IM420 Sistemas Embarcados de Tempo Real

Notas de Aula – Semana 05

Prof. Denis Loubach
dloubach@fem.unicamp.br

Programa de Pós-Graduação em Eng. Mecânica / Área de Mecatrônica
Faculdade de Engenharia Mecânica - FEM
Universidade Estadual de Campinas - UNICAMP

1^o Semestre de 2018

Tópicos

SRI's Micro Robots Can Now Manufacture Their Own Tools

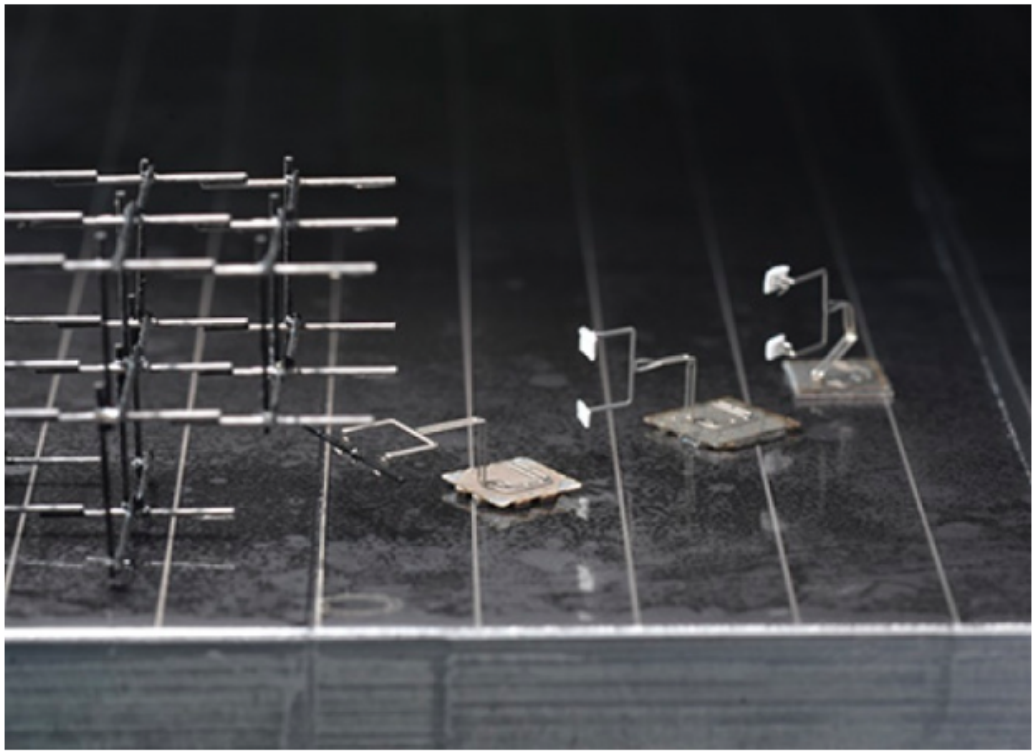

Image: SRI

"SRI's earlier generation of micro robots (each one smaller than a dime) consisted of multiple bots with different tools. Now the micro robots can not only reconfigure themselves into custom tools but also fabricate new end-effectors."

Fonte: `http://spectrum.ieee.org/automaton/robotics/industrial-robots/sri-micro-robots-can-now-manufacture-their-own-tools`

Breve histórico sobre SO (1/5)

- No início da computação, desenvolvedores criavam aplicações de software que incluíam código de máquina de baixo-nível para inicializar e interagir com o hardware diretamente
- Aplicações não-portáveis
- Pequenas mudanças no hardware ocasionavam retrabalho para o lado do software
- A indústria de software evoluiu
- SOs que proviam uma camada básica de software também evoluíram
- Facilidade da abstração da plataforma de hardware por parte do software de aplicação

Breve histórico sobre SO (2/5)

- Tal evolução dos SOs ajudou o projeto de software de aplicação
- Considerando aplicações grandes e monolíticas
- Para aplicações modulares e interconectadas
- Rodando no topo de um ambiente com um SO

Breve histórico sobre SO (3/5)

- Ao longo dos anos, muitos sistemas operacionais surgiram
- Desde SO de propósitos gerais (*General-Purpose Operating Systems* - GPOS) (ex. UNIX e *Windows*)
- Até RTOS menores e mais compactos (ex. *VxWorks*)

Breve histórico sobre SO (4/5)

- Nas décadas de 60 e 70, o UNIX foi desenvolvido para facilitar o acesso multi-usuário para um sistemas de computador caros e com disponibilidade limitada (*i.e., mainframes*)
- Nos anos 80, a *Microsoft* introduziu o *Windows* que enfatizava o ambiente de computadores pessoais
- No final da década de 80, deu-se início a era da computação embarcada
- Alguns RTOS comerciais começaram a aparecer e figurar entre os GPOS (ex. *VxWorks*)

Breve histórico sobre SO (5/5)

- Embora existam algumas similaridades entre RTOS e GPOS, muitas diferenças fundamentais existem
- Tais diferenças explicam porquê RTOS são mais aplicáveis para sistemas embarcados de tempo real

Algumas similaridades entre RTOS e GPOS

- Algum nível de multitarefa
- Gerenciamento de recursos de hardware e de software
- Provisão de serviços básicos do SO para a aplicação
- Abstração do hardware por parte do software de aplicação

Principais diferenças entre RTOS e GPOS

- Mais confiáveis no contexto de sistemas embarcados
- Escalabilidade (para baixo ou para cima) para satisfazer requisitos das aplicações embarcadas
- Preocupação com performance (tempo de execução, determinismo)
- Requisitos de memória reduzidos
- Políticas de escalonamento customizáveis para sistemas de tempo real
- Suporte para aplicações sem memórias externas (inicialização e execução a partir da ROM ou RAM)
- Portabilidade entre plataformas de hardware diferentes

Exemplos sistemas operacionais

- Windows
- Linux
- FreeBSD
- OS X

Todos possuem um *shell* (baseado em texto) e também uma *graphical user interface* (GUI)
A maioria dos SOs possuem dois modos básicos de operação

- *kernel mode* ou *supervisor mode*
- *user mode*

Um SO é considerado a parte mais fundamental do software e roda no *kernel mode*
Assim ele tem acesso completo a todo hardware e pode executar qualquer instrução que o hardware for capaz de executar

Camadas de organização do sistema genérico

O resto do software deve rodar em modo usuário, que acessa apenas um subconjunto das instruções disponíveis

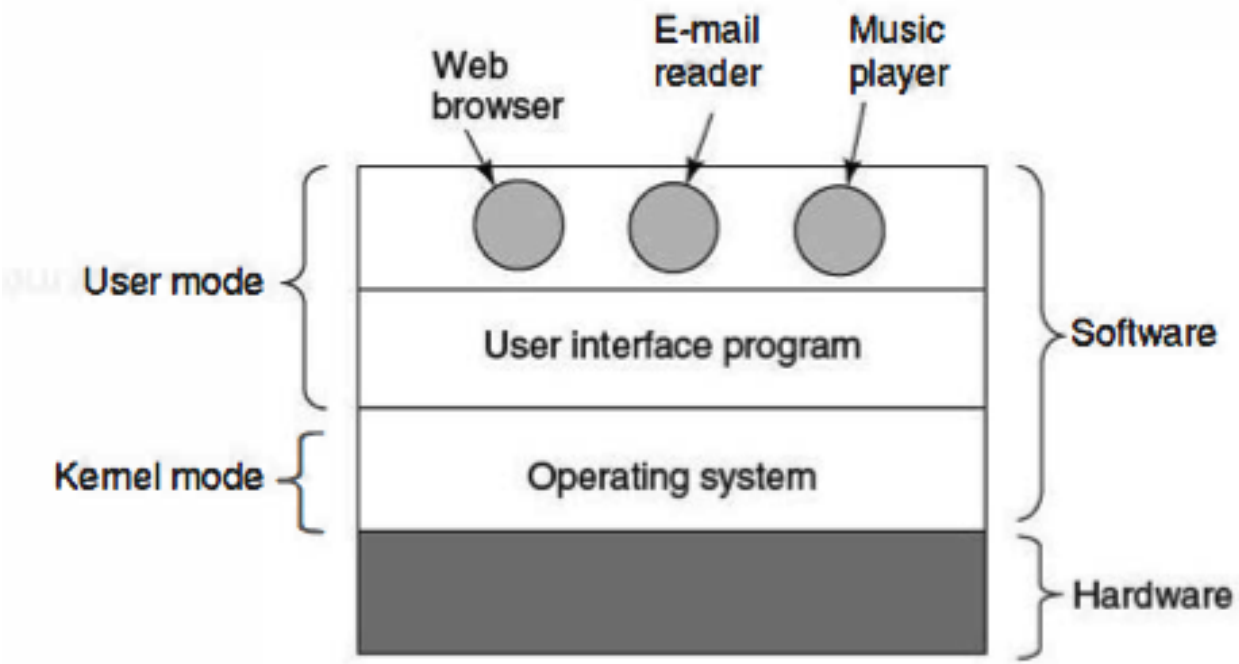

Figura: Onde o sistema operacional se encaixa

Sistemas operacionais de propósito geral são grandes (Linux possui cerca de 5 milhões de linhas de código para o núcleo somente) e complexos

Abstração

A abstração é a chave para de se gerenciar a complexidade

Isto é realizado pelas camadas apresentadas na figura anterior (ex: OS abstrai a camada de hardware de um software aplicativo)

Conforme citado em [1], o hardware é "feio"

Programar diretamente no hardware não é uma tarefa simples de ser executada, envolve possuir um *background* considerável

Uma das principais características de um SO consiste em "esconder"/abstrair o hardware e apresentar interfaces mais simples de serem programadas, mas que por sua vez fazem interface com o hardware no "baixo" nível

Interfaces de Hardware e SO

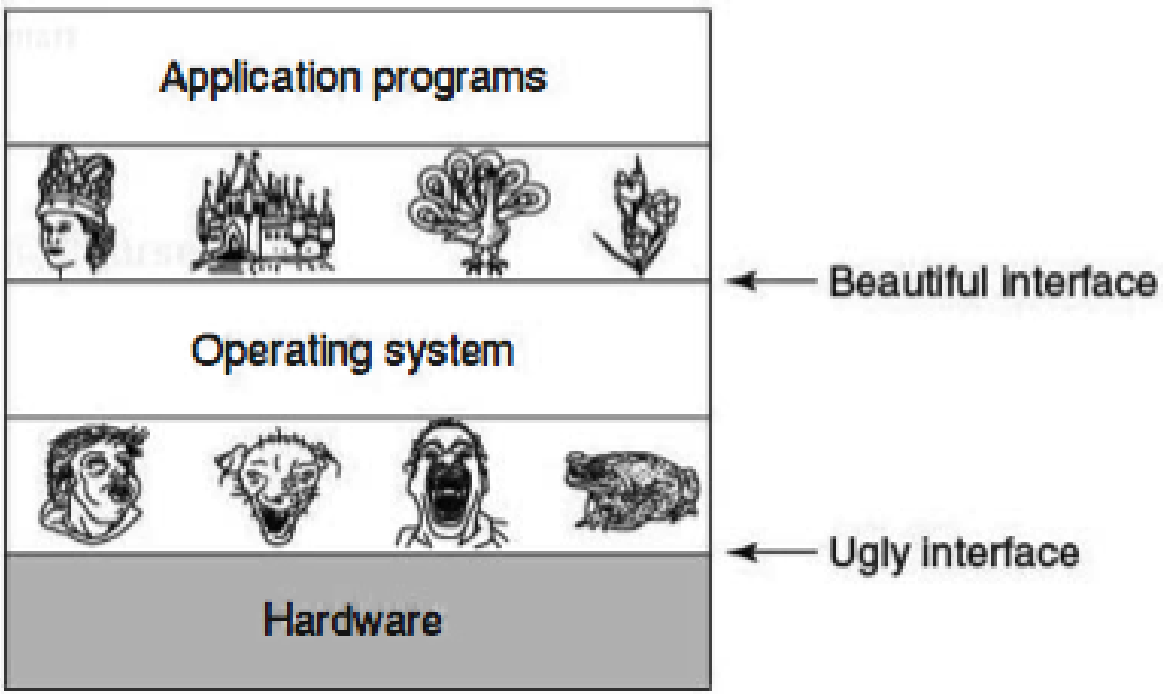

Figura: Interfaces "feias" do hardware

UNIX

Originado do projeto *MULTiplexed Information and Computing Service* (**MUTICS**)*

Previamente denominado *UNIplexed Information and Computing Service* (**UNICS**)**

Popular na academia, agências governamentais, e outras companhias

Código fonte amplamente disponível

Várias organizações desenvolveram sua "própria" variante, que também não era compatível com as demais

Obviamente, este fato tornou-se num caos

* `http://www.multicians.org`, [1], desenvolvido por MIT, *Bell Labs* e *General Electric*

** Conceito associado a Ken Thompson, cientista da computação que trabalhava na *Bell Labs*

POSIX

Visando unificar as tantas variações de UNIX existentes, o IEEE iniciou um projeto envolvendo membros neutros

Indústria, academia e governo participaram deste comitê

O projeto foi denominado **POSIX**, para *Portable Operating System Interface* adicionando-se o *X* para ficar "compatível" com o nome UNIX

O comitê elaborou um padrão conhecido como IEEE 1003.1

Conjunto de procedimentos (biblioteca) que todo sistema conformante com UNIX deve fornecer

Padrões POSIX (1/2)

- **IEEE Std 1003.1-1988**
  POSIX.1, Core Services (incorporates Standard ANSI C)
- **IEEE Std 1003.1b-1993**
  POSIX.1b, Real-time extensions (later appearing as librt - the Realtime Extensions library)
- **IEEE Std 1003.1c-1995**
  POSIX.1c, Threads extensions
- **IEEE Std 1003.2-1992**
  POSIX.2, Shell and Utilities

Padrões POSIX (2/2)

- **IEEE Std 1003.1-2001**
  POSIX.1-2001, equates to the Single UNIX Specification version 3
- **IEEE Std 1003.1-2004**
  involved a minor update of POSIX.1-2001
- **IEEE Std 1003.1-2013**
  As of 2016, Base Specifications, Issue 7 2013 edition represents the current version

MINIX

Em 1987, Andrew Stuart Tanenbaum[1], lançou uma versão clonada e reduzida do UNIX para propósitos acadêmicos

Esta versão foi denominada **MINIX**

11k8 linhas de código em C, e 800 em *assembly*

Funcionalmente similar ao UNIX incluindo suporte POSIX

Este sistema operacional pode ser encontrado em `http://www.minix3.org` (grátis e com código fonte aberto)

Linux

O desejo por uma versão de produção livre (oposta a educacional) de MINIX levou o então estudante Linus Torvalds a escrever o **Linux**, um outro clone reduzido do UNIX

Segundo [1], Linux foi diretamente inspirado e desenvolvido em cima do MINIX e, originalmente suportou várias características do MINIX (ex: sistema de arquivos)

A versão inicial era monolítica, ao invés de *micro-kernel*, com o sistema operacional inteiro no *kernel*

9k3 linhas de código em C, e 950 em *assembly*

Principais GUI: GNOME e KDE

Principais tipos de sistemas operacionais

- *Mainframe*
- Servidor
- *Multiprocessador*
- Computador pessoal
- Embarcados
- *Sensor node*
- *Real time*
- *Smart card*

Sistema operacional de *Mainframe*

Computadores que ocupam uma sala completa, ainda encontrados em grandes *data centers*

Capacidade de I/O

O sistema operacional para este tipo de computador é extremamente orientado para o processamento de vários *jobs* de um única vez

Esses por sua vez, necessitam de grandes capacidades de I/O (ex: reservas de companhias aéreas)

Sistema operacional de Servidor

Computadores grandes, *workstations*, ou mesmo *mainframes*

Objetivo de servir a múltiplos usuários ao mesmo tempo numa rede

Permitem o compartilhamento de hardware e software

Sistema operacional de *Multiprocessadores*

Computador com múltiplas CPUs num mesmo sistema

Sistema operacional que requer características especias para tratar de características como comunicação, conectividade e consistência

Sistema operacional de Computador pessoal

Suporte para multiprogramação

Vários software aplicativos sendo iniciados logo após o *boot*

Sistema operacional de Sistemas Embarcados

Footprint deve ser controlado

Dever ser extremamente customizável para poder ser aplicado aos diversos requisitos e restrições de sistemas embarcados, principalmente ligados a memória, performance e consumo de energia

Sistema operacional de *Sensor node*

Redes de pequenos sensores estão sendo desenvolvidas e aplicadas para uma quantidade enorme de aplicações

Cada nó consiste num computador que precisa se comunicar com outros ou com uma estação base

Os nós são alimentados (geralmente) por baterias

Estes rodam um sistema operacional real, porém pequeno. Também dever ser robusto para suportar possíveis falhas de comunicação

Sistema operacional de *Real time*

Foco deste curso

Deve ser capaz de responder aos eventos considerando *hard* ou *soft real time*

Sistema operacional de *Smart card*

Considerado o menor sistema operacional para rodar em *smart cards* (i.e., dispositivos do tamanho de cartão de crédito contendo uma CPU)

Restrições severas de poder de processamento e memória

Alguns são alimentados por contato com os leitores e outros sem contato por indução

Pipeline

Para melhorar a performance, CPUs modernas abandonaram o modelo simples de: **busca**, **decodificação** e **execução** de uma instrução por vez

Muitas CPUs modernas tem a habilidade de executar mais de um instrução ao mesmo tempo

Unidades separadas para busca, decodificação e execução das instruções

Enquanto estiver executando a instrução I_n, pode estar decodificando a instrução I_{n+1} e buscando a instrução I_{n+2}

CPU superescalar (1/3)

Conceito mais avançado que o *pipeline*

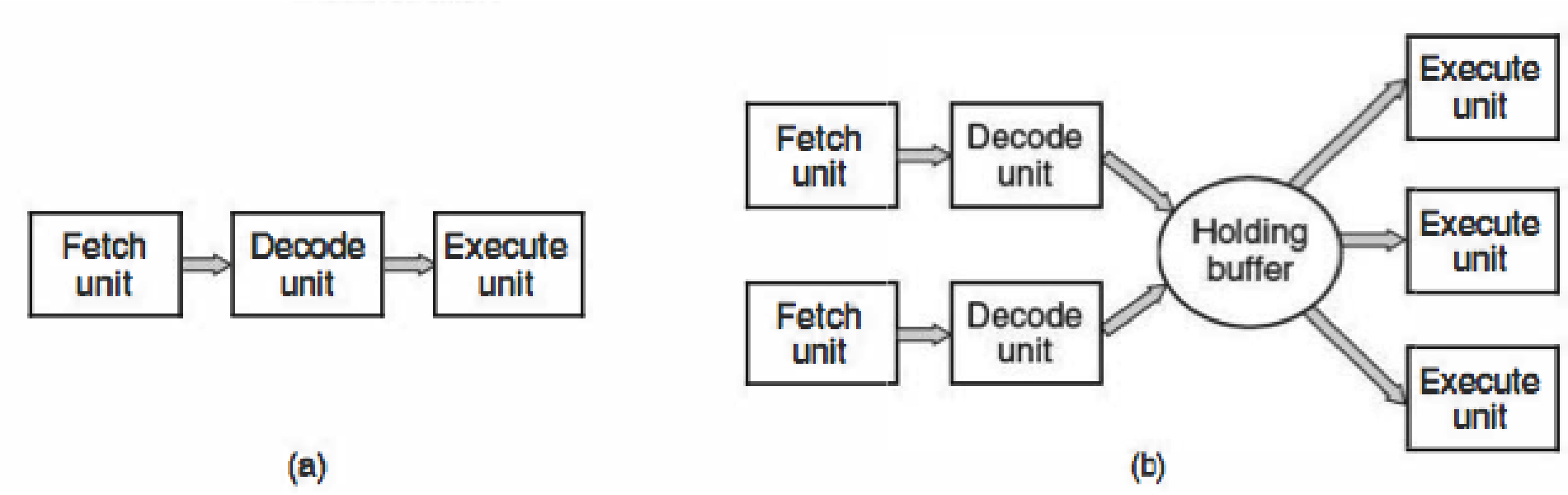

Figura: (a) *Pipeline* de 3 estágios, (b) CPU superescalar

CPU superescalar (2/3)

Múltiplas unidades de execução presentes (ex: uma para aritmética de inteiros, uma para aritmética de ponto flutuante, uma para operações *boolean*)

Duas ou mais instruções são **buscadas** de uma vez, **decodificadas** e lançadas num *buffer* até que possam ser **executadas**

Tão logo uma unidade de execução esteja livre, ela busca no *buffer* por um nova instrução, caso positivo esta instrução é removida do *buffer* e executada

CPU superescalar (3/3)

Este *design* implica na execução fora de ordem das instruções do software

A maior responsabilidade de garantir que o resultado produzido seja equivalente a uma execução sequencial é do hardware

Processos

Processo é um conceito chave em sistemas operacionais

Processo

Um processo é basicamente um programa em execução

Cada processo possui um espaço de endereço associado a ele

Espaço de endereço

Um espaço de endereço é uma lista de posições de memória indo de `0x0000` a um máximo, onde o processo pode ler/escrever

Este espaço contém o programa executável, dados e pilha

Pseudo-paralelismo

Num sistema **multiprogramação** ou **multitarefa**[1], o SO é capaz de chavear entre os vários possíveis processos/tarefas existentes. Isto está relacionado a questão de troca de contexto

Num sistema uniprocessado, a CPU só pode executar um processo por vez no tempo

Entretanto com a questão do chaveamento entre processos, tem-se a impressão/ilusão de **execução em paralelo**

De fato os processos são executados de forma sequencial, assim este conceito é conhecido como **pseudo-paralelismo**

[1] O termo multitarefa é algumas vezes utilizado para indicar várias tarefas num mesmo programa para serem gerenciadas de forma concorrente pelo SO. Multiprogramação seria para se referir a múltiplos processos de múltiplos programas. Entretanto, prefere-se equalizar estas definições de acordo com o *IEEE Std 100-1992*

Criação/Finalização de processos

Processo pode ser criado principalmente:

- Inicialização do sistema
- Execução de uma chamada de sistema para criação de um processo
- Outros

Processo pode ser terminado principalmente:

- Saída normal (voluntário)
- Saída por erro (voluntário)
- Erro fatal (involuntário)
- Finalizado por outro processo (involuntário)

Threads

Num sistema operacional tradicional, cada processo tem seu espaço de endereço e uma única *thread* de controle

Entretanto, podem existir situações onde é desejável ter múltiplas *threads* de controle num mesmo espaço de endereço rodando em *quasi*-paralelo

Neste caso, estas *threads* são quase que um processo separado, a não ser pelo fato de compartilharem um mesmo espaço de endereço

Threads

Threads podem ser definidas como miniprocessos

Razão de várias *Threads*

A principal razão de se ter múltiplas *threads* num processo é que múltiplas atividades estão acontecendo num "mesmo tempo"

Algumas dessas atividades podem ser **bloqueadas** de tempos em tempos

Então, pela decomposição de um processo em múltiplas *threads* (que rodam em *quasi*-paralelo), o modelo de programação se torna mais simples

Outro argumento é que *threads* são consideradas como **processos leves** (*light weight process*)

Finalmente, quando este processo de múltiplas *threads* for executado num multiprocessador, o paralelismo verdadeiro deverá ser mais diretamente/facilmente implementado

Exemplo de *threads* com POSIX [1]

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #define NUMBER_OF_THREADS  10
5
6  void *print_hello_world(void *tid)
7  {
8    /* This function prints the threads identifier an then exits */
9    printf("Hello World. Greetings from thread %d0", tid);
10   pthread_exit(NULL);
11 }
12
13 int main(int argc, char *argv[])
14 {
15   /* The main program creates 10 threads and then exits */
16   pthread_t threads[NUMBER_OF_THREADS];
17   int status, i;
18
19   for(i=0; i < NUMBER_OF_THREADS; i++)
20   {
21     printf("Main here. Creating thread %d0", i);
22     status = pthread_create(&threads[i], null, print_hello_world, (void *)i);
23
24     if (status != 0)
25     {
26       printf("Oops. pthread_create returned error code %d0", status);
27       exit(-1);
28     }
29   }
30   exit(NULL);
31 }
```

Espaços de endereços

Todo computador possui uma memória utilizada para guardar uma imagem executável

Num sistema operacional simples, apenas um programa encontra-se na memória por vez

Para rodar um segundo programa, o primeiro deve ser removido para o segundo entrar na memória

Sistemas operacionais mais sofisticados permitem que vários programas estejam na memória num mesmo tempo

Memória virtual

Num caso simples, a quantidade máxima de espaço de endereço que um processo possui é menor que a memória principal do sistema

Em vários casos o espaço de endereço é de 2^{32} ou 2^{64}

E se o processo tiver um espaço de endereço maior do que a memória principal do sistema?

Aplicar a técnica chamada **memória virtual** na qual o sistema operacional mantém parte do espaço de endereço na memória principal e parte numa memória auxiliar, e fica transportando as partes de/para conforme necessário

Em essência, o sistema operacional cria uma abstração do espaço de endereço

Arquivos

Uma outra característica de sistemas operacionais é possuir um sistema de arquivos

Assim, o SO deve abstrair as peculiaridades de acesso a FLASH e mecanismos de I/O, apresentado um modelo de abstração independente de máquina

I/O

Todos os computadores possuem dispositivos físicos para adquirir entradas (*input*) e produzir saídas (*outputs*)

O sistema operacional também deve ser capaz de abstrair as particularidades do hardware e prover uma API ou biblioteca (abstração) para acesso aos dispositivos de hardware de maneira menos "intrusiva" (mais alto nível)

Proteção

Relacionada, por exemplo, na proteção de acesso aos arquivos no sistema de arquivos

Exemplo: arquivos de Linux, contém 3 bits para o controle de acesso de usuários, grupos e outros

Shell

Alguns sistemas operacionais podem fornecer uma interface de linha de comando (trabalhando como um interpretador de comandos) para que o engenheiro de desenvolvimento possa verificar o estado do sistema operacional emitindo comandos (chamadas de sistema)

Instruction Set Architecture (ISA)

A ISA define o repertório de instruções de linguagem de máquina que o computador pode executar

Esta interface é uma fronteira entre software e hardware

Tanto aplicações (camada mais abstrata) quanto a camada do OS podem acessar a ISA diretamente

Para o software das aplicações um subconjunto da ISA encontra-se disponível (***user ISA***)

O OS possui acesso ao conjunto adicional da ISA (***system ISA***)

Application Binary Interface (ABI)

Define o padrão para portabilidade binária entre programas (não necessita de recompilação)

Ou seja, define as interfaces de chamadas de sistema para o sistema operacional, bem como os recursos de hardware e serviços disponíveis no sistema através da *user ISA*

Application Programming Interface (API)

A API fornece aos programas acesso aos recursos de hardware e serviços disponíveis num sistema

Isto é realizado através da *user ISA* e suplementado por chamadas de bibliotecas em linguagem de alto-nível

Geralmente, as chamadas de sistema são realizadas via bibliotecas

A utilização de API propicia que as aplicações de software sejam portadas mais facilmente, através de recompilação, para outros sistemas que suportam a mesma API

Informação ao leitor

Notas de aula baseadas nos seguintes textos:


A. S. Tanenbaum, *Modern Operating Systems*.
Pearson Prentice Hall, 3rd ed., 2008.


W. Stallings, *Computer Organization and Architecture: Design for Performance*.
Pearson Education, 9th ed., 2013.