International Journal of Computer Architecture Education (IJCAE)

Explorando a Pilha de Execução (Exploring the stack for fun and profit)

Noemi Rodriguez, Ana Lúcia de Moura
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
{noemi,amoura}@inf.puc-rio.br

***Resumo*—A disciplina de Software Básico que oferecemos na PUC-Rio adota a perspectiva de um desenvolvedor de software, enfatizando o suporte que os componentes básicos de uma arquitetura típica provê para as abstrações oferecidas por linguagens de programação. Um tópico central da disciplina é a pilha de execução, um mecanismo cujo domínio é essencial para proficiência em programação. Neste trabalho discutimos como a disciplina explora esse mecanismo e o seu suporte para abstrações típicas de programação, como variáveis locais e procedimentos, e também para multitasking a nível de aplicação.**

I. Introdução

Alunos de cursos de graduação relacionados à Computação tem contato, ao longo desses cursos, com os diversos níveis de abstração presentes em um sistema de computação. Contudo, como esses níveis são frequentemente apresentados sob diferentes perspectivas, muitas vezes esses alunos tem dificuldade em assimilar como níveis adjacentes se relacionam, e como cooperam para a construção e execução de aplicações.

Uma descontinuidade resulta das perspectivas distintas adotadas por disciplinas tradicionais de Arquitetura de Computadores e de Linguagens de Programação. Disciplinas de Linguagens de Programação introduzem diversos tipos de dados e estruturas de controle, mas raramente discutem como plataformas reais proveem suporte para essas abstrações. Por outro lado, disciplinas tradicionais de Arquitetura de Computadores apresentam representações de baixo nível – como tipos de dados básicos e linguagem de montagem – sem relacioná-las a uma linguagem de programação de nível mais alto.

A disciplina de *Software Básico* que oferecemos na PUC-Rio tem muito em comum com disciplinas tradicionais de Arquitetura de Computadores. Entretanto, ao invés de apresentar os componentes básicos de uma plataforma típica sob a visão de um arquiteto de *hardware*, nossa disciplina adota uma perspectiva que enfatiza aspectos relacionados ao *software*, mais especificamente, ao suporte que mecanismos básicos proveem para a implementação das abstrações oferecidas por linguagens de programação convencionais. Essa perspectiva oferece aos alunos uma *ponte* mais concreta entre esses dois níveis, provendo um tipo de visão "por trás dos panos".

Um tópico central de nossa disciplina é a pilha de execução. Em currículos convencionais, a pilha de execução é coberta em disciplinas de Linguagens de Programação [1]–[3] que, muitas vezes, apresentam apenas seu conceito abstrato, insuficiente para que os alunos adquiram uma compreensão mais sólida desse mecanismo. Nossa disciplina oferece aos alunos a oportunidade de atingir essa compreensão, introduzindo a implementação concreta da pilha de execução e explorando essa implementação para apresentar, ou revisitar, diversos conceitos como o modelo e as convenções de chamada de procedimentos, chamadas recursivas e classes de armazenamento de variáveis. Trabalhamos em profundidade esses conceitos em uma série de laboratórios práticos, onde, acreditamos, é o momento onde a assimilação desses conceitos de fato acontece. Além de usos mais tradicionais, nossa disciplina também explora a pilha de execução como a base para a implementação de um ambiente multi-tarefa a nível de usuário.

A perspectiva centrada em *software* que adotamos em nossa disciplina foi apresentada em um trabalho anterior [4], no qual discutimos também nossa experiência lecionando essa disciplina na PUC-Rio há mais de dez anos. Neste trabalho, nosso objetivo é apresentar e discutir em mais detalhes os tópicos de nossa disciplina que envolvem a pilha de execução.

O restante deste trabalho está organizado da seguinte forma. A Seção II apresenta um resumo do programa atual da disciplina e sua metodologia. A Seção III descreve nossa abordagem e o material de apoio para a introdução da pilha de execução e seu suporte para a implementação de procedimentos. Na Seção IV, discutimos um laboratório onde os alunos utilizam seu entendimento sobre a estrutura da pilha para implementar um ataque baseado em *buffer overflow*. A Seção V descreve como introduzimos em nossa disciplina alguns conceitos básicos de programação concorrente. Finalmente, a Seção VI apresenta algumas considerações finais.

II. Programa e Metodologia do Curso

A disciplina de Software Básico da PUC-Rio adotou um programa convencional até o início dos anos 2000. Nessa época, decidimos que a disciplina deveria adotar a perspectiva de um desenvolvedor de *software*, e não de *hardware*. Essa decisão deveu-se, parcialmente, ao fato de que muitos dos alunos fazem, mais tarde, uma disciplina de Arquitetura de Sistemas, ministrada pelo Departamento de Engenharia Elétrica. Nessa disciplina, eles têm a oportunidade de aprender, com profundidade, aspectos de projeto de *hardware*. Percebemos que havia muita redundância entre nosso programa e o da disciplina de Arquitetura e que, por outro lado, nosso programa não cobria adequadamente a interface entre o *software* e o *hardware*.

Quando decidimos explorar a forma como abstrações de dados e controle oferecidas por uma linguagem de programação convencional são mapeadas para uma arquitetura típica, descobrimos no livro de Bryant e O'Hallaron [5]

um excelente material sobre esse mapeamento. Infelizmente, esse material é bastante denso, provavelmente porque está concentrado em uma parte relativamente pequena do livro (os capítulos iniciais). Esse é o único livro texto que encontramos que adota a perspectiva de um desenvolvedor de *software*, e tem sido, desde 2004, nossa referência principal. Assim como o livro, usamos a linguagem C e a arquitetura Linux/IA-32.

Atualmente, a disciplina tem cinco módulos principais, organizados em unidades que reunem um conjunto de conceitos – apresentados pelo professor ou em vídeos – e laboratórios de exercícios, onde os são auxiliados pelo professor e, dependendo do tamanho da turma, também por um monitor. O primeiro módulo cobre representação de dados, e discutimos a representação de diferentes tipos de dados oferecidos por C (inteiros com e sem sinal, números de ponto flutuante, *arrays* e *structs*). Discutimos também as convenções de ordenamento *little* e *big-endian* e operações de manipulação de bits. No módulo seguinte, cobrimos a representação de programas. Apresentamos os componentes visíveis da CPU e introduzimos a linguagem *assembly* da arquitetura IA-32. Passamos então a explorar a "tradução" de comandos e estruturas de controle de C (atribuições, condicionais, laços) para seus equivalentes em linguagem de montagem, e cobrimos em profundidade a implementação de procedimentos. Em seguida, temos um pequeno módulo sobre interrupções e *traps*, no qual os alunos aprendem como o sistema operacional é ativado e como funcionam os tratadores de interrupções. O módulo seguinte introduz corrotinas, e nele exploramos a implementação e uso de uma biblioteca que provê essa construção. Finalmente, o último módulo cobre o processo de amarração de programas.

A carga horária da disciplina é de quatro horas semanais. No modelo inicial, essas quatro horas eram divididas em uma aula teórica e uma aula de laboratório, ambas com duas horas de duração. Com essa divisão, porém, frequentemente nos desapontávamos ao ver, no início de cada laboratório, quão pouco os alunos haviam assimilado da apresentação teórica. Aumentamos então o número de unidades por módulo, diminuindo o conteúdo de cada unidade. Isso nos permitiu apresentar esse conteúdo e realizar o laboratório correspondente numa mesma aula; quando necessário, dedicamos uma outra aula a exercícios complementares, com maior complexidade[1].

Desde o ano passado, temos também substituido a exposição presencial de alguns tópicos por vídeo-aulas. Essa substituição tem várias motivações. Como disponibilizamos cada video com antecedência, e instruimos os alunos a assistí-lo em seu tempo livre, conseguimos aumentar o número de exercícios práticos no laboratório correspondente. Além disso, uma forma interessante de motivar nossos alunos, que frequentemente perdem a concentração em exposições muito longas, é a utilização de métodos de ensino mais próximos de seus hábitos. Finalmente, os alunos podem assistir ao vídeo, ou partes dele, tantas vezes quanto necessário para adquirir um entendimento adequado dos conceitos apresentados.

[1]O programa da disciplina e os exercícios de laboratório estão disponíveis em http://www.inf.puc-rio.br/~inf1018

III. O papel da pilha de execução na implementação de chamadas de procedimentos

Ainda que a maioria dos programadores raramente lide diretamente com a pilha de execução, sua compreensão é essencial para um programador tornar-se competente. Só com essa compreensão ele poderá dominar a diferença entre diferentes classes de armazenamento, como globais, locais e `static`. A ignorância sobre a pilha pode levar a código perigosamente ineficiente. Seu entendimento é necessário para entender o custo da recursão e a importância da recursão de cauda para otimização[2]. Compreender o comportamento da pilha durante chamadas a funções também é essencial para entender ataques como o recentemente famoso *Heartbleed* [6].

A maioria de nossos exercícios combinam C e *assembly*, com uma ou duas funções escritas em *assembly* e o resto do programa escrito em C. Esse padrão de uso permite que o código *assembly* seja mais facilmente testado, e ainda motiva compilação em separado, preparando o caminho para a aula sobre ligação na parte final do curso. O uso de compilação em separado também motiva a necessidade de convenções (passagem de parâmetros, valor de retorno, salvamento de registradores, etc.) que os alunos tendem a ignorar.

Antes de chegarmos às unidades sobre pilha de execução, os alunos têm duas aulas de laboratório sobre tradução de C para *assembly*, basicamente explorando a tradução de estruturas condicionais e de repetição. Nessas aulas, pedimos que traduzam programas muito simples contendo apenas a função `main`. Para não precisar explicar o código inicial e final de uma função (*enter* e *leave*) e para permitir que chamem `printf`, fornecemos esqueletos de programas que eles preenchem. Apenas depois de explicar o funcionamento da pilha de execução é que podemos explicar todo o código de uma função.

A pilha de execução é o assunto mais importante do curso, e aquele ao qual dedicamos mais tempo: pelo menos seis aulas. Quando começamos a ensinar o curso, nossa cobertura do assunto era menos extensa, mas achamos que não era suficiente. Aos poucos quebramos o assunto nas quatro unidades seguintes: introdução, a visão da função chamadora, a visão da função chamada e variáveis locais. Normalmente alocamos duas aulas a uma combinação das duas primeiras unidades, e depois duas aulas para cada uma das outras duas. Além de mais oportunidades para exercícios *hands-on*, essa apresentação passo a passo se mostrou mais palatável para os estudantes, que ficavam afogados com o excesso de informação de cada aula na organização inicial.

A seguir descrevemos cada uma das unidades atuais.

1) *introdução*
   Começamos essa unidade descrevendo as instruções `push` e `pop` e sua relação com o registrador `%esp` (*stack pointer*). A seguir motivamos a necessidade de colocar endereços de retorno na pilha e apresentamos as instruções `call` e `ret`.
   Nesse ponto podemos pedir aos estudantes que traduzam para *assembly* a chamada de uma função que

[2]Nesse curso, muitas vezes nos deparamos com a alegria dos alunos que acabam de compreender a implementação da recursão em *assembly* e com isso podem concretizar seu entendimento de chamadas recursivas.

recebe apenas um parâmetro e não retorna resultados (a chamada à função `system` para mostrar o conteúdo do diretório corrente.) Ainda que esse seja um exercício muito simples, ele permite quebrar o gelo em relação à compilação de programas com partes em C e partes em *assembly*, e exige uma primeira passagem de argumento via pilha.

2) *visão da função chamadora*
Nesta unidade os alunos aprendem como uma função passa argumentos para outra e como recebe um resultado inteiro. Como mencionamos antes, procuramos enfatizar a necessidade de convenções, lembrando que nos exercícios do laboratório, o código escrito para uma função tem que ser compatível com aquele gerado pelo compilador para outra função.
Os estudantes desenvolvem então uma tarefa de tradução um pouco mais interessante, traduzindo uma `main` que contém uma chamada a `printf`, onde o segundo argumento é o resultado de uma chamada a outra função (escrita em C). Essa é a última aula de laboratório em que precisamos dos esqueletos de *assembly* que mencionamos, pois ainda não explicamos o código inicial e final de uma função.

3) *visão da função chamada*
Nessa unidade, finalmente os alunos dominam o código completo de uma função escrita em *assembly* (ainda sem alocação de variáveis locais). Nesse momento discutimos o papel do registrador `%ebp` no acesso aos argumentos e as convenções de salvamento de registradores. Isto permite entender os trechos iniciais e finais utilizados por padrão. Mencionamos que o registro de ativação também será usado para variáveis locais mas ainda não explicamos como é esse uso (Variáveis inteiras são frequentemente alocadas em registradores, e neste ponto do curso só tratamos com inteiros, então não é artificial adiar a questão de alocação de locais.).
Ao longo dos semestres em que ensinamos a disciplina, percebemos que é importante explorar os mecanismos apresentados até aqui com um número grande de exercícios para que os alunos possam concretizar sua compreensão. Normalmente dedicamos três aulas a exercícios de tradução de funções C de complexidade crescente. As primeiras funções recebem valores de tipos básicos e realizam tarefas simples. A seguir, apresentamos para tradução funções que recebem arrays como argumento, e depois structs, explorando diferentes níveis de indireção. Pedimos também que traduzam a função fatorial como primeiro exemplo de código com recursão, e depois aprofundamos o entendimento desse mecanismo com funções que percorrem estruturas de dados recursivas. Além de concretizar o entendimento da pilha, essas aulas são fundamentais para construir compreensão sobre ponteiros, arrays e indireção.

4) *variáveis locais*
A última unidade sobre implementação de funções trata da alocação de variáveis locais. Aqui estendemos a discussão anterior sobre registros de ativação e o uso do registrador `%ebp`. Motivamos a necessidade de alocar variáveis locais (em vez de deixá-las todas em registradores) com arrays locais e chamadas de funções que recebem endereços como argumentos (por exemplo, `scanf`). Essa é uma das unidades nas quais os alunos assistem a um video antes da aula[3].

IV. Invadindo a Pilha de Execução

Depois de alguns anos ensinando a disciplina, começamos a procurar outras oportunidades para tornar a pilha de execução mais concreta para os alunos. O livro de Bryant e O'Hallaron provê um excelente material para isso: o exercício *Buffer Bomb*. Esse exercício é um programa C que contém uma função `getbuf`; essa função usa a função da biblioteca `gets` para ler uma cadeia, armazenando-a em um vetor local. A ideia é fazer os alunos inspecionarem o código objeto, usando um *desassembly*, para poder sobrescrever a pilha de diversas formas.

Em nossa versão para o exercício, os alunos realizam apenas parte da tarefa original (as duas horas de que dispomos para o laboratório não são suficientes para a completar toda a tarefa). Em resumo, os alunos devem sobrescrever o endereço de retorno de `getbuf` com o endereço de uma outra função do programa, preparar o endereço de retorno dessa função chamada "ilegalmente" e, finalmente, preparar argumentos apropriados para uma função "invocada" dessa maneira.

O resultado deste laboratório tem sido altamente produtivo. Os alunos tem comentado que o exercício contribui significativamente para seu entendimento da pilha. Sobrescrever a posição da pilha que contém o endereço de retorno de `getbuf` com o endereço de entrada de uma outra função permite que os alunos assimilem o funcionamento de chamadas de função. A necessidade de colocar os argumentos para a função invocada artificialmente logo abaixo do endereço de retorno de `getbuf` perturba-os a princípio, mas força-os a pensar sobre a separação entre os pontos de vista da função chamadora e da função chamada. Além disso, o laboratório *Buffer Bomb* é uma excelente oportunidade para discutirmos ataques à pilha de execução e requisitos para um código seguro.

V. Corrotinas

O crescente interesse em sistemas concorrentes e paralelos motivou-nos a introduzir um tópico que apresenta uma construção básica de concorrência: corrotinas [7]. Escolhemos essa construção específica por duas razões. Em primeiro lugar, este é, geralmente, o primeiro momento em que nossos alunos tem contato com a ideia de fluxos de execução concorrentes, e corrotinas permitem-nos oferecer uma introdução mais leve à programação concorrente, evitando as complicações de mecanismos de concorrência mais complexos. Em segundo lugar, é viável implementar uma biblioteca de corrotinas bastante realista, totalmente baseada nos mecanismos que os alunos já estudaram durante a disciplina.

O tópico sobre corrotinas é trabalhado depois de completarmos as unidades que envolvem a implementação de procedimentos. Começamos introduzindo a ideia de um fluxo de execução (um tipo especial de procedimento) que pode ser "suspenso" e retomado mais tarde, continuando sua execução a partir de seu ponto de suspensão. Apresentamos a seguir o conceito de corrotinas assimétricas, conforme implementado

[3] Video disponível em https://vimeo.com/107626475, senha inf1018

pela linguagem Lua [8], e discutimos alguns de seus usos típicos: geradores/iteradores e *multitasking* cooperativo. Este é um bom momento para discutir brevemente as diferenças entre *multitasking* a nível de sistema e de aplicação, e algumas das questões envolvidas em cada um desses mecanismos.

Depois de apresentarmos o conceito de corrotinas, passamos a discutir o suporte para a implementação desse mecanismo: a necessidade de preservar estado entre invocações sucessivas, onde esse estado está localizado, e como pode ser preservado (basicamente, provendo uma pilha para cada corrotina)[4]. Ilustramos então esse suporte navegando com os alunos pelo código de uma pequena biblioteca de corrotinas que desenvolvemos para C (disponível em http://www.inf.puc-rio.br/~inf1018/coro.tar).

Os laboratórios práticos para esse tópico exploram a biblioteca para a implementação de alguns usos típicos de corrotinas. A primeira tarefa é bastante simples: os alunos devem implementar duas corrotinas semelhantes (isto é, que compartilham o mesmo código), cada uma delas provendo um contador inteiro. A cada invocação, a corrotina deve exibir quantas vezes foi invocada, incrementar seu contador local, e suspender sua execução, retornando a seu chamador o valor corrente do contador. Os alunos devem também implementar uma função `main`, que cria as duas corrotinas e as invoca repetidamente, exibindo o valor resultante de cada invocação. Como os alunos não têm experiência prévia com programação concorrente, essa tarefa é importante para que eles assimilem a ideia de suspender e retomar um fluxo de execução. Nas outras tarefas, os alunos desenvolvem alguns usos mais elaborados de corrotinas: um iterador para uma estrutura de dados e um escalonador simples para um ambiente de *multitasking* cooperativo. Ainda estamos trabalhando para melhorar esse último exercício, ilustrando esse tipo de uso com um exemplo simples, porém mais realista.

Nossa proposta de incluir conceitos básicos de concorrência na disciplina de Software Básico fez parte de um projeto da IEEE que incentiva a introdução de tópicos relacionados a paralelismo em cursos de graduação [9]. Fizemos alguns ajustes à proposta inicial, para permitir que os alunos assimilem os conceitos apresentados com mais facilidade. Trabalhamos agora com um único mecanismo de transferência de controle (*corrotinas assimétricas*) e com uma única implementação desse mecanismo (nossa biblioteca de corrotinas para C).

VI. Considerações Finais

Um entendimento sólido da hierarquia de abstrações que compõe um sistema de computação é essencial para estudantes e profissionais de Computação, e esse entendimento envolve não apenas a assimilação dos conceitos básicos de cada nível de abstração, mas também de como esses diferentes níveis interagem. Afinal, as abstrações *vazam* [10], e compreender como mecanismos mais básicos proveem suporte para representações de nível mais alto é um requisito indispensável para a proficiência em quase qualquer domínio, da programação de sistemas ao desenvolvimento de aplicações.

Nossa disciplina de Software Básico tem como foco o suporte provido pelos componentes básicos de uma plataforma de computação real às abstrações oferecidas por uma linguagem de programação, e a pilha de execução é um tópico central da disciplina. Neste trabalho, mostramos como exploramos a pilha de execução de forma a prover a nossos alunos um entendimento mais profundo desse mecanismo e de seu uso para a implementação de diversas construções de programação. Com esse entendimento, acreditamos que nossos alunos tornam-se capazes de desenvolver programas mais corretos, seguros e eficientes.

Além das exposições teóricas e exercícios práticos descritos neste trabalho, exploramos também a pilha de execução em outras iniciativas eventuais. Uma delas é em um dos trabalhos de programação que os alunos desenvolvem durante a disciplina. Neste trabalho, introduzimos o conceito de *fechos* (*closures*), e pedimos aos alunos para implementá-lo para desenvolver um *gerador de funções* dinâmico[5].

Agradecimentos

O Professor Roberto Ierusalimschy identificou a necessidade de uma disciplina que apresentasse os componentes básicos de um sistema de computação com a visão de um projetista de *software*, e foi responsável por diversas turmas ao longo dos anos. Ele também desenvolveu o primeiro conjunto de exercícios para a disciplina.

O artigo de Aleph One [11] inspirou o subtítulo deste trabalho, e é também um material complementar para nosso laboratório de invasão da pilha (*Buffer Bomb*).

Referências

[1] R. Sebesta, *Concepts of Programming Languages*, 10th ed. Pearson, 2012.

[2] M. Scott, *Programming Language Pragmatics*, 3rd ed. Morgan Kaufmann, 2009.

[3] C. Ghezzi and M. Jazayeri, *Programming Languages Concepts*, 3rd ed. Wiley, 1997.

[4] A. L. de Moura and N. Rodriguez, "Sistemas de computação sob o ponto de vista do desenvolvedor de software," in *Anais do XXXIV CSBC — WEI (Workshop de Educação em Computação)*, 2014.

[5] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 2nd ed. Prentice-Hall, 2011.

[6] (2014) Heartbleed. [Online]. Available: https://en.wikipedia.org/wiki/Heartbleed

[7] A. L. Moura and R. Ierusalimschy, "Revisiting coroutines," *ACM Transactions on Programmming Languages and Systems*, vol. 31, no. 2, pp. 6:1–6:31, Feb. 2009.

[8] R. Ierusalimschy, *Programming in Lua*, 3rd ed. Lua.org, 2012.

[9] A. Branco, A. L. Moura, N. Rodriguez, and S. Rossetto, "Teaching concurrent and distributed computing – initiatives in Rio de Janeiro," in *Proc. IEEE 27th International Parallel and Distributed Processing Simpoium Workshops & PhD Forum (IPDPSW)*, 2013.

[10] J. Spolsky. (2002) The law of leaky abstractions. [Online]. Available: http://www.joelonsoftware.com/articles/LeakyAbstractions.html

[11] (1996) Smashing the stack for fun and profit. Phrack.org. [Online]. Available: http://phrack.org/issues/49/14.html#article

[4] Para permitir a implementação de *multitasking*, trabalhamos com o conceito de corrotinas *completas* (*stackful coroutines*) [7].

[5] Um exemplo desse trabalho de programação está disponível em http://www.inf.puc-rio.br/~inf1018/2014.1/trabs/t2/trab2.html