CS510 Languages and Low Level Programming: Portfolio submission, Topic 12

Due on June 3, 2016 at 11:59pm

Mark P. Jones Spring 2016

Konstantin Macarenco

Topic 12. Explain how the requirements of low-level systems programming motivate the desire for (and benefit from) language based support.

Low-Level system programming is difficult and error prone, developing software in this environment is tedious.
Errors in Low Level Systems are critical and much more important, since they affect the entire platform, and all applications that use it. Faulty Kernel will cause failures across whole system, unlike for example broken editor, or HTTP server. Over the last decades Systems Complexity grew exponentially without any proof of correctness.
Languages used for LLP, like Assembly and C (especially Assembly) are very powerful but do very little when it comes to error checking. They don't impose any restrictions and let a programmer easily "shoot himself in a head". A special domain specific language/compiler would greatly simplify the task of system programming. Typical programming errors like buffer overflow, division by zero, null pointer dereference etc. are laborious to detect without proper operating system support. LLP specific language should be able to detect them during compilation, rather than runtime.

LLP has many repetitive, though hardware specific tasks like setting Page Directory, and Interrupt Vector Table, etc which are purely mechanical by nature and need lots of bits twiddling/manipulations. For historical reasons bit patterns in various parts of the system are very different and unpredictable due to backward compatibility requirement, since they were added as hardware matured. Many of these operations would be a lot easier, faster and reliable if supported by the language with strong typing, and good static analysis.
Another challenge in Low Level Programming is reusability of code and portability- it is hard to achieve since every application is restricted to specific hardware, there is no Kernel API that general programmers are used to. However many of the problems can be abstracted by the Language or Standard Library. Or provided on form of template/code generation (more an IDE feature). For example system calls or Paging is pretty standard, however requires modification in multiple places hence is error prone.

Understanding how different parts of the LLP application are tied together can be problematic, since vital parts of code spread out among multiple domain specific languages and physical locations, also functionality interactions can happen at unexpected places or be hidden, for example by interrupts, or system calls. This problem can be helped by providing set of abstraction and self documenting features like JavaDoc.
Human Factor plays drastic role on the top of all the problems associated with LLP. Many errors caused by the simple lack of perception. Existing tools, and systems are complicated enough to cause perceptual error i.e. when code base is huge and spread out it is very easy to oversee problems - example given in class - Page initialization loop went over the Page boundaries. It

Having specific LLP language can solve many potential problems, decrease errors, and increase performance (goal of any high level language), and surprisingly enough it still doesn't exists.