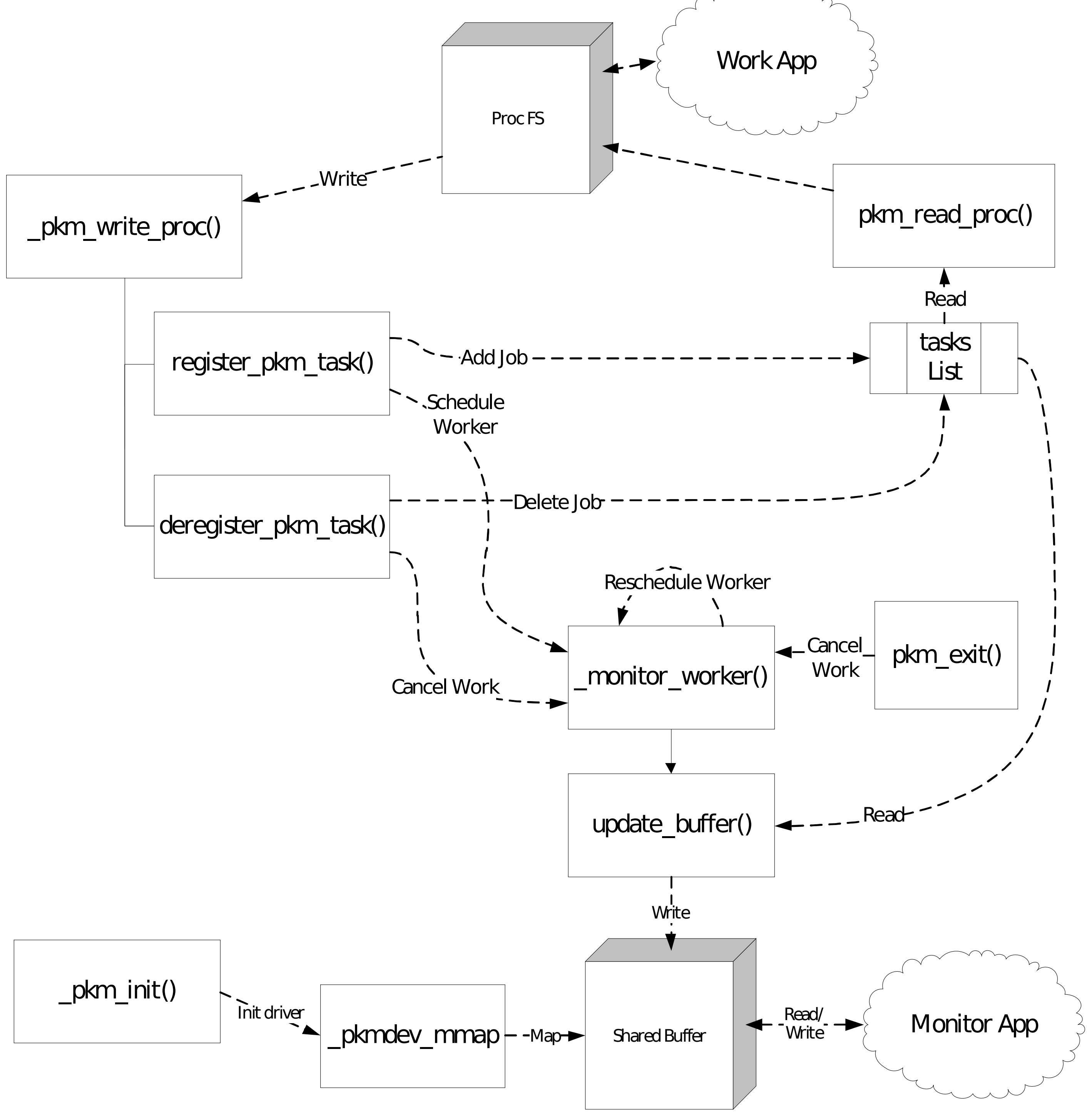


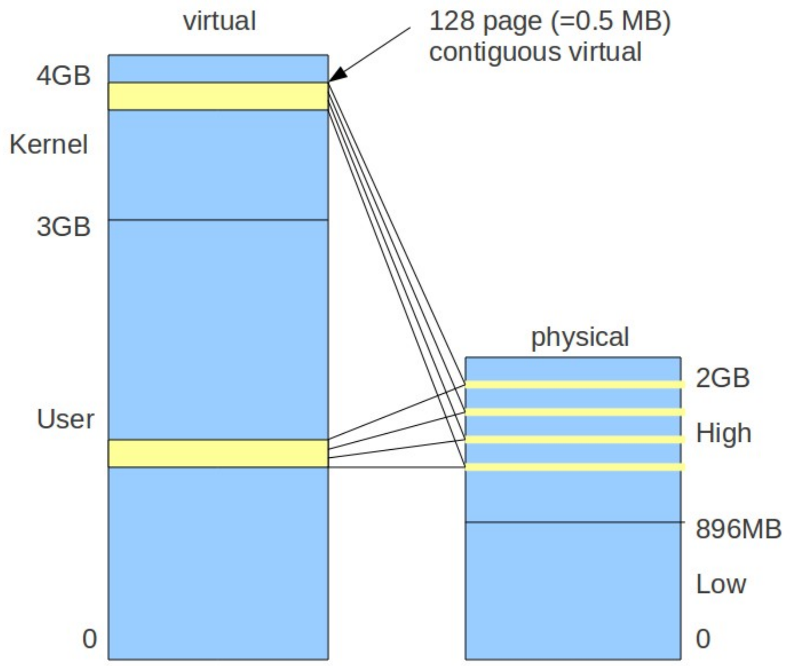
Overall Design
To implement the memory profiler, we created a kernel module that utilizes the system work queue, a memory buffer that is mapped into the user mode monitoring application's address space using a character driver, and a proc file system entry for processes to register/deregister for monitoring.

Work Queue
The system's built-in work queue was selected because this project did not require the additional performance/flexibility of managing our own unique work queue. Unlike one of our MP1 implementations which relied on a timer to queue the

worker, in this MP we utilize the kernel’s schedule_delayed_work function both to start the worker and to periodically reschedule it. The worker is created and scheduled lazily when the first process registers with the module. It then periodically collects profiling data and reschedules itself until it is canceled. Cancellation is implemented via a shared variable set by the deregister function when the last process deregisters from the module. After the deregister function sets the variable, it waits for the worker to exit by calling the kernel function cancel_delayed_work_sync. The module exit function uses the same procedure to cancel and wait for the worker in case there are any registered processes remaining when the module is removed.

Shared Buffer and Character Driver
The shared buffer is allocated during module initialization using vmalloc. Therefore, the buffer is virtually contiguous within the kernel's address space, but not necessarily physically contiguous. The page descriptors for the allocated page frames are marked as reserved to ensure they remained assigned to the buffer. The flags are removed during module exit and the buffer is deallocated.

To expose the buffer to the monitor application, we implemented a special character device driver. The device driver implements only the open, close, and mmap file operations, and of these three only mmap is implemented. Our mmap implementation iterates over the virtual pages and maps each underlying physical page to another vritual page in the user address space. The result is that the user application has a virtually contiguous view of the profile buffer. The mapping for a 32 bit x86 architecture is illustrated in the following figure.



The device driver is initialized and registered during module initialization. The major number is reserved dynamically in the process. During module exit, the major number is released and the driver is unregistered.

Proc File System Entry
Like past MPs, this MP relies on the proc file system for interacting with user-mode applications. The work.c application can write to the proc file that is registered during module initialization to indicate its desire to be monitored or to halt monitoring. As before, we use a switch statement in write_proc to key off of the first character written and invoke either the registration or deregistration functions accordingly. Work.c can then read from the file, triggering our read_proc, which provides the current list of registered PIDs.

Data Structures
To track the PIDs that are registered, we created a struct that contains the task_struct for the given process, as well as fields for major and minor fault rates, stime, and utime. The entries for each registered process are stored in the tasks

linked list, which is used by the read_proc, registration, deregistration, and update_buffer functions. Of course, it is also initialized and cleaned up in _pkm_init and pkm_exit respectively, and individual processes can be located/added/removed in the list through worker functions we added (not shown in flow diagram).

The buffer itself is just a series of unsigned longs written sequentially in groups of 4 (current jiffies value, major fault, minor fault, and utime+stime). All except the jiffies are aggregates of each registered process' relevant information since the worker thread last ran.

Finally, the kernel module contains a struct that points to _pkmdev_mmap function, as well as the _pkmdevl_open and _pkmdev_release stubs (not shown in diagram) used in the character driver. This is used for creating/initializing the driver.

Synchronization
We use two mutexes as the main synchronization primitives: one to protect the task list, and one to protect the cancellation variable. The task list mutex is acquired whenever the task list is read or updated. The cancellation mutex is acquired whenever tasks are removed from the list (deregister and module exit), since the cancellation flag may have to be set. The worker acquires the cancellation mutex to check the cancellation flag. The two mutexes are acquired and released in such a way that the removing processes can wait until the worker exists without new register processes coming in, attempting to recreate the worker at the same time.

Error Handling
Every function that makes calls that can fail (e.g. calls kmalloc) checks the return status for failures and return an appropriate error code. Care was taken to properly release any locks and free any already allocated variables in the event of errors, so the module should avoid potential leaks and deadlocks in low resource conditions.