Porting to Linux the Right Way

Migrating data between kernel and user space

Neil Horman
Red Hat
nhorman@redhat.com

Abstract

Linux has grown to be a major development platform over the last decade, often becoming the primary target for many new applications and appliances. Of course, businesses always wanting to stay current; the rate at which software has been ported to Linux has also gone on the rise. Often this is a trivial matter, especially in environments in which the development model is similar (AIX to Linux, Solaris to Linux, even Windows to Linux). However, there are environments (particularly in the embedded space) in which porting often becomes difficult. A stronger coupling of application and driver, coupled with a "just get it working fast" mentality, invariably leads to substandard porting efforts which result in products with degraded performance that leave developers and consumers alike with a bad taste in their mouths. This paper seeks to ease some of that porting effort by focusing on what has been one of the most often mis-ported areas of code: the user space / kernel space boundary, specifically the movement of data between these domains. This paper will discuss in general terms: the common monolithic application model most often associated with embedded systems development; its refactoring when porting to Linux; the modeling and description of data that must be passed between the refactored components when porting to Linux; and the selection of an appropriate mechanism for moving that data back and forth between user and kernel space. In so doing, the reader will be exposed to several mechanisms which can be leveraged to achieve a superiorly ported software product that provides both a better customer experience and a greater confidence in Linux as a future development platform.

1 Introduction

Linux has seen an almost meteoric rise in popularity over the past several years. Rather by definition, this increase in popularity has drawn developers to consider Linux as a target platform in many market segments, from servers to appliances, to small embedded solutions. Of course, along with the interest in new development on Linux comes the desire to bring older software to Linux, in an effort to leverage already existing products in potential new market spaces at reduced costs. This paper focuses on some of the pitfalls that befall developers seeking to port applications to Linux from alternative operating systems. In particular, it seeks to address the difficulties faced by embedded developers seeking to formalize the user space / kernel space split in applications which formerly were more tightly coupled in that respect. It seeks to do this by providing an overview of how the user / kernel space boundary is defined, and how to move data safely and efficiently between the two domains.

2 Defining the user / kernel space boundary

Prior to discussing the mintuae of moving data back and forth across the user / kernel space boundary, it is useful to better understand what the user / kernel space boundary is. Nominally, when discussing this particular separation, many people encounter this high level diagram:

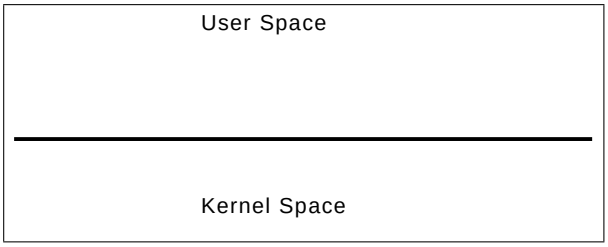

Figure 1: The most seen & worst overview of user / kernel separation

This is the sort of visual aid that is understood best by people who already understand how this separation

works, and consequently have no need for a visual aid. For everyone else, however, this is a rather lackluster description of how the two execution domains are separated. The separation of user and kernel space can be best described as having the following characteristics:

- A separation of accessible address spaces, except where granted by appropriate system calls.
- A mechanism which allows access to the functionality of kernel space code, preferably without granting the user space context direct access to the memory holding the code for that functionality.

Different architectures provide various mechanisms for enforcing the above points to various degrees, but the end result remains the same: the user / kernel space boundary enforces the isolation of an application from operating system services, forcing applications to make use of those services only in the ways defined by the operating system. This provides both stability and consistency of use.

3 Mapping strongly coupled applications to user / kernel space

Often, especially in the embedded space, the operating system is treated as a convenience library, rather than a environment to target. The application is paramount, enjoying complete or near complete control over the system's resources. In such designs, the software architecture can often be characterized as such:

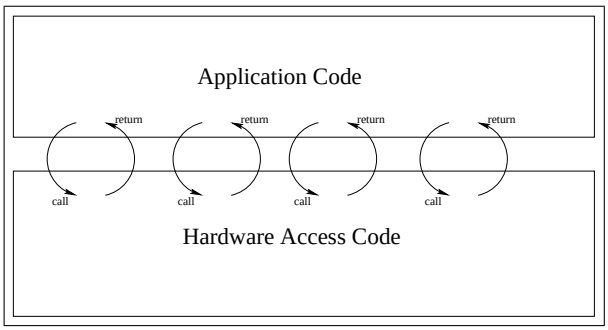

Figure 2: A common design approach to a tightly coupled embedded application

Strongly coupled applications, to whatever degree they manage to separate true application code from hardware-specific access code, interface the two with an API that can be difficult to separate. The API is often characterized by some of the following aspects:

- Customized for use by an application;
- Assumes shared memory space with application;
- Assumes common system behavioral characteristics on both sides of the API.

While some designs attempt to adhere to some level of compliance to a standard API, many do not, and the resulting temptation to speed the design cycle by taking various shortcuts invariably leads to porting difficulties when an operating system more strictly enforces the use of a pre-defined API. This leads to the inevitable question: How does one convert a system with application code and hardware access code that is tightly coupled into a system that is capable of operating in separate memory contexts using an enforced set of access methods? The answer is less difficult than many think. Generally speaking, one can accomplish this goal by:

1. Selecting/creating an API to use as a user / kernel shuttle;
2. Modeling the data that needs to cross that boundary;
3. Selecting the appropriate kernel APIs to implement that transport.

Note this will not provide the most efficient interface possible, but it will provide the most stable interface possible for the system being ported, which is really the goal here. With the time saved on avoidance of future bugs, one can focus on efficiency improvements later on.

3.1 Selecting/creating an API to use as a user / kernel shuttle

The first step in adapting an application from an strongly coupled environment is to select a good location within the system to segment it. There are many factors which can affect this decision, and will vary largely dependent on the system in question. Systems which honor their defined APIs will be more conducive to separation

than those which do not. "Thinner" APIs will make for quicker work than those with larger sets of functions. There are many facets to selecting an appropriate API for splitting an application during a porting effort, but the items below should be focused on first and foremost:

1. **Cleanliness.** Above all else, a candidate API should provide a boundary through which data can pass only in narrowly defined channels, and at approved times. APIs which provide or use information that exists external to the API implementation should not be considered unless such uses may be corrected. For instance, the use of a global variable or resource within an API's implementation is bad, simply because external access to such a variable will no longer be possible from outside the API after the port.

2. **Complexity.** The simpler an API is, the better, as migrating its implementation to one which exists across memory domains will become much easier. APIs which embody a significant amount of stateful information are difficult to manage, as that state may potentially need to be tracked, replicated, and kept in sync in both kernel and user space. Conversely, APIs which are simple, pass less data, and have fewer return codes will make for an easier split.

3. **Functional Requirements.** Be careful to closely examine an API's implementation when selecting it for use as the point where an application is separated from its hardware-dependent components. Sections within an API (and its dependencies) may require behaviors or resources that may not be available in either kernel or user space without additional work. Selecting an API that leaves a bit of code in the application that blocks while waiting for an interrupt to fire will require additional retrofitting to function properly after the port. Likewise, consider an API implementation which creates a thread that spins, polling for data. Moving it into kernel space will result in horrible inefficiencies resulting from differing scheduling behavior down in the kernel under various conditions.

4. **Size.** The more functions an API implements, the more difficult it will be to port. Quite simply, there is a quanta of work to be done for each function call implemented in an API; therefore there is a corresponding increase in the porting effort when a larger API is selected for this split. Hence, while not always possible, "thinner" APIs typically result in easier porting efforts.

5. **Volume.** Keep in mind how heavily a given API is used by an application, and under what circumstances. APIs containing very few and simple function calls are tempting to select as a candidates for a split point, but if the application must call functions in this API thousands of times to complete a given task, there will be a performance impact. Bear in mind that transitions between user and kernel space have a cost in terms of pipeline flushes, TLB and L2 cache flushes, etc. (which varies from architecture to architecture). The number of times you need to make that transition will have a significant impact on your system's performance.

While clearly one will have to make compromises when selecting a point at which to separate a software system, the above are the most important aspects to keep in mind. Neglecting any one of these aspects, while perhaps further minimizing your time to complete a port, will result in a system with stability and performance which will be at best an approximation of the original system on the original OS.

3.2 Modeling data transfers across the user / kernel space boundary

Having selected a point at which to segment the software system into a user space component and kernel space component, a developer must now re-implement the internals of that API such that data is transparently sent from user space to kernel space and back at appropriate times, and with previously provided guarantees on the data's integrity, format, etc. In most use cases, understanding the data transfers profile is fairly straightforward, but modeling data transfers in some cases can be non-obvious. All data transfers can be described in three aspects: Timing, Quantization, and Access.

3.2.1 Timing

The timing of a data transfer here refers to when data transferred across an API is acted upon by either a user of the data or the implementation of the API.

1. **Asynchronous application to driver.** These are message-based transactions. Data is accumulated into a discrete bundle of arbitrary size and passed to the driver. The return code to the submission of this transaction typically provides status of the submission effort, but not the result of the action which the data submitted embodies. Any action which the driver may take on the submitted data is deferred to an arbitrary later time, and results of those operations may or may not reported back to the application via a separate data transfer.

2. **Asynchronous driver to application.** These are the inverse of the the previous transfers. They send similarly formatted messages, only in the reverse direction. Messages generated by the driver code are queued for reception by the application through various APIs. It is interesting to note that the various APIs available for these transactions have various restrictions which do not exist in their counterparts. Those limitations will be noted in the next section. Such transactions may be driven by the previous transaction type, or may be generated independently by any number of conditions or events.

3. **Synchronous.** These are transfers which are, as the name implies, synchronous. Data passed via this profile is handed to the driver, operated on, and released at the conclusion of the API call. Return codes typically embody the result of whatever operations the driver performed on the passed data.

3.2.2 Quantization

The quantization of a data transfer refers to how the data is viewed by the API user or the API implementation. Some APIs handle data transfers as distinct units, atomic in their transfers, while others treat data as an arbitrarily sized sequence of bytes, re-segmenting the data in whatever manner is convenient.

1. **Stream data.** Stream data is unbounded. While it may contain a begin, end or other control marker, it does not normally distinguish data record boundaries. A transfer of data may contain an arbitrary number of bytes; users of the API or the implementation are not guaranteed to receive any particular amount of data during any typical transfer. Nominally, however, stream-type data is guaranteed to maintain its order (all bytes transferred in a stream are handled in the same order they are sent).

2. **Packet data.** Packet data is bounded and quantized. While the size may not remain constant from one API call to the next, packet-style data is always treated as an atomic unit. Packet data may or may not guarantee ordering of data.

3.2.3 Access

Access describes how an API makes data transfers available to either its implementation or its users. These methods are well known and well understood by all developers, but it is useful to provide a reference here so that the different methods are kept in mind when modeling your data transfers and selecting an API to use during your porting effort.

1. **Value.** Data is passed into the API and a copy is made for use internal to the API. Changes to the data made internal to the API are not visible to the user of the API.

2. **Reference.** Data itself is not passed into the API directly, but rather by a memory reference to the data's location. The user of the API and its implementation share access to the data and may need to co-ordinate that access to avoid corruption.

Using these three aspects in all their combinations, it is possible to completely model how data is moved between a user of API and its implementation. Once that is determined, the task of selecting an pre-defined kernel interface API to act as a transport for splitting a legacy software project into kernel and user space segments becomes much easier.

3.3 Selecting the appropriate kernel APIs for data transport

Now that we have described how data can be passed through APIs in general terms, we can describe the various available kernel interface in those same terms so that we can select an appropriate interface to better adapt to our existing model to optimize our porting effort. There

are several kernel interfaces to choose from, each of which offers a different set of data transfer characteristics. Each kernel interface is documented here. There are only three major interfaces from kernel space to user space: the character driver interface, the socket interface, and the signal interface. Each provides an API that allows a developer to implement various different data transfer models. Note that there are other APIs which allow for various types of data transfer (the file system interface, the block driver interface, etc.); however, the three aforementioned interfaces cover completely the types of data transfer listed in Section 3.2. While the other interfaces provide excellent mechanisms to design various types of systems, the above listed interfaces provide generic data transfer resources that allow for any application to be relatively easily split between user and kernel space.

3.3.1 Character Driver Interface

The character driver interface is seemly the de-facto interface that developers select—a transport API when segmenting a monolithic application during a porting effort. It offers several data transfer models, and as such is reasonably flexible, is fairly well understood when coming from other operating environments, and is fairly easy to implement. The user space API consists of the following elements:

1. **open**
   int open(const char *pathname, int flags)
   This is the well known and understood call to establish access to a filesystem object. Nominally, this call is used to open a regular file. However, under the Linux design model, this function can also be used to obtain access to a device through a device special file (created with the `mknod` utility or through the `udev` system). Special device files contain a major and minor number, which are used to resolve which device driver will handle operations issued to the device. Open accepts a path (nominally for the porting purposes here) to a device special file, and a set of flags which define various characteristics of the communication to the device (read/write permissions, auto close properties, etc.). It returns a descriptor to the opening process; this descriptor is used in later calls below.

2. **read**
   ssize_t read(int fd, void *buf, size_t count)
   The read call provides a synchronous, stream-oriented byte sequence passed by value to the user. Kernel modules that implement the character driver interface are notified immediately when a user space process issues a read request, and is required to return either an error code, or a data buffer (which will be copied into the process context) of a size equal to or less than the size passed in during the request.

3. **write**
   ssize_t write(int fd, const void *buf, size_t count)
   The write call provides the converse operation of the read call. It similarly provides an synchronous interface for passing stream-oriented byte sequences by value. The difference of course is that write calls allow a user process to pass data from a process to a kernel module. Unlike **read**, however, a return code is the only thing returned to the user. The conditions for success or failure in this call are dependent on the kernel module implementing the driver the data is passed to, as it what is done with the data when it arrives at the driver.

4. **ioctl**
   int ioctl(int d, int request, ...)
   `Ioctl` is the Swiss army knife of the character driver interface. It allows an arbitrary-length list of data items to be passed by value down to the driver associated with the descriptor passed in as the first argument. The flexibility and open format in the amount of data that can be transferred in this API call makes this a very attractive interface to implement the segmentation of application during porting, but it should in fact be used sparingly. Because the remaining arguments on the function are variable, only weak or non-existent type checking abilities ensue; the only way to properly interpret the arguments is via the request argument, which shares name space with every other driver layer between the application and the driver itself. While this call can be very useful, it can also introduce subtle and difficult to detect errors, and as such should be used carefully.

5. **mmap**
   void *mmap(void *start, size_t length, int prot, int flags, inf fd, off_t offset)

The `mmap` call is far more useful than developers normally give it credit for. The `mmap` call allows a user-space process to specify an address space 'hint' along with an open file descriptor and offset from that descriptor. In response, the object associated with that descriptor will map the specified length of data into the requested address range. This allows the `mmap` call to be categorized as an asynchronous kernel-to-user-space data transfer, passing packet data by reference. This call is nominally associated with regular files, in which the files contained data is mapped into process address space, allowing for direct access. What is not nominally recognized, however, is that `mmap` can be used equally well for any arbitrary device. The offset argument is passed directly to a driver, and used as an arbitrary handle, in which the driver passes back whatever data is required. For example, a character driver can be implemented to pass a stream of handles to a process via the `read` call, which can then be used in a sequence of `mmap` calls to access other out-of-band data. This provides a more efficient transfer mechanism for large volumes of data by only passing handles by value (which are much smaller than the requisite data they reference).

6. **munmap**
   int munmap(void *start, sizei_t length)
   This is of course the inverse of the previous call. It allows a block of data previously passed by reference to be released by an application. The memory referenced becomes inaccessible to the process, and the driver is informed of the release operation so that any needed clean up can be performed.

7. **close**
   int close(int fd)
   This ends a connection to a device driver/kernel module, and provides the driver the opportunity to clean up any remaining resources associated with that process connection.

The kernel space API consists of a registration and de-registration function, and several ancillary functions which map one-to-one to the user space API, each called directly in response to a user space call of the same type:

1. **register_chrdev**
   int register_chrdev(unsigned int major, const char *name, const struct file_operations *fops)
   This is the major kernel hook which allows you to register a special character device file to the kernel. The major parameter allows you to specify a major value that is matched against any special character mode device file opened in user space. A file opened containing a matching major number will be directed to this module via the function pointers passed in to the registration routine via the `file_operations` structure.

2. **unregister_chrdev**
   void unregister_chrdev(unsigned int major, const char *name)
   This is of course the converse of the above function. It allows kernel code to disconnect an association between a driver module and a major device number.

3.3.2 Socket Interface

The socket interface is far less often considered for use as an API with which to shuttle data between kernel and user space, but it should be. Highly flexible, and fairly well understood from an application standpoint, the socket API allows a developer to move data between user and kernel space in a variety of ways and formats, using both custom built protocols, and (perhaps more notably) using already existing protocols. While not often suggested, the infrastructure to create and manage sockets with the kernel has been in place for some time, making it possible to write a kernel module which can simply open a TCP, UDP, or other protocol socket, and accept incoming data from user space by sending to the opened port via the `localhost` address. Likewise, the Netlink protocol family has existed for some time (arguably in relative obscurity), for the sole purpose of connecting user space processes with kernel space services. Netlink also provides the added capability of dynamic sub-protocol registration (via the generic Netlink infrastructure), which allows for dynamic service discovery, making porting efforts even easier.

The API for working with sockets in user space is very mature and well known. While this list is not exhaustive, it enumerates the core functionality of the API:

1. **socket**
   int socket(int domain, int type, int protocol)

The socket call, like the open call, allocates a communication channel to a peer. Depending on the use of the subsequent calls, the peer could be a remote host, another process on the local host, or a service in the kernel. The domain argument specifies the address family (which specifies the format in which the peer to which you will connect is addressed). The type generally specifies whether the connection to the peer will be passing stream-oriented data or packet-oriented data, and the protocol specifies the transport layer protocol that the connection will use. While it is possible to create your own protocol (which this call can then provide access to user space to), given that the effort here is to simply migrate data between user and kernel space, it is far more efficient to simply use IPv4 (The AF_INET family), IPv6 (the AF_INET6 family), or the netlink protocol (AF_NETLINK). Any one of these protocols will allow a user-space application to take full advantage of all the features of this API for the purpose of data transfer to kernel space.

The socket call either returns a negative error message, or a positive value that represents a handle to use in subsequent socket API calls.

2. **bind**
   int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
   The bind call operation is somewhat specific to the address family, type, and protocol specified in the socket call. Generally speaking, the bind call associates a socket with an input filter. The format of the filter is specific to the protocol selected. For example, the AF_INET family allows you to bind your socket to an input address and port, so that you will only receive frames on a certain interface. AF_NETLINK, in contrast, allows you to specify a bitmask of multicast groups that you might receive frames on, in addition to messages directed at your specific process ID. This filter is passed in via the addr pointer.

3. **connect**
   int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)
   The connect call associates a socket with a peer address. Like bind associates a socket with a local endpoint, connect establishes the peer that the socket communicates with. This operation is specific to the address family and protocol which were specified in the call to socket above. Some protocols, like TCP, communicate with the remote endpoint to establish a connection, while others simply record the address of the remote endpoint.

4. **send/sendmsg/sendto**
   ssize_t send(int s, const void *buf, size_t len, int flags)
   ssize_t sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)
   ssize_t sendmsg(int s, const struct msghdr *msg, int flags)
   These functions provide various methods for the asynchronous transfer of data from user space to kernel space, passed by value, in either a stream- or packet-oriented format. There are three variants of the send routine, as different connections find different implementations more useful than others. Note that the send routine omits a remote address, which means the remote peer was specified with a prior call to connect. Conversely, the sendto operation allows for data to be sent on an unconnected socket, with each call specifying the recipient address (allowing one socket to communicate with multiple peers). Sendmsg is a variant of sendto, but uses a msghdr structure, which allows for a series of non-contiguous data pointers to be sent at once. Interestingly, all three calls may be used to send either packet data or stream data. The ability to re-segment data to split or merge data as it was sent from user space is encoded in the specific protocol as selected in the socket call.

5. **recv/recvmsg/recvfrom**
   ssize_t recv(int s, void *buf, size_t len, int flags)
   ssize_t recvfrom(int s, void *buf, sizei_t len, int flags, struct sockaddr *from, socklen_t *fromlen)
   ssize_t recvmsg(int s, struct msghdr *msg, int flags)
   These functions are the counterparts of the above send routines. They allow an application to poll a socket to see if any data is available from the peer(s) which the socket might be communicating with. Data transfer is asynchronous with its arrival at the protocol implementation in the kernel, is passed by value, and can be either stream- or packet-oriented. Overall, the operation is identical

to the `send` counterparts.

6. **setsockopt/getsockopt int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);**
   int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);

   These two calls allow for a user application to fine tune the operation of the selected communication protocol. As noted a socket can have behavior tuned at various levels (generic socket, protocol-specific, transport-specific, etc). If you are writing a custom protocol, these settings can adjust anything the developer would like (the level parameter name-space is global, but by defining a new level, the `optname` parameter becomes unique, so any number of options may be defined, unlike the `ioctl` call). Any amount of data may be passed by value via the `optval` pointer, but its interface is limited and inefficient.

7. **close int close(int fd)**
   Like the `close` call in the character driver, this call simply disconnects the descriptor in the user space application from its peer.

The kernel interface operates much like the character device kernel interface, with some enhanced abilities for fine tuning. There are two main registration and deregistration functions which allow one to add both an address family and a protocol, allowing for a larger set of communication methods. Nominally, however, the use of sockets as a data transfer mechanism doesn't require the creation of a new protocol. Developers looking to port applications to Linux and split their software into a kernel and a user space component are highly encouraged, for the sake of simplicity, to simply utilize an existing protocol for communication, such as UDP, TCP, or netlink.

3.3.3 Signal Interface

The signal interface rounds out our kernel / user space data transfer methods. The signal interface provides a data transfer model that provides one thing that the other interfaces do not. The other interfaces may provide data asynchronously or synchronously, but all require polling to retrieve that data (via the `read` or `recvmsg` family of calls). The `signal` interface provides true asynchronous data delivery, requiring no additional action to receive data beyond registering a reception function. Also, the signal interface is normally not used alone, but rather in conjunction with one of the other interfaces to provide a complete data transfer system when splitting a legacy application between user space and kernel space.

The signal interface is enumerated below:

1. **signal/sigaction**
   sighandler_t signal(int signum, sighandler_t handler)
   int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)
   The `sigaction` and `signal` calls both associate a signal value with an action, as defined by the handler pointer. The handler pointer contains a function pointer which is defined within the application, which is called each time a signal is delivered to the application process. The two calls are effectively identical; the `sigaction` call simply provides a few more options for fine-tuning signal delivery behavior. The `signum` parameter enumerates the signal value with which the action is associated. Some signal values are predefined (SIGINT, SIGSTOP, SIGKILL, etc.), and may have actions which are pre-defined. Other signals are generic and can have actions which are specific to the application being run (SIGUSR, etc.). Some signals have at-most-once semantics (multiple signal deliveries result in only one call to the action handler between certain delimiting events. Others ranges can queue their events creating a call-per-event model.

2. **sigpending**
   int sigpending(sigset_t *set)
   This call allows a user to determine which signals may be pending for delivery to the application. During periods when the application may block the delivery of signals, this calls provides a window to see what the application may be missing, so to speak.

3. **sigsuspend**
   int sigsuspend(const sigset_t *mask)
   The `sigsuspend` call provides an interface for the user-space application to temporarily block signals from being delivered.

4. **kill**
   int kill(pid_t pid, int sig) This call completes the signal API. It allows a user-space process to deliver a signal to another process, as identified by the `pid` parameter. The kernel also contains a variant of this call which allows kernel code communicate via the signal API to an application.

4 Conclusions

Legacy applications offer a mature stable code base which should not be discarded lightly. Most, if not all, applications can be ported to Linux with a minimum of effort, if proper care is taken to both segment the application properly to a kernel and a user space component and appropriate APIs are used to efficiently and correctly transfer data between the two.

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*