Construindo Sistemas Embarcados com o OpenEPOS

- - Minicurso - -

Giovani Gracioli e Antônio Augusto Fröhlich
Laboratório de Integração Software/Hardware (LISHA)
Universidade Federal de Santa Catarina (UFSC)
88040-900 – Florianópolis – SC – Brasil
{giovani,guto}@lisha.ufsc.br

9 de novembro de 2011

Resumo

Este mini-curso aborda a construção de sistemas embarcados sob a perspectiva do OpenEPOS. O mini-curso é dividido em duas partes: uma teórica e outra prática. A parte teórica apresenta os principais conceitos relacionados com o OpenEPOS. A parte prática foca em exercícios onde são implementadas aplicações utilizando o sistema operacional. As atividades práticas cobrem o desenvolvimento de aplicações multi-thread, uso da infraestrutura de mediadores de hardware e da pilha de comunicação, incluindo ICMP/IP.

1 Introdução

O projeto e a implementação de sistemas embarcados atualmente apresentam enormes desafios. As aplicações embarcados estão se tornando cada vez mais complexas conforme o avanço da indústria de semi-condutores, que tornou possível o uso de recursos computacionais que antes só eram encontrados em sistemas de computação de propósito geral. Se por um lado as aplicações estão mais complexas, do outro ainda existe a pressão do mercado para a entrega do produto de forma rápida. Neste contexto, a utilização de um sistema operacional embarcado e baseado em componentes ajuda na aceleração do projeto e desenvolvimento de aplicações embarcadas, bem como na reutilização dos componentes, tanto de hardware como de software. Este mini-curso apresenta a construção de sistemas embarcados sob a perspectiva do sistema operacional EPOS.

2 Embedded Parallel Operating System

O EPOS (*Embedded Parallel Operating System*) [1, 3] é um arcabouço baseado em componentes para a geração de ambientes dedicados de suporte de tempo de execução. O arcabouço do EPOS permite que programadores desenvolvam aplicações independentes de plataforma e ferramentas de análise permitem que componentes sejam adaptados automaticamente para atender aos requisitos destas aplicações particulares. Por definição, uma instância do sistema agrega todo

suporte necessário para a sua aplicação dedicada, e nada mais.

Abstrações (ou componentes) no EPOS representam abstrações tradicionais de sistemas operacionais e implementam serviços como gerenciamento de memória e de processos, sincronização de processos, gerenciamento de tempo e comunicação. Todas as funções dependentes de arquitetura são abstraídas através de mediadores de hardware, que exportam para as abstrações as funcionalidades necessárias através de interfaces independentes de plataformas [4]. Mediadores de hardware são funcionalmente equivalentes aos *drivers* de dispositivos em SOs baseados em UNIX, mas não apresentam uma camada de abstração de hardware (HAL - *Hardware Abstraction Layer*) tradicional. Mediadores provêm uma interface entre os componentes do SO e o hardware através de técnicas de metaprogramação estática que "diluem" o código do mediador nos componentes em tempo de execução. Consequentemente, o código gerado não possui chamadas a métodos, nem camadas ou mensagens, atingindo uma maior portabilidade e reúso em comparação com as HALs tradicionais.

2.1 Gerenciamento de Processos

No EPOS, processos são gerenciados pelas abstrações `Thread` e `Task`. Cada thread armazena seu contexto em sua própria pilha. O mediador de hardware `CPU` define o conjunto de dados que precisa ser armazenado para um fluxo de execução e, deste modo, cada arquitetura define seu próprio contexto. Já o gerenciamento de memória é realizado pela família de mediadores `MMU` (*Memory Management Unit*). A Figura 1 apresenta um exemplo da criação de uma `Thread`.

```
#include <utility/ostream.h>
#include <thread.h>
#include <alarm.h>
__USING_SYS
const int iterations = 100;
int func_a(void);
Thread * a;
OStream cout;

int main() {
    cout << "Thread test\n";
    a = new Thread(&func_a);
    int status_a = a->join();
    cout << "Thread test done\n";
    delete a;
    return 0;
}

int func_a(void) {
    for(int i = iterations; i > 0; i--) {
        for(int i = 0; i < 79; i++)
            cout << "a";
            cout << "\n";
            Alarm::delay(500000);
    }
    return 'A';
}
```

Figura 1: Exemplo da utilização dos componentes Thread e Alarm.

2.2 Gerenciamento de Tempo

Tempo no EPOS é tratado pela família de abstrações `Timepiece`, composta pelas abstrações `Clock`, `Alarm` e `Chronometer`. A abstração `Clock` é responsável por armazenar o tempo corrente e está disponível apenas em sistemas que possuem dispositivos de relógios de tempo-real (`RTC`). A abstração `Chronometer` é utilizada para realizar medições de tempo com alta precisão e a abstração `Alarm` é utilizada para gerar eventos, acordar uma Tarefa/Thread ou chamar uma função após um tempo definido. A família de abstrações `Timepiece` é suportada pelos mediadores `Timer`, `TimeStamp Counter (TSC)` e `Real-Time Clock (RTC)` [2]. A Figura 1 demonstra o uso do método `delay` do componente `Alarm` dentro da execução da função `func_a`. A

Figura 2 demonstra o utilização do componente `Chronometer`, que é usado para realizar medidas de tempo.

```
#include <chronometer.h>
__USING_SYS;
int main() {
    OStream cout;
    cout << "Chronometer test\n";
    Chronometer c;
    c.start();
    do_something();
    c.stop();
    cout << "do_something() has taken " << c.read() <<
          "ms" << endl;
    return 0;
}
```

Figura 2: Exemplo da utilização do componente Chronometer.

2.3 Sincronização entre Processos

A família de abstrações `Synchronizer` provê mecanismos que garantem a consistência de dados em ambientes com processos concorrentes. O membro `Mutex` implementa um mecanismo de exclusão mútua que entrega duas operações atômicas: `lock` e `unlock`. O membro `Semaphore` implementa, como o próprio nome diz, um semáforo, que é uma variável inteira cujo valor apenas pode ser manipulado indiretamente através das operações atômicas `p` e `v`. O membro `Condition` realiza uma abstração de sistema inspirada no conceito de variável de condição, que permite a uma thread esperar que um predicado se torne válido. A Figura 3 apresenta uma implementação do tradicional programa jantar dos filósofos utilizando os componentes `Thread` e `Semaphore`.

2.4 Entrada e Saída

Controle de entrada e saída (I/O) de dispositivos periféricos é disponibilizado no EPOS pelo mediador de hardware correspondente. O mediador `Machine` armazena as regiões de I/O e trata o registro dinâmico de interrupções. O mediador `IC` (*Interrupt Controller*) trata a ativação ou desativação de interrupções individuais. Para lidar com as diferentes interrupções existentes em diferentes plataformas e contextos, EPOS atribui um nome e uma sintaxe independente de plataforma a interrupções pertinentes ao sistema (e.g., interrupção de timer, interrupção de conversão completa no ADC).

3 Instalação

O OpenEPOS está disponível no endereço `http://epos.lisha.ufsc.br`. Basta fazer o registro e clicar em `EPOS Software` para ter acesso ao arquivo com o sistema. Além disso, para a compilação do sistema, é necessário fazer o download dos compiladores. Neste mini-curso serão utilizados os compiladores para IA32 e ARM7, também disponíveis na mesma página. Os compiladores devem ser descompactados em `/urs/local`:

```
/usr/local/ia32/gcc
/usr/local/arm/gcc
```

Para a execução do OpenEPOS será utilizado o emulador QEMU para IA32 e ARM7. Para instalá-lo, em máquinas utilizando Ubuntu, basta digitar:

```
sudo apt-get install qemu
sudo apt-get install qemu-kvm-extras
```

4 Conclusão

Este documento brevemente discute os principais conceitos e componentes que fazem parte do sistema operacional EPOS. O foco do mini-curso está no desenvolvimento de sistemas embarcados sob a perspectiva do EPOS.

Com a utilização de um sistema operacional orientado à aplicação, é possível construir sistemas embarcados que possuem somente o suporte necessário para a sua aplicação dedicada, sem nenhum sobrecusto adicional. Além disso, o mini-curso é dedicado a demonstrar a utilização e a eficiência do uso de técnicas avanças de engenharia de software na construção de sistemas dedicados.

Referências

[1] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.

[2] Antônio Augusto Fröhlich, Giovani Gracioli, and João Felipe Santos. Periodic timers revisited: The real-time embedded system perspective. *Computers Electrical Engineering*, 37(3):365–375, 2011.

[3] H. Marcondes, A. S. Hoeller, L. F. Wanner, and A. A. Fröhlich. Operating systems portability: 8 bits and beyond. In *ETFA '06*, pages 124 –130, Sept. 2006.

[4] Fauze Valério Polpeta and Antônio Augusto Fröhlich. Hardware mediators: a portability artifact for component-based systems. In *ICEUC'04*, volume 3207, pages 45–53, Japan, 2004.


```
#include <utility/ostream.h>
#include <thread.h>
#include <semaphore.h>
#include <alarm.h>
#include <display.h>
__USING_SYS
const int iterations = 10;
Semaphore sem_display;
Thread * phil[5];
Semaphore * chopstick[5];
OStream cout;
int philosopher(int n, int l, int c)
{
    int first = (n < 4)? n : 0;
    int second = (n < 4)? n + 1 : 4;
    for(int i = iterations; i > 0; i--) {
        sem_display.p(); Display::position(l, c);
        cout << "thinking";
        sem_display.v(); Delay thinking(100000);
        chopstick[first]->p();
        chopstick[second]->p();
        sem_display.p(); Display::position(l, c);
        cout << " eating ";
        sem_display.v(); Delay eating(500000);
        chopstick[first]->v();
        chopstick[second]->v();
    }
    return iterations;
}
int main()
{
    sem_display.p(); Display::clear();
    for(int i = 0; i < 5; i++)
        chopstick[i] = new Semaphore;
    phil[0] = new Thread(&philosopher, 0, 5, 32);
    phil[1] = new Thread(&philosopher, 1, 10, 44);
    phil[2] = new Thread(&philosopher, 2, 16, 39);
    phil[3] = new Thread(&philosopher, 3, 16, 24);
    phil[4] = new Thread(&philosopher, 4, 10, 20);
    Display::position(7, 44); cout << '/';
    Display::position(13, 44); cout << '\\';
    Display::position(16, 35); cout << '|';
    Display::position(13, 27); cout << '/';
    Display::position(7, 27); cout << '\\';
    sem_display.v();
    for(int i = 0; i < 5; i++) {
        int ret = phil[i]->join();
        sem_display.p();
        Display::position(20 + i, 0);
        cout << "Philosopher " << i << " ate " << ret
             << " times \n";
        sem_display.v();
    }
    for(int i = 0; i < 5; i++) delete chopstick[i];
    for(int i = 0; i < 5; i++) delete phil[i];
    return 0;
}
```

Figura 3: Implementação do programa jantar dos filósofos com Semaphore.