ERICSSON

Asynchronous Events on Linux

Frederic.Rossi@Ericsson.CA

Open System Lab

Systems Research

Introduction

Linux performs well as a general purpose OS but doesn’t satisfy most of Telecom requirements.

Server platform operating systems must be:

- Linearly scalable,
- Have non-stop operations,
- Have soft real-time responsiveness.

$\Rightarrow$ Look at standard mechanisms

Introduction

Problems with the standard mechanisms

- Scalability:
  - select() and poll() are O(n) interfaces,
  - SIGIO requires de-multiplexing in O(n),
- Soft real-time responsiveness:
  - Real-time signals have fixed priorities,
  - RT signal priorities cannot be used to increase responsiveness,
- Reliability:
  - RT signals and signals cannot live in the same world: order is not guaranteed,
  - Signal delivery is guaranteed but not the number of signal delivered,
  - Latency of signal delivery.

Introduction

Problems with the standard mechanisms

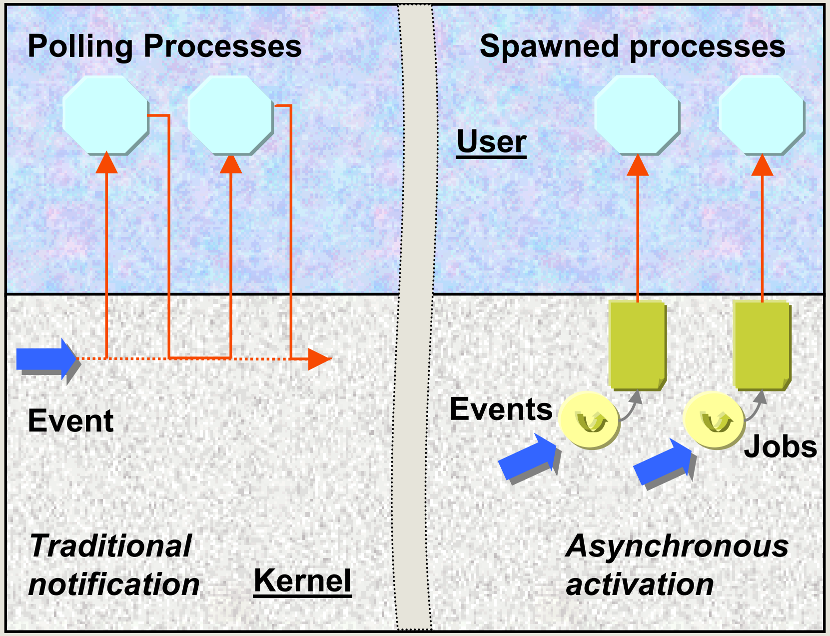
- Multi-threading
  - Standard mechanisms implies the use of multi-threading to handle multiple simultaneous connections,
  - Requires locking mechanisms for concurrency,
  - Increases scheduling latency,
  - Resources consumer: 1 thread per connection,
  - Difficult to port, to maintain. [threads] Implementation dependant,
  - Many libraries are not thread-safe,
  - Thread implementation is changing.

Introduction

Alternative mechanism

- Event-driven mechanisms:
  - One event per resource in input (socket, file, load, number of tasks…),
  - Registration for interests in some events. No specific software architecture is required,
  - Easier to program. Just provide call-backs for event handlers,
  - Used whenever concurrency between data is not needed,
  - Handlers for events are executed asynchronously,

Event-driven mechanisms



Synchronous vs. Asynchronous Execution

Asynchronous execution

- Handlers are executed asynchronously,
- This mechanism consumes no kernel resources, no CPU time.

Event-driven mechanisms

Existing asynchronous mechanisms

Microsoft I/O completion port (IOCP):

→ Completion ports are associated with descriptors,

→ Use of threads to wait for completion,

→ Applications are provided functions to get I/O completion packets from the IOCP,

→ Must provide a valid pointer and length to data location.

POSIX Asynchronous I/O (AIO):

→ Notify user processes upon completion of some operations,

→ Use of signals to notify users. Can use RT signals to take benefits of their priorities,

→ Works on (file, socket) descriptors,

→ Provides IOCP with RT signals. User processes must provide valid pointers to data location.

Event-driven mechanisms

Existing asynchronous mechanisms

Microsoft I/O completion port (IOCP):

→ Not Linux !!

POSIX Asynchronous I/O (AIO):

→ Not yet supported on Linux, not fully supported by other Unix!

→ Problems:

- Event completion is not transparent,
- Restricted to stream descriptors,
- Scalability,
- Soft real-time responsiveness.

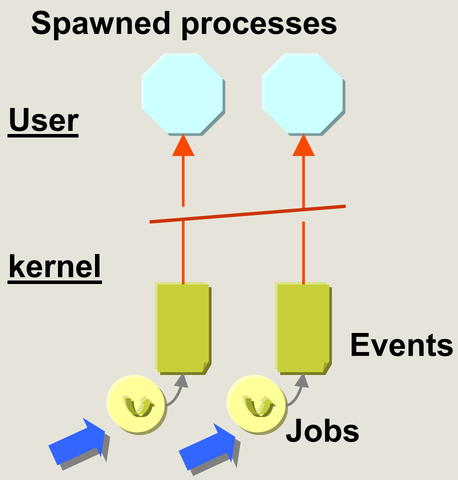
⇒ Our solution: the *asynchronous event mechanism (AEM)*

Asynchronous Event Mechanism

Architecture overview

General features

- ⇒ It's a Linux kernel enhancement,
- ⇒ Provides an event-driven methodology of development
- ⇒ Scalability and soft real-time responsiveness!



⇒ No multithreading; Executes user land call-backs from kernel space,

⇒ User processes request interest in some events, and then do something else; Non blocking mechanism,

⇒ Event loops are per process and handled from the kernel,

⇒ User call-backs are executed by context-switching the current process.

Asynchronous Event Mechanism

Architecture overview

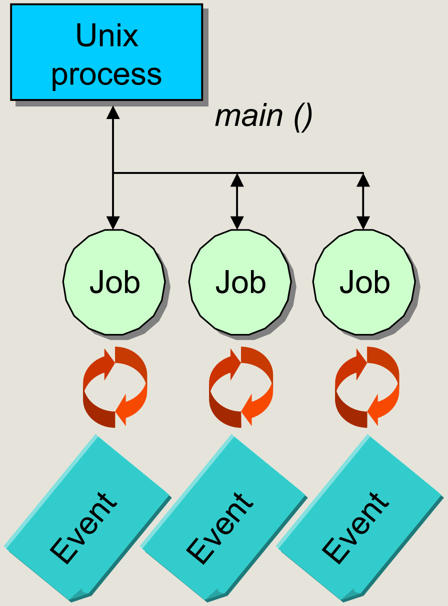
Technical features

⇒ A set of new system calls for event registration,

⇒ An *event* is an object in the system that is monitored periodically or awaiting for a change of state.

⇒ Some *Jobs* run at the level of interrupt handlers attached to some process to monitor events,

⇒ Example of events: asynchronous read on sockets, timers…



Jobs

Overview

- Jobs are processing elements executing inside the kernel at (soft) interrupt time,
- Light mechanism ; provide no execution context as opposed to processes or threads,
- Fast mechanism; jobs are high-frequency entities,
- Perfectly serialized ; a job run until completion before another one is executed,
- Benefit of SMP architecture ; jobs can execute in parallel,

$\Rightarrow$ Jobs are used for event activation

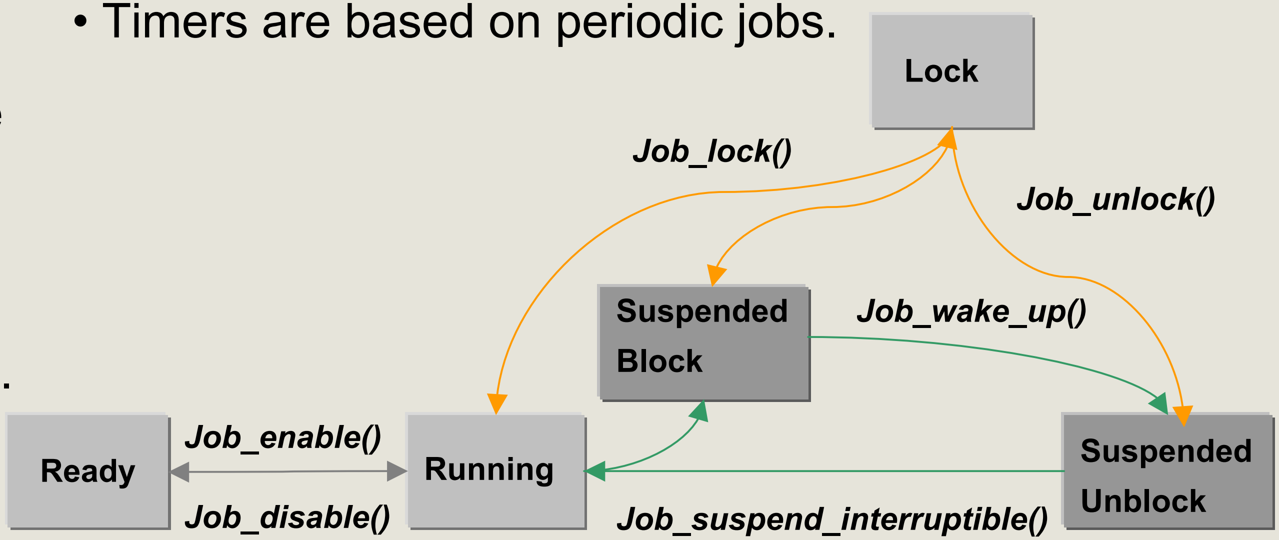
Jobs States

Periodic jobs

- These are jobs used to *periodi*cally lookup for event,
- Inserted by the dispatcher into the scheduler queue,
- Are defined by a *frequency* and a *timeout* values,
- Timers are based on periodic jobs.

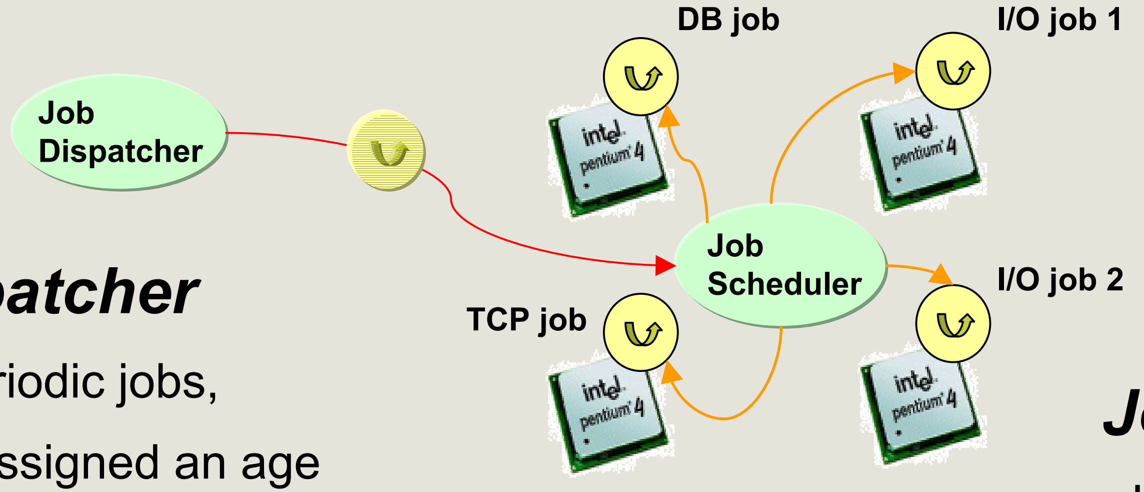
Reactive jobs

- These are jobs used to wait for some event,
- Explicitly awaken by the underlying implementation,
- Are not scheduled in order to reduce latency time of handler execution.



Job state automaton

Jobs Scheduling



Job scheduling

Job dispatcher

- Handle periodic jobs,
- Jobs are assigned an age spent in the queue,
- When a job's age reaches 0 then it becomes *ready* to execute,
- Insert ready to execute jobs into the scheduler queue.

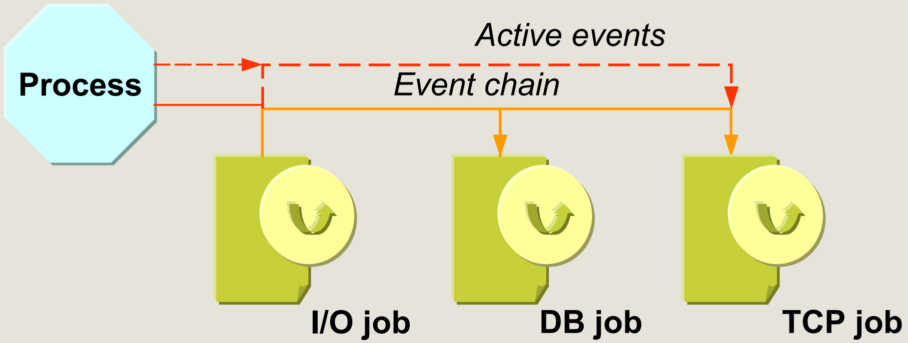
Job scheduler

- Update jobs' state according to the automaton's rule…
- Then execute the job,
- Jobs execute in parallel on SMP.

Events

Overview

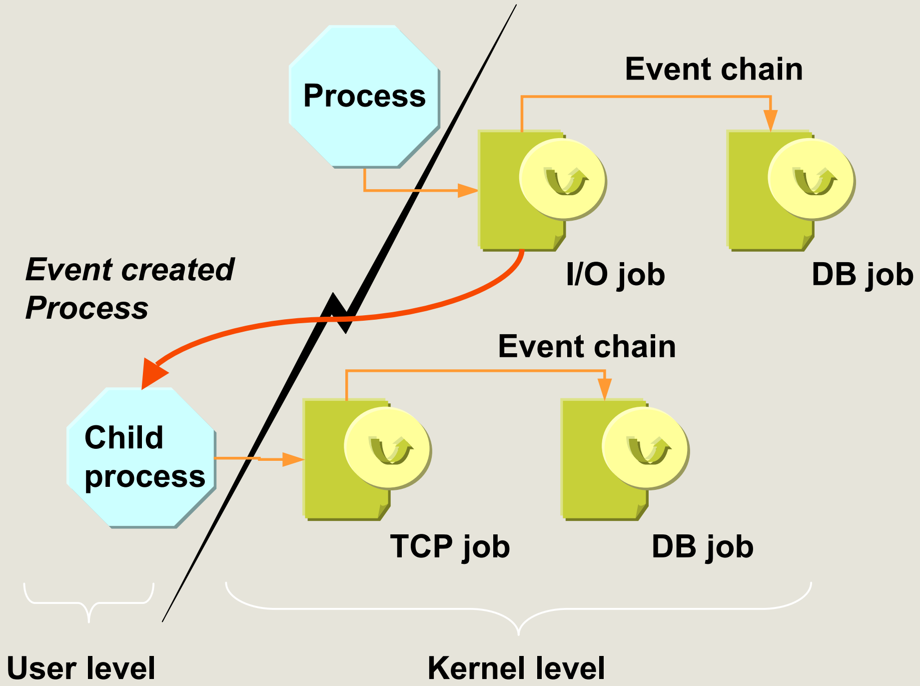
- An event defines the execution context for a user land handler and a job,
- An event defines its relationship with other events (sibling, child, parent),
- A list of active events is maintained for each process,
- Active events for scheduled processes are checked during each clock tick.



List of events per process

- When an event is activated the corresponding handler is executed,
- Scalable! no single thread of control and fine grained mechanism.

Events



Event Handler execution

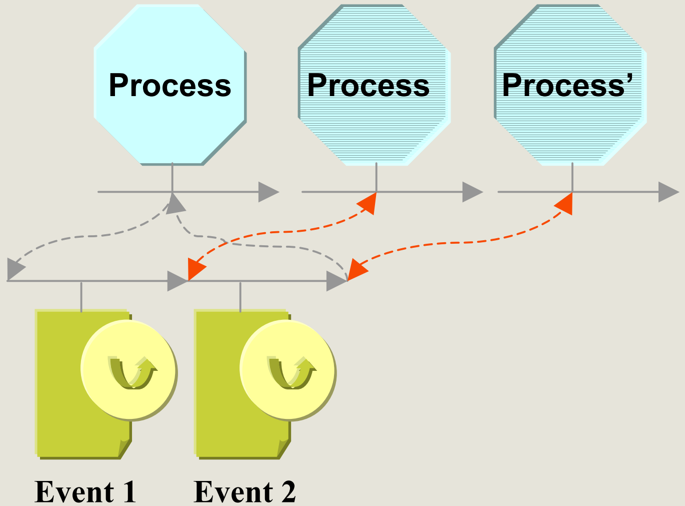
Event created processes

- New processes are created by event activations,
- Each process implements a *resource container* to handle event related data (jobs, handler arguments…),
- When creating a new process the parent can be told to not *wait* for its child when it exists.

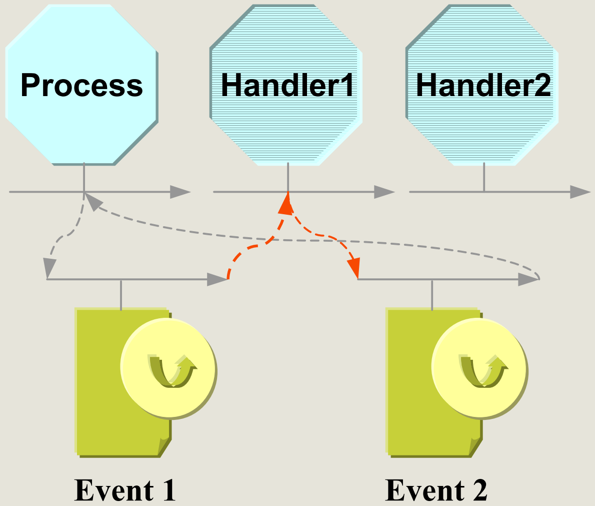
Events

Two event handler types

- Handlers can either be serialized like signals or be forked processes,



Forked event handlers



Serialized event handlers

Memory Management

Motivation

- In the POSIX definition for AIO a valid pointer must be provided by the application,
- In AEM we manage user process memory from inside the kernel. No need to pre-allocate memory from applications. This is handled at the time call-backs are executed,

⇒ This pool can be used as a *resource container* for event related data.

Memory Management

Motivation

We implemented a specific buddy allocator in order to:

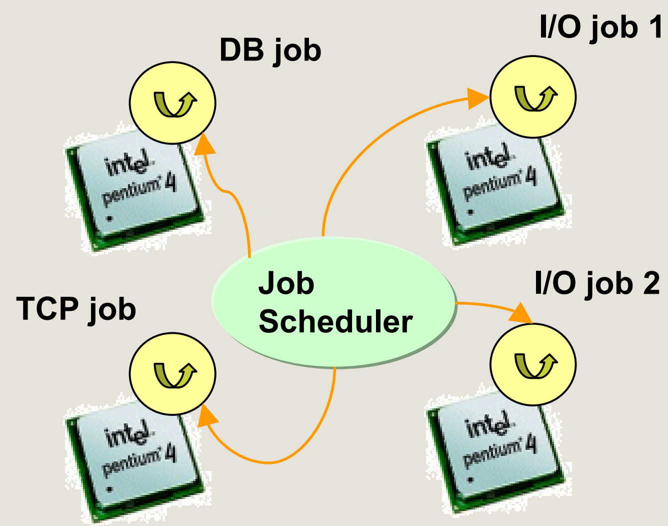
- → Prevent memory fragmentation, swapping and page faulting caused by successive allocations,
- → Encourage reuse of memory locations,
- → Allocate quickly,
- → No waste of resource. Memory space is requested when event handlers are executed,
- → Provides fine grained blocks of different size for the applications,
- → Provides big blocks to be used as pools,
- → The size of blocks fits with the event requests,
- → Can use mapping of user memory or direct copy to prevent a time consuming copy of data.

Scalability

Fine grained mechanism

- No list of port to scan (*select()* is linear in the number of *fd)*
- Event data completion; no need to lookup for information (like for SIGIO)

No single thread of control !



Soft real-time responsiveness

Soft pre-emption

- We use event priorities to increase process weights,
- So that it influences scheduling decision regarding process selection,

for a process *P*:

$$\Rightarrow srt\_priority\ (P) = \sum_{P} activated\ event\ priorities$$

and

$$\Rightarrow weight\ (P) = srt\_base + srt\_priority\ (P)$$

$\Rightarrow$ Load control problem…

Soft real-time responsiveness

Load control

- It is based on the total number of event handlers executed in the system during the last second,
- The *load* is the number of estimated events per process per time slice,
- All processes are influenced equally.

global estimation of event load,

$\Rightarrow$ *srt_base* = max [0, *cste* – *calc_decay (load)*],

and *calc_decay (load)* is the proportional increase of the load, *cste* is the maximum allowed,

$$\Rightarrow load = \frac{\sum_{\text{elapsed second}} events}{\left(Hz \cdot \sum_{\text{now}} tasks \right)}$$

Thus conservation of order is insured in the same time interval,

$$srt\_priority\ (P_1) < srt\_priority\ (P_2) \Rightarrow weight\ (P_1) < weight\ (P_2),$$

Soft real-time responsiveness

Process time slice allocation

- It is computed when selected for the first time,
- Its allocated quantum of execution is proportional to its number of events,
- So that it gives a chance to other processes,
- Small quantum time values are allocated to event handlers to improve responsiveness ($\leq$ 20 ms).

for a process *P*:

$$\Rightarrow time\_slice\ (P) = \frac{\min[weight\ (P), cste]}{cste}$$

Interface

Actual AEM user interface

- Socket interface,
- Timer interface,
- Control functions.

eventdesc_t request (handler_t *handler*, unsigned long *evflags,,,)*

- *ret* is an event descriptor if the request is successful,
- *ret* is negative if an error occurred.

evflags tells how the handler is going to be executed:

- EVF_ONESHOT
- EVF_FORK
- EVF_NOCLDWAIT

void handler (eventdesc_t *ed*,,,)

Interface

Socket interface

```
main ()
{
    int sfd = socket (…);
    bind (sfd,…); listen (sfd,…);
    id = sockasync_accept (h_accept, 0, sfd);
    while (1);
}
```

```
void h_accept (jid_t id, int sfd, int nfd)
{
    id = sockasync_read (h_read,
                 EVF_FORK|EVF_CLDNOWAIT, nfd);
    ….
    id = sockasync_close (h_close, EVF_ONESHOT, nfd);
}

void h_read (jid_t id, int fd, char *data, int len)
{ … }

void h_close (jid_t id, int fd)
{ … }
```

Interface

Socket interface

```
id = sockasync_sk (h_sk_state, EVF_FORK, sfd, TCP_ESTABLISHED);
id = sockasync_sock (h_sock_state, 0, sfd, SS_CONNECTED);
```

```
void h_sock_state (jid_t id, int fd, int state)
{ … }
```

```
void h_sk_state (jid_t id, int fd, int state)
{ … }
```

Interface

Timer interface

- Based on the implementation of periodic jobs,
- Must be provided with an interval and an optional period.

```
main ()
{
    struct timespec interv;

    interv.tv_sec   = 1;
    interv.tv_nsec = 0;

    evtimer (h_timer,0,&interv,
                 NULL,);

    while (1);
}
```

```
void h_timer (int id, struct timespec to)
{
    printf ("Timer event desc. %d: %us: %uns\n",
                                   ed, to.tv_sec, to.tv_nsec);
}
```

Interface

Control function

- Control function: *evctl,*
- Somewhat equivalent to *ioctl*,
- Controls event properties from the application.

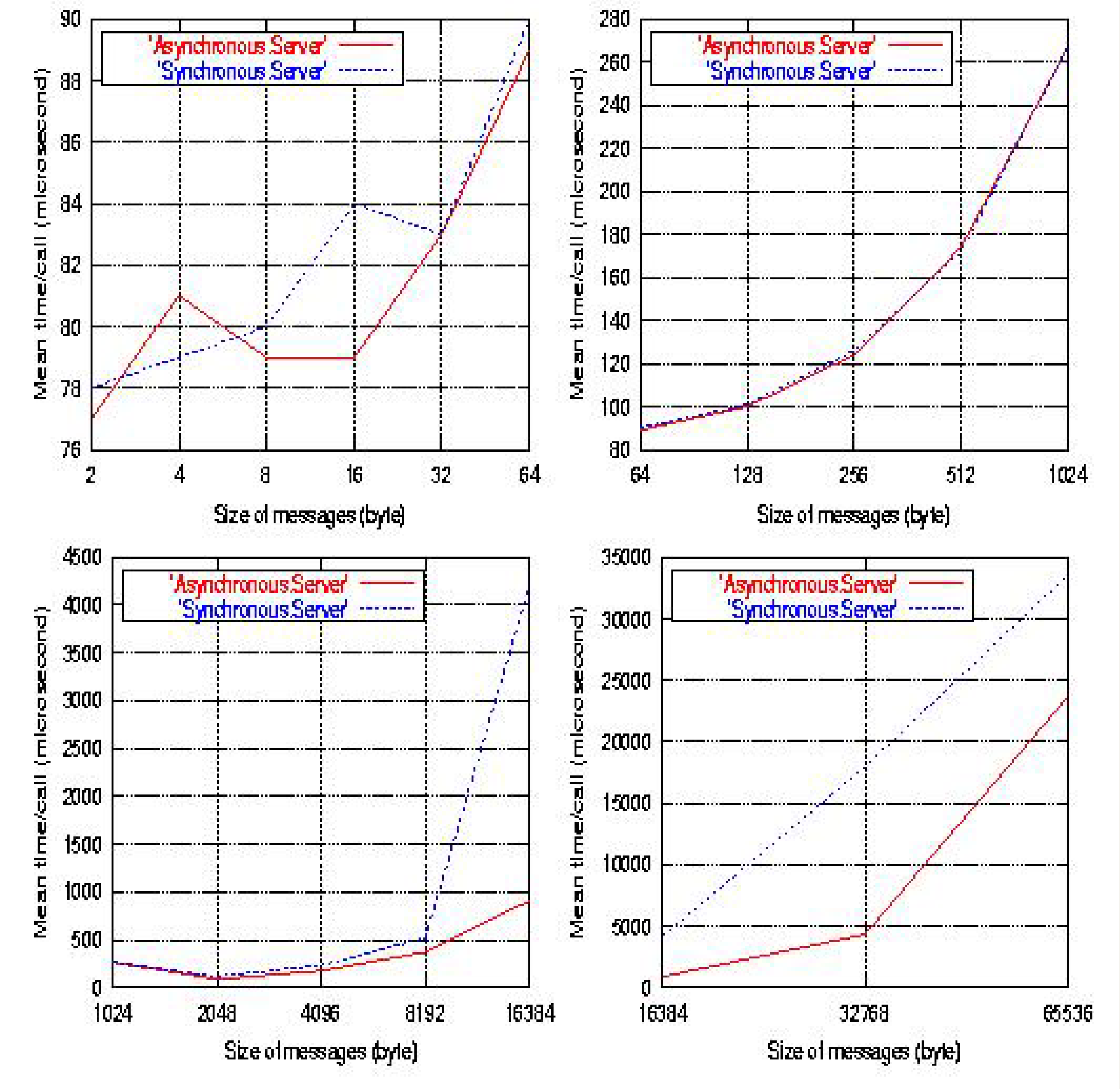
```
int id = sockasync_sk (h_connected, EVF_ONESHOT, sfd, TCP_ESTABLISHED);
if (id<0) {
    perror ("sender sockasync: ");
    exit (1);
}

evctl (id, EVJOBPRIO, JOB_HIGH);
```

Event Id — Control flag — Argument pointer/value

Performance

Context-switch measurement

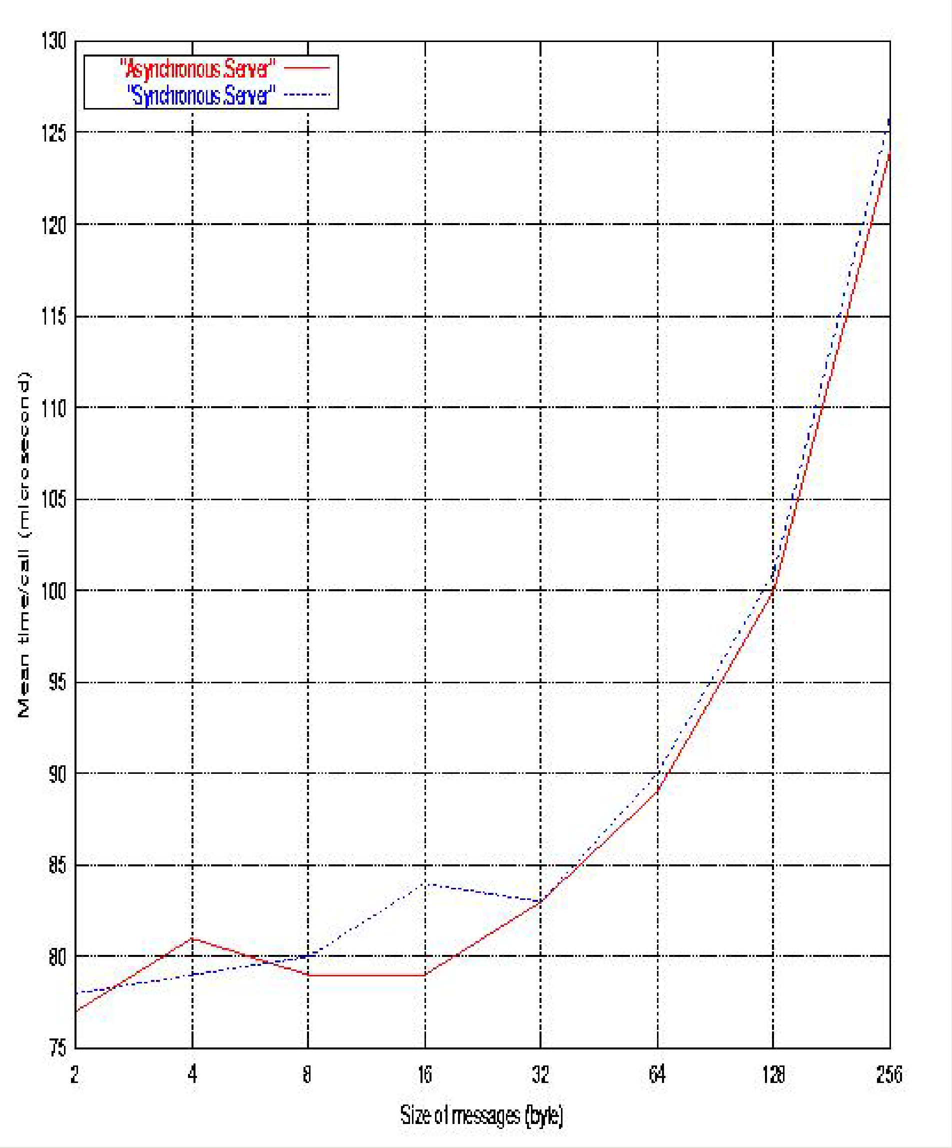
- Purpose is to run the AEM at full speed for the worst case,
- One opened socket,
- 50,000 ping-pong messages between two remote processes,
- Size of messages vary between 2 and 65536 bytes,
- Client is synchronous in both cases,
- Server is asynchronous with serialized event handlers. Not forked.



Performance

Context-switch measurement

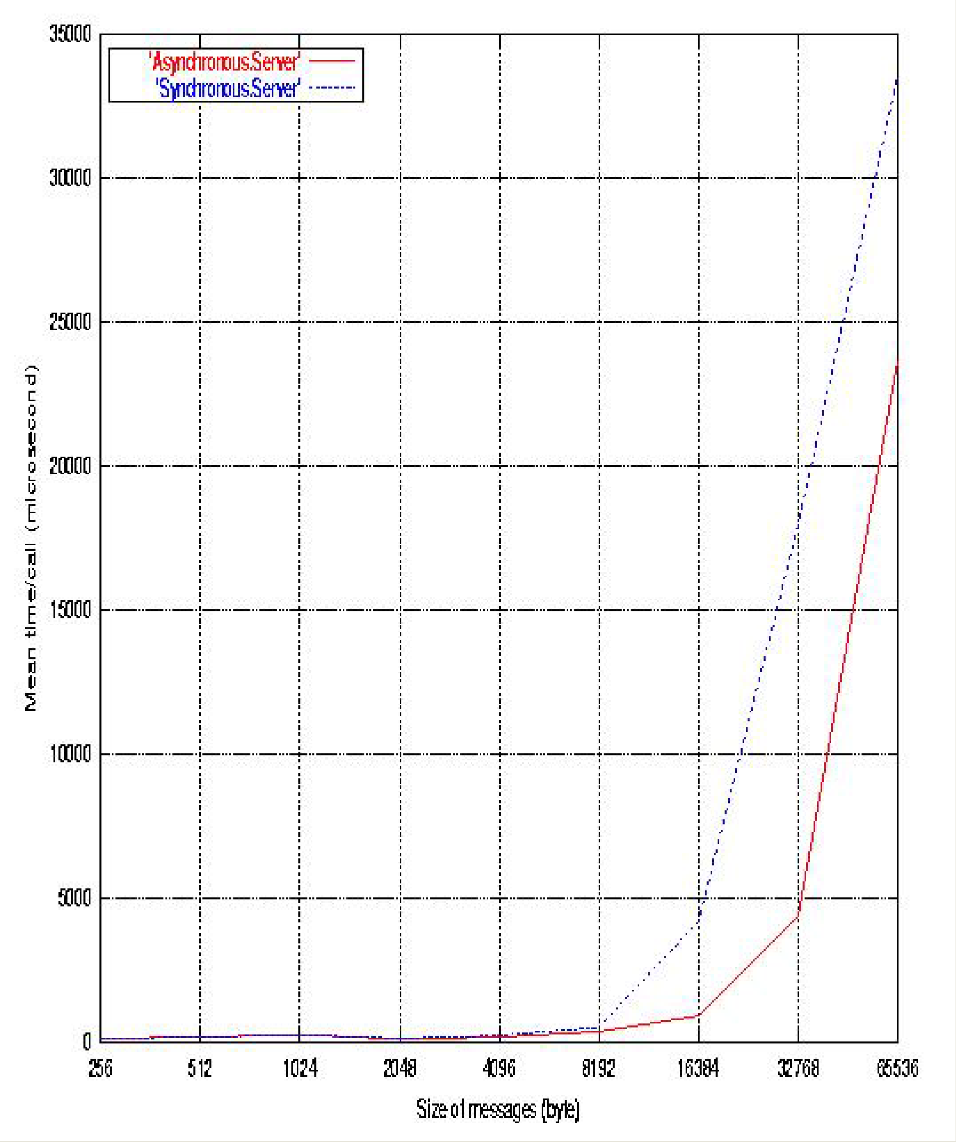
- Same graph for messages between 2 and 256 bytes with a larger scale.



Performance

Context-switch measurement

• Same graph for messages between 2 and 65536 bytes with a larger scale.



Conclusion

- *New model in the Unix world,*
- *Implemented in the Linux kernel 2.4.6,*
- *Ensure scalability,*
- *Ensure soft real-time responsiveness,*
- *Provide a secure event-driven interface to Linux for the development of highly available applications,*
- *Flexible: expandable with new system calls,*

Thank you for your attention !