Read-Copy-Update (RCU)

(..contd)

RCU differs from read-write locks, in that, it allows the read-path to execute without any synchronization, i.e., at full speed, and yet ensure correct behaviour in presence of occasional concurrent updates.

As we saw in our last lecture, this can be achieved by ensuring that updates are done in an atomic fashion, e.g., if the updates can be written as one value swap, then atomic CAS instructions can be used.

However, even if updates to a global data structure can be written through a single CAS instruction, there may be other operations, such as free() (memory reuse) that may be required to complete the operation. In the searchable stack example, `free()` can only be called once we are sure that no other thread could be holding a reference to the location being freed.

One way to ensure this is through reference counting, wherein each time a thread holds a pointer, it increments its reference count. However maintaining reference counts involve write operations, and suffer from the same problems as those suffered by read-write locks.

The other approach is to use information about the software (of which this data structure is a part). For example, if this data structure were implemented as part of the Linux kernel, and if we know that the Linux kernel is non-preemptible (i.e., a thread cannot be preempted while it is executing in kernel space), then we could devise an algorithm.

First, let's understand what we mean by the Linux kernel being non-preemptible. This means that a thread cannot be context switched-out in the middle of kernel execution. Even if a timer interrupt occurs in the middle of thread execution, the timer interrupt is recorded but the execution of the current thread is resumed (till it reaches a safe point), before context switching to another thread.

Assuming that the searchable stack structure is a part of the Linux kernel and that a thread cannot be context switched out in the middle of the `search()` procedure, we can be sure of the following things after a `pop()` update:

- Only threads currently running in kernel mode on other CPUs could be holding a pointer to the `oldtop`
    - Threads that are in ready list or in user mode are definitely not inside the `search()` procedure. If they ever call `search()` (e.g., in future), they will start at `newtop`.
- After executing `pop()` using CAS, we just need to wait for all other executing CPUs to execute a context switch. After a context switch, that CPU cannot be holding a reference to `oldtop` (as it must have exited `search()`).

The RCU algorithm waits for all other CPUs to perform a context switch before freeing the old `top`. One way to accomplish this is to schedule a small dummy thread on each CPU. If all the dummy threads execute, a context switch has definitely taken place.

In a way, RCU trades space for time. It allows the common case (read accesses) to execute faster, by allowing unused locations to not get freed for an extended period of time (i.e., for time greater than what it strictly needs to be). This is a useful approach only if the memory is plentiful, which is true for modern hardware environments.

Applications of RCU

RCU requires information about software and so can be used in special environments, like the kernel. Here are two compelling examples where RCU is used in the Linux kernel:

- Network routing table: If a Linux machine is used for routing, the routing table is dereferenced on every incoming packet (read-only access). Occasionally, however, the routing table may be updated by the network-level route-discovery protocols.

  Using read-write locks would penalize routing table lookups. However, RCU allows lookups to execute unsynchronized (full speed) and yet maintains correctness with respect to occasional updates.

- Loadable Modules: Linux kernel modules are object files that can be loaded into the kernel at runtime. The module object files contain code (callable functions) and data that can be reached from the kernel. At the same time, it is possible for users to load or unload modules at will. We need to ensure that a module that is currently being referenced (e.g., a CPU is executing a function inside it) is not unloaded.

  This can be accomplished using read-write locks, but that would mean that all module accesses (e.g., function calls) are preceded by a read-acquire. A much faster method would be to use RCU, where function calls can execute without synchronization, but updates (loading/unloading) will need to be done carefully (CAS, wait for other CPUs to context switch before freeing).

OS Organizations

The UNIX-like abstractions that we have studied so far, are also grouped into what are called *monolithic kernels*, wherein all kernel services live in one shared address space (of the kernel), and the applications invoke system calls to access these services.

Figure showing monolithic kernel and Apps. The monolithic kernel has subsystems like filesystem, virtual memory, scheduler, device drivers, etc.

The primary weaknesses of the monolithic organization are:

- All subsystems share the same protection domain. A bug or security hole in one subsystem can bring down the entire machine. For example, bugs in device drivers are fairly common, and can completely compromise the system's security.
- The subsystems have to be shared by all applications, and thus they need to cater to all types of applications. For example, the same cache replacement policy or filesystem layout and routines will be used for all applications. This can cause performance problems, if the application needs do not match the kernel policies.

There are alternate OS organizations that have been proposed to alleviate these limitations of the monolithic kernel, two of which are called the *microkernel* and the *exokernel*.

Microkernels

Microkernels organize the kernel subsystems as "servers", each running in isolated address spaces (or processes). The kernel only provides fast interprocess communication and protection mechanisms between various processes.

Figure showing microkernel (IPC+protection), servers (FS, VM, Drivers, Scheduler, TCP/IP, etc.), and apps.

The kernel becomes much thinner, and isolated from all the other services. A bug or a security hole in one service does not affect the other services or the kernel. Also, different implementations of subsystems can be used for different applications: e.g., different FS implementations can co-exist (as executables and processes) and different apps can use different FS implementations (perhaps tuned for themselves), assuming that the FS implementations are operating on different parts of the disk. While monolithic kernels also support multiple filesystems, the flexibility of a microkernel is far greater. For example, a database-like application that cares about persistence and has structured records, can choose its own filesystem where a file looks very similar to a database table and has stronger persistence guarantees.

The downside of the microkernel architecture is performance: IPC, no matter how fast, is decidedly slower than direct interaction with the kernel. For example, monolithic kernels allow communication between application and the kernel service through pointer exchanges. Such communication will now require marshalling and unmarshalling of data structures across address space boundaries.

While microkernels are not very popular in modern desktop and server environments, microkernels (e.g., L4) are often used in embedded environments that require strong isolation, reliability, and security guarantees.

Exokernel

The philosophy of the exokernel is to abstract at a very low level (e.g., at the hardware level). Instead of executing the kernel services inside the kernel, the services should run as a part of the application, and the kernel should allow these services to be implemented at the user-level by exporting a rich-enough system call API.

Figure showing App with FS, VM, scheduler subsystems within the App, and the kernel providing low-level syscall API.

Exokernel example for virtual memory:
Here are the set of kernel abstractions:

- Downcalls (system calls)
  - `pa = AllocPage()`
  - `DeallocPage(pa)`
  - `CreateMapping(va, pa)`
- Upcalls (kernel->app calls, similar to UNIX signals but more general)
  - `PageFault(va)`
  - `PleaseReleaseAPage()`

Notice that unlike UNIX, an application is fully aware of the physical address space, and uses `AllocPage()` and `DeallocPage()` to control its physical memory footprint. The `Createmapping()` syscall is used for creating a mapping from a process-private virtual address to a physical address. The process must be authorized to map the physical address for the CreateMapping() call to succeed.

Similarly, there are upcalls which allows the kernel to invoke process-registered functions (similar to signal handlers on UNIX) for a page fault and for requesting the process to release a page. On a page fault, the process's page fault handler gets called (unlike UNIX where the page fault handler was inside the kernel), and decides what to do: e.g., choose a physical page for replacement, create a new mapping, etc. This allows a process to decide its own replacement policy and other mechanisms, thus providing far greater flexibility. Also, the `PleaseReleaseAPage()` upcall can be used by the kernel to request the process to release a page (using DeallocPage()). The kernel may expect the process to respond to an upcall within a bounded time, after which it may take coercive action (e.g., forcibly take a page away, kill the process, etc.).