their scheduling decisions at run time; the latter make them before the system starts running. Let us briefly consider a few of the dynamic real-time scheduling algorithms. The classic algorithm is the rate monotonic algorithm. In advance, it assigns to each process a priority proportional to the frequency of occurrence of its triggering event. For example, a process to run every 20 msec gets priority 50 and a process to run every 100 msec gets priority 10. At run time, the scheduler always runs the highest priority ready process, preempting the running process if needed.

Another popular real-time scheduling algorithm is earliest deadline first. Whenever an event is detected, its process is added to the list of ready processes. The list is kept sorted by deadline, which for a periodic event is the next occurrence of the event. The algorithm runs the first process on the list, the one with the closest deadline.
A third algorithm first computes for each process the amount of time it has to spare, called its laxity. If a process requires 200 msec and must be finished in 250 msec, its laxity is 50 msec. The algorithm, called least laxity, chooses the process with the smallest amount of time to spare.

While in theory it is possible to turn a general-purpose operating system into a real-time system by using one of these scheduling algorithms, in practice the context-switching overhead of general-purpose systems is so large that real-time performance can only be achieved for applications with easy time constraints. As a consequence, most real-time work uses special real-time operating systems that have certain important properties. Typically these include a small size, fast interrupt time, rapid context switch, short interval during which interrupts are disabled, and the ability to manage multiple timers in the millisecond or microsecond range.

3.8 Policy versus Mechanism

Up until now, we have tacitly assumed that all the processes in the system belong to different users and are thus competing for the CPU. While this is often true, sometimes it happens that one process has many children running under its control. For example, a data base management system process may have many children. Each child might be working on a different request, or each one might have some specific function to perform (query parsing, disk access, etc.). It is entirely possible that the main process has an excellent idea of which of its children are the most important (or time critical) and which the least. Unfortunately, none of the schedulers discussed above accept any input from user processes about scheduling decisions. As a result, the scheduler rarely makes the best choice.

The solution to this problem is to separate the scheduling mechanism from the scheduling policy. What this means is that the scheduling algorithm is parameterised in some way, but the parameters can be filled in by user processes. Let us consider the data base example again. Suppose that the kernel uses a priority scheduling algorithm but provides a system call by which a process can set (and change) the priorities of its children. In this way the parent can control in detail how its children are scheduled, even though it itself does not do the scheduling. Here the mechanism is in the kernel but the policy is set by a user process.
4 L4’s TASK SCHEDULER

A scheduling in L4 is preemptive, i.e. a scheduler can suspend running thread before its termination and start another one. Policy and mechanism of scheduling are separated, such that the scheduler deals only with mechanism. Policy of scheduling is defined by user by means of two properties associated with each thread: priority and time slice. See the sections 3.2 and 3.1 of previous chapter to get idea about meaning of this terms.

The scheduler uses a multi-level priority round-robin scheme, i.e. there is a fixed number of levels of priority and all threads with maximal priority share processor time with respect to the time slice associated with each thread. Thread scheduling in L4 works with respect to the following principles:
1. The scheduler always selects the thread with the highest priority.
2. The kernel does not change the priority of any thread.
3. CPU time is shared among threads with the highest priority with respect to the value of time slice associated with each thread.
4. The thread which is to be scheduled can donate its timeslice to any other thread.

4.1 Data structure of the scheduler

The data structure of the scheduler consists of two queues. The first queue is called ready queue, it contains all ready threads, i.e. threads which are able to be executed. The second queue is called wakeup queue. It contains threads which are not ready at the moment, but each thread in the wakeup queue has associated value of time at which it should become ready and be moved into the ready queue.

4.1.1 Idle thread

It is possible, that there are no ready threads at the moment at all. To manage such cases L4 kernel has special instance of a thread named idle thread. The idle thread is always ready. The idle thread is written as endless loop. It starts at the end of kernel initialization. When it has got a control, it performs a search for a ready thread and either switches to the ready thread or, in case there is no ready threads, calls
system_sleep() function. Implementation of function system_sleep() is hardware specific, for example it can turn computer in power-save mode etc.

4.1.2 Ready queue

The L4 kernel defines 256 levels of priority in the range [255..0] with 255 being the highest priority and 0 the lowest. All ready threads with the same level of priority are joined into the cyclic double linked list; such a list is also called round-robin queue. At any given moment of time there is a queue (possibly empty) associated with each priority level. For each particular level of priority there is a pointer which points to one of elements of the round-robin queue associated with this priority level. All these pointers taken together constitute an array of pointers. The index of each element in the array is equal to the level of priority of the associated round-robin queue. An element of a round-robin queue which is pointed by the corresponding element of the array is called head of this queue. The ready queue is a union of the array of pointers and all round-robin queues. The Figure 4.1 depicts an example of the ready queue.

![Diagram of the ready queue](image)

Figure 4.1 – Structure of the ready queue.

When we say that “a thread belongs to ready queue” we mean that “the thread belongs to a round-robin queue which contains threads with particular priority”.
In the Figure 4.1 we can see, that there are no threads with priority $P$, therefore the element of the array with index $P$ contains pointers to NULL. There are three threads with priority 0. These threads are joined into round-robin queue and the $0^{th}$ element of array of pointers contains pointer to one of the elements of the corresponding round-robin queue. This element is named \textit{head}. The round-robin queue associated with priority level 256 contains only one thread, its successor and predecessor are the same as the thread itself.

The ready queue can be changed by adding a new ready thread or by removing a thread which is no longer ready or by changing the priority of a thread which is belonging to the ready queue. A new thread is added into the particular round-robin queue of the ready queue with respect to its priority. A thread is added into tail of a round-robin queue i.e. such that it becomes the predecessor of current head of the queue.

4.1.3 Wakeup Queue

Wakeup queue consists of one round-robin queue. The idle thread is a permanent head of this queue, so there is no need to have special pointer to wakeup queue. Threads are always added in the tail of the wakeup queue, i.e. between idle thread and its predecessor. The procedure of adding a new thread into wakeup queue is illustrated in the Figure 4.2.

![Figure 4.2](image)

Figure 4.2 – Wakeup queue before (a) and after (b) adding the thread \textit{new} into it.
4.2 Implementation of the scheduler data structure in code

4.2.1 Thread control block

Operating system needs some information (like priority, current state, UID etc.) about each thread to organize thread management. This information is contained in a special data structure. In L4 this structure is called *thread control block*, it represents a thread. A thread control block (TCB) is described in C code with the help of an element of type *struct tcb_t*. We are interested only in those fields of the *tcb_t* which are used by the scheduler. These fields are listed in the Table 4.1.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Field type</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>myself</td>
<td>l4_threadid_t</td>
<td>It keeps unique identifier (UID) of the thread.</td>
</tr>
<tr>
<td>wakeup_prev</td>
<td>tcb_t*</td>
<td>These are pointers to the previous and next elements in the wakeup queue.</td>
</tr>
<tr>
<td>wakeup_next</td>
<td>tcb_t*</td>
<td></td>
</tr>
<tr>
<td>ready_prev</td>
<td>tcb_t*</td>
<td>These are pointers to the previous and next elements in a round-robin queue belonging to ready queue.</td>
</tr>
<tr>
<td>ready_next</td>
<td>tcb_t*</td>
<td></td>
</tr>
<tr>
<td>thread_stat</td>
<td>dword_t</td>
<td>It represents current state of the thread.</td>
</tr>
<tr>
<td>queue_stat</td>
<td>dword_t</td>
<td>It represents presence of the thread in queues.</td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>priority</td>
<td>dword_t</td>
<td>It specifies priority of the thread.</td>
</tr>
<tr>
<td>timeslice</td>
<td>sdword_t</td>
<td>It specifies length of the timeslice associated with the thread.</td>
</tr>
<tr>
<td>current_timeslice</td>
<td>sdword_t</td>
<td>It is used by scheduler to keep intermediate data.</td>
</tr>
<tr>
<td>absolute_timeout</td>
<td>qword_t</td>
<td>For threads in wakeup queue, it specifies the time when the thread should be moved into ready queue.</td>
</tr>
</tbody>
</table>

Unique identifier of a thread (see the chapter 2.2.5) is contained in the field `l4_threadid_t myself`. The type `l4_threadid_t` has length of 32-bit. The bits with indices 31-22 contain version of the task to which the thread belongs to. The bits with indices 21-16 contain number of the thread within the task. The bits 15-8 contain an UID of the task, and the bits 7-0 contain an UID of the chief of the task to which the thread belongs.

Fields `ready_prev`, `ready_next`, `wakeup_next` and `wakeup_prev` are pointers to the elements of the same type as a TCB itself. If a thread belongs to the ready queue, then it belongs to the one of the round-robin queues. The field `ready_prev` points to the previous member of round-robin queue and `ready_next` points to the next member of the queue. This is illustrated in the Figure 4.3. The fields `wakeup_prev` and `wakeup_next` are used for the same purpose within wakeup queue (see the Figure 4.4).

From the point of view of the scheduler, a state of a thread is only used to determine whether the thread is runnable or not. State of a thread is defined as combination of last four bits in the field `thread_state`. These bits have the following names: TSB_READY, TSB_POLLI NG, TSB_LOCKED, TSB_XCPU. Bit TSB_READY indicates whether the thread is able to be executed or not. Indeed, the scheduler needs only value of the bit TSB_READY to decide whether the thread is runnable or not. We are not interested in values of other bits since scheduler does not deal with them.
Ready thread can be either running or suspended. In first case CPU executes thread’s code. In second case the thread stays ready, but some other ready thread is currently executed by CPU. A TCB contains no information about whether the thread is suspended or running.

There are four queues in the L4 kernel, namely present queue, ready queue, wakeup queue and send queue. The present queue contains all threads which were created until current moment. The send queue is used only by IPC mechanism. We are interested only in ready queue and wakeup queue, since scheduler does not deal with the other two. Field queue_state indicates the presence of a thread in kernel queues. Only the four last bits of the field queue_state are in use. These bits have the following names: TS_QUEUE_READY, TS_QUEUE_PRESENT, TS_QUEUE_WAKEUP and TS_QUEUE_SEND. Bit TS_QUEUE_READY is set if and only if ready queue is containing the thread. The same rule holds for bits TS_QUEUE_PRESENT, TS_QUEUE_WAKEUP and TS_QUEUE_SEND. They describe the thread presence in present queue, wakeup queue and send queue respectively. The bits are used by functions which add a thread into particular queue or remove it from queue. For example: function which removes a thread from ready queue tests the bit TS_QUEUE_READY to check whether the thread is belonging to ready queue or not. If the thread is not in ready queue, then function makes nothing. If the thread is in ready queue, then the function clears the TS_QUEUE_READY bit and removes the thread from ready queue.

Field priority indicates the current priority of a thread. It is important to note that kernel itself does not change the priority of threads. Priority of a thread can be changed only by chief of the clan which the thread belongs to. There is a restriction on priority changing: chief cannot set the priority of any thread in its clan greater than its own priority.

The scheduler periodically compares value of the field absolute_timeout of each thread in wakeup queue with current system time. The field absolute_timeout of particular thread specifies the time, when this thread should become ready and be added into the ready queue. See the description of the function parse_wakeup_queue in the section 4.3.2.2 for more details.
If a thread has been scheduled, it will be executed for some particular time, then it will be suspended and next ready thread will be scheduled. Field `timeslice` specifies the amount of the time for which thread will be executed. This field is of type integer. It is interpreted as number of microseconds. Value of this field can be varying between individual threads of a task. Note, that a thread's timeslice is in no way determined by its priority. It is valid for threads of the same priority to have different timeslice lengths. Only threads with positive value of the field `timeslice` are allowed to be executed and hence, to be in ready queue. This field is set during creation of a thread to the value of the integer constant DEFAULT_TIMESLICE and stays the same until it will be changed by chief of the thread. Its value is interpreted as number of microseconds. In the given implementation of L4 the value of DEFAULT_TIMESLICE is equal to 10240.

The value of the field `current_timeslice` specifies how long the currently executed thread should be still executed since the current moment of time. When the scheduler has just selected a thread to execute, the value of the `current_timeslice` is equal to the value of `timeslice`. Each time the timer interrupt occurs the scheduler decreases the value of field `current_timeslice` by value of TIME_QUANTUM. TIME_QUANTUM is an integer constant. Its value is interpreted as length of the time period between two successive timer interrupts in microseconds. When the value of `current_timeslice` becomes negative or equal to zero (it means, that currently executed thread should be suspended and the next thread should get a control), the handler assigns to `current_timeslice` the value of the field `timeslice`, and initiates a suspension of the thread and a search for a next one. We will use the term “scheduling event” to denote a time when `current_timeslice` of currently executed thread has just expired and the scheduler starts to search for the next thread.

4.2.2 Organization of system time in L4

The system time is organized as follows: kernel information page (a certain area in the kernel memory space represented by an element of `struct kernel_info_page_t`) contains a 64-bit field `qword_t clock`. A value of this field is interpreted as number of microseconds since the time that L4 has been booted. Initially this field contains zero. The handler of the timer interrupt increases the field `clock` by value TIME_QUANTUM
every time it is invoked. In the given implementation of the kernel the value of TIMES_QUANTUM is equal to 2048.

4.2.3 Implementation of the idle thread

The TCB of the idle thread is represented by static variable tcb_t __idle_thread. Pointer to the idle thread is accessible through the call of the function tcb_t* get_idle_tcb(). Variable __idle_thread belongs to kernel memory space, it gets initial value during system initialization.

4.2.4 Implementation of the ready queue

Ready queue consists of array of pointers to TCB and set of round-robin queues. The array is represented in code by element tcb_t* prio_queue[256].

Size of prio_queue is equal to number of levels of priority, it has valid indices from 0 to 255. If there are no ready threads with priority i, then i-th element of prio_queue is equal to NULL. Otherwise the element of prio_queue with index i points to some element of a non-empty round-robin queue. All elements of this queue have priority i. An element of a particular round-robin queue to which points the element of prio_queue is called head of this round-robin queue. TCBs in queues with greater index have greater priority, while members of the round-robin queue pointed by element of prio_queue with index zero have the minimal priority. An example of ready queue is depicted in the Figure 4.3.

The kernel contains an integer variable current_max_prio. At any time current_max_prio is equal to the index of a non-empty queue with the maximal priority (see Figure 4.3). If there is no ready TCB at all, then current_max_prio is equal to -1 and idle thread is executed.

The kernel contains a pointer to the TCB of the currently executed thread. This pointer is represented in code by element tcb_t* current. It is changed whenever the scheduler suspends currently executed thread and runs the next one.
In the Figure 4.3 we can see, that there are no threads with priority 254 and an element of \textit{prio\_queue} with corresponding index value of NULL. There are three threads with priority 255. TCBs of these threads are joined into round-robin queue and element of \textit{prio\_queue} with index 255 points to one of elements of round-robin queue named \textit{head}. In the Figure 4.3 the variable \textit{current\_max\_prio} is equal to 255. Round-robin queue associated with priority level 0 contains only one TCB, its successor and predecessor are the same as the TCB itself.

4.2.5 Implementation of the wakeup queue

As mentioned above each thread in wakeup queue has associated value of time at which it should be woken up. When we say that a thread was woken up it means that thread became ready and was moved into the corresponding round-robin queue in the
ready queue with respect to its priority. Wakeup queue is implemented as round-robin queue. TCBs are joined into wakeup queue by using the fields `wakeup_prev` and `wakeup_next`. The idle thread is a permanent head of this queue, so there is no need to have a special pointer to wakeup queue. Of course, idle thread is never “asleep” and can not be waked up. Organization of the wakeup queue is illustrated in the figure 4.4.

![Figure 4.4 – Organisation of the wakeup queue (extended)](image)

4.3 Functions used by the scheduler

The functions used by the scheduler can be divided into two groups. The first is formed by auxiliary functions, which have no independent meaning. The second group of functions implements concepts of scheduling.

4.3.1 Auxiliary functions used by the scheduler

4.3.1.1 `void switch_to_idle(tcb_t* current)`

This function is processor-dependent. It performs switching to idle thread by changing the content of the hardware registers. It receives as parameter the pointer to the TCB but this pointer is not used in the current implementation of the function.
4.3.1.2 \textbf{void} switch\_to\_thread(tcb\_t* \textit{next}, tcb\_t* \textit{current})

This function is also processor-dependent. It performs switching to the thread pointed by \textit{next} by changing content of the hardware registers. The second parameter is not used in current implementation of the function.

4.3.1.3 \textbf{void} thread\_enqueue\_ready(tcb\_t* \textit{t})

This function enqueues given thread \textit{t} into the ready queue. If transferred thread is not in ready queue, the function places it into the tail of a corresponding queue. If the queue associated with corresponding level of priority is empty at the moment, then the added tcb constitutes this queue. The function also increases the \textit{current\_max\_prio} if the priority of added thread is greater than \textit{current\_max\_prio}. Note: it is the only function which is able to increase \textit{current\_max\_prio}.

4.3.1.4 \textbf{void} thread\_dequeue\_ready(tcb\_t* \textit{tcb})

The function removes given tcb from the ready queue.

4.3.1.5 \textbf{void} dispatch\_thread(tcb\_t* \textit{t})

This function works as follows: if \textit{t} points to the TCB of currently executed thread, then it makes nothing. If \textit{t} does not points to TCB of currently executed thread, then it calls either \textit{switch\_to\_idle} function (if \textit{t} points to TCB of idle thread) or \textit{switch\_to\_thread} otherwise.

Let us denote TCB, to which \textit{t} points by TCBt. Consider a case when the TCBt belongs to ready queue, and let priority of the thread associated with TCBt be equal to \textit{P}. In this case element of \textit{prio\_queue} with index \textit{P} points to one of the elements of the round-robin queue, and TCBt is a member of this round-robin queue. The function assigns to the element of \textit{prio\_queue} with index \textit{P} value of \textit{t}, therefore the TCBt becomes head of round-robin queue. This procedure is illustrated in the Figure 4.4.
Figure 4.5 – Fragment of ready queue before (a) and after (b) dispatch_thread(t) call.

4.3.2 Basic functions

4.3.2.1 tcb_t* find_next_thread()

This function returns pointer to the TCB of a thread which should be executed next time. Such a thread should satisfy the following requirements:

- maximal priority.
- positive value of timeslice.
- ready state.

This function works as follows: it tries to find appropriate thread in round-robin queue which is associated with level of priority current_max_prio. It starts the search from the successor of the head of the queue (if the queue contains only one thread, then the successor of head is head itself). If the successor of head satisfies described above
requirements, then the function returns pointer to it. If the successor violates any requirement it is removed from round-robin queue (it means also removing from ready queue) and the function proceed with successor of head again. The queue contains finite number of threads, therefore either appropriate thread will be found or the queue becomes empty. Associated with level of priority current_max_prio queue initially contains currently executed thread. On last scheduling event the scheduler has selected this thread and run it, therefore the thread was satisfying the requirements. State or priority or timeslice of any thread can be changed by its chief at arbitrary time, therefore the queue associated with level of priority can contain no threads satisfying the requirements when scheduler searches a next thread for execution.

If the function has not found an appropriate thread in round-robin queue associated with level of priority current_max_prio, then it decreases variable current_max_prio by one and repeats the search with new value of current_max_prio. If appropriate thread was not found in the whole ready queue (it means that round-robin queues associated with all levels of priority are empty), then function returns pointer to the idle thread and assigns to current_max_prio value -1.

4.3.2.2 tcb_t* parse_wakeup_queue(dword_t current_prio, qword_t current_time)

This function is invoked from handle_timer_interrupt function with variable current_max_prio and current system time as values of input parameters. The function compares value of the field absolute_timeout of each thread which belongs to wakeup queue with value of current_time. If the current_time is greater than or equal to absolute_timeout of some thread, then the function dequeues the thread from the wakeup queue, sets the bit TSB_READY in the field thread_state of the thread’s TCB and enqueues thread into ready queue by calling thread_enqueue_ready function. The function starts from successor of the idle thread and proceed with successor of successor of idle thread, and so on for all members of wakeup queue. The function returns NULL if among the moved threads there are no one with priority greater than current_prio, otherwise it returns the pointer to the moved thread which has maximal priority among all moved threads.
4.3.2.3 **void** handle_timer_interrupt()

This function handles timer interrupts. It manages the scheduling at the high level performing several actions. As first, the function increments system time by one TIME_QUANTUM. Then it decreases value of the field `current_timeslice` in the TCB of the currently executed thread by one TIME_QUANTUM.

Then it calls `parse_wakeup_queue` function with values of `current_max_prio` and current system time as parameters. If function `parse_wakeup_queue` returns non-NULL value, then it calls `dispatch_thread` function with returned pointer as parameter. It means that a thread with priority greater than `current_max_prio` has arrived (was woken up), such thread should get control first.

If the function `parse_wakeup_queue` has returned NULL, then the function proceeds in the following way: it tests value of the field `current_timeslice` of the TCB of the currently executed thread and if this value is non-positive then the function assigns to the field `current_timeslice` value of the field `timeslice` and performs a switch to the next thread by call of `dispatch_thread(find_next_thread())`.

4.4 System calls

The interface between OS and the user programs is defined by set of functions that the OS provides. These functions are also called system calls. In this section we describe system calls for managing a process of scheduling.

4.4.1 **void** sys_schedule(schedule_param_t param, l4_threadid_t tid)

This function can be used to set a priority of a “child” thread. The `param` is a structure where we are interested only in the field `unsigned prio`. The value of this field is interpreted as integer. The field `prio` contains a new value of priority for the thread specified by `tid`. An UID of the thread can be transformed into pointer to the TCB of the thread through the call `tcb_t* tid_to_tcb(l4_threadid_t tid)`. To use this function a thread-chief should create a variable of type `schedule_param`, set the field `prio` of this variable to new value of priority and then call the function with the variable and the identifier of thread-child as parameters.
4.4.2 **void** sys_thread_switch(l4_threadtid_t tid)

This function can be used by a thread to donate its timeslice to the thread specified by tid. This system call allows to violate the normal order of scheduling. There are two restrictions on using this function. A thread cannot donate its timeslice to itself or to non-ready thread. In case of attempt to use the function with incorrect parameter, next thread will be scheduled in normal order, i.e. through the call of find_next_thread function.

4.5. Order of scheduling

Let ready queue contain some ready threads. An element of prio_queue with index current_max_prio points to the head of the round-robin queue which contains all ready threads with maximal priority. On each scheduling event, the scheduler will always select successor of head of the highest priority queue that is currently non-empty and run it. This fact follows from algorithm of the find_next_thread function.

All threads which were executed form a sequence of executed threads. To be in this sequence a thread should be picked either by the scheduler or by some running thread through the system call sys_thread_switch. The order of scheduling is called normal if the sequence is entirely formed by the scheduler, i.e. each thread in the sequence was picked by the scheduler at the same time. In this case current always points to the same thread as prio_queue[current_max_prio], this fact follows from algorithm of dispatch_thread and handle_timer_interrupt functions.

Normal order of scheduling can be changed by using the sys_thread_switch function. The thread pointed to by current differs from the “head of round-robin queue with maximal priority” only if head has donated its timeslice to another thread. The sentence “thread A has donated its timeslice to thread B” means, that thread A was to be executed and has called the sys_thread_switch function with identifier of the thread B as parameter. As result thread B has become running i.e. is the currently executed thread. In this case current points to B although A is head of round-robin queue with maximal priority.
An arbitrary algorithm of scheduling can be implemented at the user level. The “internal” or “kernel” scheduler and the function `sys_thread_switch` provide powerful framework for this purpose.

4.6 Architecture of user’s scheduler

The OS personalities built on the top of L4 can implement its own scheduling algorithm as follows: the user-level scheduler is (the only) thread running at highest priority, so it will be run by the kernel scheduler on each scheduling event. The user-level scheduler then selects the next thread it wants to run and donates its time slice to it. If there is only one thread with highest priority, then the period of time between two successive scheduling events is equal to `timeslice` of this thread.
5 Verification

The formal proof of correctness of any real system usually includes the following steps:

- formulation of the correctness criteria in some human language
- elaboration of an abstract model of the real system by means of some formal language (for instance Higher-Order Logic)
- translation of the correctness criteria into used formal language
- formal correctness proof of abstract model

Note that result proof does not validate correctness of the source code. Only correctness of abstract model can be obtained in such a way. To verify formally correctness of the source code the semantic of the programming language (C in our case) should be formally defined.

Dealing with abstract model allows to consider only important properties hiding the low level details of the real system. It is usually very useful since it allows to make the task of formal verification of large and complex system manageable.

5.1 The correctness criteria

The developers of the L4 do not provide any correctness criteria about L4 or its parts. The criteria listed below were obtained by analyzing those parts of source code which contain implementation of the scheduler.

Usually claims about a system which is verified can be divided into two classes: safety and liveness. Informally, safety means “something terrible will never occur” and liveness means “something good will eventually occur”.

The following claims state safety of the scheduler:

1. A thread with low priority will be not scheduled if there exists a runnable thread with higher priority.
2. A non-ready thread will never be scheduled.

Liveness of the scheduler is stated by following claims:

1. A period of time between two successive scheduling events is equal to value of timeslice of the currently executed thread.
2. Threads with highest priority share CPU with respect to their timeslices.
The claims stating liveness and safety of the scheduler are formulated at the highest level of abstraction. In order to prove them, we need a large number of more detailed criteria, and reasoning at a lower level of abstraction. The criteria are:

1. The current_max_prio lies within the interval 0...255.
2. In case of normal order of scheduling priority of current is equal to current_max_prio.
3. The idle thread is executed if and only if value of any entry of prio_queue is equal to NULL.
4. The idle thread is executed if and only if current_max_prio is equal to –1.
5. If during execution of thread t function parse_wakeup_queue always returns NULL (i.e. there were no threads with priority greater than priority of t woken up before t was suspended) then t will continue its execution for the period of time determined by its timeslice.
6. If thread t is the only ready thread with priority current_max_prio, then it will be executed until either it terminates or a ready thread with priority equal or greater than priority of t appears in prio_queue.
7. Let
   - the round-robin queue, which is pointed to by prio_queue[current_max_prio], contain k ready threads
   - the execution of the first thread (head) has been just started
   - sum of the values of timeslices of all k threads excepts the last one be equal to T
   - no threads with priority greater than current being woken up during T i.e.
     current_max_prio has a constant value during time T
     then the last thread will be scheduled no later than T.
8. Immediately after the call of parse_wakeup_queue the value of absolute_timeout of all threads in wakeup queue is greater than current system time.
9. Let the value returned by find_next_thread point to some thread t, then the prio_queue does not contain a thread with priority greater than priority of t.
10. A thread in wakeup queue has an expired timeout if value of the field absolute_timeout in its TCB is smaller than or equal to current system time. Let
    - wakeup queue contain some threads with expired timeout
    - thread t have maximal priority among the threads with expired timeout
among the threads in wakeup queue which have expired timeout and priority
equal to priority of \( t \), thread \( t \) be the first thread which will be woken up (see the
section 4.3.2.2 for more details)
then thread \( t \) will be scheduled no later than \( \text{TIME\_QUANTUM} \) microseconds.

11. Let a ready thread \( t \) have priority greater than \( \text{current\_max\_prio} \) and \( t \) is the only
thread which was added into ready queue (not by the call \( \text{parse\_wakeup\_queue} \)).
Then \( t \) will be scheduled no later than value of \( \text{current\_timeslice} \) of \( \text{current} \).

12. Any ready thread will not be scheduled as long as there exists at least one ready
thread with greater priority.

5.2 PVS

PVS is a verification system: an interactive environment for writing formal
specifications and checking formal proofs. It builds on nearly 20 years experience at
SRI in building verification systems, and on substantial experience with other systems.
The distinguishing feature of PVS is its synergistic integration of an expressive
specification language and powerful theorem-proving capabilities. PVS has been
applied successfully to large and difficult applications in both academic and industrial
settings.

PVS provides an expressive specification language that augments classical
higher-order logic with a sophisticated type system containing predicate subtypes and
dependent types, and with parameterized theories and a mechanism for defining abstract
data types such as lists and trees. The standard PVS types include numbers (reals,
rationals, integers, naturals, and the ordinals), records, tuples, arrays, functions, sets,
sequences, lists, and trees, etc. The combination of features in the PVS type-system is
very convenient for specification, but it makes typechecking undecidable. The PVS
typechecker copes with this undecidability by generating proof obligations for the PVS
theorem prover. This liberation from purely algorithmic typechecking allows PVS to
provide relatively simple solutions to issues that are considered difficult in some other
systems (for example, accommodating partial functions such as division within a logic
of total functions), and it allows PVS to enforce very strong checks on consistency and
other properties (such as preservation of invariants) in an entirely uniform manner.
PVS has an interactive theorem prover/proof checker. The basic deductive steps in PVS are large compared with many other systems: there are atomic commands for induction, quantifier reasoning, automatic conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision diagrams.

The distinctive feature of formal specifications is that they support formal deduction: it is possible to reduce certain questions about a formal specification to a process that resembles calculation and that can be checked by others or by machine. Thus, reviews and inspections can be supplemented by analysis of formal specifications, and those analysis can be mechanically checked.

PVS is a verification system: a specification language tightly integrated with a powerful theorem prover and other tools.

5.3 Specification of double linked list

We will use the following notations:

- A variable \( x \) of type \( T \) will be referred as \( x: T \)
- A function \( \text{func} \) which returns an element of type \( T \) and receives parameters \( \text{param}_1, \text{param}_2, \ldots \) of type \( T_1, T_2, \ldots \) respectively will be referred as \( \text{func}(\text{param}_1: T_1, \text{param}_2: T_2, \ldots): T \)

As it was described in chapter 4, round-robin queue (or double linked list) is widely used in data structure of scheduler. In this section we will formally define double linked list. We will use the identifier \( \text{ring} \) to denote a type of double linked list. The properties of the particular data structure (like list, stack, double linked list, array, etc.) are independent from type of elements (for instance integer, boolean or real) which are joined into data structure. The type of elements can be given into data structure as parameter. So, a particular instance of double linked list of elements of type \( T \) may be formally described as an element of type \( \text{ring}[T] \). The abstract data type \( \text{ring} \) is defined in PVS as follows:
ring[T:TYPE]: DATATYPE
BEGIN
    null: null?
    enq(body: ring, last: T): nonempty?
END ring

Where:
- ring is identifier of data type
- TYPE, DATATYPE, BEGIN and END are keywords used to construct the type ring
- null and enq are constructors
- null? and nonempty? are recognizers
- body and last are accessor

A constructor is used to define a way in which an element of type ring can be created. A constructor can be considered as a function which returns an element of constructed type. The type ring has two constructors:
- null is used to construct an empty double linked list, i.e. a list which contains no elements
- enq(body: ring, last: T) is used to construct an element of type ring from another element of the type ring and one element of type T, where T is a type of elements in the ring.

A recognizer is used to identify the way in which an element of the type ring was created. Each constructor has a dedicated recognizer. The recognizers null? and nonempty? are predicates over ring type that are true when their argument is constructed using the corresponding constructor. More formally, let r be an element of type ring, then the following two claims hold:
1. null?(r)=true iff r=null
2. nonempty?(r)=true iff r=enq(body_el, last_el)

where body_el is some element of type ring and last_el is some element of type T.

Given a ring element that is known to be nonempty?, the accessor body and last may be used to extract the first and the second arguments of the constructor.
5.3.1 Properties of the ring data type

To express properties of a double linked list (in C code) the additional properties of the data type ring are needed. The set of properties of ring defines which properties of a C-list we are able to model. Through the definitions of properties we will use the following notation:

- \( r, r_1 \) denote some elements of the type ring
- \( T \) denotes type of elements of ring (parameter of ring type)
- \( x, y \) denote some elements of type \( T \)
- \( n \) denotes some natural number

Note, \( r = \text{enq}(r_1, x) \) is equivalent to \( \text{last}(r) = x \) and \( \text{body}(r) = r_1 \).

The first property of the type ring we are interested in is the length of list i.e. the number of elements in some element of type ring. This property is defined recursively as follows:

\[
\text{length}(r) = \begin{cases} 
0, & \text{if } r = \text{null} \\
\text{length}(\text{body}(r)) + 1, & \text{otherwise}
\end{cases}
\]  

(5.1)

In order to know whether some \( x \) belongs to \( r \) or not, we define the predicate \( \text{member}(r, x) \) as follows:

\[
\text{member}(r, x) = \begin{cases} 
\text{false}, & \text{if } r = \text{null} \\
\text{true}, & \text{if } x = \text{last}(r) \\
\text{member}((\text{body}(r), x), & \text{otherwise}
\end{cases}
\]  

(5.2)

The constructor \( \text{enq} \) creates the exemplar of type ring independently from whether the second argument is a member of the first, i.e. a list created by \( \text{enq} \) can contain the same element more than once. The uniqueness of elements in the list is the next property we are interested in. This property is expressed by the predicate \( \text{each_unique}(r) \), which is defined recursively as follows:

\[
\text{each_unique}(r) = \begin{cases} 
\text{true}, & \text{if } r = \text{null} \\
\neg \text{member}(r_1, x) \land \text{each_unique}(r_1), & \text{if } r = \text{enq}(r_1, x)
\end{cases}
\]  

(5.3)

The mechanism of adding a new element into the ring provided by PVS works as for stack, i.e. for given \( r \) known to be nonempty? only last element may be extracted immediately by accessor last. Obviously, there is a need to associate with each element of \( r \) an index. It allows us to provide an access to n-th element of \( r \) by means of our own
accessor \( nth \), which returns an element of a nonempty \( r \) with given index. The definition of \( index \) and \( nth \) follows.

\[
index(r, x) = \begin{cases} 
length(r) - 1, & \text{if } x = \text{last}(r) \\
index(body(r), x), & \text{otherwise}
\end{cases}
\]  

where \( x \in \{i \mid \text{member}(r, i) \text{ and } i \text{ has type } T\} \).

\[
nth(r, n) = \begin{cases} 
x, & n = length(r) - 1 \\
nth(body(r), n), & \text{otherwise}
\end{cases}
\]  

where \( n \in [0, ..., length(r)-1] \).

The key property of the double linked list is that given an element of the list, its predecessor and successor may be accessed immediately. To maintain this property let us introduce natural functions \( next \) and \( prev \) in the following way:

\[
\begin{align*}
next(r, n) &= \text{mod}(n + 1, length(r) - 1) \\
prev(r, n) &= \text{mod}(n - 1, length(r) - 1)
\end{align*}
\]

where \( n \in [0, ..., length(r)-1] \) is an index of some element in \( r \). The successor and predecessor of the element may be received with a help of the \( nth \) accessor.

There is a need to have a reference to the first element of the list. We will refer the first element of a list as its \( head \), it is defined as follows:

\[
head(r) = nth(r, next(r, length(r)-1))
\]

### 5.3.2 Operations over ring data type

For us an operation is a function which receives an element of type \( ring \) and maybe some additional variables as parameters and returns an element of type \( ring \). A set of operations defines the ways in which an element of the type \( ring \) can be changed. The operations differ from constructors and accessors in the way, that an operation can be defined to change an element of \( ring \) in an arbitrary way.

The first operation which we want to specify adds an element into a list, but it is intended to keep uniqueness of elements in the list. It is defined as follows:

\[
append(r, x) = \begin{cases} 
enq(r, x), & \text{if } \neg\text{member}(r, x) \\
r, & \text{otherwise}
\end{cases}
\]

The next operation removes an element \( x \) from \( r \). If \( r \) contains \( x \) many times, then only the last instance of \( x \) in \( r \) will be removed (\( last(r) \) is tested first, \( head(r) \) is tested last). It is defined as follows:
The operation \textit{concatenate} performs concatenation of two lists. It is defined as follows:

\[
\text{concatenate}(r, l) = \begin{cases} 
  r, & \text{if } r = \text{null} \\
  \text{enq}(\text{concatenate}(r, \text{body}(l)), \text{last}(l)), & \text{otherwise}
\end{cases}
\]  

The operation \textit{shift} performs cyclic shift of the list such that the \textit{last} becomes the \textit{head} as result. The \textit{shift} is defined as follows:

\[
\text{shift}(r) = \begin{cases} 
  r, & \text{if } r = \text{null} \\
  \text{concatenate}(\text{enq}(\text{null}, \text{last}(r)), \text{body}(r)), & \text{otherwise}
\end{cases}
\]  

We extend the last operation by introducing an extra parameter of natural number type. The second parameter will denote number of elementary shifts defined by (5.10). The extended version of \textit{shift} is defined recursively as follows:

\[
\text{shift}(r, n) = \begin{cases} 
  r, & \text{if } r = \text{null} \lor n = 0 \\
  \text{shift}(r), & \text{if } n = 1 \\
  \text{shift}(\text{shift}(r, n-1)), & \text{otherwise}
\end{cases}
\]  

Note that the first case in definition (5.11) was introduced only to extend a set of valid values of second parameter by zero. The termination condition of recursion is defined by second case.

The last operation we need for \textit{ring} is the \textit{move\_head} operation. It changes the received element of type \textit{ring} such that its n-th element becomes the head, where n is value of second parameter. The operation is defined as follows:

\[
\text{move\_head}(r, n) = \begin{cases} 
  r, & \text{if } r = \text{null} \\
  \text{shift}(r, \text{mod}(\text{length}(r) - n, \text{length}(r))), & \text{otherwise}
\end{cases}
\]  

5.3.3 Lemmas stating correctness of specification of the double linked list.

In order to be sure that formally defined properties meet their intended meaning, some amount of lemmas for each defined property is required. The number of lemmas for data type \textit{ring} is about 50. In this chapter we describe only some of them, to give an idea. The complete listing of lemmas can be seen in Appendix XX. All lemmas are proved in PVS.
To explain meaning of lemmas we will use the same notation for $r$, $y$ and $x$ as in the previous section.

Lemma **append_uniq**. If all elements of double linked list $r$ are unique then all elements of $append(r, x)$ are unique, in other hands operation $append$ preserve uniqueness of elements in the element of type $ring$. This lemma is formulated in PVS as follows:

```plaintext
append_uniq: LEMMA
    each_unique(r) IMPLIES each_unique(append(r, x))
```

where **LEMMA** and **IMPLIES** are keywords, $r$ is a some element of type $ring$ and **append_uniq** is an identifier of the lemma.

Lemma **deque_mem**. Let double linked list $r$ be nonempty and $x$ be a member of $r$. If all elements in $r$ are unique, then $x$ is not a member of $deque(r, x)$. This lemma is formulated in PVS as follows:

```plaintext
deque_mem: LEMMA
    FORALL (r:(nonempty?), x:{k:T| member(r,k)}):
        each_unique(r) IMPLIES not member(deque(r, x),x)
```

Lemma **deque_mem2**. If $x, y$ are members of $r$ and $x \neq y$, then $y$ is a member of the $deque(r, x)$. This lemma is formulated in PVS as follows:

```plaintext
deque_mem2: LEMMA
    member(r,x) AND member(r,y) AND x/=y
    IMPLIES member(deque(r, x), y)
```

Lemma **deque_mem3**. If $y$ is a member of $deque(r, x)$, then $y$ is a member of $r$. This lemma is formulated in PVS as follows:

```plaintext
deque_mem3: LEMMA
    member(deque(r,x),y) IMPLIES member(r,y)
```

5.4 Formal specification of the *struct tcb_t*

We need an abstract model of the thread to specify formally the scheduler data structure. We will use PVS syntax for specification in this section (see [19] for description of the PVS syntax).

The type *l4_threadid* is defined to represent an UID of threads. We do not need any details about implementation of *l4_threadid* in C code. Consequently, we model UIDs as elements of non-empty type:
The only assumption made on this type is that it is disjoint from all other types.

Since in our work we only need to distinguish the idle thread from the other threads, we define the constant IDLE_ID of type \textit{l4\_threadid}. This constant will represent the UID of the idle thread.

\textbf{IDLE\_ID: \textit{l4\_threadid}}

The C type \textit{struct \_tcb\_t} is modeled by PVS record \textit{tcb}. Any field of an element of record type can be accessed by associated name. The \textit{tcb} is defined as follows:

\begin{verbatim}
tcb:TYPE=[# myself: \textit{l4\_threadid},
        thread_state:[# TSB\_READY:bool  #],
        queue_state:[# TS\_QUEUE\_WAKEUP: bool,
                     TS\_QUEUE\_READY: bool #],
        priority:  {i:nat| i < 256},
        timeslice:   int,
        current_timeslice: int,
        absolute_timeout: below[2^64]  #]
\end{verbatim}

The fields of the \textit{tcb} type has the following meaning:

- \textit{myself} represents an UID of a thread
- \textit{thread\_state} is of record type. The scheduler can distinguish only between ready and non-ready threads, therefore only one field TSB\_READY of boolean type is needed to describe a state of a thread
- \textit{queue\_state} is of record type. Fields TS\_QUEUE\_WAKEUP and TS\_QUEUE\_READY represent fields of \textit{struct \_tcb\_t} of the same name, i.e. the field is true if a thread belongs to correspondent queue
- \textit{priority} represents a priority of the thread, its type is a set of natural numbers from 0 to 255
- \textit{timeslice} represents the field of the same name of \textit{struct \_tcb\_t}, it is of integer type
- \textit{current\_timeslice} represents the field of the same name of \textit{struct \_tcb\_t}, it is of integer type
- \textit{absolute\_timeout} represents the field of the same name of \textit{struct \_tcb\_t}, its type is a set of all natural numbers from 0 to \(2^{64}-1\). The values of the bounds
are chosen accordingly to declaration of the field in C code as \texttt{qword_t absolute_timeout}

To specify the fact that threads are joined into round-robin queue and to model access to the elements of this queue we can use the type \texttt{ring} (described in previous section). Therefore, in PVS type \texttt{tcb} we do not need to have a representation of the fields of \texttt{struct tcb_t} which are used to join TCBs into queues.

Some functions used by scheduler returns a pointer to TCB as result of their work. In some cases a NULL pointer may be returned. Therefore we need a formal representation of the object which has type \texttt{tcb_t} and value of NULL. This is done by introducing special constant \texttt{empty_tcb} of type \texttt{tcb}. The only usable property of this constant is that it differs from all over objects of the same type. Uniqueness of \texttt{empty_tcb} is ensured by introducing constant \texttt{ERROR_ID} of type \texttt{l4_threadid}.

We define \texttt{empty_tcb} as follow:

\begin{verbatim}
ERROR_ID: l4_threadid
empty_tcb: tcb=(#
    myself:= ERROR_ID,
    % parameters below are never used,
    % but we cannot leave them undefined
    thread_state:=(# TSB_READY := true   #),
    queue_state:=(# TS_QUEUE_WAKEUP := true,
                  TS_QUEUE_READY :=true   #),
    priority := 0,
    timeslice := DEFAULT_TIMESLICE,
    current_timeslice := 0,
    absolute_timeout := 2^64-1     #)
\end{verbatim}

The formal definition of ready thread can be given basing on the formal specification of the TCB. This is done by introducing the predicate \texttt{is_ready?} which evaluates to true if a thread has positive value of the field \texttt{timeslice} and set bit TSB READY in the field \texttt{thread_state}. The predicate is defined as follows:

\begin{verbatim}
is_ready?(t:tcb): bool =
    t`thread_state`TSB_READY AND (t`timeslice>0)
\end{verbatim}
To build a formal model of scheduler we need a formal description of the idle thread. TCB of the idle thread is represented by constant *idle* of type *tcb*. A constant of record type in PVS is defined by assigning a particular value to each field. Indeed, the values of all fields of *idle* are not used, except for the field *myself* which is used to identify *idle*.

\[
\text{idle}: \text{tcb} = (\#
\text{myself} := \text{IDLE\_ID},
\text{thread\_state} := (\# \text{TS\_READY} := \text{true} \ #),
\text{queue\_state} := (\# \text{TS\_QUEUE\_WAKEUP} := \text{true},
\text{TS\_QUEUE\_READY} := \text{true} \ #),
\text{priority} := 0,
\text{timeslice} := \text{DEFAULT\_TIMESLICE},
\text{current\_timeslice} := 0,
\text{absolute\_timeout} := 2^{64}-1 
\)
\]

### 5.5 Formal specification of the ready queue

The formal model of the ready queue is build on the base of the formal model of the double linked list. Since we have defined types *ring* and *tcb*, we can easily define double linked list of TCB as *ring[tcb]*. An instance of ready queue will be formally described by element of type *prio\_queue*. Type *prio\_queue* is defined as follows:

\[
prio\_queue: \text{TYPE} = [\#
\text{current\_max\_prio}: \{i: \text{int}|i<256 \ \text{AND} \ i>=-1\},
\text{elements}: \text{ARRAY}[\{i: \text{int}|i<256 \ \text{AND} \ i>=-1\} \rightarrow \text{ring[tcb]}] \ #]
\]

where:

- The field *current\_max\_prio* represents the priority of the ready thread with maximal priority. Valid values of this field are integer numbers from –1 to 255. For convenience it has been defined as a part of the formal model of ready queue.
- The field *elements* is an array of elements of type *ring[tcb]* with the valid indices in the range [-1, ..., 255]. The array *elements* can also be considered as a function with domain “set of integer numbers from –1 to 255” and range *ring[tcb]*.
Note that C array *prio_queue* is a part of ready queue (it is presented in section 4.2.4) and type *prio_queue* is the formal model of whole ready queue in PVS.

The case of absence of ready threads is considered to be a special case of scheduling. Switching to the idle thread is managed in nonuniform way. However, we are not interested to model these details. To simplify the formal model of the scheduler we have defined the range of possible indices of *elements* with extra (vs. implemented in C code array *prio_queue*) index -1. The *idle_queue* formally describes virtual round-robin queue, which consists only of one idle thread (really, in C code priority level -1 is undefined and ready queue does not contain the idle thread). In the array *elements* *idle_queue* has index -1. It is a constant of type *ring[tcb]* and defined as follows:

\[
\text{idle\_queue: ring[tcb]}=\text{enq(null, idle)}
\]

As we see, *idle_queue* is constructed from empty element of type *ring* and *idle* by using constructor *enq*.

Since *head* is defined only for such objects of type *ring* which satisfy the predicate *nonempty?*, we need some function with range *tcb* and domain *ring[tcb]* which will return *head* or *empty_tcb* depending on particular value of the object. This function, named *current_thread*, is defined as follows:

\[
\text{current\_thread(r: ring[tcb])}: \text{tcb} =
\begin{cases}
  \text{head(r)} & \text{if nonempty?(r)} \\
  \text{empty_tcb} & \text{else}
\end{cases}
\]

The *current* (pointer to the TCB of currently executed thread) in case of normal order of scheduling points to the head of round-robin queue which is associated with priority level *current\_max\_prio*. Since we have a formal representation for ready queue and variable *current\_max\_prio* we can formally describe *current* by PVS function *current* as follows:

\[
\text{current(pq: prio\_queue)}: \text{tcb} =
\text{current\_thread(pq\_elements(pq\_current\_max\_prio))}
\]

5.5.1 Formal description of the initial state of ready queue

The state of ready queue is identified by the value of *current\_max\_prio* and content of all round-robin queues.
Let us consider the initial state of scheduler data structure, i.e. state at the time when L4 has just finished initialization. At this time there exist only the idle thread, thus all elements of the C array \textit{prio\_queue} of ready queue are equal to NULL and the value of \textit{current\_max\_prio} is equal to -1. To represent this state of ready queue we define a constant \textit{empty\_prio\_queue} of type \textit{prio\_queue} as follows:

\begin{verbatim}
empty_prio_queue: prio_queue= (#
current_max_prio:=-1,
elements:=(LAMBDA(index:{i:int|i<256 AND i>=-1}): IF index=-1 THEN idle_queue ELSE null ENDIF)
#)
\end{verbatim}

5.5.2 Operations over the formal model of ready queue

In this section we will specify two functions used by the scheduler: \textit{thread\_enqueue\_ready} and \textit{thread\_dequeue\_ready} (see sections 4.3.1.3 and 4.3.1.4 for details).

With the help of an auxiliary predicate \textit{member} we can establish whether a thread belongs to ready queue or not. For given \textit{t} of type \textit{tcb} and \textit{pq} of type \textit{prio\_queue} it evaluates to true if and only if \textit{t} belongs to some element of the array \textit{elements} of \textit{pq}. It is defined as follows:

\begin{verbatim}
member(pq: prio_queue, t:tcb):bool=
   IF FORALL(index:{i:int|i<256 AND i>=-1}): NOT member(elements(pq)(index), t)
   THEN false
   ELSE true
ENDIF
\end{verbatim}

Note that in body of this definition is used predicate \textit{member(r: ring[tcb], t: tcb)} which determines whether the thread belongs to the round-robin queue, its definition can be found in section 5.3.1.

The C function \textit{thread\_enqueue\_ready} is represented in PVS by function \textit{enqueue\_ready}. It updates ready queue by adding a new ready thread in correspondent round-robin queue. The first input argument \textit{old} describes initial state of ready queue.
The state of ready queue would be not changed if the thread described by the second argument \( t \) is not ready or it belongs already to \( old \). The function is defined as follows:

\[
\begin{align*}
\text{enqueue\_ready}(\text{old:prio\_queue}, \text{t:tcb}): \text{prio\_queue} &= \text{IF member(\text{old, t}) OR (NOT is\_ready?(t)) THEN old ELSIF old`current\_max\_prio < t`priority THEN old WITH [`current\_max\_prio:=t`priority,`elements(t`priority):=append(old`elements(t`priority), t)] ELSE old WITH [`elements(t`priority):=append(old`elements(t`priority), t)] ENDIF}
\end{align*}
\]

The C function \textit{thread\_dequeue\_ready} is represented by function \textit{dequeue\_ready}. It removes the given thread from ready queue. The ready queue would not be changed by the function if the given thread does not belong to ready queue. We model the function as follows: the state of ready queue before application of the operation is described by the first argument \( old \), the second argument \( t \) describes the thread which will be removed. If predicate \textit{member} (it determines whether the thread belongs to ready queue) evaluates to false, then the unchanged value of \( old \) is returned.

\[
\begin{align*}
\text{dequeue\_ready}(\text{old:prio\_queue}, \text{t:tcb}): \text{prio\_queue} &= \text{IF NOT member(\text{old, t}) THEN old ELSE old WITH [`elements(t`priority):=deque(old`elements(t`priority), t)] ENDIF}
\end{align*}
\]

5.5.3 Formal specification of properties of ready queue

To state some property of ready queue we define a predicate which evaluates to true if and only if the correspondent property holds. The predicates which we introduce in this section state the following properties of ready queue:

1. Uniqueness of any thread in ready queue (there do not exist two or more threads with the same UID in ready queue).
2. Correct placing of threads in round-robin queues (i.e. according to their priority).
3. Correctness of the value of \textit{current\_max\_prio} (the value is correct if there exist no ready threads with priority greater than \textit{current\_max\_prio} and there exist at least one ready thread with priority equal to \textit{current\_max\_prio})

4. Absence of non-ready threads in ready queue

Let us introduce an auxiliary predicate \textit{no\_common}? which will be used in definition of another predicate stating the property of uniqueness of any thread in ready queue. Let $r1$ and $r2$ be two variables of type \textit{ring[tcb]}. Predicate \textit{no\_common}? evaluates to true if and only if two round-robin queues, which are represented by the value of input parameters $r1$ and $r2$, are completely disjoint, i.e. any member of the first queue does not belong to the second queue. It is defined as follows:

\[
\text{no\_common?}(r1:\text{ring[tcb]}, r2:\text{ring[tcb]}): \text{bool} = \\
\text{IF FORALL}(t:\{x:\text{tcb}\mid \text{member}(r1,x)}): \\
\quad \text{NOT member}(r2, t) \text{ THEN true} \\
\text{ELSE false} \\
\text{ENDIF}
\]

The predicate stating uniqueness of all threads in ready queue can be specified now with the help of predicate \textit{no\_common}?. The predicate \textit{each\_unique} evaluates to true if and only if all threads in ready queue (represented by the input parameter $pq$) are unique. The predicate is defined as follows:

\[
\text{each\_unique}(pq:\text{prio\_queue}): \text{bool} = \\
\text{FORALL}(\text{index}:\{i:\text{int}\mid i<256 \text{ AND } i>=-1\}): \\
\quad \text{each\_unique}(pq`\text{elements(index)}) \text{ AND} \\
\quad \text{FORALL}(k:\{i:\text{int}\mid i<\text{index} \text{ AND } i>=-1\}): \\
\quad \text{no\_common?}(pq`\text{elements(index)}, pq`\text{elements(k)})
\]

As it was described in previous chapter, priority of a ready thread determines to which particular round-robin queue within ready queue the thread belongs to. The predicate \textit{correct\_placing} states that:

1. Any thread belonging to ready queue (represented by the input parameter $pq$) has been placed into round-robin queue according to its priority, i.e. round-robin queue with index $i$ contains only threads with priority level $i$, where $i \in [0..255]$. 

2. The virtual round-robin queue associated with priority level –1 solely consists of the idle thread.

The predicate is defined as follows:

\[
\text{correct\_placing} (pq: \text{prio\_queue}): \text{bool} = \\
(\text{FORALL}(\text{index}: \{i: \text{nat} | i<256\}, \\
  k: \{i: \text{nat} | i<\text{length}(pq`\text{elements(index))}\}): \\
  \text{nth}(pq`\text{elements(index)}, k)`\text{priority} = \text{index} ) \text{ AND} \\
pq`\text{elements}(-1)=\text{idle\_queue}
\]

To state the readiness of all threads in ready queue we introduce predicate \textit{correct\_elements}. It evaluates to true if and only if all threads in ready queue (represented by the value of input parameter \(pq\)) are ready. The readiness of a single thread is stated by predicate \textit{is\_ready?}. Predicate \textit{correct\_elements} is defined as follows:

\[
\text{correct\_elements} (pq: \text{prio\_queue}): \text{bool} = \\
\text{FORALL}(\text{index}: \{i: \text{int} | i>=-1 \text{ AND } i<256\}, \\
  k: \{i: \text{nat} | i<\text{length}(pq`\text{elements(index))}\}): \\
  \text{is\_ready?}(\text{nth}(pq`\text{elements(index)}, k))
\]

Correctness of the value of \textit{current\_max\_prio} is stated by predicate \textit{correct\_cmp}. It evaluates to true if and only if ready queue and \textit{current\_max\_prio} satisfy the following two claims:

1. Ready queue does not contain ready threads with priority greater than \textit{current\_max\_prio}.
2. The round-robin queue associated with priority level \textit{current\_max\_prio} is nonempty.

The predicate is defined as follows:

\[
\text{correct\_cmp} (pq: \text{prio\_queue}): \text{bool} = \\
(\text{FORALL}(k: \{i: \text{nat} | i<256 \text{ AND } i>pq`\text{current\_max\_prio}\}): \\
  \text{null?}(pq`\text{elements(k)}) \text{ AND} \\
  \text{nonempty?}(pq`\text{elements}(pq`\text{current\_max\_prio}))
\]
We are also interested in the weak version of the predicate described above. Predicate `correct_cmp_weak` states only the first claim. This predicate is defined as follows:

```plaintext
correct_cmp_weak(pq: prio_queue): bool =
(FORALL(k:{i:nat|i<256 AND i>pq`current_max_prio}) :
null?(pq`elements(k)))
```

5.5.4 Ready queue correctness proof sketch

Let a reachable state of ready queue be a state which can be reached from the initial state of ready queue by applying of operations `enqueue_ready` and `dequeue_ready`. In this section we introduce formal definition of a correct state of ready queue (or just correct ready queue). After this we describe how we can prove that any reachable state of ready queue is correct.

Ready queue is correct if it satisfies the predicates `each_unique`, `correct_placing`, `correct_cmp` and `correct_elements` presented in previous section.

Let us introduce some notation. Let there be an instance of ready queue RQ which satisfies property stated by predicate P, this fact is denoted as P(RQ). We will say that “operation OP preserves a property of RQ stated by P” if the claim “P(RQ) implies P(OP(RQ))” holds.

Proof of correctness of any reachable state of ready queue is based on induction scheme. The first step is to prove induction base, i.e. that formal model of ready queue in initial state satisfies the predicates. State of ready queue can be changed only by the operations described in section 5.5.2. In induction step we have to prove that operations `dequeue_ready` and `enqueue_ready` preserve all wanted properties, i.e. that any state of ready queue obtained by application of the operation to correct ready queue is also correct. We should prove that the operations preserve all wanted properties.

Such a strategy does not ensures correctness of any instance of ready queue, but we are interested only in states obtained by application of two particular operations to initial state.
5.5.5 Lemmas about operations over ready queue

The four lemmas listed below state the correctness of initial state of ready queue. Each lemma states that the initial state of ready queue satisfies the particular predicate stating some property.

- **empty_each_unique**: LEMMA each_unique(empty_prio_queue)
- **empty_correct_cmp**: LEMMA correct_cmp(empty_prio_queue)
- **empty_correct_placing**: LEMMA correct_placing(empty_prio_queue)
- **empty_correct_elements**: LEMMA correct_elements(empty_prio_queue)

Four lemmas are required for each operation to prove that it preserves the wanted properties. The following four lemmas state that operation $enqueue_{ready}$ preserves the properties of ready queue stated by predicates $each\_unique$, $correct\_placing$, $correct\_cmp$ and $correct\_elements$.

- **enqueue_unique**: LEMMA each_unique(pq) IMPLIES each_unique(enqueue_ready(pq, t))
- **enqueue_placing**: LEMMA correct_placing(pq) AND t \`myself/=IDLE_ID IMPLIES correct_placing(enqueue_ready(pq, t))
- **enqueue_elements**: LEMMA correct_elements(pq) IMPLIES correct_elements(enqueue_ready(pq, t))
- **enqueue_cmp**: LEMMA correct_cmp(pq) IMPLIES correct_cmp(enqueue_ready(pq, t))

The next three lemmas state that operation $dequeue_{ready}$ preserves the properties of ready queue stated by predicates $each\_unique$, $correct\_placing$ and $correct\_elements$.

- **dequeue_unique**: LEMMA each_unique(pq) IMPLIES each_unique(dequeue_ready(pq, t))
- **dequeue_placing**: LEMMA correct_placing(pq) AND t \`myself/=IDLE_ID IMPLIES correct_placing(dequeue_ready(pq, t))
- **dequeue_elements**: LEMMA correct_elements(pq) IMPLIES correct_elements(dequeue_ready(pq, t))
The operation does not preserve the property stated by predicate \textit{correct\_cmp}. The reason is that if the round-robin queue associated with priority level \textit{current\_max\_prio} contains only one thread and this thread should be removed by the function \textit{thread\_dequeue\_ready}, then after removing of the thread the round-robin queue associated with priority level \textit{current\_max\_prio} will be empty. This fact violates the second requirement of predicate \textit{correct\_cmp}. However, the lemma which states that if ready queue initially satisfies \textit{correct\_cmp}, then after removing of any thread it satisfies \textit{correct\_cmp\_weak} can be proved. This lemma is formulated as follows:

\textbf{dequeue\_cmp1: LEMMA} \textit{correct\_cmp}(pq) IMPLIES \textit{correct\_cmp\_weak}(\text{dequeue\_ready}(pq, t))

In C code function \textit{thread\_dequeue\_ready} does not update the value of \textit{current\_max\_prio} (it is done by function \textit{find\_next\_thread}). Therefore, we will be able to formulate necessary lemmas about the value of \textit{current\_max\_prio} only having a formal specification of \textit{find\_next\_thread} function.

To be sure that operations were specified correctly, i.e. according to their intended meaning we formulate and prove the following four lemmas. Two of these lemmas informally can be called “positive” and other two “negative”. “Positive” lemmas state validity of some proposition whereas “negative” state invalidity.

An arbitrary thread $t$ can be either ready or non-ready. Independently from its readiness the thread either belongs to ready queue or not. The following lemma states that when function \textit{thread\_enqueue\_ready} has been terminated, the thread $t$ given to the function as parameter does not belong to ready queue if it was non-ready and it was not a member of ready queue before invocation of the function. This can be formulated as following “positive” lemma:

\textbf{enqueue\_member: LEMMA} \textit{member}(enqueue\_ready(pq,t),t) or \textit{(not member}(pq,t) AND \textit{not is\_ready?}(t)))

Roughly speaking lemma states that the function enqueues ready threads into ready queue.

Let some thread $t$ belong to ready queue. Then after calling of \textit{thread\_enqueue\_ready} with any thread as parameter thread $t$, will still belong to ready queue. This is stated by the following lemma:

\textbf{enqueue\_member\_safe: LEMMA} \textit{member}(pq,t) IMPLIES \textit{member}(enqueue\_ready(pq,t1),t)
Actually this is a “negative” lemma, it states that the function does not remove threads from ready queue.

Let the function \textit{ready_dequeue_thread} to be invoked with thread \( t \) as parameter. If
- threads in ready queue are unique
- \( t \) differs from the idle thread
- threads in ready queue are placed correctly,

then \( t \) does not belong to ready queue after invokation of the function. This is stated by following “positive” lemma:

\[
\text{dequeue_member: LEMMA } t \neq \text{IDLE_ID} \\
\text{and each_unique(pq)} \\
\text{and correct_placing(pq)} \\
\text{IMPLIES not member(dequeue_ready(pq,t),t)}
\]

Obviously, the function should remove exactly the thread given as parameter and should not remove any other thread. This claim is stated by following “negative” lemma:

\[
\text{dequeue_member_safe: LEMMA member(pq,t) and } t \neq t_1 \\
\text{IMPLIES member(dequeue_ready(pq,t_1),t)}
\]

All lemmas described in this section have been proved in PVS.\(^1\)

5.6 Formal specification of \textit{find_next_thread} function

In this section we will specify formally \textit{find_next_thread} function – one of the three basic scheduler functions.

At first we introduce an auxiliary predicate \textit{contains_ready?} which evaluates to false if and only if a round-robin queue represented by input parameter \( r \) does not contain ready threads. It is defined as follows:

\[
\text{contains_ready?}(r: \text{ring[tcb]}): \text{RECURSIVE bool}= \\
\text{CASES } r \text{ OF} \\
\text{null: } \text{false},
\]

\(^1\) The proof of presented lemmas rely on some auxiliary lemmas. See appendix A for complete list of lemmas.
enq(bod, last):
    IF is_ready?(last) THEN true
    ELSE contains_ready?(bod) ENDIF
ENDCASES
MEASURE length(r)

Let us consider algorithm of \textit{find_next_thread}. We can easily realize, that the function would be decomposed into several parts. One of the parts can be considered as a subfunction which looks through the elements in the round-robin queue given as parameter for any ready thread. It starts the search from the successor of \textit{head} and removes from the round-robin queue all non-ready threads which were met during the search. If the appropriate thread was found, then the subfunction returns pointer to this thread. Otherwise NULL is returned (it means also that all threads were removed and round-robin queue is empty).

We specify this subfunction as a PVS function \textit{next_ring}. The C-prototype of \textit{next_ring} (the subfunction) gets a round-robin queue as input parameter and returns a pointer to thread (now we are only interested in types of input and output parameters). We define \textit{next_ring} as function with domain ring[tcb] (it corresponds to round-robin queues). Since \textit{next_ring} deals with models of round-robin queues of arbitrary length, using of recursion is the most simple method. Since definition of \textit{next_ring} is recursive, its range is the same as its domain (ring[tcb]). Thus, some convention is required to specify returned value. As it was described in the section 4.3.2.1, the value returned by function \textit{find_next_thread} will be eventually transferred into \textit{dispatch_thread} function as parameter. Function \textit{dispatch_thread} performs such permutation in ready queue, that the thread transferred as parameter becomes \textit{head} of round-robin queue which it belongs to. We specify returned value as follows:

1. If the appropriate thread has been found, then it will be the \textit{head} of returned round-robin queue.
2. Otherwise descriptor of empty round-robin queue is returned (i.e. element of type ring[tcb] constructed by \textit{null} constructor)

The function \textit{next_ring} uses operation \textit{move_head} to specify the permutation of \textit{head}. Function \textit{next_ring} is defined as follows:

\[
\text{next\_ring}(r: \text{ring}[tcb]) : \text{RECURSIVE ring}[tcb] =
\]
IF length(r)=0 THEN r
ELSIF is_ready?(nth(r,next(r,0))) THEN move_head(r,1)
ELSE next_ring(deque(r,nth(r,next(r,0))))
ENDIF
MEASURE length(r)

Now we can specify function \textit{find_next_thread} using \textit{next_ring} and \textit{is_ready?}. During the search of a next thread function \textit{find_next_thread} decreases \textit{current_max_prio} if the round-robin queue associated with this (equal to value of \textit{current_max_prio}) level of priority becomes empty. As result of analyzing \textit{find_next_thread} we conclude, that using of recursion on \textit{current_max_prio} is the most suitable variant of specification. The value of -1 will correspond to case satisfying termination condition since the virtual round-robin queue associated with priority level -1 contains the always-ready idle thread.

Function \textit{find_next_thread} is formally specified as recursive function \textit{find_next}. The next thread for execution is specified as a head of the round-robin queue (within ready queue represented by returned value of \textit{find_next}) associated with priority level \textit{current_max_prio}. The \textit{find_next} is defined as follows:

\texttt{find_next}(pq): \textbf{RECURSIVE} prio_queue =
\hspace{1cm} \text{LET} cmp=pq`current_max_prio IN
\hspace{1cm} \text{IF} cmp=-1 OR contains_ready?(pq`elements(cmp))\text{THEN}
\hspace{1cm} \hspace{1cm} pq WITH [`elements(cmp):=next_ring(pq`elements(cmp))] 
\hspace{1cm} \text{ELSE} find_next(pq WITH 
\hspace{2cm} [`elements(cmp):=null,`current_max_prio:=cmp-1])
\hspace{1cm} ENDIF
\hspace{1cm} MEASURE pq`current_max_prio+1

5.6.1 Lemmas concerning \textit{find_next_thread} function

The lemmas described in this section state that formal specification of \textit{find_next_thread} function meets its intended semantic.

Lemma \textbf{fn_placing} states that function \textit{find_next} preserves property of ready queue stated by predicate \textit{correct_placing}. It is formulated as follows:

\texttt{fn_placing: \textbf{LEMMa} correct\_plac\_ing(pq) IMPLIES \textit{correct\_plac\_ing(find\_next(pq))}
Lemma **correct_cmp** states that \textit{find\_next\_thread} function assigns to \textit{current\_max\_prio} correct value (see section 5.5.3 for definition). It is formulated as follows:

\[
\text{fn\_cmp: LEMMA correct\_cmp(find\_next(pq))}
\]

Lemma **fn\_member\_safe** states that \textit{find\_next\_thread} does not remove ready threads from ready queue. It is formulated as follows:

\[
\text{fn\_member\_safe: LEMMA member(pq, t) AND is\_ready?(t) IMPLIES member(find\_next(pq),t)}
\]

Lemma **fn\_member** states that \textit{find\_next\_thread} does not add any thread into ready queue. It is formulated as follows:

\[
\text{fn\_member: LEMMA member(find\_next(pq),t) IMPLIES member(pq,t)}
\]
Conclusion

My diploma thesis is a part of the project "Verification of the L4 operating system kernel". The aim of this project is to formally verify the L4 kernel in order to guarantee its correctness. In my thesis we have described the implementation of the scheduler mechanism in the L4 kernel. We have picked the main properties and correctness criteria of this mechanism. We also proved the correctness of a part of the scheduler. This was done in three steps: As the first step we have created the formal specification of the basic scheduler data structure. After that we built a model of that data structure in PVS and we proved that created model fulfills its specification. After that we built a model of a part of L4's scheduler in PVS by means of defined data structure. As last step we have specified the invariants concerning modeled part of the scheduler in PVS and proved that this invariants holds in our model. The model which is built in this work, is not meant to be complete and precisely representation of the original scheduler. We can not treat proofs of such models as proofs for given C code. We hope such models can help to write a correct software.
References

1. Christopher B. Browne's Home Page
   http://www.cbbrowne.com/info/microkernel.html
3. The Mach Project
4. The L4ka Project http://l4ka.org
5. QNX http://www.qnx.com/
6. Liedtke J. μ-Kernels Must And Can Be Small
   MIT Exokernel Operating System http://www.pdos.lcs.mit.edu/exo.html
7. A Caching Model of Operating System Kernel Functionality
    1996, pages 125-130.
13.Liedtke J. Clan\&Chiefs. // In 12. GI/ITG-Fachtagung Architektur von
14.Liedtke J. Improving IPC by Kernel Design.// 14th ACM Symposium on OS
    http://www.csl.sri.com/sri-cls-fm.html


    http://pvs.csl.sri.com/

Appendix A

PVS files.

File ring.pvs

ring[T:TYPE]: datatype
BEGIN
  null: null?
  enq(body: ring, last: T): nonempty?
END ring

ring_props[T: TYPE]: THEORY
BEGIN
  importing ring[T]
  importing bitvectors@mod_rules

  r, r1, r2, r3: VAR ring[T]
  rn: var (nonempty?)
  x,y: VAR T
  Q, P: pred[T]
  n: var nat

  length(r): recursive nat=
  cases r of
    null: 0,
    enq(body, last): length(body)+1
  endcases
  MEASURE reduce_nat(0,(\(n: nat),(x: T): n+1))

    len1: lemma length(null)=0
    len2: lemma length(enq(r, x))=length(r)+1
    len5: lemma nonempty?(r) implies length(r)>0

  member(r, x): recursive bool=
  cases r of
    null: false,
    enq(bod, tl): tl=x OR member(bod, x)
endcases
measure length(r)

null_member: LEMMA member(r, x) implies not null?(r)

append(r, x): (nonempty?)=
   if member(r, x) then r else enq(r, x) endif

each_unique(r): recursive bool=
cases r of
   null: true,
   enq(bd, tl): (not member(bd, tl)) and each_unique(bd)
endcases
measure length(r)

   append_uniq: lemma each_unique(r) implies
   each_unique(append(r,x))

nth(r, (n:below[length(r)])): RECURSIVE T =
   IF n = length(r)-1 THEN last(r) ELSE nth(body(r), n) ENDIF
MEASURE length(r)

member_nth: lemma member(r, x) implies
   exists(n: below[length(r)]): x=nth(r, (n))

nth_member: lemma n<length(r) implies member(r,nth(r,n))

index(r: (nonempty?), x:{i: T | (member(r, i))}):recursive nat=
   if x=last(r) then length(r)-1 else index(body(r), x) endif
measure length(r)

   index_length: Lemma member(r, x) implies index(r,
   x)<length(r)

   index_nth: lemma member(r,x) implies nth(r, index(r, x))=x

next(r: (nonempty?), n: below[length(r)]): nat = mod(n+1, length(r))
prev(r: (nonempty?), n: below[length(r)]): nat = mod(n-1, length(r))
head(r: (nonempty?): T = nth(r, next(r, length(r)-1))

next_prev: lemma n<length(r) implies next(r, prev(r, n)) = n
prev_next: lemma n<length(r) implies prev(r, next(r, n)) = n
head_last: lemma length(r)=1 implies
  ( head(r)=last(r) and
    enq(null, head(r))=enq(null, last(r))
    and r=enq(null, last(r)))
next_length: lemma n<length(r) implies next(r,n)<length(r)
prev_length: lemma n<length(r) implies prev(r,n)<length(r)
deque(r, x): recursive ring[T]=
  if not member(r,x) then r
  elsif x=last(r) then body(r)
  else enq(deque(body(r),x), last(r)) endif
measure length(r)

deleque_last: lemma body(rn)=deque(rn,last(rn))

enq_deque: lemma enq(deque(rn, last(rn)),last(rn))=rn

deqeque_enq: lemma deque(enq(rn, x), x)=rn

deqeque_mem: lemma
  forall (r:(nonempty?), x:{k:T| member(r,k)}):
    each_unique(r) implies not member(deque(r, x),x)
deqeque_mem2: lemma member(r,x) and member(r,y)and x/=y
  implies member(deque(r, x), y)
deqeque_mem3: lemma member(deque(r,x),y) implies member(r,y)
deqeque_length: lemma length(deque(r,x))<= length(r)

deqeque_length2: lemma length(deque(r,x))>= length(r)-1
deqeque_length3: lemma length(deque(r,x))=length(r) iff
  not member(r,x)
deqeque_uniq: lemma %member(r,x) and
  each_unique(r) implies each_unique(deque(r, x))
deqeque_nth: lemma member(r,x) and n<length(r)-1 implies
nth\(\text{deque}(r,x),n) = \begin{cases} 
\text{if } n < \text{index}(r,x) \text{ then } \text{nth}(r,n) \\
\text{else } \text{nth}(r,n+1) \end{cases}
\]

\text{concate}(r_1, r_2): \text{RECURSIVE} \ \text{ring}[T] = 
\begin{cases} 
\text{null}: r_1, \\
enq(bod, tl): \text{enq}(\text{concate}(r_1, bod), tl) 
\end{cases}
\]

\text{MEASURE length}(r_2)

\text{concate}\_null: \text{LEMMA} \ \text{concate}(\text{null}, r) = r

\text{concate}\_assoc: \text{LEMMA}
\text{concate}(\text{concate}(r_1, r_2), r_3) = \text{concate}(r_1, \text{concate}(r_2, r_3))

\text{length}\_concate: \text{LEMMA}
\text{length}(\text{concate}(r_1, r_2)) = \text{length}(r_1) + \text{length}(r_2)

\text{mem}\_concate: \text{lemma} \ \text{member}(\text{concate}(r_1, r_2), x) \iff 
\text{member}(r_1, x) \text{ or member}(r_2, x)

\text{concate}\_uniq: \text{lemma} \ \text{each}\_unique(\text{concate}(r_1, r_2)) \implies 
\text{each}\_unique(r_1) \text{ and each}\_unique(r_2)

\text{con1}: \text{lemma} \ \text{each}\_unique(r) \text{ and (not member}(r,x) \text{) implies} 
\text{each}\_unique(\text{concate}(\text{enq}(\text{null}, x), r))

\text{con2}: \text{lemma not member}(r, x) \text{ implies} 
\text{index}(\text{concate}(\text{enq}(\text{null}, x), r), x) = 0

\text{con}\_enq: \text{lemma} \ \text{concate}(r_1, \text{enq}(r, x)) = \text{enq}(\text{concate}(r_1, r), x)

\text{con}\_index: \text{lemma} \ \text{member}(r_1, x) \text{ and each}\_unique(r_1) \implies 
\text{index}(\text{concate}(r_2, r_1), x) = \text{index}(r_1, x) + \text{length}(r_2)

\text{concate}\_ins: \text{lemma} \ \text{member}(r, x) \text{ and each}\_unique(r) 
\text{implies} \ \text{index}(\text{concate}(\text{enq}(\text{null}, y), r), x) = \text{index}(r, x) + 1

\text{shift}(r): \text{ring}[T] = 
\begin{cases} 
\text{null}: r, \\
enq(bd, tl): \text{concate}(\text{enq}(\text{null}, tl), bd) 
\end{cases}
\]

\text{shift}(r, n): \text{recursive} \ \text{ring}[T] = 
\begin{cases} 
\text{if } (r = \text{null or } n = 0) \text{ then } r \text{ elsif } n = 1 \text{ then } \text{shift}(r) 
\end{cases}
\]
else shift(shift(r, n-1)) endif
measure n

shift_mem: lemma member(r,x) implies member(shift(r),x)
shift_mem2: lemma member(shift(r),x) implies member(r,x)
nshift_mem: lemma member(r,x) implies member(shift(r,n),x)
nshift_mem2: lemma member(shift(r,n),x) implies member(r,x)

shift_uniq: lemma each_unique(r) implies each_unique(shift(r))
nshift_uniq: lemma each_unique(r) implies each_unique(shift(r,n))

shift_length: lemma length(r)=length(shift(r))
nshift_length: lemma length(r)=length(shift(r,n))

shift_index: lemma (member(r,x) and each_unique(r)) implies
index(shift(r),x)=mod(index(r,x)+1,length(r))
nshift_index: lemma (member(r,x) and each_unique(r)) implies
index(shift(r,n),x)=mod(n+index(r,x),length(r))

% performs cyclic shifting, n - index of element which becomes head
move_head(r:ring[T], n): ring[T]=
cases r of
    null: r,
    enq(en1,en2): shift(r, mod(length(r)-n, length(r)))
endcases

move_length: lemma length(r)=length(move_head(r,n))

move_member: lemma %forall(r:ring[T], n:below[length(r)]):
    member(r, x) implies member(move_head(r,n), x)

move_member2: lemma %forall(r:ring[T], n:below[length(r)]):
    member(move_head(r,n), x) implies member(r, x)

move_nonempty: lemma nonempty?(r) implies
    not null?(move_head(r,n))
mh_uniq: lemma %forall(r:(nonempty?), n:below[length(r)]):
  each_unique(r) implies each_unique(move_head(r, n))

j: var nat
mh_index: lemma (each_unique(r) and member(r, x) and
  index(r, x) = j)
  implies index(move_head(r, n), x) = mod(j - n, length(r))

mh_np_inv: lemma forall(r, n, (x, y:{k:T|member(r, k)})):
  (each_unique(r) and next(r, index(r, x)) = index(r, y) implies
   next(move_head(r, n), index(move_head(r, n), x)) =
   index(move_head(r, n), y)) and
  (each_unique(r) and prev(r, index(r, x)) = index(r, y) implies
   prev(move_head(r, n), index(move_head(r, n), x)) =
   index(move_head(r, n), y))

every_concate: LEMMA
  every(P)(concate(r1, r2)) IFF (every(P)(r1) AND every(P)(r2))

every_disjunct1: LEMMA
  every(P)(r) IMPLIES every(LAMBDA (x: T): P(x) OR Q(x))(r)

every_disjunct2: LEMMA
  every(Q)(r) IMPLIES every(LAMBDA (x: T): P(x) OR Q(x))(r)

every_conjunct: LEMMA
  every(LAMBDA (x: T): P(x) AND Q(x))(r) => (every(P)(r) AND
  every(Q)(r))

END ring_props

File l4_types.pvs

l4_types_defs: THEORY
BEGIN

  MAX_PRIO: posnat=255
  TIME_QUANTUM: posnat=2%*1024
DEFAULT_TIMESLICE: posnat=10%*1024

% schedule_param: TYPE =[# prio:below[2^8],
%     small_as:below[2^8],
%     cpu:below[2^4],
%     time_exp:below[2^4],
%     time_man:below[2^8] #]

l4_threadid: type+

IDLE_ID: l4_threadid
ERROR_ID: l4_threadid

% error_id is used to represent empty ring when function should
% return tcb and we can't use null instance of ring to detect and
% handle such situation

tcb: type=[# myself: l4_threadid,
    thread_state:[# TSB_LOCKED: bool,
        TSB_POLLING: bool,
        TSB_READY: bool
        #],
    queue_state: [# TS_QUEUE_WAKEUP:bool,
        TS_QUEUE_READY: bool
        #],
    priority: {i: nat| i<MAX_PRIO+1},
    timeslice: int,% sdword
    current_timeslice: int,%sdword
    absolute_timeout: below[64] %below[2^64] %qword
#

is_ready?(t:tcb): bool = t\'thread_state\'TSB_READY and (t\'timeslice>0)

idle: tcb=(# myself:= IDLE_ID,
    thread_state:=(#TSB_LOCKED := false,
        TSB_POLLING := false,
        TSB_READY := true
        #),
    queue_state:=(# TS_QUEUE_WAKEUP := true,
        TS_QUEUE_READY :=true
    )
empty_tcb: tcb=(#
    myself:= ERROR_ID,
    % parameters below are never used, but we can't leave them undefined
    thread_state:=(# TSB_LOCKED := false,
                      TSB_POLLING := false,
                      TSB_READY := true
                      #),
    queue_state:=(# TS_QUEUE_WAKEUP := true,
                      TS_QUEUE_READY := true
                      #),
    priority := 0,
    timeslice := DEFAULT_TIMESLICE,
    current_timeslice := 0,
    absolute_timeout := 63 %2^64-1
#)

importing ring[tcb]

idle_queue: ring[tcb]=enq(null, idle)

END l4_types_defs

File prio_queue.pvs

prio_queue : THEORY
BEGIN
    importing l4_types_defs
    importing ring_props[tcb]

    r1, r2: VAR ring[tcb]
current_thread(r1): tcb=
    if nonempty?(r1)
        then head(r1)
        else empty_tcb
    endif

prio_queue:type= ['#
    current_max_prio:{i:int|i<MAX_PRIO+1 and i>=-1},
    elements: array[{i:int|i<MAX_PRIO+1 and i>=-1}]->ring[tcb]
%    current: tcb = current_thread(elements(current_max_prio))
#]

current(pq: prio_queue): tcb =
    current_thread(pq`elements(pq`current_max_prio))

empty_prio_queue: prio_queue= (#
    current_max_prio:=-1,
    elements:=(lambda(index:{i:int|i<MAX_PRIO+1 and i>=-1}): if index=-1 then idle_queue else null endif)
#)

% returns TRUE iff input prio_queue coincides with empty_prio_queue
empty?:[prio_queue->bool]
=lambda(pq:prio_queue):pq=empty_prio_queue

idle_current_empty: lemma forall (pq:prio_queue):
    empty?(pq) implies current(pq)=idle

% returns TRUE iff pq contains t (as minimum one time)
member(pq: prio_queue, t:tcb):bool=
    if forall(index:{i:int|i<MAX_PRIO+1 and i>=-1}): not member(elements(pq)(index), t)
        then false
        else true
    endif
% if input tcb is ready and isn't member of input prio_queue then it returns
% prio_queue where t is placed in proper list according its priority,
% otherwise returns pq without any changes
enqueue_ready(old:prio_queue, t:tcb): prio_queue=
  if member(old, t) or (not is_ready?(t)) then old
  elsif old`current_max_prio < t`priority then
    old with [ `current_max_prio := t`priority,
               `elements(t`priority):= append(old`elements(t`priority),
               t)]
  else old with [ `elements(t`priority):=append(old`elements(t`priority), t)]
  endif

dequeue_ready(old:prio_queue, t:tcb): prio_queue=
  if not member(old, t) then old
  else old with [ `elements(t`priority):=deq(old`elements(t`priority), t)]
  endif

% -- some intermediate definitions --

% returns false iff as minimum one element of first list belongs to second list or vice versa
no_common?(r1, r2): bool=
  if forall(t:{x:tcb| member(r1,x)}):not member(r2, t) then true
  else false
  endif

%======= properties of prio_queue =======
% returns true iff each element of pq is not equal to all other
each_unique(pq: prio_queue): bool=
  forall(index:{i:int|i<MAX_PRI0+1 and i>=-1}):
    each_unique(pq`elements(index)) and
    forall(k:{i:int|i<index and i>=-1}):
      no_common?(pq`elements(index), pq`elements(k))
% % returns true iff each element of pq is not equal to all other
% each_unique(pq: prio_queue): bool=
%  forall(index:{i:int|i<MAX_PRIO+1 and i>=-1}): 
%   each_unique(pq`elements(index)) and
%  forall(k:{i:nat|i<length(pq`elements(index))}, j:{j:int| j<index and j>=-1}):
%    not member(pq`elements(j), nth(pq`elements(index), k))
%%%!!!TODO: the definition of the predicate above can be simplified

% returns TRUE iff all threads in pq are placed according
% with their priority
correct_placing(pq: prio_queue): bool=
  (forall(index:{i:nat|i<MAX_PRIO+1},
   k:{i:nat|i<length(pq`elements(index))}):
   nth(pq`elements(index), k)`priority = index ) and
   pq`elements(-1)=idle_queue

% returns TRUE iff all threads which are contained in pq has
% status TS_READY
% and timeslice>0
correct_elements(pq: prio_queue): bool=
  forall(index:{i:int|i>=-1 and i<MAX_PRIO+1},
   k:{i:nat|i<length(pq`elements(index))}):
   nth(pq`elements(index),k)`thread_state`TSB_READY and
   nth(pq`elements(index),k)`timeslice>0

% returns TRUE iff current_max_prio points to nonempty list,
% (it may be idle_queue if current_max_prio=-1)
% and all lists with index greater than current_max_prio are empty
correct_cmp(pq: prio_queue): bool=
  (forall(k:{i:nat|i<MAX_PRIO+1 AND i>pq`current_max_prio}): 
   null?(pq`elements(k)) and non-empty?(pq`elements(pq`current_max_prio))

correct_cmp_weak(pq: prio_queue): bool=
  (forall(k:{i:nat|i<MAX_PRIO+1 AND i>pq`current_max_prio}): 
   null?(pq`elements(k))

% -- simple lemmas --
pq: VAR prio_queue
t, t1: VAR tcb
i, j: VAR {i:int| i>=-1 and i<= MAX_PRIO}

cmp_inv uniq: lemma each_unique(pq) implies
  each_unique(pq with [current_max_prio := i])

append_nocom: lemma (not member(pq, t)) and (i/=j) and
  each_unique(pq) implies
  no_common?(append(pq`elements(i),t), pq`elements(j))

deque_nocom: lemma no_common?(r1, r2) implies
  no_common?(deque(r1,t), r2)

nocom_sym: lemma no_common?(r1,r2) iff no_common?(r2,r1)

cmp_weakcmp: lemma correct_cmp(pq) implies correct_cmp_weak(pq)

%%% lemmas about correctness of prio_queue if it can be represented
%%% as
%%% result of applying enqueue_ready() or dequeue_ready() to another
%%% instance
%%% of correct prio_queue. + lemmas about correctness
empty_prio_queue

empty_each_unique: lemma each_unique(empty_prio_queue)
empty_correct_placing: lemma correct_placing(empty_prio_queue)
empty_correct_elements: lemma correct_elements(empty_prio_queue)
empty_correct_cmp: lemma correct_cmp(empty_prio_queue)

enqueue_unique: lemma each_unique(pq) implies
  each_unique(enqueue_ready(pq, t))
enqueue_placing: lemma correct_placing(pq) and t`myself/=IDLE_ID im-
  plies
  correct_placing(enqueue_ready(pq, t))
enqueue_elements: lemma correct_elements(pq) implies
  correct_elements(enqueue_ready(pq, t))
enqueue_cmp: lemma correct_cmp(pq) implies
  correct_cmp(enqueue_ready(pq, t))
deque_unique: lemma each_unique(pq) implies each_unique(dequeue_ready(pq, t))

deqeue_placing: lemma correct_placing(pq) and t`myself/=IDLE_ID implies correct_placing(dequeue_ready(pq, t))

deqeue_elements: lemma correct_elements(pq) implies correct_elements(dequeue_ready(pq, t))

deqeue_cmp1: lemma correct_cmp(pq) implies correct_cmp_weak(dequeue_ready(pq, t))

% ---------------- Lemmas about correctness of operations enq/deq -- -----

enqueue_member_safe: lemma member(pq,t) implies member(enqueue_ready(pq,t1),t)

enqueue_member: lemma (member(enqueue_ready(pq,t),t) or (not member(pq,t) and not is_ready?(t)))

deqeue_member_safe: lemma member(pq,t) and t/=t1 implies member(dequeue_ready(pq,t1),t)

deqeue_member: lemma t`myself/=IDLE_ID and each_unique(pq) and correct_placing(pq) implies not member(dequeue_ready(pq,t),t)

% ==== specification of find_next_thread() ====
% returns false iff r1 contains no ready threads
contains_ready?(r: ring[tcb]): recursive bool=
    cases r of
        null: false,
        enq(bod, last):
            if is_ready?(last) then true
            else contains_ready?(bod) endif
    endcases
    measure length(r)
% removes from given queue all non-ready and non-positive_timeslice
% threads. Indeed it's find_next_thread() for one level of priority.
% If returned queue isn't empty then its head is a new thread for
% execution.

next_ring(r: ring[tcb]): recursive ring[tcb]=
    if length(r)=0 then r
    elsif is_ready?(nth(r,next(r,0))) then move_head(r,1)
    else next_ring(deque(r,nth(r,next(r,0))))
    endif
    measure length(r)

% ---- lemmas about next_ring -----

neri_member: lemma member(next_ring(r1), t) implies member (r1, t)

% returns changed prio_queue which is obtained from given
% by removing non-ready threads (which were met during search)
% and cyclic shifting of threads in entry with maximal priority
% such, that head of pq(cmp) is the next thread for execution.
% if there is no ready threads in whole pq, it will return pq where
% all entries are null and cmp=-1
% if pq initially contains idle_thread then entry of pq with index -
% 1
% can't be equal and idle_tcb is returned if there are no other
% ready threads

find_next(pq): recursive prio_queue= let cmp=pq`current_max_prio
in
    if cmp=-1 or contains_ready?(pq`elements(cmp))
    then pq with [``elements(cmp):=next_ring(pq`elements(cmp))]
    else find_next(pq with
    [``elements(cmp):=null,``current_max_prio:=cmp-1])
    endif
    measure pq`current_max_prio+1

%--------- Lemmas about correctness of find_next ---------------

% find_next should preserves correct_placing, correct_elements,
% each_unique, correct_cmp and membership under
% some circumstances

fn_placing: lemma correct_placing(pq) implies correct_placing(find_next(pq))
fn_elements: lemma correct_elements(find_next(pq))
fn_cmp: lemma correct_cmp(find_next(pq))
fn_member_safe: lemma member(pq, t) and is_ready?(t)
  implies member(find_next(pq), t)
fn_member: lemma member(find_next(pq), t) implies member(pq, t)

END prio_queue