Sistemas de Computação sob o Ponto de Vista do Desenvolvedor de Software

Noemi Rodriguez, Ana Lúcia de Moura

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)

{noemi,amoura}@inf.puc-rio.br

Abstract. *Most undergraduate programs related to Computer Science offer a Computer Systems course with a systems architect's perspective. The Systems Software course we have been teaching at PUC-Rio over the last ten years emphasizes instead a software perspective; its main goal is to make students understand how data and control abstractions offered by C are mapped to the basic mechanisms of a typical architecture. In this paper we describe our experience teaching this course as well as the materials we have developed for it over the years. We also discuss our recent introduction of a new topic on basic concurrency constructs.*

Resumo. *O conteúdo tradicional de uma disciplina de Sistemas de Computação adota a perspectiva de um projetista desses sistemas. A disciplina que oferecemos há mais de dez anos a alunos de graduação na PUC-Rio (Software Básico) adota uma perspectiva que enfatiza o suporte que componentes básicos de uma arquitetura típica provêem para a implementação das abstrações de dados e de controle oferecidas por uma linguagem de programação convencional. Este artigo descreve nossa experiência ministrando esse curso e o material que desenvolvemos ao longo dos anos. Ele discute também a introdução de um novo tópico que apresenta construções básicas de concorrência.*

1. Introdução

Grande parte dos cursos de graduação relacionados à Computação oferece uma disciplina de Sistemas de Computação que adota a perspectiva de um arquiteto desses sistemas. Uma disciplina baseada nessa perspectiva tem, tipicamente, como foco principal o projeto e a implementação de componentes de arquiteturas de hardware e de *software básico*.

Na PUC-Rio, oferecemos três cursos de graduação ligados à área de Computação: Engenharia de Computação, Sistemas de Informação e Ciência de Computação. Os três cursos incluem uma disciplina chamada *Software Básico*, que inclui parte do conteúdo de cursos tradicionais de Arquitetura de Computadores, mas que adota uma perspectiva diferente, com ênfase em aspectos ligados à programação. O objetivo dessa disciplina é apresentar componentes básicos de uma arquitetura típica sob o ponto de vista de um desenvolvedor de software, discutindo e explorando o suporte que esses componentes oferecem para a implementação de abstrações de dados e estruturas de controle oferecidas por uma linguagem de programação convencional. Esses conceitos constroem a base para as disciplinas de Compiladores e de Sistemas Operacionais, e também possibilitam um

entendimento mais sólido do processo de desenvolvimento de programas, auxiliando na construção de programas mais corretos e eficientes.

O restante deste trabalho está organizado da seguinte forma. A seção 2 apresenta o programa atual da disciplina e descreve, com alguns exemplos, o material desenvolvido ao longo dos anos. A seção 3 discute nossa experiência ministrando a disciplina a alunos dos três cursos de graduação da PUC-Rio. A seção 4 discute a adição recente de um novo tópico ao programa da disciplina, que introduz e implementa uma construção básica de concorrência. Finalmente, a seção 5 apresenta algumas considerações finais.

2. Programa do Curso

Desde a concepção da disciplina, em 2000, a idéia foi imprimir um enfoque orientado a software. No entanto, não havia na época um bom livro texto para apoio a essa visão, e considerávamos o uso de um livro didático essencial para uma disciplina obrigatória. Em 2003, Bryant e O'Hallaron publicaram a primeira edição do livro *Computer Systems: A Programmer's Perspective* [Bryant and O'Hallaron 2011], baseado no material desenvolvido pelos autores para uma disciplina da Universidade de Carnegie Mellon. O livro tem uma visão bastante similar à nossa, propondo o estudo de componentes de uma arquitetura de hardware sob o ponto de vista do *usuário*, o projetista de software, e não do construtor dessa arquitetura. A partir de 2004 adotamos o texto de Bryant e O'Hallaron como nossa principal referência. Também disponibilizamos uma página com as aulas e laboratórios semana a semana, além de outros materiais de apoio (`www.inf.puc-rio.br/~inf1018`).

A disciplina é fortemente baseada em exercícios práticos: todos os os conceitos apresentados são exercitados em aulas de laboratório. Além disso, os estudantes desenvolvem dois trabalhos mais extensos. Tipicamente o primeiro trabalho explora conceitos de representação de dados (codificação, compactação, etc), enquanto o segundo aborda mecanismos de tradução de código.

O conteúdo da disciplina é organizado em quatro módulos principais: representação de dados, representação de programas, interrupções e o processo de compilação e ligação de programas. Apresentamos a seguir cada um desses módulos.

2.1. Representação de Dados

Neste módulo, apresentamos a representação dos tipos básicos e estruturados de C: números inteiros e de ponto flutuante, ponteiros, *arrays* e *structs*. Aqui também discutimos organizações de memória *little* e *big-endian* e restrições de alinhamento.

Para essa parte, usamos extensivamente uma função que faz `dump` de uma região de memória byte a byte:

```
void dump (void *p, int n) {
  unsigned char *p1 = (unsigned char *) p;
  while (n--) {
    printf("%p - %02x\n", p1, *p1);
    p1++;
  }
}
```

Usando essa função, os alunos podem observar concretamente como cada tipo de dado é representado na memória (organização *little-endian*, representação em complemento a dois, alinhamento, etc).

Este módulo também é importante como uma retomada de C com maior atenção para ponteiros e endereços. Os estudantes têm experiência anterior de programação em C, mas aqui aprofundamos assuntos como aritmética de ponteiros e a relação entre *arrays* e ponteiros.

Os tópicos abordados são os seguintes:

1. Memória principal: bits e bytes; palavras e endereçamento; notação binária e hexadecimal; inteiros sem sinal; os tipos *unsigned* de C; códigos de caracteres; limitações de representação e o conceito de *overflow*.
   Nos exercícios referentes a este tópico, a função `dump` é utilizada para conferir resultados de exercícios onde se discute a implementação dos tipos inteiros sem sinal de C. Outro grupo de exercícios relembra a codificação de caracteres, trabalhando com uma função semelhante a `atoi` de transformação de strings em inteiros.

2. Operações de manipulação de bits: *and*, *or*, *xor*, deslocamentos para esquerda e direita, lógicos e aritméticos.
   Nos exercícios deste tópico os alunos trabalham manipulações de bits com exemplos motivados pelo tratamento de campos de protocolos de rede e pela implementação de operações aritméticas.

3. Representação de números negativos; alternativas; complemento a 2; overflow; os tipos *signed* de C; extensão de representação.
   Novamente, recorremos à função função `dump` para que os estudantes adquiram domínio da representação em complemento a 2. Programas envolvendo comparações são usados para motivar a necessidade de conjuntos diferentes de instruções para valores inteiros com e sem sinal.

4. Implementação de arrays e registros; *padding*; cálculo de tamanhos e de deslocamentos;
   Aqui exploramos a relação entre o `sizeof` de um `struct` com o tamanho ocupado por um array de structs, e as restrições de alinhamento envolvidas na construção de registros e uniões. Os exercícios exploram diferentes exemplos de alinhamento e novamente os alunos utilizam a função `dump` para conferir suas respostas.

5. Representação de números em ponto flutuante; sinal e magnitude; representações IEEE e excesso 2^k.
   Apresentamos o padrão IEEE754 e as limitações da representação para cada precisão definida. Discutimos as aproximações envolvidas e os cuidados que devem ser tomados pelo programador ao trabalhar com valores desse tipo (por exemplo, na comparação entre dois valores de ponto flutuante). Os exercícios solidificam a compreensão dessas representações, explorando operações de manipulação de bits para implementar conversões de valores inteiros para ponto flutuante e vice-versa.

2.2. Representação de Programas

Este módulo introduz a linguagem *assembly*, utilizando a arquitetura IA-32 com o compilador `gcc` em ambiente Linux. Exploramos apenas um subconjunto das instruções e modos de endereçamento, que é suficiente para realizar a tradução de código C e para ilustrar características típicas de arquiteturas de hardware. O uso da arquitetura mais presente nos dias de hoje motiva os alunos e permite que desenvolvam os exercícios em múltiplos ambientes. Por outro lado, a limitação a um subconjunto de instruções facilita a utilização de uma arquitetura que é muito complexa para uma primeira aproximação com o assunto.

Os tópicos abordados são os seguintes:

1. CPU: ciclo de execução, registradores; apresentação de elementos da linguagem IA-32 e classes típicas de instruções; estrutura básica de um programa.
   Como observamos que o contato inicial dos estudantes com um programa *assembly* é um pouco assustador, o primeiro laboratório nesse módulo parte de um programa simples, já pronto, pedindo alterações relativamente pequenas para permitir que os alunos se ambientem com sintaxe e semântica.

2. Tradução de estruturas de controle de C.
   A partir desse tópico, todo os exercícios com *assembly* são pedidos de tradução de trechos de C; neste caso específico, pedimos que os alunos apliquem os padrões de tradução apresentados em sala para a tradução de comandos condicionais e de repetição (`if`, `while` e `for`).

3. Pilha de execução: registro de ativação; modelo de chamadas e passagem de parâmetros; suporte a chamadas recursivas; variáveis locais.
   Este assunto é central ao curso e consome em torno de seis aulas, entre teoria e prática. A pilha de execução é uma abstração presente em muitas discussões em outras disciplinas de programação, e aqui os alunos têm a chance de concretizar seu entendimento.
   Como nosso objetivo não é que o estudante se torne um exímio programador *assembly*, os laboratórios tipicamente misturam programação C e *assembly*, com apenas uma ou duas funções escritas em *assembly* e o restante do programa em C. Além de permitir que os conceitos sejam testados com mais simplicidade no desenvolvimento, esse padrão de uso motiva a compilação em separado e a necessidade de convenções (passagem de argumentos, retorno, uso de registradores, etc.). O uso da compilação em separado também permite que o professor discuta, em paralelo ao assunto principal, o processo de compilação e ligação, que é retomado de forma explicita em um módulo futuro.

4. Código de Máquina.
   Esse tópico tem caráter puramente prático. Através de exercícios, os estudantes travam contato com a representação hexadecimal de instruções de máquina. Observamos que, para a maioria dos alunos, a idéia de que o programa executável se resume a uma sequência de bytes na memória é de difícil assimilação, e esse tópico oferece uma oportunidade de concretização desse conceito.

Inicialmente, pedimos aos alunos que obtenham um `dump` do código gerado para uma função simples e que preencham manualmente uma região de memória com a sequência de bytes obtida. Depois disso, pedimos que façam um `cast` do endereço desse trecho de memória para um ponteiro de função e chamem a função obtida. O mesmo laboratório é utilizado para introduzir a noção de relocação através do cálculo manual do deslocamento entre o endereço de uma função e de seu ponto de chamada.

2.3. Interrupções e Traps

Neste módulo apresentamos os mecanismos de exceção, motivando a interrupção no fluxo normal de controle para casos síncronos (*traps*) e assíncronos (interrupções). Discutimos os requisitos para que o tratamento de exceções seja transparente para o programa em execução (salvamento de contexto, desvio para tratadores específicos, etc.). Apresentamos também o uso de *traps* para comunicação com o sistema operacional.

No exercício prático, exploramos a comunicação com o sistema operacional Linux via *traps*, implementando chamadas diretas ao sistema para realizar leituras e escritas em um arquivo.

2.4. Compilação e Ligação de Programas

O último módulo da disciplina trata do processo de ligação de módulos para a geração de um programa executável. Apresentamos as informações que compõem um módulo objeto, e o uso dessas informações pelo ligador para a execução de suas tarefas básicas (resolução de referências externas, relocação, etc.). Essa discussão organiza a experiência dos alunos com o uso de compilação em separado, explorado na maioria dos exercícios da disciplina.

Nas disciplinas de programação anteriores, os estudantes tipicamente utilizam ambientes integrados de desenvolvimento (IDEs). Esta disciplina cumpre o papel de desvendar as etapas de geração de um programa executável, escondidas por trás das interfaces desses ambientes.

3. Experiência

Um aspecto de resultado muito positivo da disciplina é seu caráter prático. Nos primeiros anos, a disciplina era organizada no formato de uma aula teórica e uma prática a cada semana. Ao longo do tempo, tornou-se cada vez mais claro que os estudantes fixavam apenas parte do conteúdo na aula expositiva, e que as aulas práticas eram fundamentais para a maior compreensão e assimilação do tópico exposto. Além disso, a divisão em tópicos não correspondia exatamente a essa estrutura, pois em alguns casos o conteúdo teórico não era suficiente para as duas horas de aula expositiva e, em muitos outros, o tempo de aula prática era insuficiente para os alunos completarem um conjunto adequado de exercícios. As aulas práticas também constituem a parte mais motivadora da disciplina, e oferecem muito mais oportunidades de interação. Nas aulas expositivas, são poucos os alunos que levantam dúvidas ou questionamentos, enquanto que nas aulas práticas não há como avançar nos exercícios sem entender o conteúdo envolvido.

De alguns anos para cá, adotamos uma nova organização, onde todas as aulas são dadas em laboratório. Mesmo que algumas (poucas) aulas sejam predominantemente

expositivas, procuramos sempre propor pelo menos um exercício prático para fixação do conteúdo. Para isto, reestruturamos a divisão dos módulos em tópicos de forma a tornar as exposições menores e introduzir exercícios mais seguidamente. Esse foi um aprendizado importante para nossa didática: retiramos alguns detalhamentos das aulas expositivas e, na maioria das vezes, tivemos oportunidade de esses mesmos detalhes em exercícios práticos.

Nos primeiros períodos em que ministramos a disciplina, o conteúdo de cada módulo descrito na Seção 2 era esgotado antes de passarmos ao próximo módulo. Percebemos, contudo, que seria interessante antecipar a chegada à programação *assembly*, por duas razões. Em primeiro lugar porque o *assembly* fornece um acesso concreto à memória, consolidando a compreensão da representação de dados. Em segundo lugar, a novidade de uma forma diferente de programar é um fator de motivação para os alunos, além de representar uma quebra em uma sequência de aulas, já bastante extensa, sobre assuntos relacionados. Assim, atualmente postergamos a apresentação da representação de ponto flutuante para depois do estudo de tradução básicas de programas.

Em termos do conteúdo apresentado, a disciplina permaneceu bastante estável ao longo dos anos, com uma única exceção. Há cerca de dois anos incluímos um novo tópico que apresenta o conceito e implementação de uma construção básica de concorrência: corotinas. Da mesma forma que a exposição dos alunos aos assuntos de tradução de programas permite que entendam melhor processos de compilação, o estudo de corotinas é um primeiro passo para a compreensão e uso de mecanismos de concorrência mais complexos. Na próxima seção descrevemos esse novo tópico.

4. Introdução de Conceitos Básicos de Concorrência

A principal motivação para a introdução de um tópico sobre corotinas [Moura and Ierusalimschy 2009] foi o interesse corrente em modelos de concorrência e paralelismo. Muitos desses modelos exploram fluxos de execução de peso leve implementados no nível de aplicação. Em nossa experiência com ensino de concorrência, percebemos que os estudantes têm muita dificuldade em entender concretamente qual a diferença de se trabalhar com fluxos de nível de aplicação e de nível de sistema operacional e como uma implementação de fluxos concorrentes em nível de aplicação pode ser realizada. Neste novo tópico, mostramos uma implementação de corotinas que é bastante realista e que é totalmente baseada em mecanismos estudados ao longo da disciplina. Essa nossa experiência fez parte de um projeto da IEEE de incentivo à introdução de tópicos ligados a paralelismo em cursos de graduação [Branco et al. 2013].

Dedicamos três aulas ao assunto, apresentadas depois que os alunos já dominam os conceitos relacionados a subrotinas, em especial a pilha de execução. No novo tópico, os alunos aprendem o que são corotinas e seus usos típicos. Para a apresentação teórica, nos inspiramos parcialmente no material do livro de Linguagens de Programação de Michael Scott [Scott 2009] e na implementação de corotinas oferecida pela linguagem Lua [Ierusalimschy 2012]. Inicialmente, utilizamos corotinas em Lua também para os primeiros exemplos em laboratório, mas concluímos que era confuso para os alunos utilizarem duas linguagens com nível de abstração tão diferentes.

Desenvolvemos uma pequena biblioteca de corotinas em C (`www.inf.puc-rio.br/~inf1018/coro.tar`), que utilizamos tanto na aula expositiva como

nas aulas práticas. Na parte expositiva, recorremos à explicação do código, passo a passo, para discutir uma possível implementação de fluxos concorrentes a nível de aplicação. Na parte prática, os alunos usam essa biblioteca para desenvolver alguns dos exemplos típicos de aplicação de corotinas.

O primeiro exercício é bastante simples. Nele os alunos criam duas corotinas que incrementam e devolvem o valor de um contador local, e no programa principal invocam alternadamente essas duas corotinas. Como os estudantes não tem experiência anterior com programação concorrente, esse exercício é importante para assimilarem a idéia de suspender um fluxo de execução e retomá-lo posteriormente, no mesmo contexto em que se encontrava quando foi suspenso. A partir daí, os alunos desenvolvem outros exemplos mais elaborados. O primeiro deles é a implementação de um iterador para uma estrutura de dados e o segundo um escalonador simples para um ambiente de tarefas cooperativas.

Os alunos demonstraram bastante interesse nesse novo tópico, em especial com os resultados obtidos no laboratório. Ficou claro que nem todos os alunos atingiram o mesmo grau de compreensão, o que parece razoável dada a complexidade dos conceitos envolvidos.

5. Considerações Finais

Apesar de não termos uma avaliação formal do resultado da introdução dessa disciplina nos currículos dos cursos de graduação da PUC-Rio, há alguns indicadores bastante positivos e encorajadores. Nas disciplinas de Sistemas Operacionais e de Compiladores, os professores já partem da base construída em Software Básico. Em conversas informais com antigos alunos, eles muitas vezes mencionam situações onde o conteúdo apresentado na disciplina, e sua forma de apresentação, foi importante. A PUC-Rio também mantém um sistema de avaliação por questionários, oferecido aos alunos na renovação de matrícula semestral. A resposta dos alunos é altamente positiva tanto no que diz respeito ao conteúdo quanto ao formato da disciplina. Outro indicador que consideramos importante é a crescente presença, na disciplina, de alunos de outras Engenharias (Automação, Elétrica e até Mecânica), para os quais a disciplina não é obrigatória.

Acreditamos também que a disciplina vem contribuindo para a formação de programadores que fazem melhor uso dos mecanismos oferecidos por linguagens convencionais, por entenderem o que está por trás desses mecanismos. Um bom entendimento das representações de dados é importante para entender e resolver diversos erros comuns de programação causados pelas limitações impostas por essas representações. A concretização da pilha de execução é fundamental para a formação de um programador proficiente. Além disso, o domínio do modelo de execução baseado em pilha permite entender questões básicas de segurança como, por exemplo, ataques de *buffer overflow*.

Agradecimentos

Gostaríamos de agradecer aos colegas que contribuiram para o trabalho descrito neste artigo. Os Profs Renato Cerqueira, Markus Endler e Luiz Fernando Seibel foram responsáveis por turmas dessa disciplina em alguns semestres e contribuíram para a evolução de seu conteúdo. O Prof. Roberto Ierusalimschy, também responsável por diversas turmas ao longo desses anos, foi quem identificou a necessidade de uma disciplina com a visão do projetista de software e foi também quem desenhou o primeiro conjunto de aulas práticas.

Referências

Branco, A., Moura, A. L., Rodriguez, N., and Rossetto, S. (2013). Teaching concurrent and distributed computing – initiatives in Rio de Janeiro. In *Proc. IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1318–1323.

Bryant, R. E. and O'Hallaron, D. R. (2011). *Computer Systems: A Programmer's Perspective*. Prentice-Hall, 2nd edition.

Ierusalimschy, R. (2012). *Programming in Lua*. Lua.org, thirs edition.

Moura, A. L. and Ierusalimschy, R. (2009). Revisiting coroutines. *ACM Transactions on Programmming Languages and Systems*, 31(2):6:1–6:31.

Scott, M. (2009). *Programming Language Pragmatics*. Morgan Kaufmann, 3rd edition.