OSDL CGL Requirements for Low-Level Asynchronous Events

Contact: Frederic.Rossi@Ericsson.CA
Date: March 24, 2003
Revision: Draft 0.6

1 Introduction

OSDL CGL WG specifies that Carrier-Grade Linux 2.0 shall provide a very efficient capability for handling asynchronous events.

CGL need to have an efficient mechanism that provides the Linux kernel with advanced carrier-grade capabilities. The motivation for the asynchronous event mechanism is to enforce the system scalability and soft real-time responsiveness by reducing contentions appearing at the kernel level especially under high load.

In this document we associated priorities with requirements. The reason to push some requirements to CGL 3.0 is that the implementation might require a rework and the APIs might be changing. Also, some items pushed to priority 2 imply tight cooperation with the process scheduler:

- Priority 1 - CGL 2.0
- Priority 2 - CGL 3.0

2 Description

The purpose of this document is to describe the requirements for an efficient implementation of an asynchronous event mechanism.

The following sections describe AEM (The Linux Asynchronous Event Mechanism) [aem], which is a kernel implementation for the management of asynchronous system events in complex software architectures especially in the context of Telecom applications.

Carrier-grade systems are facing severe conditions of execution and all kind of hazards. In many common situations these platforms have to handle thousands of transactions per second, hundred simultaneous connections per node while taking care of databases containing millions of entries. The downtime of such systems should not exceed 5 minutes per year, comprising software upgrade, operations and maintenance and hardware failures. Also the response time should be kept in an acceptable range of some few seconds even under high load. An efficient mechanism is then necessary to dispatch low-level events to applications.

Traditional dispatch mechanisms, which either implement (based on a multithreaded architecture) or partially provide (like AIO, epoll) a kind of event-driven paradigm, fail to satisfy telecom requirements in these conditions. They basically partially solve the problem behind event management, because both the notification and the delivery of information is a critical function of the system under a variable load. In [faq] we provide a functional comparison between AEM, AIO, select and epoll. Basically AEM is meant to be complementary to those existing systems by providing a support for the asynchronous execution of handlers related to low-level events.

It is important that system resources are uniformly distributed between applications. Resource greedy applications could easily create delays or points of failure in a node because they implement their own dispatch mechanism. This is also a question of reliability and security. This is why it is necessary to provide a generic support for event management directly in the Linux kernel that is usable by all applications and more controllable by the kernel.

AEM aims at implementing kernel components that scale linearly with the number of events while providing good response time to applications. It basically provides a notification mechanism and completion of event data directly to application handlers.

AEM focuses on implementing a native support to interrupt-driven software components like network protocols or event driven software like the X windows system, Corba or TIPC. More precisely, AEM aims at reducing complexity of multi-layer software architectures or applications that require a high level of con-
currency between components. Some of its characteristics are:

- Uses a new object as event references (not descriptors),
- Is not restricted to file and socket descriptors,
- Implements sporadic or periodic events monitoring,
- Makes use of priority enforcement,
- Manage memory on behalf of processes,
- Can execute handlers in the same execution context or in a new process context,
- Provides routines to stop/start or shutdown events explicitly.

AEM belongs to a group of applications providing asynchronous execution of processes. Other kinds of similar mechanisms are the x-kernel, active messages or even the Microsoft IOCP. It provides basic building blocks to build a higher level asynchronous interface. AEM is not limited to socket operations and network protocols. It is possible to implement callbacks for overload notifications or for whatever critical events. Of course, even if it is a generic and very flexible, the support in the kernel must be given for all events.

In a carrier-grade context, computing nodes should also be secured to limit the number of events or restrict the type of events applications can register. AEM can be improved with specific algorithms to provide QoS or load control at the process level, which would be complex to put in place otherwise.

2.1 The event-driven model

The purpose of this section is not to be exhaustive, but to simply illustrate the difference between a blocking system call and an event-driven mechanism during the execution of an application. The example we have in mind is a simple application using networking functions like read, select or accept.

2.1.1 Synchronous interface

Figure 1 illustrates a typical blocking system call. During the system call phase in 1) the process enters a *sleeping* state 2). This state is periodic as illustrated in 3) and the process can either oscillate between a *sleeping* state and a *running* state or stay sleeping until it is awaken by the underlying subsystem. During this period of time, which can be undetermined, the application is simply frozen until the corresponding event occurs to wake it up.

It is important to note that the application must be aware of the number of file descriptors to listen to in the case of the select. This is an important limitation because is forces to really plan and design the application in advance or to set it to the maximum available by the system resources. This last solution increases the response time when using select().

2.1.2 Event-driven interface

Figure 2 illustrates the behavior of an application using AEM. Event registration in 1) is a necessary step an application has to issue prior in order to handle an event via the corresponding event handler. This later must be exist for that purpose. The goal is to fill internal structures with information provided by the application like the address of the event handler. Once the event is registered the application can execute in 2) without interruption or without waiting explicitly for this event. Once an event has been detected and activated in 3), the handler will be executed soon. This occurs when returning from any system call (there is currently no

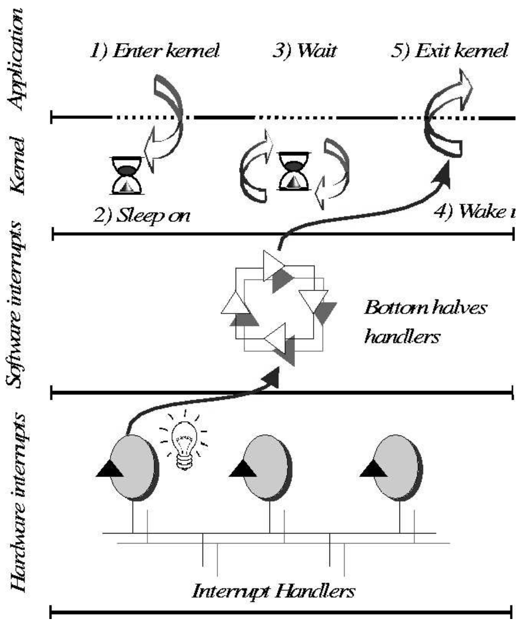

Figure 1: *Operating system layers view of a synchronous interface.*

restriction regarding the type of system call allowed to precede the execution of an event handler). A context-switch is then performed and the handler is executed in 4). The application is resumed in 5) when the handler exits.

There is one point regarding AEM that compares advantageously to the previous example we illustrated with select. There is no restriction or explicitly limitation on the number of sockets to listen to; basically because there is no equivalent to select at all. As long as the event is not disabled by the application it will continue to execute the event handler.

In the case of an event-driven scheme, the accept system call is a non-blocking operation as we illustrated in the next example. This scheme allows the application to continue execution and accept incoming connections asynchronously without any disturbance.

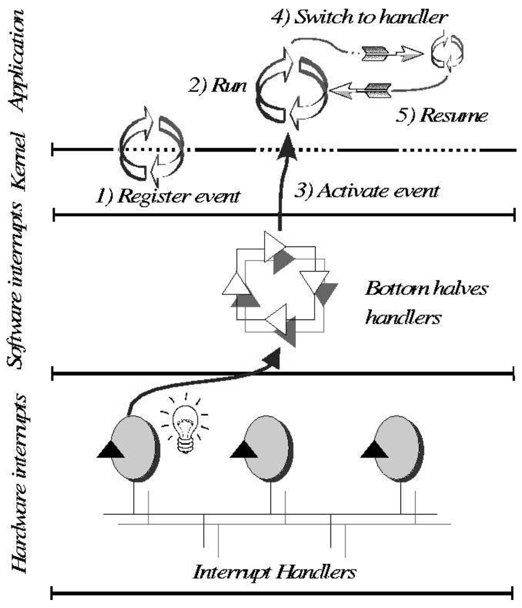

Figure 2: *Operating system layers view of an event-driven interface.*

2.1.3 Example of an event-driven server

The following example illustrates the part of an asynchronous server performing the accept of incoming connections. In this example, the procedure is executed in the same thread of execution. Once registered the server is kept alive. When a connection is coming in the handler of accept is executed.

```
/*
 * This event handler is executed upon connection.
 */
void accepted (int jid, int old_sockfd, int new_sockfd)
{
    /* handle accept */
}


main ()
{
    int sfd;
    int err;
    int c_len;
    struct sockaddr_in s_addr;

    c_len = sizeof (c_addr);

    if ((sfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
         perror ("");
         exit (1);
    }

    bzero ((char *)&s_addr, sizeof (s_addr));
    s_addr.sin_family           = AF_INET;
    s_addr.sin_addr.s_addr = inet_addr (SERVER_ADDR);
    s_addr.sin_port              = htons (SERVER_PORT);


    if (bind (sfd, (struct sockaddr_in *)&s_addr, sizeof (s_addr)) < 0) {
        perror ("");
        exit (1);
    }

    listen (sfd, 5);

    /* Event registration for accept - non blocking system call */
    id = sockasync_accept (sfd, (struct sockaddr *)&c_addr, &c_len,
                            0, accepted);
    if (id<0) {
        perror ("");
        exit (1);
    }

    enter_keepalive ();
}
```

2.1.4 Note on transparency

As opposed to other mechanisms, applications are not required to use AEM exclusively. Also AEM will not disturb applications that use it partially (the synchronous model inside the asynchronous model). The goal is to provide both in the same time. This is possible because AEM makes use of different internal mechanisms which are not used by other solutions. For example, applications can take benefit of select inside event handlers.

2.2 Scalability

Scalability is achieved by ensuring that all sub-components used by the applications are executed in a time-bounded interval in the worst case. In this situation, kernel components must be able to handle variations of the workload linearly.

AEM provides a fine-grained implementation of event activation points using a mechanism similar to wait-queues for processes. This ensures that only active events consume system resources. Most applications based on a multi-threading architecture need a lot of threads to handle system events. AEM provides a native event-driven methodology, which prevents from introducing a bottleneck at the scheduler level. To achieve this it introduces a kernel processing element called job that takes care of events prior to execute related handlers. These elements are developed specifically to answer the need of one type of event. In particular they can aggregate or not multiple arrivals of the same event and they are active only
when awaken by the corresponding events.

Scalability must be a linear function of the number of events and the latency for the execution of one event handler.
[TODO: Figures of scalability, Figures of latency,]

2.3 Soft real-time

Soft real-time characteristic defines a system that is able to meet its deadlines statistically. Priorities can be set during event registration in such a way that this interval (response time latency) varies with event priorities. An application that is about to receive an event of high priority must be able to receive it before any other processes. This implies a global management of events and a cooperative kernel mechanism.

The need for such a mechanism comes from the fact that we need to bypass a significant section in the kernel and invoke directly a user thread with the appropriate data to go and handle the event. But in order to maintain a soft-real time execution profile we need to create a tight relationship or an affinity between
a process and its events. In this case, communications between the kernel level and the user level is reduced to execute event handlers inside applications context.
This is motivated by the fact that the latency time between a notification sent by the kernel and the effective execution of the corresponding action by the process is not/badly controllable by other mechanisms.

Memory allocation on behalf on processes is a critical part of event management. Generic memory managers have always been a problem for applications close to real-time requirements. Patterns of allocations become persistent as a system is up and running provoking shortage of memory for some specific block sizes and aggravating the system latency. For this reason AEM implements a per process memory management that insures fast memory availability to store event information. The memory manager, *vmtable*, handles pools of memory using a buddy algorithm adapted for this purpose.

Also, AEM aims at putting more pressure on processes (or execution agents). A process priority-driven mechanism solves many problems. In particular it permits to apply directly (to a certain extent since most applications run in user space) some of the concepts already developed for hard real-time systems. This is achieved in AEM by modifying dynamically process priorities according to pending events priorities.

2.4 Reliability

Reliability is the capability of a system to resist against errors, hazards and faults. Failures due to shortage of resources or faults happening while serving a request are not rare and must be handled carefully.

Asynchronous execution of processes brings lot of complexity from a computational point of view. Software synchronizations and locks of critical sections mainly cause this increase of complexity. Architectures based on multi-threads usually take care of synchronizing and protecting shared data to the extent of increasing the space complexity of the overall software. This kind of architecture self-encodes the state of the software in the form of a global state machine, for which threads are the functions that represents the transitions between those states. Bringing a small modification to the applications require also updating the underlying state machine mechanism, that is the all logic of the software. This kind of architecture clearly increases the space complexity of software and the time of development. Also modifications brought into an existing source code increase the chance of introducing new points of failure.

In a carrier-grade environment it is almost impossible to rely on this kind of architecture. For example, software upgrades don't necessarily imply upgrade of the all system or upgrade of all the running applications. These applications should be flexible enough and distributed in their design and their implementation to be upgraded process by process in order to satisfy the 99.999% uptime.

AEM brings this flexibility to software and reduces their space complexity especially in the case of multi-layer software architectures. It provides inside the kernel a generic framework to handle system events and dispatch them to the applications that did the registrations with only slight adaptations (event registrations).

Also, resource greedy applications could create a bottleneck or increase vulnerabilities of a node. AEM is centralized as a part of the Linux kernel and is able to control and restrict resource usage.

Handling of errors is quite important to provide reliability to carrier-grade systems. For example shortage of critical resources like memory must be handled carefully without killing the transactions or the applications. AEM tries to be as reliable as possible while serving processes and tries to guaranty the execution of event handlers even in case of internal errors as much as possible. For example, AEM always falls back onto the applications' decision when memory is not available for event handlers.

2.5 Performance

AEM targets performance in term of throughput and response time either in the case of streaming or transactional applications. Improving throughput or response time is a matter of scheduling and resource management.

In order to try to achieve performance for wide types of applications, AEM implements the job processing elements, executing in the kernel on behalf on processes. A job is the latest element in the chain responsible for activating the execution of event handlers. It is executed with a very small latency from wait queues or by the job dispatcher. Since it can be executing sporadically or periodically a job is well adapted for many types of applications.

Performance at the scheduling level is achieved by enforcing process priorities creating a tight affinity between a process and its events. A priority is associated to events during registrations and the scheduler uses this priority to make decisions. When an event is activated the corresponding process priority is raised according to the event's priority.

AEM implements a direct data copy mechanism when event handlers are to be executed in a new process. Direct data copy is necessary to prevent extra buffering in the kernel, which tends to aggravate resource consumption and penalize performance. This is achieved via the vmtable memory manager, which provides routines to transparently manipulate memory located in another process context. This is particularly useful in the case of network transactions that have to be received inside another process context.

2.6 Portability

Portability of applications is the main advantage behind an event driven methodology. The intermediate layer that is used to supply the management of events and that is usually built up from on a multi-threaded architecture is no more necessary since the operating system kernel provides a native support.

Cross-platform portability is not really an issue here since OSDL CGL is running Linux, but a standard API would really be a benefit to ease port of applications to other operating systems.

3 Requirements

3.1 Application Programming Interface

3.1.1 Event registration

Registration must be done by the application in order to receive notification of events. A unique identifier is returned to the application upon registration if successful otherwise a standard error value. It is possible to register an event more than once, for the same event handler with the same priority. It is then up to the event data completion mechanism to maintain the coherence.

There should not be no restriction regarding multiple registrations of the same event to execute different handlers. Although it can depend on an event type basis if this is possible.

Priority 1

3.1.2 Handlers API

All events handlers should provide their own parameter list but follow the same API scheme. Event data completion is achieved by supplying event handlers with all the required information in the parameter list of the user space callbacks. We mean by required information what is necessary to reference the event structures.

Priority 1

3.1.3 Existence of handlers

Event handlers must be defined by the applications for all of the required events.

Priority 1

3.1.4 Event identifier

An identifier is returned as a result of a registration or a standard error value in case of failure. This identifier is directly or indirectly usable by an application for accessing the structure describing the registered event.

Priority 1

3.1.5 Event priority

User processes can set a priority during registration. This represents the priority given to a particular registration and not the the event itself. It defines how fast the event handler should be executed on a global (scheduler) scope once activated. Priority is optional and a default one is assigned otherwise. Priorities can be set during or modify after event registration.

Priority 2

3.1.6 Executor agents

Event handlers can run inside the same context or a copy of the context of the original process. In either way, it is up to user processes to define this option during registration.

Priority 1

3.1.7 Low level control API

Primitives to enable, disable, filter, lookup, set time-out, clean events are provided through a system call for allowing a single thread to deal with them. Event registration returns a unique identifier as a reference for that purpose. The implementation should allow a user process to interact with the event subsystem unless
restrictions apply for other reasons (security, resource limit...)
Priority 1

3.2 Soft real-time

3.2.1 Event priority

A priority can be set during or after event registration. This priority should guaranty a fixed latency between event activation and the execution of the corresponding event handler [orders are to be defined].
Priority 2

3.2.2 Process priority enforcement

Process priority should be changed according to priorities of pending events [the algorithm to use is left to the implementation].
Priority 2

3.2.3 Direct process invocation

An asynchronous event mechanism performs direct invocation of execution agents (handlers) with the data for the events in parameters. An execution agent can be a process, a thread or a function (entry point) in the current execution context.
Priority 1

3.3 Scalability

3.3.1 Event system

Scalability is achieved by ensuring that all sub-components of the event subsystem are executed in a time-bounded interval in the worst case. Scalability cannot be ensure when blocking routines are used (select()...).

Priority 1

3.4 Performance

3.4.1 Event data completion

When an event handler requires memory allocation for event data completion, this is handled inside the kernel on behalf of processes. No intervention from user processes should be required to either allocate memory or to free this memory. An application should be able to use its own memory allocation scheme.

Priority 1

3.4.2 Direct Data Copy

Event data completion raises a problem when the executor agent type is a process. In that case the completion is happening in the parent process context whereas it should be reported in the child process context. In order to keep good performance of event handlers' execution, a direct data copy mechanism should be employed when building child process parameters.

This mechanism should be flexible to take benefits of some of the kernel facilities improving performance [like DIRECT IO].

Priority 1

3.5 Reliability

3.5.1 Guaranty of delivery

All activated events should guaranty to trigger their corresponding executor agent. No event should be dropped. Multiple occurrences should be managed internally.

Priority 1

3.5.2 Error handling

An application must be able to handle errors occurring in the event subsystem (like shortage of memory) while fetching data. Notification should be done in these cases and it should be left to the application to decide what to do.

Priority 1

3.5.3 Security

Security procedure could be set to control the following items:

- Types of system calls that can (or cannot) be executed prior running a handler,
- Type of events an application can register,
- Type of executor agents

Priority 2

3.5.4 Resource limit

Limits should be set on the following items:

- Number of events registered
- Number of simultaneous event activated
- Type of events
- Type of executor agents

Priority 2

3.6 General behavior

3.6.1 Queuing

Events need to be queued as they arrive in the system until the corresponding event handlers are executed. For that matter, it is performed for each process by the event mechanism if the controlled objects cannot pro vide queuing.

Priority 1

3.6.2 Atomicity

Only one event handler should be executed at a time in the same execution context.

Priority 1

3.6.3 Timers and periodic events

Should provide timers and periodic event monitoring.

Priority 1

3.6.4 Inheritance

A forked process can either inherit its parent list of events or start from a new fresh list. This is to conform to the semantics of forked processes.

Priority 2

3.6.5 Event dispatching

Dispatching of the same event data to different event handlers should be possible. This mainly depends on the underlying system to which events belong [i.e. TCP stack]. Otherwise it can be left to handlers to perform their own dispatching [scalability issue here???].
Priority 2

4 Supported Interfaces

[This section describes the supported interface either towards the user or towards the kernel implemented in AEM. This interface is flexible and is going to change in the kernel 2.6. One of the reasons is because it consumes many system calls entries. I propose to have a system call entry per family of functions. For example, sockasync() for the network, evtimer() for timers, fsasync() for file systems.]

4.1 Control functions

- [int] evctl ([int] job identification, [uint] operation, [ulong] arguments)

Control function. Used to control jobs and events related structures from user space. It returns 0 or a positive value on success, a negative value upon error.

- [void] enter keepalive ([void])

Enters an infinite loop in the kernel. Exits when it receives a signal.

4.2 Network API

These functions return an event identifier or a negative value upon error.

- [int] sockasync accept (<same as accept>, [ulong] flags for handler, [ulong] handler)

Creates an asynchronous event for the TCP accept method.

- [int] sockasync read([int] socket fd, [ulong] flags for handler, [ulong[handler)

Creates an asynchronous event for reading data on a socket.

- [int] sockasync close([int] socket fd, [ulong] flags for handler, [ulong] handler)

Creates an asynchronous event for the close operation on a socket.

- [int] sockasync sk ([int] socket fd, [ulong] flags for handler, [ulong] handler)

Creates an asynchronous event to monitor TCP DFA states for a socket.

- [int] sockasync sock ([int] socket fd, [ulong] flags for handler, [ulong] handler)

Creates an asynchronous event to monitor a socket state.

4.3 Timer API

This function returns an event identifier or a negative value upon error.

- [int] evtimer ([struct timespec] timer interval, [struct timespec] timer period, [uint] flags, ([ulong] han dler,)

Creates a periodic timer.

5 References

[aem] AEM - The Linux Asynchronous Event Mechanism Home Page, http://sourceforge.net/projects/aem/ and http://aem.sourceforge.net/.

[faq] AEM - FAQ, available at http://sourceforge.net/projects/aem/ and http://aem.sourceforge.net/.