Kernel Korner

AEM: A Scalable and Native Event Mechanism for Linux

Give your application the ability to register callbacks with the kernel.

by Frédéric Rossi

In a previous article [“An Event Mechanism for Linux”, *LJ*, July 2003], we introduced the necessity for Linux to adopt a native and generic event mechanism in the context of telecom. Many existing solutions attempting to increase the capabilities of Linux have failed thus far. Others did not reach our level of satisfaction, because carrier-grade platforms have different levels of real-time requirements. In order to succeed, such an event mechanism must be bound tightly to the host operating system and take advantage of its capabilities in order to deliver better performance.

At the Open Systems Lab (Ericsson Research) in Montréal, Canada, we started a project in 2001 to develop a generic solution, the Asynchronous Event Mechanism (AEM). AEM allows an application to define and register callback functions for some specific events and lets the operating system execute these routines asynchronously when the events have been activated.

AEM provides an event-driven methodology of development. This is achieved through the definition of a natural user interface in which event handlers contain in their parameter lists all of the data necessary for their execution sent directly by the kernel.

AEM also is motivated by the fact that complex distributed applications based on multithreaded architectures have proven difficult to develop and port from one platform to another because of the management layer. The objective of AEM is not only to reduce the software's development time, but also to simplify source code generation in order to increase portability between different platforms and to increase the software's life cycle.

The biggest challenge of this project was designing and developing a flexible framework such that adding or updating a running system with new event-handling implementations is possible. The constraint was to be able to carry out system maintenance without rebooting the system. The modular architecture of AEM offers such capabilities.

AEM is a complementary solution to other existing notification mechanisms. One of its great benefits is the possibility to mix event-driven code and other sequential codes.

AEM: Architecture Overview

AEM is composed of one core module and a set of loadable kernel modules providing some specific event service to applications, including soft timers and asynchronous socket interfaces for TCP/IP (Figure 1). This flexible architecture permits AEM capabilities to be extended at will.

There is no restriction on what a module can implement, because each exports a range of independent pseudo-system calls to applications. In fact, this allows two different modules to make available the same

functionality at the same time. Interestingly, this offers the possibility of loading a new module to implement an improved revision without breaking other applications—they continue to use the older version. This design provides the ability to load the necessary AEM modules depending on the applications' need or to upgrade modules at runtime.

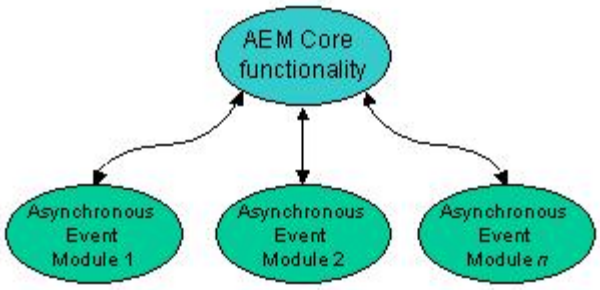

Figure 1. AEM is based on one core kernel module providing the basic event functionalities and a set of independent kernel modules providing asynchronous event services to applications.

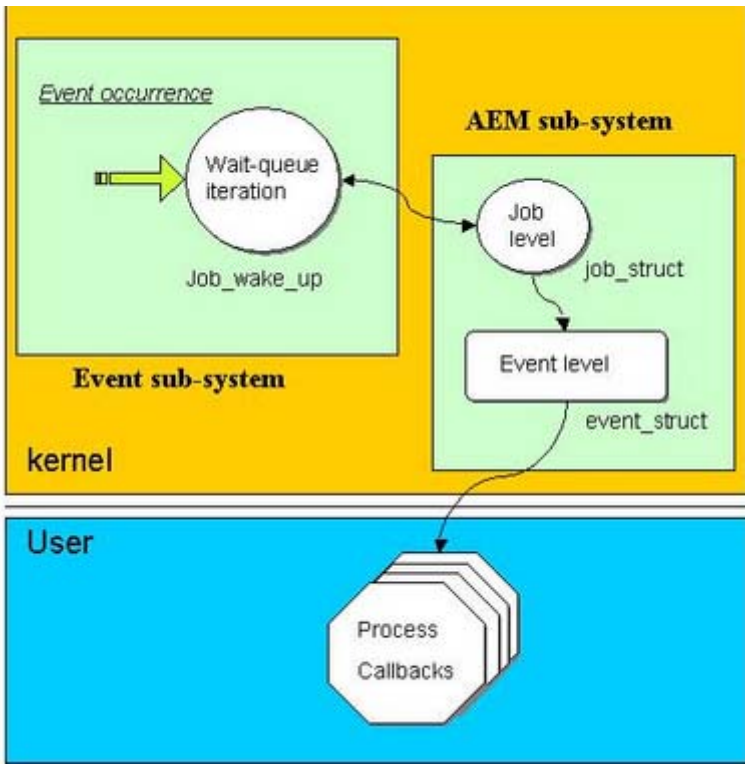

Figure 2. Architecture behind event activation and process notification in AEM. Event wait queues containing sleeping jobs are scanned, and all concerned jobs wake up at the occurrence of an event. It immediately follows the activation of the corresponding event for each related process.

The condition for such flexibility is the presence of event activation points located at strategic places in the kernel (Figure 2). Each activation point is a specific AEM queue used to activate events. In the following sections, we describe the internals of AEM in detail.

AEM: Internals Overview

The concept of asynchrony is a major problem when the main flow of a program's execution is broken without warning in order to execute event handlers. Input requests then are handled without the knowledge of

previous input states. These are delivered directly by the core kernel or the interrupt handlers and are received without presuming any kind of order. This situation constitutes a problem for some applications, including those based on TCP/IP, which rely on the transaction state to proceed.

AEM is a three-layer architecture composed of a set of pseudo-system calls for event management, a per-process event_struct performing event serialization and storing context information in order to execute user callbacks and a per-event job_struct performing event activation.

Events

From the AEM perspective, an event is a system stimulus that initiates the creation of an execution agent, the event handler. Support is provided by the event_struct, which is a structure initialized during event registration that contains the context necessary to execute one event handler. Some of the main fields are address of a user-space event handler, constructors and destructors for the event handler and relationship with other events (list of events, child events and active events—see Figure 3).

There can be as many registered events per process as necessary. When an event is detected, we say it is activated, and the user-defined callback function soon is executed. Events are linked internally in the same order as they arrive into the system. It then is up to each handler constructor to manage the data correctly and keep the events serialized for each process without presuming any order of arrival.

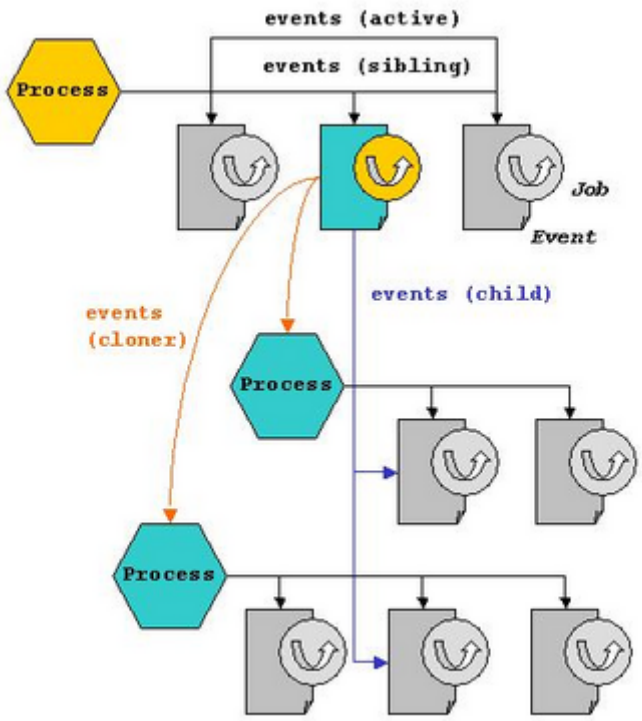

Figure 3. The Relationship between a Process and Its List of Events.

Some process events are active and linked to an active events list. Upon activation, an event can create a process. These events are called cloners, and the relationships between these events and created processes are recorded internally. An event registered by the top process in Figure 3 has created two new processes below it. They remain attached to this event and keep their own list of events.

Event handlers are used during event registrations and must be implemented at the user level. They define

their own fixed set of parameters in order to provide event data completion directly to the user-space process. This operation is done by event constructors and destructors executed right before and right after handlers are called. Event handlers are executed in the same context as the calling process. The mechanism is safe and re-entrant; the current flow of execution is saved and then restored to its state prior to the interruption.

A priority is associated with each event during registration that represents the speed at which an application is interested to receive notification. It is possible to register twice for the same event with two different priorities.

Other real-time notification mechanisms, such as the real-time extension of POSIX signals, do not consider priorities during the scheduling decision. This is important, because it allows a process receiving a high-priority event to be scheduled before other processes. In AEM, the occurrence of an event pushes the event handler to be executed depending on its priority. To some extent, an event handler is a process, because it has an execution context. Changing process priorities dynamically is a real issue when the rate of event arrival is high, because priorities are updated quickly at the same rate. We solved this problem by introducing a dynamic soft real-time value calculated using a composition of event priorities. This value influences the scheduling decision without affecting the Linux scheduler and brings soft real-time responsiveness to applications.

Jobs

A job is a new kernel abstraction introduced to serve events before notifying processes. It is not a process, although both share the same conceptual idea of executable entity. One typical action performed by a job is to insert itself into a wait queue and stay there until something wakes it up. At that point, it quickly performs some useful work, such as checking for data validity or availability before activating the user event, and goes back to sleep. A job also guarantees that while it is accessing some resource, no other job can access it. Several jobs can be associated with one process, but there is only one job per event.

This abstraction layer between the kernel and the user process is necessary. Otherwise, it is difficult to ensure consistency in checking for data availability or agglomerating multiple occurrences of the same event when the handler is executed. If something goes wrong, the process wastes time handling the event in user space. Deciding whether to concatenate several notifications is event-specific and should be resolved before event activation.

A generic implementation of jobs would consider software interrupts in order to have a short latency between the time an event occurred and the time the process is notified. The goal is to execute on behalf of processes and provide the same capabilities as both an interrupt handler and a kernel thread, without dragging along a complete execution context.

Two types of jobs are implemented, periodic jobs and reactive jobs. Periodic jobs are executed at regular intervals, and reactive jobs are executed sporadically, upon reception of an event. Jobs are scheduled by their own off-line scheduler. According to the real-time theory of scheduling, both types of jobs could be managed by the same scheduler (see the Jeffay et al. paper in the on-line Resources). In our context, a job is a nonpreemptive task. By definition, jobs have no specific deadlines, although their execution time should be bounded implicitly because of their low-level nature. This assumption simplifies the implementation. The constraint in our case is for reactive jobs to be able to execute with a negligible time interval between two invocations so as to satisfy streaming transfer situations.

Our implementation of periodic and reactive jobs is different in both cases in order to obtain a better throughput in case of sporadic events. A job scheduler and a dispatcher handle periodic jobs, whereas reactive jobs change state themselves for performance reasons. Figure 4 describes the job state evolution and

functions used to move from one state to another.

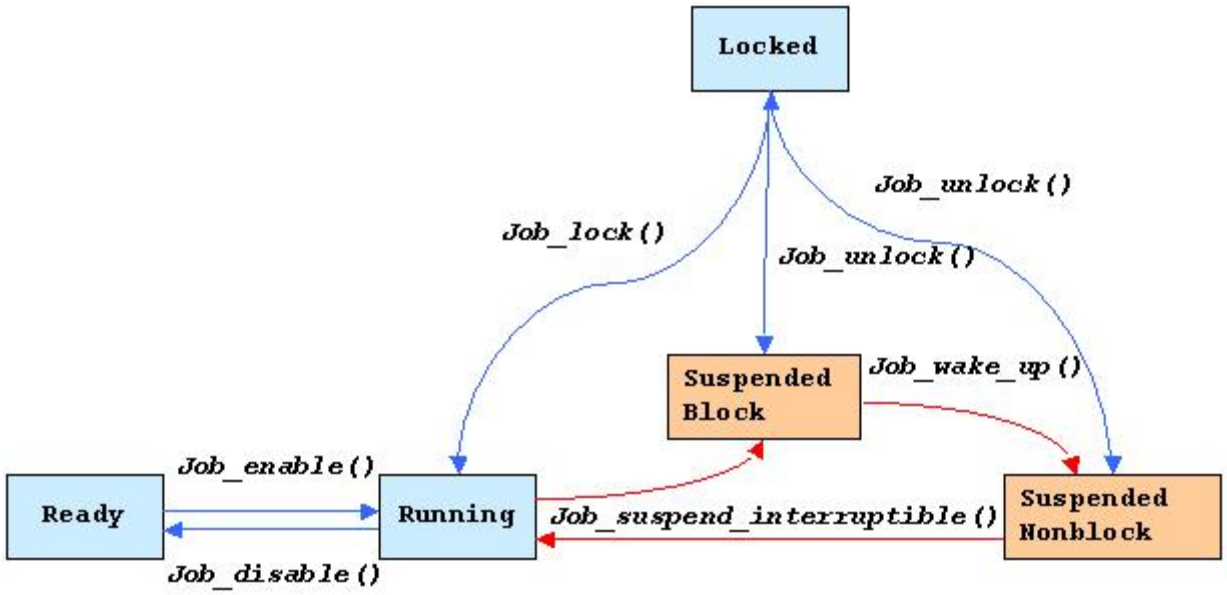

Figure 4. The State Transition Graph for Periodic Jobs and Reactive Jobs

Once a job has activated the corresponding event, either a process is executed asynchronously or the current flow of execution of the user program is redirected somewhere else in the code. The user decides how to handle events at registration time.

Asynchronous Execution of Processes

Event handlers can be executed either by breaking the main thread's flow of execution or by creating a new process to handle the event and execute it in parallel. In either case, the event is managed transparently and asynchronously without explicitly polling for its occurrences. This management has some important implications, because an application does not have to reserve and consume system resources before it is needed. For example, we adapted a simple HTTP server for benchmarking AEM, which is fully single-threaded and capable of quite good performance for that type of server. It is described at the end of this article.

Sometimes situations arise in which it is necessary to create a new process as a response to an event. Unfortunately, creating processes dynamically is resource consuming; during that period, neither the new nor the parent process is able to handle new requests. In some critical situations, there might be no resource left for that purpose, such as a shortage of system memory. This is a problem because emergencies might require creating new processes to shut down the system gracefully or to carry out handover procedures.

For this reason, we introduced a new concept called capsules. A capsule is a task structure already initialized and part of a pool containing other free capsules. When a process wants to create a new execution context, a capsule is linked out from that pool and is initialized with only a few of the current process parameters.

During event registration, specific flags indicate whether a handler is to be executed inside a process. No parameter or a 0 means the handler is to be executed by breaking the current flow of execution. These flags are:

- EFV_FORK: to create a process with the same semantic as fork().

- EVF_CAPSULE: to create a process from the capsule pool.
- EVF_NOCLDWAIT: this flag has the same semantics as the signal SIG_NOCLDWAIT. When the child process exists, it is re-parented to the capsule manager thread.
- EVF_KEEPALIVE: to prevent the process/capsule from exiting by entering into a kernel loop—like while(1); in user space.

Memory Management

Memory management is a major issue for an event-driven system, because events are handled by executing callback functions located in the application space directly from the kernel. The simplest solution would be to allocate memory when the process registers for an event. Consider, though, either the case of a huge number of events to register or the case of a process that needs to be restarted, needs new events to be added or needs events to be unregistered. In case of failure in event management, the system integrity becomes inconsistent. It is less critical for the operating system kernel to manage such resources itself and allocate memory on demand on behalf of processes.

Regarding performance, it is necessary to be able to allocate this memory inside a pre-allocated pool to prevent memory fragmentation and maintain soft real-time characteristics. Generic memory allocators, such as glibc's malloc(), are not aligned with this requirement even if they provide good performance for general purposes.

Some data types, such as integers, are passed to user space easily, but more complex types, such as character strings, require specific implementations. Process memory allocation is managed by the glibc library. This becomes complicated if we want to allocate memory from the kernel, because we have to take care that this new address is located correctly or mapped into the process space. Allocating memory efficiently and simply on behalf of user processes is something currently missing in the Linux kernel but needed.

AEM's vmtable is filling the gap in this area of memory allocation. It implements a variation of the binary buddy allocator—covered in Knuth, see Resources—on top of a user process memory pool. This permits the management of almost all kinds of data of unplanned sizes. In the case of a real shortage of memory, it can fall back on the user decision by using event handlers. This feature offers the possibility of relying on glibc as a last resort if something bad happens.

AEM has been designed to return a valid pointer in constant time when free blocks are available. This often is the case for telecom applications, which are most likely to receive requests of the same size for the same type of application. We also want to prevent memory fragmentation caused by a flood of requests of different sizes in a short time interval.

vmtable also has an interesting extension: it can be used to provide user-space initialization procedures for device drivers. This is possible using a pointer, allocated from the kernel by the AEM subsystem and then passed up to the user process in a callback function. It then is given back to the kernel when the function returns. Callback functions are not only usable as a response to an event, but also as a means to communicate safely with the kernel.

In this scenario, each user process is assigned a pool of memory called a vmtable. Forked processes and capsules automatically inherit a vmtable, depending on their parents' vmtable. Two different types of strategies are implemented:

1. VMT_UZONE: allocation is done inside process' heap segment. This provides fast access but consumes the user process address space.

2. VMT_VZONE: allocation is done in the kernel address space and mapped inside process' address space. This provides minimal memory consumption, but accesses take more time due to the page faults handling.

Both strategies carry some advantages and disadvantages, depending on the situation. The preferred strategy is chosen at runtime when starting the application. Figure 5 illustrates the architecture layout of vmtable.

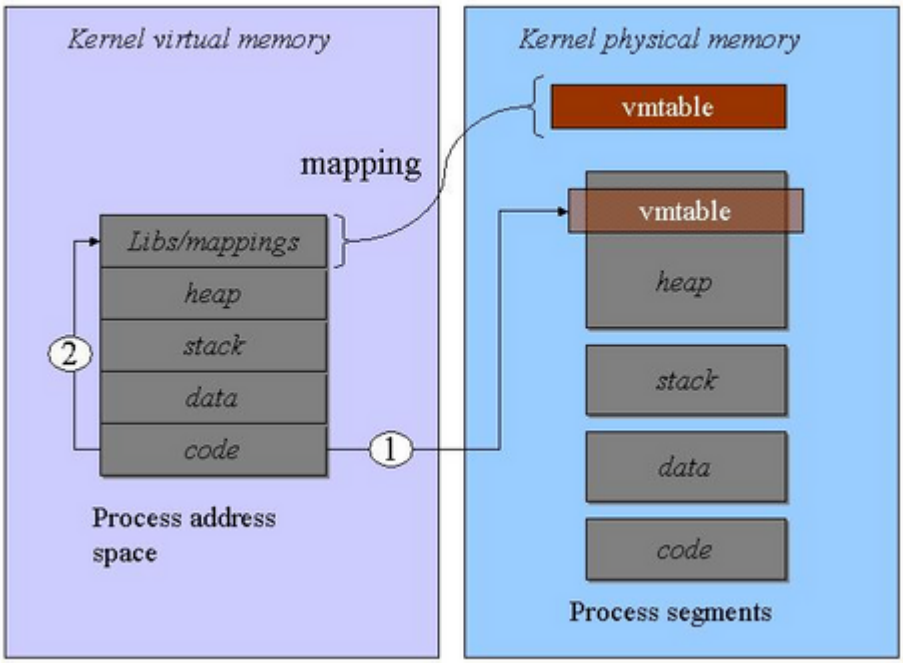

Figure 5. The Architectural Layout of vmtable

In the current implementation, physical pages are allocated explicitly by the vmtable subsystem. This is done to ensure that they really are allocated contiguously, so the memory pool can be used safely for I/O operations. A future extension is to use vmtable for direct I/O in the context of network performance.

vmtable exports a simple and easy-to-use interface to AEM users and module developers, hiding the complexity of memory allocation in kernel space. Simple routines are implemented in the AEM core module to allocate and free memory in a transparent manner. This interface greatly simplifies maintenance of modules and their adaptations between different Linux kernel releases.

Scalability

We performed testing to measure the behavior of AEM during a simple exchange between two remote processes. This test was done to ensure the time needed to context switch an event handler did not cause a performance problem. More recently, we also performed benchmarking to measure the scalability of AEM and to test the internal implementation of jobs and wait queues, which represent the base functionality of AEM.

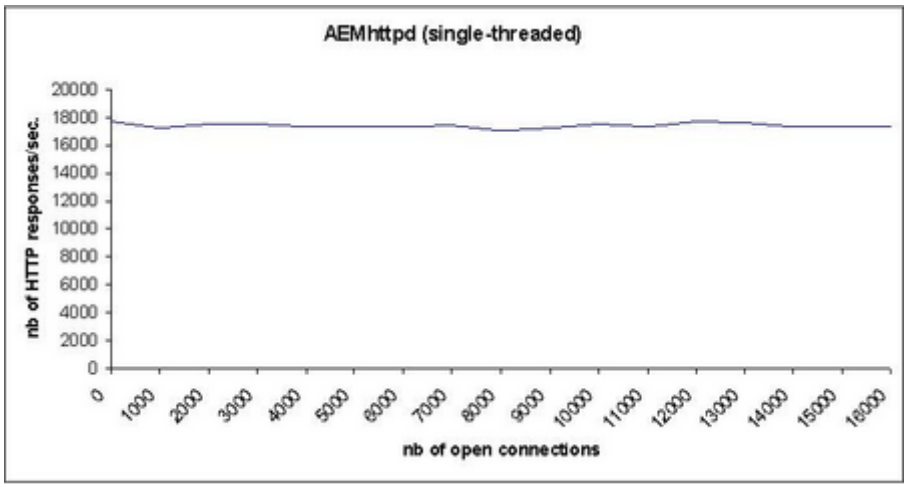

Figure 6. AEMhttpd, a single-threaded HTTP server used to make scalability measurements of the AEM internal implementation. Here we have performed 100 active connections for each sample.

In order to generate figures we could compare easily, we decided to use an existing Web server and adapt it with the AEM interface. AEMhttpd is a simple single-threaded HTTP server (see Resources.) A single-threaded server runs entirely in the main thread of execution. It means that neither kernel threads nor user threads are created to handle HTTP requests. The measurements done with this type of server focus on the implementation capabilities rather than on the performance of the server itself.

In the example illustrated in Figure 6, we have run 100 active transactions. For each transaction, we increased the number of open connections to increase the number of jobs in the sockets' connection wait queues. In a standard server implementation based on select(), this would have increased the time to respond to requests, because all descriptors would have been scanned in sequence. With AEM, only active jobs (that is, sockets with data ready) execute their corresponding event handlers. This proves that AEM provides a generic and scalable implementation.

Conclusion

Linux is widely used in the industry, imposing itself as the operating system of choice for enterprise-level solutions. It also is on the edge of becoming the operating system of choice for the next-generation IP-based architectures for telecom services. There already is a wide acceptance of Linux capabilities, but providing enhancements at the kernel level will attract the next-generation service providers that demand scalability, performance and reliability.

AEM is a solid attempt to provide the asynchronous notification of processes together with event data completion. It also brings an event-driven methodology that enables a secure programming paradigm for application developments. In addition, AEM implements a mechanism that focuses on increasing application reliability and portability by exporting a simple user interface.

Acknowledgements

All of the researchers at the Open Systems Lab and Lars Hennert at Ericsson for their useful comments, and Ericsson Research for approving the publication of this article.

Resources for this article: www.linuxjournal.com/article/7746

Frédéric Rossi (Frederic.Rossi@ericsson.ca) is a researcher at the Open Systems Lab at Ericsson Research,

in Montréal, Canada. He is the creator of AEM and the main driver behind its development.
