pkMem: A transactional, zero copy, shared memory based mechanism for IPC

by

Hareesh Nagarajan

A project submitted in partial fulfillment of the requirements for the degree

Master of Science in Computer Science

Approved, Masters Project Committee

Jon Solworth, Adviser
Associate Professor of Computer Science

V.N. Venkatakrishnan
Assistant Professor of Computer Science

Chicago, Illinois
April, 2006
This project is dedicated to my parents and their fathers.
Acknowledgments

I would like to thank my adviser Jon Solworth for having been most patient and supportive of me, during the entire duration of my time in Graduate school.

I would also like to thank Mike Ter Louw for many engaging discussions and brain storming sessions which greatly helped me refine the implementation of our system.

And finally I would like to thank the many friendly people who hung out on the following IRC channels: #xemacs, #emacs, #kernelnewbies and #lse.
| A.3  | advance_queue(struct pk_task_share *task_share, unsigned long status) : The conflicting requester’s attempt to advance the wait queue | 59 |
| A.4  | How to obtain pkMem | 61 |
| A.5  | Installation | 61 |
| A.6  | Feedback | 62 |
Summary

For this Masters Project I have implemented a new mechanism for Inter-Process Communication (IPC) called pkMem for the Linux kernel (2.6). There has been some work (Refer [1], [2]), done at UIC on this mechanism in the past, but none of them have attempted to implement a general purpose mechanism that incorporates transactional semantics.

A large part of this report deals with the design and implementation of pkMem for Linux and hence it is only natural that many of the ideas presented are specific to Linux.

Later in the report I have documented the results of my attempt at implementing the classic producer/consumer problem using pkMem. This has been done to show that the mechanism is sufficiently general purpose, is efficient and has a very clean API.

The report ends with describing the various directions for future work that are possible with this new mechanism.
Abstract

pkMem is a shared memory space that is mapped into every process on the system at exactly the same virtual address. By default, a process cannot access this space.

During transactions, parts of pkMem are made accessible to the process for reading or writing. The transactional semantics means that either the entire transaction occurs or that none of the transaction occurs and moreover that transactions appear to occur at a single instant in time without interleaving from other transactions—in other words, the transaction is atomic. The semantics of pkMem are very clean.

The most common alternative to pkMem are threads, in which a set of threads share the same address space, with full access rights. In the pkMem mechanism for Inter-Process Communication (IPC), processes share this well defined region and unlike threads, must use the transactional mechanism to access pkMem.

Other benefits include:

1. An object in pkMem has the same address in every process. This implies that pointer-based data structures such as trees, lists and graphs may be accessed identically from every process on the system, thus eliminating the need to convert pointer-based structures into linear form or to remap pointers.

2. Because pkMem is shared memory, during IPC zero copies of the data are required. But, unlike traditional shared memory, processes using the pkMem mechanism do not need to use semaphores or any other form of locking, to protect concurrent access to the region.

3. Decreased context switches (assuming there is sufficient room to put all the data). The pkMem system was built targeting 64-bit machines which have large amounts of physical memory.

\[1\text{Which essentially means that the processes cannot directly mprotect()} \text{pkMem}. \text{(A check is added to the mprotect()} \text{system call to deny the mprotect()} \text{in this case.)}\]
1 Terminology

In this Section, we will define some of the terms that will be used throughout this report:

1. **pkMem space**: A contiguous, page aligned region of memory that is mapped into the address space of every process on the system. This space starts at a fixed virtual address on every process. A pointer that belongs to the pkMem space in one process is valid (although not necessarily accessible) and equivalent in another process. Moreover, the same pointer is valid and equivalent in the kernel as well.

2. **pkMem mechanism**: The APIs that are used by processes to communicate with one another using the pkMem region.

3. **pkMem**: In this report, any reference to pkMem could (depending on the context of course), refer to the pkMem space or to the pkMem mechanism.

4. **locks**: A process obtains a lock to a shared region by calling the pk_get system call. If a request for a read lock is made, then the kernel sets read protection to the desired shared region. Similarly, if a request for a write lock is made, the kernel sets write protection to the shared region. At any point in time, multiple readers can hold a read lock on a region, but only a single writer can hold a write lock on a region. Requests to lock a shared region are serviced in first come, first serve order (FCFS). When a process revokes a lock on a region, the kernel removes all protection associated with the region.

   From the discussion below it will be clear that pkMem makes use of Rigorous 2 Phase Locking.

5. **Shared region**: A contiguous, page aligned region of memory that is a part of the larger pkMem space. Shared regions are created with the pk_alloc system call, which provides access to the shared region. An existing shared region can be made accessible with the pk_get system call. A region can be released back into the free pool by a call to pk_free. There is (currently) no restriction on which process can allocate, get, or free shared memory.

6. **Log region**: A contiguous, page aligned region of memory that is a part of the larger pkMem space. Log regions are created with the pk_alloc_log call; they are local to a process (not shared across processes) and cannot be obtained with the pk_get system call. These regions are not released at the end of transactions; they are only released when the creator of the region calls the pk_free_log system call, or when the process that created the region exits without releasing the region.

7. **Transactions**: Transactions are the heart and soul of the pkMem mechanism. Later in this report we will see the pkMem API, which can be used to obtain locks on regions (pk_alloc, pk_get, and pk_free implicitly obtain a lock) and release locks on all the regions that we have obtained (pk_end). A transaction begins when we obtain a lock and ends when we (invoke the call to) release all locks that we have obtained during the course of a transaction.

   For all purposes, the following definition should suffice: A transaction begins with a pk_alloc, pk_get or a pk_free (obtain locks) and ends with a pk_end (revoke all locks). The following terms are usually associated with a transaction.

   (a) **aborts**: If a process holds a lock on a region and another process requests a conflicting lock, then the requester will wait for a while for the current lock holder to release to lock on its own accord. If the lock is not released within the allotted time, the kernel on behalf of the requester, will abort the current holder. Aborting the transaction entails:

   i. Releasing all the locks held by the current holder

   ii. Undoing any changes (rollback) the current holder made to shared regions that it had a write lock on. This is done by traversing the undo log (refer Section 6 for more information).
(b) **undo log:** A process that holds a write lock on a shared region can modify the region. Since the process could be aborted, it needs to maintain an undo log (which is contained in the log region) of the changes it has made to the shared region during the course of the transaction. This undo log is used during an abort (by the kernel) to rollback the modified shared memory. Once a call to `pk_end` is executed, the transaction has committed and therefore the undo log is truncated.

(c) **rollback:** Is the database term, that brings a shared `pkMem` space that was modified during the course of an aborted transaction into a state that it was in prior to the start of the transaction.

### 1.1 The view from user space

The virtual memory of a process is divided into two regions. On x86 systems which have physical memory less than 1 GB, the kernel constant `TASK_SIZE` is set to `0xc0000000` (or 3GB). This constant defines the upper limit of the accessible memory for code that works at user space (or the lowest privilege level). The memory above this limit contains kernel code and data structures (at highest privilege level).

Figure 1 depicts the memory layout of a process without `pkMem`. Figure 2 depicts the memory layout of a process that has `pkMem` mapped into its virtual address space.

The memory map of a process such as `init` on a system with `pkMem`, would typically look like this:

```
00b45000  84K r-x--- /lib/ld-2.3.3.so
00b5a000  4K r------- /lib/ld-2.3.3.so
00b5b000  4K rw------ /lib/tls/libc-2.3.3.so
00b5e000 1108K r-x--- /lib/tls/libc-2.3.3.so
00c73000  8K r------- /lib/tls/libc-2.3.3.so
```
As can be seen, the **pkMem** space is mapped into the process, but access to the region is inaccessible. If the process were to, at this point, de-reference a pointer in the **pkMem** space, then it would receive a **SIGSEGV**.

## 2 A brief description of the API

Processes can invoke the following system calls with **pkMem** as part of a transaction:

1. `pointer_to_allocated_region = pk_alloc(pages);` Allocates a new contiguous, shared region of `pages * PAGE_SIZE` bytes, in the **pkMem** space; the process has an write lock to the region.

2. `number_of_pages = pk_get(pointer_to_allocated_region, lock_requested);` The process obtains either a read lock or a write lock on an already allocated shared region. The process holds the lock on the region, until the process invokes a **pk_end()**. At any given point in time, multiple processes can hold a read lock (on a shared region) but only one process can hold a write lock (on a shared region).
For example, if a shared region is currently locked for writing by a process, then the caller to `pk_get()` for that shared region will block (irrespective of the type of lock requested), until the current holder of the region releases it. If the region is not released via a call to `pk_end()` during the allotted time, then the current holders (transactions) will be aborted by the kernel. As a part of the abort, the kernel will perform a rollback based on the contents of the undo log (which is maintained by the holder) to restore the contents of the shared region to a state that it was in before, the holder modified the region. For more information on the undo log refer Section 6.

3. `pk_free( pointer_to_allocated_region )`: This call de-allocates and returns a shared region back into `pkMem`’s free pool. Any process can free any shared region in `pkMem`. The region is truly freed only, when the transaction successfully commits.

4. `pk_end(void)`: The call to `pk_end()` releases all the locks that a process has obtained during the course of a transaction. If a process was aborted in the middle of a transaction, then all calls up to and including `pk_end()` to `pkMem` will fail and return with an error, saying the transaction has been aborted.

The above calls happen during a transaction. The next calls are somewhat independent of (and unaffected by) transactions.

1. `pointer_to_log_region = pk_alloc_log(pages)`: By invoking this call, the kernel maps in a contiguous, private region `pages ∗ PAGE_SIZE` bytes inside `pkMem` which is for logging use only. The region cannot be `got` or `freed` by other processes. This region will be used by the process, to write the undo logs to the shared regions that it modifies during the course of a transaction. We note that there are no locks associated with a log region. A process can invoke this call at any time.

2. `pk_free_log(pointer_to_log_region)`: This call de-allocates and returns a log region back into `pkMem`’s free pool. Only the creator of the log region can release the log region back into the free pool. This call successfully executes only when a transaction is not in progress.

3 How do we map `pkMem` into the address space of every process?

To map the `pkMem` space into the same virtual address of every process, we create an in-memory file in `RAMFS`. We observe, that there are two places inside the kernel when a new user-space address map is created:

1. The first is when the kernel creates the first process on the system called `init`.
2. The second is when the kernel executes the `exec()` call

Following this observation we create a one time in-memory file inside `RAMFS`. And then during every call to `do_execve()` we attach the file to the process’s memory map. This is done with the help a call to `do_mmap()`.

This is similar to the approach taken in Linux to implement SVR4 Shared Memory. Here an in-memory file is created on a `shm_get()`. Then this file is attached (mapped into the address space of the process) to the process on an `shm_at()`.

The virtual start address and the size of the region are specified using the following constants.

---

2 we do not run the risk of mapping the `pkMem` region into an address (which in our case is `PKMEM_START`) that perhaps already has something mapped into it, because we perform the mapping of `pkMem` early, right when the process has mapped via the `exec()`
PKMEM\_START is an arbitrary address that lies somewhere between the stack and the heap in the process address space. The constant PKMEM\_PAGES refers to the size of the pkMem region in terms of the number of pages. These parameters are specified at compile time.

For more information on RAMFS one could look up the following file Documentation/filesystems/tmpfs.txt in the Linux kernel sources.

4 What do we need to track?

The main entities in the pkMem mechanism are:

**Processes** which during the course of a transaction hold locks on zero or more shared regions.

**Shared Regions** which can be locked at any time by an arbitrary number of processes. In addition, if some process has a lock on the shared region, there could also be a queue of processes which must wait to get a lock on the region.

**Free Pool** which is used to keep a track of the unused regions in pkMem.

4.1 Keeping track of processes

In Linux, all processes are described by the struct task\_struct structure. We add to the struct task\_struct a field which points to a structure that holds the pkMem information for that process:

```
struct pk_task *pk;
```

The struct pk\_task structure has the following definition:

```
struct pk_task {
    spinlock_t lock;
    unsigned long status;
    struct list_head log_head;
    struct list_head share_head;
};
```

The lock field controls access to this structure. This report will not go into the details of protecting the pkMem data structures with spinlocks. For now, it should suffice that our implementation is SMP safe and the locking of the internal pkMem data structures is relatively fine grained.

The status field refers to the state of the current ongoing transaction (We defer the discussion of the various states a transaction can be in to Section 4.5).

The share\_head list tracks all of the shared regions that the process has either allocated, freed or got during the course of the transaction. This list is traversed when the transaction has come to an end (via pk\_end() or during an abort).

**aborted** If the transaction is aborted (because it held the lock to a region for too long), then the kernel will traverse the task’s list to release all the regions that had been allocated during the course of this
transaction to the free pool; similarly freed regions need to be un-freed. The share_head list links objects of structures of type pk_task_share.

**ended** Once a transaction has come to an end, this list is traversed and the locks that have been obtained are released and shared regions that have been freed are returned to the free pool.

The log_head list keep a track of all the log regions that a process has allocated during the course of its lifetime. In Section A.1 we will see that, on process termination, this list is used for releasing log regions back into the free pool.

Nodes in the log_head list have the following structure:

```c
struct pk_basic {
    struct list_head list;
    unsigned long start;
    unsigned long pages;
};
```

The start field refers to the address at which the region starts.

The pages field refers to the size of the shared region (in pages).

### 4.2 Keeping track of pkMem shared regions

A shared region could have one or more processes holding a lock on it. In addition, it could also have a queue of processes waiting for a lock which cannot yet be granted. The following structure is used to track the per region information,

```c
struct pk_share {
    struct rw_semaphore sem;
    unsigned long start;
    unsigned long pages;
    struct list_head list;
    spinlock_t lock;
    struct list_head task_head;
    unsigned long status; // PK_FL_RDLCK | PK_FL_WRLCK
    // PK_ST_DEL | PK_ST_DEL_FINAL
    unsigned int refcnt;
};
```

The list field links the shared region to all the shared regions in the pkMem system. The head of this list is pk_shareHEAD.

The task_head field is the head node, to all the tasks that currently hold a lock on the region.

The sem field arbitrates first in, first out access to all the processes that wish to access this shared region (internally semaphores uses a queue). By using a read-write semaphore, our implementation guarantees, that only one process will get a write lock to the shared region at any given point in time. And, multiple processes that request a read lock can simultaneously obtain a read lock on the region.

The start field refers to the address at which the shared region starts.

The pages field refers to the size of the shared region (in pages).

The status field describes if processes current has shared access (PK_FL_RDLCK) or exclusive access (PK_FL_WRLCK) on the region. The flag PK_ST_DEL is set when the region has been freed during the course of the transaction.
(if perhaps, another process were to call a pk_get() on this region, then the call to pk_get() would fail, because the region has been freed). Now if the transaction (during which the region was freed) successfully ends (via a pk_end()) then status of the region is set to PK_ST_DEL_FINAL, which means the region can now be reclaimed back into the free pool. But on the other hand, if the transaction were aborted, then the PK_ST_DEL flag is removed. By doing so other processes can now call pk_get() and pk_free() on the region (because the region has effectively been un-freed).

To describe the task_head list field we shall make use of an example. Assume three processes currently holding a read lock on a shared region, then the task_head list field of the region would point to the corresponding struct pk_task_share nodes of each of these processes (refer Section 4.3). Now, if a new request for a write lock were to be made, then the requester will iterate over the task_head list in question and will wait for the processes that hold the lock (the processes that hold the lock can be obtained by iterating over the task_head list of the region in question), to release the locks within their allotted time. If however, the processes use up their allotted time without completing, the requester will abort each of the processes. Once it has done so, it will add itself to the task_head list. This example is described in detail in Section 5.

The allotted time is specified at compile time by the constant PK_TIMEOUT. In our current implementation we have set the timeout to 3 seconds.

```c
#define PK_TIMEOUT (3 * HZ)
```

So we can informally say, that the task_head list field maintains a list of all the processes that are currently holding a lock on the region.

### 4.2.1 What condition needs to be met for a shared region to be reclaimed?

A shared region is reclaimed back into the free pool only when its refcnt field is zero and its status field is set to PK_ST_DEL_FINAL (and not when the status is set to PK_ST_DEL) (Refer Figure 3).

The refcnt field keeps a count on the number of processes that have expressed an interest in the shared region. When a pk_get(), a pk_alloc() or a pk_free() is called, then the refcnt of the region is incremented. When a transaction ends or when a transaction gets aborted, the refcnts of each of the regions that were incremented during the course of the transaction are decremented.
4.3 The structure that links regions with processes

In the `struct pk_task` structure we observed a field called `share_head` which contains the list of all shared regions that a process has obtained a lock on. The nodes in this list have the following structure:

```c
struct pk_task_share {
    struct list_head list ;
    struct task_struct *task ;
    spinlock_t lock ;
    unsigned int status ; // PK_PROC_GOT_WR | PK_PROC_GOT_RD
    // PK_PROC_ALLOC | PK_PROC_FREE
    struct task_struct *task ;
    struct pk_share *share ;
    unsigned long jiffies ;
};
```

The `list` field points to the next `struct pk_task_share` structure for the given process.

The `list_task` field points to the next `struct pk_task_share` structure which along with the given process, currently holds a lock on the shared region.

The `status` field refers to how the process obtained the shared region. It could have been with a `pk_get()`, a `pk_alloc()` or a `pk_free()`.

The `share` field points to the shared region in question.

The `task` field points to the `struct task_struct` structure of the process.

The `jiffies` field stores the time at which the process obtained the lock on the region. According to [11] `jiffies` is a variable that keeps increasing forever until it wraps back to zero at which time it still keeps increasing. It gets increased at the HZ (a defined constant) rate as a result of a hardware interrupt. It is used for various timing functions within the kernel.

4.4 Keeping track of the unused regions (the free pool) and the shared regions

The free pool maintains a list of all the unused regions in `pkMem`. The head of this list is called `pk_freeHEAD`. Just like the nodes in `log_head list` field of a process’s `struct pk_task` structure, the nodes that constitute the free pool share the same data structure:

```c
struct pk_basic {
    struct list_head list ;
    unsigned long start ;
    unsigned long pages ;
};
```

As we mentioned in Section 4.2, the shared regions that have been allocated in `pkMem` are maintained in the `pk_shareHEAD` list. We observe that we do not need to maintain a list of log regions primarily because these regions are private entities (In Figure 3 we see the two global lists: `pk_shareHEAD` and the `pk_freeHEAD` along with a task’s `logsH` list). No global search operation (such as the ones that are internally performed during a `pk_get()` and `pk_free()`) need to be performed on these regions. Moreover, on the termination of every process on the system, the kernel releases any log regions the process might have allocated back into the free pool. This is unlike shared regions, which `persist` long after the process that created them dies.

A new entry is added into the `pk_shareHEAD` list whenever a region is allocated via a `pk alloc()`. An entry is removed from the list, only when the region has been freed and has been successfully reclaimed back into
the free pool.

In the following example, we see the process has allocated a shared region and has freed a shared region after which it got aborted. We observe that on an abort any regions that have been allocated during the course of the failed transaction are released back in the free pool and similarly regions that have been freed are unfreed.

```c
... start1 = pk_alloc(16);
pages1 = pk_get(start2);
pk_free(start3);
...
<--- Aborted!
...

// start1 no longer points to a valid shared region of size 16 bytes.
// The region pointed to by start3 is valid and has been valid prior to
// the start of this transaction and hence can be freed or got by
// any process on the system including this process (only after this
// process has called pk_end() and has internally reset its status to PK_TR_STOPPED)
```

In Figure 4 we can see the log regions a process has allocated (in addition to the two aforementioned lists).

### 4.5 The states a transaction can be in

The `struct pk_task` structure’s status field maintains the current status of the transaction. In Figure 5, we have shown the states that a process can be in. As we know, an abort occurs when a process has failed to release a lock on a shared region (within the allotted time) which is contended by one or more processes. It could occur asynchronously when a `pk_*()` system call is in progress or it could occur synchronously, when no `pk_*()` system call is in progress.

The following are the states that a task (that makes use of `pkMem`) can be in:

1. **Task is yet to call a `pk_*` system call**: Every task on the system starts off at this state. Once the task invokes any of the following system calls: `pk_alloc()`, `pk_alloc_log()`, `pk_free()`, `pk_free_log()`, its state is set to `PK_TR_STARTED | PK_SYS_INPROGRESS`. 

Figure 4: A view of the log regions a process has allocated and the important global lists
Figure 5: The various states of a transaction that a process (using \texttt{pkMem}) can be found in

2. **PK\_TR\_STARTED | PK\_SYS\_INPROGRESS**: This state refers to the fact that a transaction has been started and a \texttt{pkMem} system call is in progress.

3. **PK\_TR\_STARTED**: Once the \texttt{pkMem} system call has completed, the task goes into this state.

4. **PK\_TR\_ABORT\_PENDING**: If an abort were to occur during the execution of a \texttt{pkMem} system call \(^3\), then the state of the transaction is set to PK\_TR\_ABORT\_PENDING.

5. **PK\_TR\_ABORTED**: If an abort is pending that means, the abort could not complete, as a \texttt{pkMem} system call was in progress. But once the \texttt{pkMem} system call has completed, the \texttt{do_abort()} routine (Refer section A.2) is called, which sets the status of the transaction to PK\_TR\_ABORTED (asynchronous abort).

   On the other hand, if no \texttt{pkMem} system call is in progress (which means the task is in the PK\_TR\_STARTED state), then the \texttt{do_abort()} routine sets the status of the transaction to PK\_TR\_ABORTED (synchronous abort).

6. **PK\_TR\_STOPPED**: From the PK\_TR\_STARTED state, a call to \texttt{pk\_end()} will result in the task going to the stopped state via the PK\_TR\_STARTED | PK\_SYS\_INPROGRESS state. As we mentioned above, once the \texttt{pk\_end()} system call is in progress, the task can no longer be aborted.

   Moreover, if the task is in the PK\_TR\_STOPPED state then further calls to \texttt{pk\_end()} and \texttt{pk\_free\_log()} will result in the task staying in the same state.

   Once a task has been aborted and is in the PK\_TR\_ABORTED state, a call to \texttt{pk\_end()} will set the state of the task to PK\_TR\_STOPPED.

\(^3\)except \texttt{pk\_end()}; since the transaction is being committed, we just ignore the abort
Figure 6: A process has a read lock on a shared region

Figure 7: Another process successfully obtains a read lock on the shared region
5 The relationship between the various data structures

In this Section, we shall describe the relationships between the data structures in `pkMem` with the help of the same example that we used in Section 4.2 to describe the need for the `task_head` list field.

1. In Figure 6 we can see that a process has obtained a read lock on a shared region. The shared region is described by the `struct pk_share` structure. This structure is a node in the `pk_shareHEAD` list. The `task_head` list in the `struct pk_share` structure maintains a pointer to the `struct pk_task_share` structure of the process.

2. In Figure 7 we see that another process has obtained a read lock on the same shared region. This process got the lock to the region immediately (The reason will be outlined below). We can see the `list_task` entry of the `struct pk_task_share` of the first process points to the `struct pk_task_share` structure of the second process.

3. In Figure 8 we now see a process that has obtained a write lock on the shared region. This process in all likelihood did not get the lock to the region immediately (The reason will be outlined below). As we can see the first two processes no longer hold their read locks on the shared region. They probably invoked the `pk_end()` call and revoked all their locks (or they could have gotten aborted because they held the lock for too long; this case will be considered next). We also note that the `task_head` list in the `struct pk_share` structure now points to the `struct pk_task_share` structure of this new process. In Section 5.1 we look at this scenario from userspace.

4. If however, the processes did not release the region in their allotted time, then requester for the write lock, will abort each of the processes that are currently holding the read lock on the shared region. This is made possible by iterating the `task_head` list which is found in the `struct pk_share` structure. This list, points to the current holders of the region. Once, the requester for the write lock, has obtained the lock on the region, the `task_head` list in the `struct pk_share` structure now points to the `struct pk_task_share` structure of this process. This can be seen in Figure 9. In Section 5.2 we look at this scenario from userspace.

5.1 An example without an abort

In the previous section we saw an example where two readers are concurrently holding a read lock on a shared region after which a writer comes in and makes a request for a write lock on the same shared region. The readers then release the region (of their own accord) within `PK_TIMEOUT` seconds (from the time they obtained the lock on the region). The code sequence and the memory mapping would perhaps look like this.

For the readers:

```c
int main()
{
    unsigned long start = 2616205312; // or 0x9bf02000 in hexadecimal
    int err = -1; read;
    if (pk_get(start, PK_FL_RDLCK) < 0) { pkerror("pk_get"); goto out; }

    // Read from the region
    read = *(int*) start;
    if (pk_end() < 0) { pkerror("pk_end"); goto out; }

    err = 0;
out:
```

For the writers:
Figure 8: A third process now successfully obtains a write lock on the shared region without having to abort the first two processes.
Figure 9: A third process now successfully obtains a write lock on the shared region after having aborted the first two processes
For the writer:

```c
int main()
{
    unsigned long start = 2616205312; // or 0x9bf02000 in hexadecimal
    int err = -1, write = 100;

    if (pk_get(start, PK_FL_WRLCK) < 0) { pkerror("pk_get"); goto out; }

    // Write to the region
    *((int *) start) = write;

    if (pk_end() < 0) { pkerror("pk_end"); goto out; }
    err = 0;
out:
    return err;
}
```

The `pkMem` memory mapping of the first reader after obtaining the read lock:

- 9bef8000 40K s /pkmem (deleted)
- 9bf02000 16K s /pkmem (deleted)
- 9bf06000 1992K s /pkmem (deleted)

The `pkMem` memory mapping of the second reader after obtaining the read lock:

- 9bef8000 40K s /pkmem (deleted)
- 9bf02000 16K s /pkmem (deleted)
- 9bf06000 1992K s /pkmem (deleted)

**Note:** The 40KB region that begins at 0x9bef8000 is perhaps being used in other transactions, either for logging or sharing or both, on the system. In this report we will not delve into the mechanics of how Linux handles virtual memory address mappings. For more information on this subject refer [3].

The `pkMem` memory mapping of the writer before obtaining the write lock:

- 9bef8000 2048K s /pkmem (deleted)

The `pkMem` memory mapping of the first reader after releasing the read lock:

- 9bef8000 2048K s /pkmem (deleted)

The `pkMem` memory mapping of the second reader after releasing the read lock:

- 9bef8000 2048K s /pkmem (deleted)

The `pkMem` memory mapping of the writer after obtaining the write lock:

- 9bef8000 40K s /pkmem (deleted)
- 9bf02000 16K rw-s /pkmem (deleted)
- 9bf06000 1992K s /pkmem (deleted)

We also observe the exit values of the readers and we find that they completed their transactions.
5.2 An example with an abort

In our ongoing example, an abort would occur if the readers held the read lock on the region for a period of time that is greater than PK_TIMEOUT. Let us assume that between obtaining the lock (pk_get()) and releasing the lock (pk_end()) the readers perhaps called sleep(4) or invoked some computation that took considerable amounts of time (enough for the writer to abort the readers).

Then we observe that the writer after it has obtained the lock on the region, any pointer de-referenced by the readers will result in a SIGSEGV being sent.

```
./getrd
Segmentation fault
```

But if on the other hand, we were able to catch the signal, or if the reader did not de-reference any pointer in the shared region after the reader lost access to the region then then we observe that the pk_end() call notifies the reader that the transaction was indeed aborted.

```
./getrd
Caught SIGSEGV
pk_end(): Process was aborted
```

The readers would receive the SIGSEGV when they de-reference a pointer in the shared region

```
read = *((int *) start);
```

5.3 The cases to consider when a requester requests a lock

There are various cases to consider, when a requester requests a lock on a shared region. These cases are outlined below:

- If there is no process holding a lock on a region, then a request for either a read or a write is granted immediately.

- If there are one or more readers holding a read lock on a region, and have held the lock for some time where, \( time > PK_{\text{TIMEOUT}} || time < PK_{\text{TIMEOUT}} \), then a requester’s request (assuming it is the first request in FCFS order) for a read lock will be granted immediately.

- If there are one or more readers holding a read lock on a region, and have held the lock for some time where, \( time > PK_{\text{TIMEOUT}} || time < PK_{\text{TIMEOUT}} \), then a requester’s request (assuming it is the first request in FCFS order) for a write lock will result in the requester waiting for the readers to release the region in their allotted time. If any of readers fail to release their locks in their allotted time, then the requester will abort those readers.

- If there is a writer holding a write lock on a region, and has held the lock for some time where, \( time > PK_{\text{TIMEOUT}} || time < PK_{\text{TIMEOUT}} \), then a requester’s request (assuming it is the first request in FCFS order) for a write lock or a read lock will result in the requester waiting for the writer to release the region in its allotted time. If the writer fails to do so, then the requester will abort the writer.

Once this is done, the requester like the holders did before her, will add herself to the share->activeH list.

The ideas that have been presented here have been incorporated in the algorithm given in Section A.3.
undo logs and rolling back aborted transactions

Every process that makes use of pkMem typically accrues log regions over the course of its lifetime. It uses these regions to write the undo log, whenever it modifies the contents of a shared region. The system does not force the user to write these logs as we will see in the examples presented later in this report. If a process were to get aborted during the course of its transaction, the kernel will use the undo log, prepared by the process, to restore the modified region into a state that it was in prior to modification.

For the example below, it must be clear that an undo log need not be written. We do not care what is written into the shared region pointed to by ptr, because if the process were to get aborted (if it held the lock to the shared region pointed to by ptr2 for too long), then the kernel will release the shared region pointed to by ptr back into the free pool.

```
ptr = pk_alloc(16);
    // Start Transaction
*ptr = 200;
    // Modified the region
pk_get(ptr2, PK_FL_RDLCK);
<-- Aborted!
pk_end();
    // End Transaction
```

But, for the following example, an undo log needs to be written because. This is because, if the transaction were to be aborted (if it held the lock to the shared region pointed to by ptr2 for too long), the shared region pointed to by ptr2 needs to be restored, to a state that it was in, prior to any modification. This can only be done with the help of an undo log.

```
ptr = pk_alloc(16);
    // Start Transaction
pk_get(ptr2, PK_FL_WRLCK);
*ptr2 = 100;
    // Modified the region
<-- Aborted!
pk_end();
    // End Transaction
```

We could handle rollbacks, by entirely doing away with log regions and instead we could create a copy of the region, every time a write lock is obtained on a shared region. So for the above example, the kernel could internally create a copy of the shared region (pointed to by ptr2) when the write lock is granted. But doing so would increase the number of intermediary copies created. This is far from ideal. By design pkMem is a zero copy mechanism, which means, no intermediary copies are created during IPC.

So the above code snippet, could be rewritten (making use of undo logs) like this:\n
```
ptrlog1 = pk_alloc_log(8);
...
ptr = pk_alloc(16);
    // Start Transaction
pk_get(ptr2, PK_FL_WRLCK);
oldValue = *ptr2;
writeLog(ptrlog1, ptr2, oldValue);
*ptr2 = 100;
    // Modified the region
<-- Aborted!
pk_end();
    // End Transaction
```

The astute reader might ask the following questions:

1. This is scary! The kernel actually trusts user space to write the undo logs correctly? Yes. But since pkMem will primarily be used by library writers who could have access to secure debugging environments, like the one described in Section 10.3, the chances of error are reduced.

\[4\text{this is just an example}\]
2. *Can the logs get very big?* We only need to maintain the logs for the current transaction and transac-
tions are typically quite short. Once a transaction ends with a `pk_end()`, the kernel no longer considers
the contents of the logs to be valid. For upcoming transaction user space can resume writing its undo
logs into the same log regions. But yes, the logs can get potentially very big. We have not written
applications that can make use of `pkMem` as yet so we are unable to comment on how big these logs
can actually get.

6.1 `rollback()`

The `rollback()` routine which has been described in A.2 has not yet been implemented in `pkMem` version
0.2SMP. The routine, takes the start address of first log region that the process has allocated. It will then
walk the data structures according to the selected format, and bring the modified shared regions back into
a consistent state.

6.2 Format of the logs

`pkMem` version 0.2SMP does not incorporate support for logging just as yet. We are working on a format for
the logs that can be used by:

1. User space: for writing the logs.
2. Kernel space: for rolling back the modified shared region to a consistent state.

The format of the logs is yet to be finalized. But, figure 10 showcases the `pkMem` log format which is currently
being worked on. It shows the relationship between the three different types of log entries:

1. Header Entry
2. Write Entry
3. Continue Entry

In addition, the figure shows the ability of the log to span multiple log regions. The first log region must
begin with a header entry, which contains a pointer to the last log entry. Write entries are followed by the
undo data structure required to roll back the write operation. Write entries that have been undone have the
`undone` flag set in their type field. The `continue` log entry links separate log regions together.

Details on the log format will be released in the future on the `pkMem` website.

7 Performance of `mprotect()`

One of the main internal operations in `pkMem` is in changing the memory protection that is associated with
a shared region. Therefore, in this section we look at the relative performance of the `mprotect` primitive in
Linux.

We do a similar test to what was done in [7]. There 2000 pages were protected and then unprotected and
the entire process was repeated 50 times. In our test we protect and un-protect regions that range from 1
Figure 10: Format of the undo log
MB to 64 MB and we repeat the process 4096 times. The number reported is the average number of these pairs of operations which were accomplished in a second.

The results on an Intel(R) Xeon(TM) CPU 3.06GHz with 2 GB of RAM are as follows:

<table>
<thead>
<tr>
<th>Megabytes</th>
<th>1MB</th>
<th>2MB</th>
<th>4MB</th>
<th>8MB</th>
<th>16MB</th>
<th>32MB</th>
<th>64MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of times the region can be protected/unprotected in a second</td>
<td>149,198</td>
<td>126,705</td>
<td>125,134</td>
<td>125,505</td>
<td>125,805</td>
<td>124,336</td>
<td>123,072</td>
</tr>
</tbody>
</table>

And the results on an Intel(R) Pentium(R) M processor 1600MHz with 768 MB of RAM are as follows:

<table>
<thead>
<tr>
<th>Megabytes</th>
<th>1MB</th>
<th>2MB</th>
<th>4MB</th>
<th>8MB</th>
<th>16MB</th>
<th>32MB</th>
<th>64MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of times the region can be protected/unprotected in a second</td>
<td>373,744</td>
<td>238,186</td>
<td>220,414</td>
<td>235,987</td>
<td>229,007</td>
<td>219,763</td>
<td>179,878</td>
</tr>
</tbody>
</table>

We draw a similar inference to what was drawn in [7]. We expect the faster Xeon to perform better than the slower Pentium M on this test, but that is not the case. We can comfortably say that the performance of mprotect() varies greatly and it is difficult to give a bound on the time taken by this operation.

7.1 _flush_tlb_range ()

One would expect a full TLB flush every time an operation such as an mprotect() or a munmap() is called. Fortunately, that is not the case. A call to mprotect() internally calls the _flush_tlb_range () function which removes the ranges of TLB entries selectively and rather intelligently, rather than flushing the entire TLB.

Since in pkMem calls to mprotect() are frequent this optimization is a good thing. For more information on this topic refer [3].

8 Event notification

When processes communicate with each other using a shared region there needs to be a way for them to suspend execution and relinquish the processor until some predicate on the shared data is satisfied or until some event is signaled. pkMem does not provide any native mechanism for event notification because such mechanisms already exist on Linux.

Some of the ways of notifying related processes of an event (or condition) are delineated below.

8.1 pthread condition variables

pthread condition variables have been built using with the extremely efficient mechanism known as futexes which has been available in the 2.6 Linux kernel. [4] describes a futex quite nicely: A futex consists of a piece of memory (an aligned integer) that can be shared between processes; it can be incremented and decremented by atomic assembler instructions, and processes can wait for the value to become positive. Futex operations are done almost entirely in userspace; the kernel is only involved when a contended case needs to be arbitrated. This allows locking primitives implementing used futexes to be very efficient: since most

\[^{5}\]mprotect() is called internally, via the wrapper mprotect_for_task()
operations do not require arbitration between processes, most operations can be performed without needing to perform a (relatively expensive) system call.

For more information on futexes refer [5], [6].

To use condition variables with pkMem one can typically create a shared memory segment\(^6\) which is then attached to address space of the communicating processes. This shared segment could hold the shared mutex and the shared condition variable cond.

```c
pthread_mutex_t *mutex;
pthread_cond_t *cond;
```

If a process wishes to wait for a specific condition (perhaps, the shared region is empty and the process wants to be woken up when the region is full), then it would typically make the following calls:

```c
pthread_mutex_lock(mutex);
pthread_cond_wait(cond, mutex);
pthread_mutex_unlock(mutex);
```

Similarly if a process wishes to signal the waiting process, then it would typically make the following calls:

```c
pthread_mutex_lock(mutex);
pthread_cond_signal(cond);
pthread_mutex_unlock(mutex);
```

Figure 11 describes the sequence of events. We have used pthread condition variables as a synchronization device for implementing the producer consumer problem. Our experiment is described later in Section 9.1.1.

### 8.2 epoll() / poll() / select() on futex FD’s

Using epoll() or poll() or select() on a futex based file descriptor appears to be a popular choice, for event notification\(^7\). It is known to scale quite well. A description and usage of these mechanisms is not in the scope of this report. For more information read the C10k Problem from [8].

---

\(^6\) using POSIX shared memory, or SVR4 shared memory, or one could just use the /dev/shm pseudo file

\(^7\) Typically, this would mean, that there is data to be read
8.3 Variants of semaphore down() and semaphore up()

For relatively simple problems semaphores can be used for event notification. In the producer consumer problem that is described below, we have a common buffer into which the producer puts data. Now, once the buffer is full, the producer must go to sleep and must wake up when the consumer has consumed the data. Similarly, the converse applies to consumers. They need to sleep when the buffer is empty and wake up when the producer has filled the buffer.

We can clearly see that to implement such an event notification mechanism, a shared binary semaphore (mutex) should suit our needs just fine. Other mechanisms that exhibit the same behavior as that of semaphores, have been described in the sections 9.1.2 and 9.1.3.

The main downside with using semaphores is that, one cannot selectively wake up processes.

9 Producer/Consumer problem

The producer consumer problem is a well known problem in synchronization and can be found in any decent operating systems textbook. We quote [9] to describe the problem statement and the need for event notification: Two processes share a fixed-size buffer. One process produces information and puts it in the buffer, while the other process consumes information from the buffer. These processes do not take turns accessing the buffer, they both work concurrently. Herein lies the problem. What happens if the producer tries to put an item into a full buffer? What happens if the consumer tries to take an item from an empty buffer?

In this section, we implement the problem using pkMem for IPC. While, this might not be the most ideal application of our mechanism, the implementation goes to show that our mechanism is general enough for one to implement synchronization problems such as this.

From the implementations below, it should be clear that we use pthread condition variables and mutexes, futexes, FIFOs, solely for event notification and not for controlling access to the shared fixed buffer (which in our case is a pkMem shared region). From the description of pkMem so far, it should also be clear that controlling access to the shared region is inherently provided by the mechanism.

A note to the reader In the implementations that follow, we do not make use of undo logs. We also assume, that neither the producer or the consumer will hold a lock on the fixed buffer (which in this case is the shared region) for period which is greater than PK_TIMEOUT. This also goes to show, that pkMem is just as usable if the programmer does not wish to use the logging functionality that is provided. The tests were performed on an AMD Athlon(tm) 64 Processor 3000+ (2009.977 MHz) with 1024 MB of RAM.

9.1 Fixed-size buffer implemented with pkMem

We implement the fixed-size buffer using a shared region. We note the memory throughput (rate at which data has been transferred from the local memory of the producer into the local memory of the consumer) by making use of shared regions of different sizes.
9.1.1 Using pthread condition variables for event notification

We described the general mechanism in section 8.1.

The code for the producer is as follows (we have omitted error checks and initialization):

```c
pthread_mutex_lock(mutex);
while(1) {
    pk_get(start, PK_FL_WRLCK);
    if(GET_STATUS(start) & FULL) {
        pk_end();
        pthread_cond_wait(empty, mutex);
        continue;
    }
    if(cur_mmap - start_mmap >= filesz) {
        SET_STATUS(start, COMPLETED);
        pk_end();
        pthread_cond_signal(full);
        break;
    }
    offset = ((pages * PAGE_SIZE) - sizeof_int);
    if(cur_mmap - start_mmap + offset > filesz)
        offset = filesz - (cur_mmap - start_mmap);
    memcpy((char *)(start + sizeof_int), cur_mmap, offset);
    cur_mmap += offset;
    SET_STATUS(start, EMPTY);
    pk_end();
}
pthread_mutex_unlock(mutex);
```

start points to the pkMem fixed buffer. The data that we wish to transfer lies in the following address range: [start_mmap, start_mmap + filesz]. The first 4 bytes in the pkMem fixed buffer store control information, such as the status of the buffer (if the buffer is FULL or EMPTY or if the transfer has COMPLETED) in addition to the size of data that is stored in the buffer. The same idea applies even when we use other event notification mechanisms which we have described in Sections 9.1.2 and 9.1.3.

Similarly, the code for the consumer is as follows:

```c
pthread_mutex_lock(mutex);
while(1) {
    pk_get(start, PK_FL_WRLCK);
    if(GET_STATUS(start) & COMPLETED) {
        pk_end();
    }
    if(GET_STATUS(start) & EMPTY) {
        pk_end();
        pthread_cond_wait(full, mutex);
        continue;
    }
    offset = GET_STATUS(start);
    offset &= ~(EMPTY | FULL | COMPLETED);
    memcpy(cur, (void *)(start + sizeof_int), offset);
    cur += offset;
    SET_STATUS(start, EMPTY);
    pk_end();
}
```

8The data from the producer is stored in buffer; cur points to a location in buffer
Observation  The performance numbers can be seen in Table 1.

Some purists may be of the opinion, that we need not hold the mutex lock for the entire duration of the transfer, but we do so to avoid lost signals (wake-ups) without cluttering the code with additional (event notification) logic. The pthread library supports rich and varied mechanisms for event notification in addition to being familiar to most programmers. Hence we are of the opinion that users of pkMem are recommended to use this robust mechanism for event notification, even though the other implementations detailed below, offer significant performance benefits.

9.1.2 Using FIFOs for event notification

In this section we make use of named pipes (FIFOs) for event notification. This is rather unconventional, but surprisingly, we will see that it performs quite well.

We make use of the following idea: We create two named pipes, namely FULL_FIFO and EMPTY_FIFO. To signal to the consumer that the fixed buffer is full, the producer will write 1 byte to the FULL_FIFO which will wake up the consumer that is indefinitely, blocking on a read(2). Similarly, to signal to the producer that the fixed buffer is empty, the consumer will write 1 byte to the EMPTY_FIFO which will wake up the producer that is indefinitely, blocking on a read(2).

We create the FIFO (which denotes an event) using the following code:

```c
eventfd = open("event", O_RDWR);
```

We signal the event using the following code:

```c
write(eventfd, &dummy, sizeof(int));
```

We wait on the event using the following code:

```c
read(eventfd, &dummy, sizeof(int))
```

The code for the producer is as follows:

```c
while(1) {
    pk_get(start, PK_FL_WRLCK);
    if (GET_STATUS(start) & FULL) {
        pk_end();
        read(emptyfd, &what, sizeof(int));
    }
}
```
continue;
}
if ((cur_mmap - start_mmap) >= filesz) {
    SET_STATUS(start, COMPLETED);
pk_end();
    write(fullfd, &what, sizeof(int));
    break;
}

offset = ((pages * PAGE_SIZE) - sizeof(int));
if ((cur_mmap - start_mmap + offset > filesz)
    offset = filesz - (cur_mmap - start_mmap);
memcpy((char *)(start + sizeof(int)), cur_mmap, offset);
SET_STATUS(start, FULL);
SET_SIZE(start, offset);
pk_end();
write(fullfd, &what, sizeof(int));
cur_mmap += offset;
}

Similarly, the code for the consumer is as follows 9:

while(1) {
    pk_get(start, PK_FL_WRLCK);
    if ((GET_STATUS(start) & COMPLETED) {
        pk_end();
    }
    if ((GET_STATUS(start) & EMPTY) {
        pk_end();
        read(fullfd, &what, sizeof(int));
        continue;
    }
    offset = GET_STATUS(start);
    offset &= ~(EMPTY | FULL | COMPLETED);
    memcpy(cur, (void *)(start + sizeof(int)), offset);
    cur += offset;
    SET_STATUS(start, EMPTY);
pk_end();
    write(emptyfd, &what, sizeof(int));
}

<table>
<thead>
<tr>
<th>Amount of data to be transferred</th>
<th>4KB</th>
<th>8KB</th>
<th>12KB</th>
<th>16KB</th>
<th>20KB</th>
<th>24KB</th>
<th>28KB</th>
<th>32KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB</td>
<td>133.23</td>
<td>167.77</td>
<td>207.19</td>
<td>210.92</td>
<td>229.23</td>
<td>248.95</td>
<td>254.27</td>
<td>252.8</td>
</tr>
<tr>
<td>2 MB</td>
<td>141.88</td>
<td>181.31</td>
<td>214.38</td>
<td>231.49</td>
<td>241.33</td>
<td>255.44</td>
<td>269.09</td>
<td>275.56</td>
</tr>
<tr>
<td>4 MB</td>
<td>144.67</td>
<td>185.12</td>
<td>216.9</td>
<td>242.21</td>
<td>253.77</td>
<td>268.74</td>
<td>275.89</td>
<td>281.97</td>
</tr>
<tr>
<td>8 MB</td>
<td>151</td>
<td>191 .88</td>
<td>222.05</td>
<td>250.03</td>
<td>262.37</td>
<td>276.2</td>
<td>281.94</td>
<td>294.62</td>
</tr>
<tr>
<td>16 MB</td>
<td>151.2</td>
<td>190.8</td>
<td>227.02</td>
<td>253.98</td>
<td>268.64</td>
<td>283.02</td>
<td>288.98</td>
<td>297.55</td>
</tr>
</tbody>
</table>

Table 2: Memory throughput in MB/second while transferring data of size ‘x’ MB using a pkMem shared region (buffer) of size ‘y’ KB

**Observation**  The performance numbers can be seen in Table 2. The high memory throughput is very encouraging, but this implementation for event notification does not scale very well when there are additional

---

9The data from the producer is stored in buffer; cur points to a location in buffer
consumers. At best, it can be described as an ‘optimization’, that improves performance in the case where there are a few consumers.

9.1.3 Using the user space futex library for event notification

In this section we use the semaphore interface written by Rusty Russell which internally makes use of futexes. The interface and the associated library can be obtained from [10].

To wait on an event, we do `futex_down(&event)` And to signal an event, we do `futex_up(&event)`.

The code for the producer is as follows:

```c
while(1) {
    pk_get(start, PK_FL_WRLCK);
    if((GET_STATUS(start) & FULL) {
        pk_end();
        futex_down(empty_futex);
        continue;
    }
    if(cur_mmap - start_mmap >= filesz) {
        SET_STATUS(start, COMPLETED);
        pk_end();
        futex_up(full_futex);
        break;
    }
    offset = ((pages * PAGE_SIZE) - sizeof_int);
    if(cur_mmap - start_mmap + offset > filesz)
        offset = filesz - (cur_mmap - start_mmap);
    memcpy((char*)(start + sizeof_int), cur_mmap, offset);
    SET_STATUS(start, FULL);
    SET_SIZE(start, offset);
    pk_end();
    futex_up(full_futex);
    cur_mmap += offset;
}
```

Similarly, the code for the consumer is as follows:

```c
while(1) {
    pk_get(start, PK_FL_WRLCK);
    if((GET_STATUS(start) & COMPLETED) {
        pk_end();
        break;
    }
    if(GET_STATUS(start) & EMPTY) {
        pk_end();
        futex_down(full_futex);
    }
    offset = GET_STATUS(start);
    offset &=(EMPTY | FULL | COMPLETED);
    memcpy(cur, (void*)(start + sizeof_int), offset);
    cur += offset;
    SET_STATUS(start, EMPTY);
    pk_end();
    futex_up(empty_futex);
}
```

The data from the producer is stored in buffer; cur points to a location in buffer.
Observation The performance numbers can be seen in Table 3. As we can see, among all the event notification mechanisms we have used so far, the highest throughput have been obtained using futexes.

9.2 Fixed size buffer implemented with named pipes (FIFO)

In this section we implement the producer consumer problem using named pipes (FIFOs). No mechanism for event notification is required, as the kernel puts the producer to sleep, when it has filled the buffer. And similarly, the kernel puts the consumer to sleep when the buffer is empty. The size of the pipe is fixed to 4096 bytes by the constant `PIPE_BUF` defined in `include/linux/limits.h`.

As expected, from Table 4, we can see that the performance is excellent and the memory throughput one can achieve are very high.

<table>
<thead>
<tr>
<th>Amount of data to be transferred</th>
<th>Throughput (MB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB</td>
<td>171.38</td>
</tr>
<tr>
<td>2 MB</td>
<td>166.68</td>
</tr>
<tr>
<td>4 MB</td>
<td>176.88</td>
</tr>
<tr>
<td>8 MB</td>
<td>178.35</td>
</tr>
<tr>
<td>16 MB</td>
<td>178.35</td>
</tr>
</tbody>
</table>

Table 4: Memory throughput in MB/second while transferring data of size ‘x’ MB using a named pipe (with a fixed buffer size)

9.2.1 Observation: pkMem versus named pipes

It would be unfair to compare FIFOs with pkMem, as pkMem is a much higher level construct, that is capable of so much more than what FIFOs are capable of. Moreover, FIFOs have been known to excel at this very task - single producer, single consumer.

But still, we observe (from Table 5) that using pkMem with a fixed buffer (shared region) of size 1 KB and futexes for event notification we achieve nearly as good performance as that of FIFOs. The average throughput with FIFOs is 176.15 MB/Sec and pkMem with futexes comes in at a close second with 154.57 MB/Sec. But if we use larger shared regions (>= 2 KB) for the transfer then we see that pkMem comfortably beats FIFOs (refer Table 6).

---

Table 3: Memory throughput in MB/second while transferring data of size ‘x’ MB using a pkMem shared region (buffer) of size ‘y’ KB

<table>
<thead>
<tr>
<th>Amount of data to be transferred</th>
<th>4KB</th>
<th>8KB</th>
<th>12KB</th>
<th>16KB</th>
<th>20KB</th>
<th>24KB</th>
<th>28KB</th>
<th>32KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB</td>
<td>144.79</td>
<td>178.93</td>
<td>205.33</td>
<td>235.39</td>
<td>250.57</td>
<td>241.9</td>
<td>249.7</td>
<td>250.68</td>
</tr>
<tr>
<td>2 MB</td>
<td>153.25</td>
<td>186.19</td>
<td>212.38</td>
<td>233.12</td>
<td>250.02</td>
<td>263.03</td>
<td>258.68</td>
<td>267.92</td>
</tr>
<tr>
<td>4 MB</td>
<td>156.08</td>
<td>193.84</td>
<td>224.13</td>
<td>251.25</td>
<td>264.19</td>
<td>274.68</td>
<td>279.55</td>
<td>294.33</td>
</tr>
<tr>
<td>8 MB</td>
<td>158.72</td>
<td>198.36</td>
<td>233.76</td>
<td>254.6</td>
<td>267.95</td>
<td>280.93</td>
<td>289.13</td>
<td>296.79</td>
</tr>
<tr>
<td>16 MB</td>
<td>160.02</td>
<td>200.63</td>
<td>235.93</td>
<td>258.49</td>
<td>276.09</td>
<td>287.66</td>
<td>294.46</td>
<td>302.26</td>
</tr>
</tbody>
</table>

---

11 bytes in atomic write to a pipe
<table>
<thead>
<tr>
<th>Amount of data to be transferred</th>
<th>Throughput with FIFOs (MB/sec)</th>
<th>Throughput with pkMem (1KB buffer) + futexes (MB/sec)</th>
<th>Throughput with pkMem (2KB buffer) + futexes (MB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB</td>
<td>171.38</td>
<td>144.79</td>
<td>178.93</td>
</tr>
<tr>
<td>2 MB</td>
<td>176.21</td>
<td>153.25</td>
<td>186.19</td>
</tr>
<tr>
<td>4 MB</td>
<td>176.88</td>
<td>156.08</td>
<td>193.84</td>
</tr>
<tr>
<td>8 MB</td>
<td>177.94</td>
<td>158.72</td>
<td>198.36</td>
</tr>
<tr>
<td>16 MB</td>
<td>178.35</td>
<td>160.02</td>
<td>200.63</td>
</tr>
</tbody>
</table>

Table 5: Comparing memory throughput FIFOs v/s pkMem

<table>
<thead>
<tr>
<th>Size of pkMem shared buffer (KB)</th>
<th>Throughput with pkMem + futexes (MB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>154.57</td>
</tr>
<tr>
<td>2</td>
<td>191.59</td>
</tr>
<tr>
<td>3</td>
<td>222.3</td>
</tr>
<tr>
<td>4</td>
<td>246.57</td>
</tr>
<tr>
<td>5</td>
<td>261.76</td>
</tr>
<tr>
<td>6</td>
<td>269.64</td>
</tr>
<tr>
<td>7</td>
<td>274.3</td>
</tr>
<tr>
<td>8</td>
<td>282.39</td>
</tr>
</tbody>
</table>

Table 6: Memory throughput that are possible with pkMem (using futexes) for the Producer/Consumer problem

9.3 Using large pkMem shared buffers for transfer

In this section, we shall look at the data throughput that are possible when we use larger pkMem shared buffers. In the previous sections we were using, small shared buffers (of the order of kilobytes), for transferring data between the producer and the consumer because we wished to compare it with named pipes. Since, pkMem would ideally, be used in systems, that have a great deal of physical memory, we would assume, that in such systems, the pkMem space is large (of the order of gigabytes) and therefore the pkMem shared regions that are created for communication between processes could also be equally large.

So in Table 7 we observe the memory throughput that are possible, using the various event notification mechanisms, when we make use of a shared buffer which is the same size as the data that needs to be transferred, between the producer and consumer.

<table>
<thead>
<tr>
<th>Data to be transferred (in pages)</th>
<th>Size of pkMem shared region (in pages)</th>
<th>futexes (MB/sec)</th>
<th>fifo (MB/sec)</th>
<th>pthread (MB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 (1 MB)</td>
<td>256</td>
<td>295.34</td>
<td>285.33</td>
<td>280.49</td>
</tr>
<tr>
<td>512 (2 MB)</td>
<td>512</td>
<td>306.42</td>
<td>318.29</td>
<td>316.49</td>
</tr>
<tr>
<td>1024 (4 MB)</td>
<td>1024</td>
<td>318.20</td>
<td>337.88</td>
<td>330.11</td>
</tr>
<tr>
<td>2048 (8 MB)</td>
<td>2048</td>
<td>327.74</td>
<td>326.11</td>
<td>332.48</td>
</tr>
<tr>
<td>4096 (16 MB)</td>
<td>4096</td>
<td>328.66</td>
<td>306.63</td>
<td>335.55</td>
</tr>
<tr>
<td>8192 (32 MB)</td>
<td>8192</td>
<td>344.82</td>
<td>329.08</td>
<td>342.08</td>
</tr>
<tr>
<td>16384 (64 MB)</td>
<td>16384</td>
<td>345.34</td>
<td>319.84</td>
<td>338.54</td>
</tr>
</tbody>
</table>

Table 7: Memory throughput that are possible with pkMem for the Producer/Consumer problem using pkMem shared buffers that are of the same size as the data that needs to be transferred

The contents of 7 are plotted in figure 12. Just as we observed before, here too, we observe, that with futexes we obtain the best memory throughput. The largest throughput obtained is 345.34 MB/Sec.
9.4 Performance when we only read the pkMem shared buffer instead of \texttt{memcpy()}ing the contents of the shared buffer

Since, \texttt{memcpy()} is the most expensive operation performed, by the consumer, in this section, we look at the throughput that are possible, when the consumer only reads the contents of the shared buffer. That is instead of copying the contents of the shared buffer into a local buffer via a \texttt{memcpy()}...:

```c
// cur is a pointer to the buffer on the stack
// start is a pointer to the pkMem shared buffer
memcpy(cur, (void *)(start + sizeof(int), offset);
```

... the consumer only reads the contents of the shared buffer:

```c
// cur is a pointer to the buffer on the stack
// start is a pointer to the pkMem shared buffer
volatile char *curptr = (char *)(start + sizeof(int), ch;
while(curptr - ((char *)(start + sizeof_int)) < offset) ch = *curptr++;
```

Now, this is not very useful in itself, but it goes on to show of the throughput that are possible with pkMem when one does away with the expensive \texttt{memcpy()} operation. Now, understandably, the throughput that we obtain are significantly larger than the throughput obtained when the consumer invokes a \texttt{memcpy()}.

Table 8 documents the results. The contents of 8 are plotted in figure 13. Just as we observed before, here too, we observe, that with futexes we obtain the best memory throughput. The largest throughput obtained is \textit{612.65 MB/Sec}. Surprisingly, using fifo’s for event notification, results in very poor throughput.
Memory throughputs that are possible with pkMem
for the Producer/Consumer problem using pkMem shared buffers
that are of the same size as the data that needs to be transferred.
Here the Consumer only reads the contents of the buffer
and does not copy it (via a memcpy()).

![Graph](image)

Figure 13: A plot of Table 8

<table>
<thead>
<tr>
<th>Data to be transferred (in pages)</th>
<th>Size of pkMem shared region (in pages)</th>
<th>futex (MB/sec)</th>
<th>fifo (MB/sec)</th>
<th>pthread (MB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 (1 MB)</td>
<td>256</td>
<td>494.98</td>
<td>273.92</td>
<td>447.06</td>
</tr>
<tr>
<td>512 (2 MB)</td>
<td>512</td>
<td>540.11</td>
<td>296.11</td>
<td>533.83</td>
</tr>
<tr>
<td>1024 (4 MB)</td>
<td>1024</td>
<td>570.72</td>
<td>315.61</td>
<td>563.21</td>
</tr>
<tr>
<td>2048 (8 MB)</td>
<td>2048</td>
<td>583.07</td>
<td>313.59</td>
<td>580.87</td>
</tr>
<tr>
<td>4096 (16 MB)</td>
<td>4096</td>
<td>612.66</td>
<td>320.11</td>
<td>606.34</td>
</tr>
<tr>
<td>8192 (32 MB)</td>
<td>8192</td>
<td>694.13</td>
<td>336.83</td>
<td>607.78</td>
</tr>
<tr>
<td>16384 (64 MB)</td>
<td>16384</td>
<td>573.92</td>
<td>320.23</td>
<td>520.87</td>
</tr>
</tbody>
</table>

Table 8: Memory throughputs that are possible with pkMem for the Producer/Consumer problem using pkMem shared buffers that are of the same size as the data that needs to be transferred. But here the consumer only reads the contents of the buffer and does not copy it (via memcpy()).

10 Future work

Additional features need to be built into pkMem for it to be truly useful. There are a number of loose ends that need to be tied up and in this section we shall have a look at some of them.
10.1 Assigning name spaces to shared regions

When we talk of SVR4/POSIX Shared Memory, we observe that the shared regions of memory are treated as objects and are given a name using which unrelated communicating processes can refer to them.

For example, in SVR4 Shared memory shared segments are assigned keys. The following call is used to create a shared memory segment.

```c
int shmget(key_t key, int size, int shmflg);
```

On success an identifier is returned which is used to refer to the region. So now, if two processes wish to talk to each other using the same shared segment then they would refer to the same key.

Similarly, in POSIX Shared memory objects are given names using which unrelated processes can refer to them. The following call is used to create (or open an existing) shared memory object.

```c
int shm_open(const char *name, int oflag, mode_t mode);
```

pkMem lacks a mechanism for naming shared region. Once a process creates a shared region, it only has a pointer to that region. If an unrelated process wished to lock this region, then the only way out is for these processes to decide on exchanging this information using a conventional IPC mechanism for communicating this piece of information.

In our implementation of the producer-consumer problem (Section 9), we made use of a POSIX shared memory object to store the start addresses of the shared regions (the buffer), condition variables, mutexes and other information. In this way the producer and consumer could communicate using the same shared region (fixed buffer).

We did toy with the idea of building a simple key-value pair kernel interface which could be used to assign names to shared memory objects, but we decided against it. Instead a robust mechanism for storing references to shared regions in name spaces was proposed (this report will not go into the details of regarding the proposed design of this name space). A name space such as this could also be used to determine the appropriate privileges a process has based on the pkMem name space that it has acquired shared region(s) in.

10.2 Security

The following section describes the various aspects of pkMem which could potentially make the mechanism a secure one.

10.2.1 Quotas and prevention of Denial of Service

In the current implementation nothing prevents a process from allocating all the pkMem memory that is available on the system. With quotas one could specify limits on the amount of pkMem memory a process could allocate.

These quotas could perhaps be implemented by adding additional resources to the *rlimit family of system calls.

```c
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
```
For **pkMem**, the `resource` argument could be one of these values:

- **RLIMIT_PKALLOC** – The maximum number of bytes of pkMem memory that may be allocated, during the course of a transaction.
- **RLIMIT_FREE** – A value which determines if a process can free a pkMem share region, or if it can only free shared regions that it creates, or if it cannot free a shared region at all.
- **RLIMIT_PKSIZE** – The maximum number of bytes of pkMem memory that may be allocated, by a process during a single `pk_alloc()`.
- **RLIMIT_TIMEOUT** – The default abort timeout for every process is set to `PK_TIMEOUT`. With this limit a process could change its abort timeout value.

By setting up quotas and resource limits we can lower many obvious Denial of Service (DoS) attacks to a considerable extent.

Associated with the resource limits that we mentioned above, we would also need to setup appropriate capabilities for **pkMem**. For example:

- **CAP_SYS_PKTIMEOUT** – If a process has the following capability then it can change its abort timeout value.

### 10.2.2 Security Labels

For every **pkMem** shared region that is allocated, the process (or the kernel) could assign a security label. This label could be used to determine the processes that have the right to obtain locks on the shared region. Security models that focus on confidentiality such as the [Bell-LaPadula Model](#) could be implemented over **pkMem**.

Moreover, by using security labels one could try to integrate **pkMem** with [kernelSec](#).

### 10.3 Build a safe programming environment for debugging

From Section 6, we know that when a transaction has obtained a write lock on a shared region, any write to that region must typically be preceded by writing an undo log entry into the log region of the process.

It is important that the undo log is written correctly (according to the format that the kernel and user space have agreed upon) because, if the transaction were aborted and the kernel rolled back the contents of the shared region, the kernel trusts that the user space application that wrote the undo log wrote the log correctly. If the log has been written incorrectly, the contents of the **pkMem** shared region will be inconsistent.

For this reason, we need an environment, perhaps aided by the kernel, which could be used for debugging **pkMem** applications primarily for notifying the programmer that his undo log, if replayed (during rollback) will result in inconsistencies.

An inefficient implementation could make use of the following idea:

1. A system call called `pk_debug(int flag)` could be implemented. If the user application invokes the system call with an argument of `true` prior to the start of a transaction, then the kernel will create a copy of every region that the transaction gets a write lock on.
2. The user application goes ahead with writing the undo log to every shared region that it modifies during the course of its transaction into its log region(s).

3. Prior to invoking the `pk_end()` system call, the application could perhaps call a system call called `pk_isconsistent()` which would replay the undo log (written in the log region) on the modified shared regions and compares it with the copies created prior to the modification of the shared region. If the copies are identical, then the log has been written correctly and the call returns with a 0. If not the call returns with a pointer to the first inconsistency it finds.

10.4 Performance on SMP systems

`pkMem` version 0.2SMP has been written with SMP systems in mind. Careful planning has gone into the locking of the internal `pkMem` data structures. But, at the time of writing this report we have not had the opportunity to stress test applications that make use of `pkMem` on an SMP machine. So it would be interesting to benchmark the system on an SMP machine to especially observe what effects our IPC mechanism has on the cache-line.

11 system call Internals

The code commentary that appears in this section documents how `pkMem` Version 0.2SMP works.

11.1 `pk_alloc_log` *(unsigned long pages)* system call

<table>
<thead>
<tr>
<th>Parameters:</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned long pages – Specifies the number of pages that need to be allocated for creating a log region within <code>pkMem</code>. The size of the region is (PAGE_SIZE * pages) bytes.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return value on success:</th>
</tr>
</thead>
<tbody>
<tr>
<td>An unsigned long value that points to the region of memory that has been allocated for logging.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return value on error:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOMEM – Out of memory</td>
</tr>
<tr>
<td>EABRT – Transaction has been aborted</td>
</tr>
<tr>
<td>EMPROTECT – Kernel was unable to set the protection on region</td>
</tr>
</tbody>
</table>

Since a transaction that generates a lot of undo entries could be heavy on log usage, this call can be successfully invoked during the course of a transaction. This call will fail, if the transaction has been aborted.

1. If the status of the transaction `current->pk->status` is set to `PK_TR_STOPPED` or `PK_TR_STARTED` then we set the status to `PK_TR_SYS_INPROGRESS`. If the status of the transaction is in any other state (Section 4.5), then we return with an error. The `setup_transaction()` function handles these checks:

```
err = setup_transaction(PK_TR_SYS_INPROGRESS);
if (err == -ENOMEM || err == -EABRT) ...;
```

---

This report will not get into how the `pkMem` internal data structures are protected using *spin locks*. In some of the code snippets that appear below, I have rewritten the code, to enhance readability.
2. We iterate over the pk_freeHEAD (this list is sorted in decreasing order of the number of pages) list to see if there is a large enough (contiguous, of course) region that can be split to satisfy the allocation. The following call does this for us:

```c
struct pk_basic *log = alloc_log_pages (pages);
if (!log) { err = -ENOMEM; ...; }
```

3. If `alloc_log_pages()` is successful, then the log region that is returned is protected:

```c
mprotect_for_task(current, log->start, pages * PAGE_SIZE, (PROT_READ | PROT_WRITE));
```

4. Next we add the log node to the tail of the list of all the log regions the process has acquired, during its lifetime. During process exit, these log regions are returned back into the free pool (refer Section A.1).

```c
list_add_tail (&log->list, &current->pk->logs_head);
```

5. Before we return, we check the status of the transaction to see if we were aborted along the way. If yes, then we call the abort routine and return error –EABRT.

```c
if (current->pk->status & PK_TR_ABORT_PENDING)
    err = _do_abort(current, 1);
```

If no, then we reset the status of the transaction and return log->start

```c
current->pk->status &= ~(PK_TR_SYS_INPROGRESS);
return log->start;
```

### 11.2 pk_free_log (unsigned long start) system call

Parameters:
- `unsigned long start` – Specifies the start address of the log region within pkMem that needs to be freed.

Return value on success:
- 0

Return value on error:
- ENOTSTOPPED – The transaction is still in progress.
- ENOEXIST – The log region doesn’t exist
- EMPROTECT – Kernel was unable to set the protection on region

This call will succeed only if the transaction is stopped. This is to prevent, a process from freeing the log regions it has acquired during the course of the transaction.

1. If the status of the transaction `current->pk->status` is set to PK_TR_STOPPED then we proceed. If the status of the transaction is in any other state (Section 4.5), then we return with an error –ENOTSTOPPED.

```c
if (current->pk) {
    if (current->pk->status & PK_TR_STOPPED == 0) {
        err = -ENOTSTOPPED; ...
    }
}
```
2. We search the process’s pk_task->log_head list, to see if a region beginning at address start exists. If the region does not exist then we return error -ENOEXIST.

```c
struct pk_basic *remove = search_logs_head(start);
if (!remove) return -ENOEXIST;
```

If the region does exist then we remove the region from the process’s pk_task->log_head list

```c
list_del (&remove->list);
```

3. We then remove the protection associated with the region

```c
mprotect_for_task(current, remove->start, remove->pages * PAGE_SIZE, PROT_NONE);
```

4. And finally, we release the region back into the free pool and return 0.

```c
_insert_into_freelist (remove);
return 0;
```

11.3 pk_alloc (unsigned long pages) system call

Parameters:
- unsigned long pages – Specifies the number of pages that need to be allocated for creating a shared region within pkMem. The size of the region is (PAGE_SIZE * pages) bytes.

Return value on success:
- An unsigned long value that points to the region of memory that has been allocated for sharing.

Return value on error:
- ENOMEM – Out of memory
- EABRT – Transaction has been aborted
- EMPROTECT – Kernel was unable to set the protection on region

1. If the status of the transaction current->pk->status is set to PK_TR_STOPPED or PK_TR_STARTED then we set the status to PK_TR_SYS_INPROGRESS. If the status of the transaction is in any other state (Section 4.5), then we return with an error. The setup_transaction() function handles these checks

```c
err = setup_transaction(PK_TR_SYS_INPROGRESS);
if (err == -ENOMEM || err == -EABRT) ...;
```

2. We iterate over the pk_freeHEAD (this list is sorted in decreasing order of the number of pages) list to see if there is a large enough (contiguous, of course) region in the free pool that can be split to satisfy the allocation. The following call does this for us:

```c
struct pk_share *share = _alloc_share_regions (pages);
if (!share) { err = -ENOMEM; ... }
```

3. If _alloc_share_pages () is successful, we initialize a struct task_share node and link it with the share node that has just been allocated. We then set the status flags. The code sequence is as follows:

```c
task_share = init_pk_task_share ();
task_share->share = share;
task_share->status = PK_PROC_ALLOC | PK_FL_WRLCK;
```
After which we mprotect the shared region:

```c
mprotect_for_task(current, share->start, pages * PAGE_SIZE, (PROT_READ | PROT_WRITE));
```

4. Now that memory protection has been setup, we link this task_share node with the task’s share_head list. Then we link the task_share node with region’s task_head list and finally we increment the refcnt of the region

```c
list_add (&task_share->list, &current->pk->share_head);
list_add_tail (&task_share->list_task, &kshare->task_head);
share->refcnt++;  
```

5. And finally, we add the shared region to the global shared region list

```c
list_add (&kshare->list, &pk_shareHEAD);
```

6. Before we return, we check the status of the transaction to see if we were aborted along the way. If yes, then we call the abort routine and return error –EABRT.

```c
if (current->pk->status & PK_TR_ABORT_PENDING)
err = _do_abort(current, 1);
```

If no, we reset the status of the transaction and return share->start

```c
current->pk->status &~ (PK_TR_SYS_INPROGRESS);
return share->start;
```

11.4 `pk_get(unsigned long start, unsigned long flags)` system call

<table>
<thead>
<tr>
<th>Parameters:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned long start</td>
<td>Specifies the start address of the shared region within pkMem that needs to be locked.</td>
</tr>
<tr>
<td>unsigned long flags</td>
<td>PK_FL_WRLCK for write lock PK_FL_RDLCK for read lock</td>
</tr>
</tbody>
</table>

Return value on success:
An unsigned long value that refers to the size of the shared region that has been successfully locked (in pages).

Return value on error:
EINVAL – Invalid flag passed
ENOEXIST – The shared region doesn’t exist
ENODGRADE – A write lock has already been obtained on the region; hence lock cannot be downgraded to a read lock
ENOUGRADE – A read lock has already been obtained on the region; hence lock cannot be upgraded to a write lock
EDELETED – The region has been marked for deletion by another task. (The other task could be us; If the other task were to get aborted, and the pk_free() never committed, then a subsequent call to pk_free() could be successful)
ENOMEM – Internal kernel structures could not be allocated
EABRT – Transaction has been aborted
EMPROTECT – Kernel was unable to set the protection on region

1. First off, we check if the flag parameter to this call is correct:

```c
if (flag & ~(PK_FL_WRLCK | PK_FL_RDLCK)) { err = -EINVAL; ...; }
```
2. If the status of the transaction current->pk->status is set to PK_TR_STOPPED or PK_TR_STARTED then we set the status to PK_TR_SYS_INPROGRESS. If the status of the transaction is in any other state (Section 4.5), then we return with an error. The setup_transaction() function handles these checks

```c
err = setup_transaction(PK_TR_SYS_INPROGRESS);
if (err == -ENOMEM || err == -EABRT) ...;
```

3. We search the pk_shareHEAD list to see if the region exists in the first place; the call to search_share_head() accomplishes this task; if the call does find the region, then it increments share->refcnt. If the region does not exist we return error -ENOEXIST.

```c
struct pk_share *share = search_shareHEAD(start);
if (!share) {
  err = -ENOEXIST; ...;
}
```

4. If the region does exist, then we need to make sure that we haven’t already got, freed or allocated the same region in the current transaction. We first search the share_head list of the process.

```c
task_share = search_share_head(share);
```

If search is successful, then we do the following checks and return:

(a) If we have already allocated this region:
   i. If a request for PK_FL_WRLCK is made then the request is ignored and we return share->pages.
   ii. If a request for PK_FL_RDLCK is made then we return with an error -ENODGRADE, because we do not support lock downgrades.

(b) If we have already freed this region then return with error -EDELETED because we have already freed this region in the current transaction and it makes no sense to obtain a RDLCK or a WRLCK on the region.

(c) If we have already got this region ...
   i. and hold a PK_FL_WRLCK on it then the scenario is identical to 4(a)i.
   ii. and hold a PK_FL_RDLCK on it then if a request for a PK_FL_WRLCK is made then we return with an error -ENODGRADE because we do not support lock upgrades during the course of a transaction. But, if a request for PK_FL_RDLCK is made then the request is ignored and we return share->pages.

The code sequence for these steps looks like this:

```c
if (task_share->status & PK_PROC_ALLOC || task_share->status & PK_PROC_GOT_WR) {
  if (flag & PK_FL_WRLCK) { err = 0; goto decrement_the_refcnt; ...; }
  else { err = -ENODGRADE; goto decrement_the_refcnt; ...; }
} else if (task_share->status & PK_PROC_FREE) {
  err = -EDELETED; goto decrement_the_refcnt; ...;
} else if (task_share->status & PK_PROC_GOT_RD) {
  if (flag & PK_FL_WRLCK) { err = -ENOUGRADE; goto decrement_the_refcnt; ...; }
  else {
    err = 0; goto decrement_the_refcnt; ...;
  }
}
```

5. We initialize a struct task_share node and then link it with the share node that was returned by search_shareHEAD(). We set the status flags of the task_share node and we determine the protection flag which will be used for protecting the region.
task_share = init_task_share();
if(!task_share) { err = -ENOMEM; goto decrement_the_refcnt; ...; }

prot = PROT_READ;
task_share->share = share;
if(flag & PK_FL_WRLCK) { prot |= PROT_WRITE; task_share->status = PK_PROC_GOT_WR; }
else { task_share->status = PK_PROC_GOT_RD; }

6. We then invoke the routine to advance the wait queue (described in Section A.3). We need to invoke
this routine, because if there are any processes that have a lock on the region, then we need to wait
for them to release these locks within their allotted time slot. But, if they don’t release the region in
their time slot, then we need to make sure these processes are aborted (to revoke their locks on the
region). The call to _advance_queue() can return -EDELETED.

err = _advance_queue(task_share, flag);
if(err < 0) { ...; }

On success, the _advance_queue() routine sets, the time the task obtained the lock on the region.
task_share->jiffies = jiffies ;

7. Now, that we’ve obtained the required lock on the region, we protect the region and we add the
task_share node to the task’s share_head list.
mprotect_for_task(current, share->start, share->pages * PAGE_SIZE, prot);
list_add(&task_share->list, &current->pk_share_head);

8. Before we return, we check the status of the transaction to see if we were aborted along the way. If
yes, then we call the abort routine and return error -EABRT.

if(current->pk->status & PK_TR_ABORT_PENDING)
  err = _do_abort(current, 1);

If no, we reset the status of the transaction and return share->pages

current->pk->status &= ~(PK_TR_SYS_INPROGRESS);
return share->pages;

11.5  pk_free(unsigned long start) system call

Parameters:
  unsigned long start – Specifies the start address of the shared region
  within pkMem that needs to be freed.

Return value on success:
  0

Return value on error:
  ENOMEM – Out of memory
  EABRT – Transaction has been aborted
  ENOEXIST – The log region doesn’t exist
  EDENIED – A read lock has already been obtained on the region;
            So we cannot free this region
  EDELETED – The region has been marked for deletion by another task.
            (The other task could be us; If the other task were to get
            aborted, and the pk_free() never committed, then a subsequent
            call to pk_free() could be successful)
1. If the status of the transaction current->pk->status is set to PK_TR_STOPPED or PK_TR_STARTED then we set the status to PK_TR_SYS_INPROGRESS, else we return with an error. The setup_transaction() function handles these checks

```
err = setup_transaction(PK_TR_SYS_INPROGRESS);
if (err == -ENOMEM || err == -EABRT) ...;
```

2. We search the pk_shareHEAD list to see if the region exists in the first place; the call to search_share_head() accomplishes this task; if the call does find the region, then it increments the share->refcnt. If the region does not exist we return error -ENOEXIST.

```
struct pk_share *share = search_shareHEAD(start);
if (! share) {
    err = -ENOEXIST; ...;
}
```

3. If the region does exist, then we need to make sure that we haven’t already got, freed or allocated the same region in the current transaction. We first search the share_head list of the process.

```
task_share = search_share_head(share);
```

If search is successful, then we do the following checks and return:

(a) If we have already allocated the region, then we set the status of the shared region to PK_ST_DEL
(b) If we have already freed the region then we return 0
(c) If we have already got ...
   i. A PK_FL_WRLCK on the region then all we have to do is to set the status of the shared region to PK_ST_DEL
   ii. A PK_FL_RDLCK on the region, then we deny the operation and return -EDENIED

The code sequence for these steps looks like this:

```
if (task_share->status & PK_PROC_ALLOC || task_share->status & PK_PROC_GOT_WR) {
    task_share->status |= PK_PROC_FREE;
    share->status |= PK_ST_DEL;
    err = 0; goto decrement_the_refcnt;
} else if (task_share->status & PK_PROC_FREE) {
    err = 0; goto decrement_the_refcnt;
} else if (task_share->status & PK_PROC_GOT_RD) {
    err = -EDENIED; goto decrement_the_refcnt;
}
```

4. We initialize a struct task_share node and then link it with the share node that was returned by search_shareHEAD(). We set the status flags of the task_share node.

```
task_share = init_pk_task_share();
if (! task_share) { err = -ENOMEM; ...; }
task_share->share = share;
task_share->status = PK_PROC_FREE;
```

5. We then invoke the routine to advance the wait queue (described in Section A.3). We need to invoke this routine, because if there are any processes that have a lock on the region, then we need to wait for them to release these locks within their allotted time slot. But, if they don’t release the region in their time slot, then we need to make sure these processes are aborted (to revoke their locks on the region). The call to _advance_queue() can return -EDELETED.
err = _advance_queue(task_share, flag);
if (err < 0) { ...; }

On success, the _advance_queue() routine sets, the time the task obtained the lock on the region.

task_share->jiffies = jiffies ;

6. Now that we’ve obtained exclusive access to the region, we mark the region for deletion.

share->status |= PK_ST_DEL;

7. We add the task_share node to the task’s share head list.

list_add (&task_share->list, &current->pk->share_head);

8. Before we return, we check the status of the transaction to see if we were aborted along the way. If yes, then we call the abort routine and return error –EABRT.

if (current->pk->status & PK_TR_ABORT_PENDING)
err = _do_abort(current, 1);

If no, we reset the status of the transaction and return 0

current->pk->status &= ~(PK_TR_SYS_INPROGRESS);
return 0;

11.6 pk_end(void) system call

Parameters:
None

Return value on success:
0

Return value on error:
EABRT – Transaction has been aborted
EABRTONGOING – Transaction is being aborted
EMPROTECT – Kernel was unable to set the protection on region

This system call releases the locks to all the regions that have been obtained in the current transaction. It also makes sure that shared region(s) that have been freed (during the course of the transaction) are reclaimed into the free pool as well as shared region(s) that have been allocated (during the course of the transaction) are made available system wide.

1. We first need to do some checks on the status of the transaction. If the status current->pk->status of the transaction is set to ...:

(a) PK_TR_STOPPED then there is nothing there to do, so we return 0
(b) PK_TR_ABORTED then we set the status to PK_TR_STOPPED and return an error which indicates that the transaction aborted but the status was successfully reset.
(c) PK_TR_ABORT_INPROGRESS then we return an error –EABRTONGOING, which would indicate to user space that it would need to call pk_end() again.
d) PK_TR_STARTED then we set the status to PK_TR_SYS_INPROGRESS

The code sequence for these steps looks like this:

```c
if (current->pk) {
    if (current->pk->status & PK_TR_STOPPED)
        return 0;
    else if (current->pk->status & PK_TR_ABORTED) {
        current->pk->status |= PK_TR_STOPPED;
        return -EABRT;
    }
    else if (current->pk->status & PK_TR_ABORT_INPROGRESS)
        return -EABRTONGOING;
    else if (current->pk->status & PK_TR_STARTED)
        current->pk->status |= PK_TR_SYS_INPROGRESS;
    }
else { return 0; }
```

2. For each of the shared regions in the task’s share_head list, we do the following:

(a) We delete the task_share node from the process’s share_head list.

(b) If the shared region has been freed (that is, PK_PROC_FREE is set), then we make sure the region gets reclaimed back into the free pool. This is accomplished by setting the region’s status flag to PK_ST_DEL_FINAL.

(c) Next we remove the protection for shared region. If the region has only been freed (PK_PROC_FREE only) then we do not need to remove the protection (because we never added any protection to begin with).

(d) After which, we delete the task_share node from the shared region’s task_head list.

(e) Then we decrement the refcnt of the shared region by calling __dec_share_refcnt () which handles region reclamation.

(f) And finally, we free the task_share node.

The code sequence for these steps looks like this:

```c
while (!list_empty(&current->pk->share_head)) {
    idx = current->pk->share_head.next;
    list_del (idx);
    task_share = list_entry (idx, struct pk_task_share, list );

    if (task_share->status & PK_PROC_FREE)
        task_share->status |= PK_ST_DEL_FINAL;

    if (!((task_share->status & PK_PROC_FREE) && (!task_share->status & PK_PROC_GOT_WR) && (!task_share->status & PK_PROC_ALLOC)))
        mprotect_for_task(current, task_share->share->start, task_share->share->pages * PAGE_SIZE, PROT_NONE);
    list_del (&task_share->list_task);
    __dec_share_refcnt (task_share->share);
    kmem_cache_free(pk_task_share_cachep, task_share);
}
```

3. Finally we, set the status of the transaction to PK_TR_STOPPED and return 0.

```c
current->pk->status = PK_TR_STOPPED;
return 0;
```
12 High memory in the Linux kernel and pkMem

If one wishes to use large amounts of pkMem memory (1GB or more)\textsuperscript{13}, on either 32 bit or 64 bit machines, then no changes are required in the implementation of pkMem. All one needs to make sure is that \((PKMEM\_START, PKMEM\_START + PAGE\_SIZE \times PKMEM\_PAGES)\) lies within the address space of the process. That is the \(pk\text{Mem}\) region lies between \((0, TASK\_SIZE)\) bytes (Refer figure 1).

1. On 32 bit machines: Linux is capable of accessing physical memory that is greater than 1 GB in size using PAE extensions on an x86 architecture. The maximum physical memory that can be addressed is 64 GB. To read more about this refer \[14], \[15].

2. On 64 bit machines: A 64 bit Linux kernel can natively access upto 64 GB of physical memory. If the user space is also 64 bit, then user processes will have \(2^{64}\) bytes of virtual memory.

13 Conclusion

pkMem is perhaps the first transactional IPC mechanism that has been implemented in Linux. It performs quite well on traditional synchronization problems such as the producer/consumer. Its true potential has not yet been exploited, since applications that could benefit from it, have yet to be identified and written.

\textsuperscript{13}referred to as High memory
References


A Appendix

The code commentary that appears in this section documents how pkMem Version 0.2SMP works.

A.1 pkMem region reclamation on process exit

If a process were to exit (or was made to exit, via a signal) while it was in the middle of a transaction, then we need to make sure, that all the changes that were made during the course of the uncommitted transaction are undone. In addition, all the log regions that have been allocated by the process need to be released back into the free pool.

The following changes were integrated into the common exit point inside the kernel (do_exit() which is defined in kernel/exit.c).

1. If the task’s last transaction has been aborted, or if an abort is in progress, or if the transaction has been stopped, then we do not bother calling the _do_abort() routine which is described in section A.2

2. If that is not the case, then we call the _do_abort() routine, with its second argument, the call_mprotect flag set to 0 (since the process is exiting, we need not bother removing the memory protection associated with its pkMem regions)

3. Finally we call the _release_log_regions () routine, which will release the log regions allocated by the process (if any) back into the free pool.

The code sequence for these steps looks like this:

```c
if (tsk->pk) {
    if (! (tsk->pk->status & PK_TR_ABORTED || tsk->pk->status & PK_TR_STOPPED 
        || tsk->pk->status & PK_TR_ABORT_INPROGRESS))
        _do_abort(tsk, 0);
    _release_log_regions (tsk->pk);
}
```

A.2 _do_abort(struct task_struct *task, int call_mprotect): The function that handles the all important abort!

1. First off, we check if task->pk has been initialized. If it hasn’t then we return 0, because the task has not done anything with pkMem.

```c
if (! task->pk) return 0;
```

2. If we are interrupting another pk_* system call then we set the status of the task to PK_TR_ABORT_PENDING otherwise we set the status to PK_TR_ABORT_INPROGRESS to make a note of the fact, that an abort is indeed in progress. Once the status is set to PK_TR_ABORT_PENDING then it is the responsibility of the interrupted pk_* system call to call _do_abort() (We have seen this behavior in the pk_* system calls that were described in section 11).

```c
if (pk->status & PK_TR_SYS_INPROGRESS) { pk->status |= PK_TR_ABORT_PENDING; ...; }
else { pk->status |= PK_TR_ABORT_INPROGRESS; }
```
3. We then grab the first log region (if any) that has been allocated by the task and pass the region’s start address as an argument to the _rollback() routine.

```c
if (!list_empty(&pk->logs_head)) {
    idx = pk->logs_head.next;
    log_region = list_entry(idx, struct pk_basic, list);
    _rollback (log_region ->start);
}
```

4. We iterate over all the shared regions in the task’s share_head list and do the following:

   (a) We remove the task_share node from the share node’s task_head list
   (b) Next we undo any allocations or frees:
      i. If the region has been allocated and freed, then we delete the region
      ii. If the region has been freed (marked for deletion; PK_ST_DEL is set), then we undo that change
      iii. If the region has been allocated, then we delete it
   (c) If _do_abort() was called with its second argument call_mprotect set to 1, then we remove the protection associated with the region.
   (d) We decrement the refcnt of the shared region by calling _dec_share_refcnt(). As we know, this function handles region reclamation.

   The code sequence for these steps looks like this:

```c
list_for_each (idx, &pk->share_head) {
    task_share = list_entry(idx, struct pk_task_share, list);
    list_del (&task_share->list_task);

    if (task_share->status & PK_PROC_FREE && !(task_share->status & PK_PROC_ALLOC))
        task_share->status &= ~(PK_ST_DEL);
    else if (task_share->status & PK_PROC_ALLOC)
        task_share->status = PK_ST_DEL_FINAL;

    if (call_mprotect) {
        if (!(task_share->status & PK_PROC_FREE) && !(task_share->status & PK_PROC_GOT_WR)
            && !(task_share->status & PK_PROC_ALLOC)))
            mprotect_for_task(task, task_share->share->start,
                task_share->share->pages * PAGE_SIZE, PROT_NONE);
    }
    _dec_share_refcnt (task_share->share);
}
```

5. We now empty the elements in the task’s share_head list and free the task_share structure.

```c
while(!list_empty(&pk->share_head)) {
    idx = pk->share_head.next;
    list_del (idx);
    kmem_cache_free(pk_task_share_cachep, idx);
}
```

6. Set the status of the transaction to PK_TR_ABORTED and return -EABRT.

### A.3 _advance_queue(struct pk_task_share *task_share, unsigned long status) : The conflicting requester’s attempt to advance the wait queue

The code presented here is an implementation of the ideas that were presented in Section 5.3. The pk_get() (refer section 11.4) and pk_free() (refer section 11.5) system calls invoke this routine.
The status argument to this routine describes the lock that is requested by the requester. The following macros are used to post and signal the read-write semaphore in which is a field in the \texttt{struct \_pk\_share} structure (refer section 4.2).

\begin{verbatim}
#define GET_SEMAPHORE(task_share, status) \ 
  if (status & PK\_FL\_RDLCK) down_read(&task_share->share->sem); \ 
  else down_write(&task_share->share->sem);
#define RELEASE_SEMAPHORE(task_share, status) \ 
  if (status & PK\_FL\_RDLCK) up_read(&task_share->share->sem); \ 
  else up_write(&task_share->share->sem);
\end{verbatim}

1. We try to acquire the semaphore right away.

\begin{verbatim}
GET_SEMAPHORE(task_share, status);
\end{verbatim}

2. Since it might take some time for the call to \texttt{down\_\_} to return, the shared region that we are interested in could be deleted while we waited. So we check, if the region has been marked for deletion. If it has indeed been marked then we return \texttt{−EDELETED}

\begin{verbatim}
if (task_share->share->status & PK\_ST\_DEL || task_share->share->status & PK\_ST\_DEL\_FINAL) {
  err = −EDELETED; ...;
}
\end{verbatim}

3. If the region's \texttt{task\_head} list is empty then that means, that no other task is holding a lock on the region, so we update the \texttt{status} field of the \texttt{task\_share} node whose pointer was passed as an argument to \texttt{advance\_queue}. Next, we set the time the task obtained the lock on the region to the current time. And finally, before returning 0, we add the \texttt{task\_share} node to the region's \texttt{task\_head} list and release the semaphore.

\begin{verbatim}
if (list_empty(&task_share->share->task_head)) {
  task_share->share->status |= status;
  task_share->jiffies = jiffies ;
  list_add_tail (&task_share->list_task, &task_share->share->task_head);
  RELEASE_SEMAPHORE(task_share, status);
  return 0;
}
\end{verbatim}

4. If we are here, then there are other task(s) that are currently holding a lock on the shared region. If we have requested a \texttt{PK\_FL\_RDLCK} and there are currently readers who are holding a lock on the region, then we do same things, that we did in the previous step before returning 0.

\begin{verbatim}
if ((task_share->share->status & PK\_FL\_RDLCK) && (status & PK\_FL\_RDLCK)) {
  task_share->share->status |= status;
  task_share->jiffies = jiffies ;
  list_add_tail (&task_share->list_task, &task_share->share->task_head);
  RELEASE_SEMAPHORE(task_share, status);
  return 0;
}
\end{verbatim}

5. But if on the other hand, we have requested a \texttt{PK\_FL\_RDLCK} and there is currently a writer who is holding a \texttt{PK\_FL\_WRLCK} or If we have requested a \texttt{PK\_FL\_WRLCK} and there is currently a writer who is holding a \texttt{PK\_FL\_WRLCK} or If we have requested a \texttt{PK\_FL\_WRLCK} and there are currently reader(s) who are holding a \texttt{PK\_FL\_RDLCK} then we do either of the following, for each one of the task(s) that are currently holding a lock on the region:

60
(a) If the task has held the region for a period of time which is greater than the allotted time \( PK_{\text{TIMEOUT}} \), then we abort the task by calling the \_do\_abort() routine.

(b) If that is not the case, we wait for the time differential and then retry

We are done, only when there are no tasks holding a lock on the region, or in other terms the shared region’s task head list is empty.

The code sequence for these steps looks like this:

```c
while(!list_empty(&task_share->task_head)) {
    list_for_each(idx, &task_share->task_head) {
        ts = list_entry(idx, struct pk_task_share, list_task);
        current_time = jiffies;
        allotted_time = ts->jiffies + PK_TIMEOUT;
        if(time_before_eq(allotted_time, current_time)) {
            _do_abort(ts->task, 1);
            goto _continue;
        } else {
            long diff;
            diff = (long) allotted_time - (long) current_time;
            set_current_state(TASK_UNINTERRUPTIBLE);
            schedule_timeout(diff);
            goto _continue;
        }
    }
}
```

6. At this point, we have the lock on the region. We update the status field of the task_share node

```c
if(task_share->share->status & PK_FL_RDLCK && status & PK_FL_WRLCK) {
    task_share->share->status &= ~(PK_FL_RDLCK);
    task_share->share->status |= PK_FL_WRLCK;
}
if(task_share->share->status & PK_FL_WRLCK && status & PK_FL_RDLCK) {
    task_share->share->status &= ~(PK_FL_WRLCK);
    task_share->share->status |= PK_FL_RDLCK;
}
```

7. And finally, we set the time the task obtained the lock on the region to the current time. Before returning 0, we add the task_share node to the region’s task head list and we release the semaphore.

```c
task_share->jiffies = jiffies;
list_add_tail(&task_share->list_task, &task_share->share->task_head);
RELEASE_SEMAPHORE(task_share, status);
return 0;
```

### A.4 How to obtain pkMem

pkMem is licensed under the GNU General Public License Version 2.

It can be downloaded from [http://www.cs.uic.edu/~hnagaraj/pkmem](http://www.cs.uic.edu/~hnagaraj/pkmem).

### A.5 Installation

These instructions are specific to pkMem version 0.2SMP. In this release pkMem has not been integrated into the kbuild - kernel build system.
1. Download the sources of the 2.6.13.1 kernel

```bash
```

2. Extract the sources

```bash
tar jxvf linux-2.6.13.1.tar.bz2
```

3. Download the `pkMem` patch.

```bash
wget http://cs.uic.edu/~hnagaraj/pkmem/pkmem-v0.2-smp.patch
```

4. Apply the patch

```bash
cd linux-2.6.13.1
patch -p1 < ../pkmem-v0.2-smp.patch
```

5. If one wishes to change the size of the `pkMem` region, one could edit `include/linux/pkmem.h` and change the value of `PKMEM_PAGES`. Moreover, if one wishes to change the start address at which `pkMem` is mapped into every process, the value of the `PKMEM_START` can be changed.

6. Configure and compile the kernel

```bash
make menuconfig
... make selection ...
make
```

7. After compilation, the kernel image resides in:

```bash
arch/i386/boot/bzImage
```

8. Make the necessary changes to your boot loader and boot into the kernel. Once you have successfully done so, you should see the following kernel message which can be seen when you invoke `dmesg`:

```bash
pkMem: initialized
```

### A.6 Feedback

For comments and suggestions please send an email to the author at hnagaraj@cs.uic.edu