Page Fault Scalability under Linux

Illustration 1 shows the time spend in the page fault handler for anonymous page faults. The page fault handler typically acquires the page_table_lock twice. Yellow is the total time spent in the fault handler per fault. Red is the time spend allocating a page (which may include acquiring yet another spinlock that we wont consider now) and blue is the time spend zeroing the page. For one and two processors the time spent in the fault handler is dominated by the necessity to zero a page before providing the application access to it. The situation slightly changes for 4 processors. The time spent apart from zeroing and allocating increases. This is the time spend trying to acquire the page_table_lock twice.

If 8 processors are contending for a long then already more than 50% of processing time is spend acquiring the lock meaning the processors are busy causing cache lines to bounce back and forth without making much progress in doing the work that they are expected to do. The time spend on lock acquisition increases exponentially as the number of processors increase. The diagram does not contain bars for more than 16 processors because they would no longer fit onto the page.

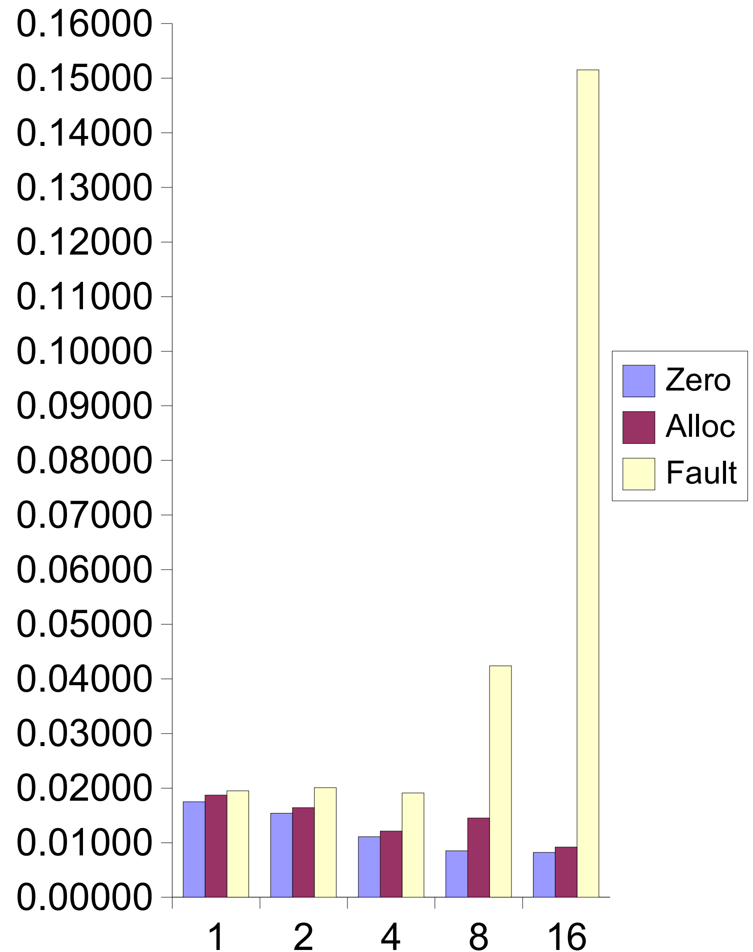

Illustration 1 Time spent in Page fault handler in ms for 1 to 16 processors

The use of the page_table_lock may be avoided by changing the locking from only relying on the page_table_lock to a combination of atomic operations plus the use of the page table spinlock.
In the Linux kernel the page_table_spinlock is acquired for any operation on the page table in order to avoid concurrent updates. However, the system also acquires a read lock on **mmap_sem** when a processes makes small changes to memory mappings like changing individual entries. **Mmap_sem** is acquired for larger changes to the memory maps as a write lock. We can therefore rely on the **mmap_sem** for protection against large scale remapping of page table entries.

One can therefore redefine the role of the page_table_lock to only offer protection against

modifications to a page table entry but allow the replacement of an empty page table entry through a new entry without the page_table_lock. However, then it must be also guaranteed that an empty page table entry can always be populated when a read lock of the mmap_sem is held regardless of the page_table_lock.

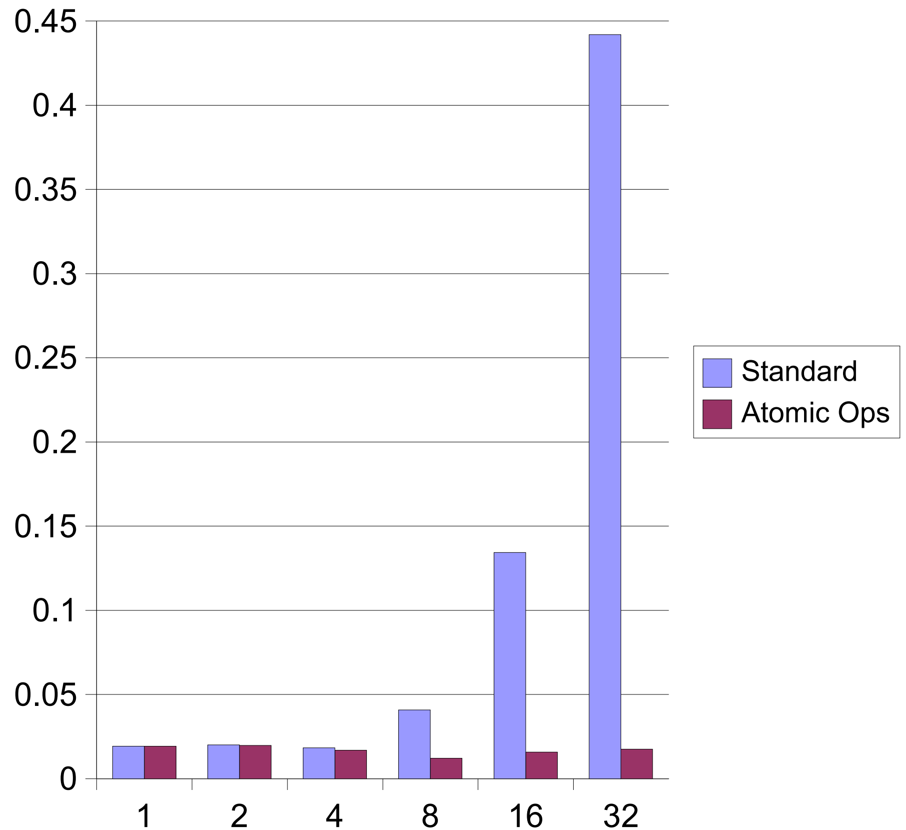

Illustration 2 Page fault time in ms per page for 1 to 32 processors

This means that all code using the page_table_lock must now insure that a page table entry is never sporadically set to empty during the changing of the value of a page table entry. Also if it is empty then the code using the page_table_lock must not assume that the page table entry will stay empty but must use atomic operations to replace values in order to guard against other processors concurrently changing the value without obtaining the lock. Holding the page_table_lock now only insures that a valid page entry is not changed. It no longer protects from empty page table entries becoming populated.

After the modification the page fault handler scales linearly. The diagram to the right compared the standard approach of using the page_table_lock with the atomic operations. The standard solution is not scaling well beyond 4 processors and becomes a major bottleneck at 16 processors.

Christoph Lameter, April 29th, 2005.