The Linux Memory Model

Francesco Zappa Nardelli
ENS & INRIA
francesco.zappa_nardelli@inria.fr

Peter Sewell
U. of Cambridge, UK
peter.sewell@cl.cam.ac.uk

The Linux kernel is an excellent example of critical concurrent code that must compile correctly on a large range of different architectures. The code is mostly written in C, with some architecture specific code expressed directly in assembly.

We know that different architectures expose wildly different memory models. For instance the x86 implements basically TSO [1], while Power and ARM allow almost arbitrary thread local reorderings and are not multi-copy atomic [2]; the situation gets more complex if we consider that the Linux kernel also runs on other exotic architectures (e.g., Itanium [3]). To guarantee portability across architectures kernel developers defined several macros that are intended to guarantee the same memory synchronisation behaviour across different architectures. The semantics of these macros, and more in general, what a kernel developer can expect as semantics of shared memory, is defined informally in [4]. The resulting model is very different from the high-level language memory models, e.g., those of C11/C++11 or Java, which are usually based on a data-race freedom assumption.

A preliminary analysis of the Linux memory model pointed out some potential problems. The model described in [4] presents several ambiguities. This is not surprising from an informal prose document, but calls for a rigorous formalisation and extensive testing. The semantics of some of the synchronisation macros is intricate, and their implementation on the different architectures should be studied and possibly proved correct (for instance, we believe that the implementation on Itanium multiprocessors is likely to be unsound). It might also be the case that the kernel code respects stronger memory invariants than those guaranteed by the memory model, and that the memory model might be strengthened.

Aim In this internship we set out to explore the design and implementation of the Linux Memory Model. By analysing the implementation of some key algorithms of the kernel, discussing with kernel developers, and reading the actual documentation, we aim at understanding the minimal assumption of the code on the shared memory system and the actual semantics of the barrier macros. We will attempt to produce a rigorous memory model [5] for the Linux kernel, suitable for reasoning about critical kernel algorithms. We might then investigate the correctness proof, or correctness validation, of some key kernel algorithms (e.g., the process scheduler). According to the interest of the student, we might also explore if the `gcc` compiler preserves the guarantees of the memory model.

Abridged bibliography

[1] x86-TSO: A Rigorous and Usable Programmer?s Model for x86 Multiprocessors.
`http://www.cl.cam.ac.uk/~pes20/weakmemory/cacm.pdf`

[2] A Tutorial Introduction to the ARM and POWER Relaxed Memory Models.
`http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`

[3] A Formal Specification of Intel Itanium Processor Family Memory Ordering.
`http://download.intel.com/design/itanium/downloads/25142901.pdf`

[4] Linux Kernel Memory Barriers.
`http://www.kernel.org/doc/Documentation/memory-barriers.txt`

[5] Relaxed memory models must be rigorous.
`http://www.cl.cam.ac.uk/~pes20/weakmemory/ec2.pdf`