ESCOLA POLITÉCNICA
UNIVERSIDADE DE SÃO PAULO

SIMULAÇÃO DE UM SISTEMA OPERACIONAL SIMPLES EM UM MOTOR DE EVENTOS DISCRETOS

PCS3446 – SISTEMAS OPERACIONAIS

Pedro Henrique Lage Furtado de Mendonça *N° USP: 8039011*

SIMULAÇÃO DE UM SISTEMA OPERACIONAL SIMPLES

PCS3446

SUMÁRIO

CÓDIGO FONTE DO MOTOR DE EVENTOS

O motor de eventos foi totalmente modernizado adequando-o aos padrões modernos de C++ e mantendo em mente as boas práticas de C++ elaboradas pelo Bjarne Stroustrup, criador e figura chave no universo C++. Dentre as adaptações feitas podem-se destacar as seguintes:

- Diversas rotinas retornavam códigos de erro (tratamento de erro estilo C) ou, pior ainda, apenas "engoliam" erros de forma invisível ao usuário. Essas rotinas foram atualizadas para operar totalmente a base de exceptions, recurso de C++ que torna o código mais seguro, simples, fácil de usar e com melhor mantebilidade.

- Reformulou-se o mecanismo de chamada de eventos. Antigamente eles eram chamados por ponteiros de funções e a mantebilidade e gerenciamento de eventos existentes e de novos eventos era demasiadamente difícil. Agora lançou-se mão de herança, polimorfismo e programação genérica, o usuário deve herdar seus eventos de uma classe puramente abstrata que apresenta ao motor uma interface simples e intuitiva.

    - Adicionalmente, anteriormente não era possível adicionar novos eventos na lista de eventos em tempo de execução, um evento não tinha acesso à lista de eventos e nem às configurações do motor. Agora, com as novas técnicas de programação implementadas, os eventos tem acesso total a tudo do motor e podem facilmente gerenciar a lista de eventos, configurações do motor e variáveis do usuário.

- Implementação do namespace "`DESE::`" para evitar conflito na resolução de nomes em tempo de compilação e permitir controle de versão com compatibilidade retroativa.

- Uso de templates para remover o uso de métodos desnecessários e repetitivos, aumentando a mantebilidade e gerando um código final menor.

- Por fim, falta atualizar partes do código para o novo padrão C++17, como por exemplo remover o uso da biblioteca terceirizada `Boost::Any` em favor da nova biblioteca padrão `STD::Any`, porém isso é uma atualização de **baixa prioridade** e não fora feito ainda.

A modernização do motor de eventos foi algo que custou caro em termos de tempo por dois motivos. O primeiro deles foi planejamento das novas interfaces, do ponto de vista desse programador, é melhor investir mais tempo em planejamento para se obter um sistema mais robusto, fácil de manter e expansível do que ficar remendando e quebrando a cabeça para mudanças futuras. O segundo motivo foi por alterar todo o funcionamento interno do motor de eventos (evidente quando se compara as classes que compõe esse novo motor com as classes antigas).

Como mudou-se totalmente todo o funcionamento interno, não foi possível fazer modificações pequenas e depurações ao longo do desenvolvimento, desenvolveu-se tudo em uma "tacada única", assim, sobraram problemas de compilação para serem resolvidos no final. Surpreendentemente, muitas linhas foram escritas e poucos erros foram encontrados, mostrando

um aumento do domínio sobre a ferramenta C++, porém os erros encontrados foram de complexidade maior, como de dependência circular e de resolução de nomes por parte do linker.

O novo código fonte totaliza cerca de 940 linhas de código comparado com mais de 1200 do motor antigo. O código fonte pode ser acessado na link abaixo:

INSERIR LINKS

O código fonte no momento da submissão e todas as imagens binárias juntamente com as listas de eventos em texto estão acessíveis descompactados e também zipados (DESEngine.zip) no github e google sites acima.

Os arquivos que devem constar no zip são:

Códigos Fonte (C++)	Binários e Listas de Eventos	
DESEngine.h/cpp Event.h EventCatalog.h/cpp EventList.h GlobalVariables.h/cpp SystemEvents.h/cpp Main.h/cpp SystemManager.h TextParser.h/cpp	FALTA ATUALIZA	

Arquivos em verde podem ser apagados pois eles serão gerados novamente ao executar o respectivo script (.txt)

Para a escrita, leitura e edição dos programas binários utilizados recomenda-se a utilização do programa HxD – editor de hexa gratuito - https://mh-nexus.de/en/hxd/

RESUMO DO MOTOR

O motor é composto de oito classes distintas, a saber: DESEngine, Event, EventCatalog, EventList, SystemEvents, SystemManager, GlobalVariables, e TextParser.

GLOBAL VARIABLES

Essa classe é a que possui maior grau de independência em termos de aplicação e pode ser utilizada completamente fora do contexto do motor de eventos. Ela é apenas um mapa desordenado de elementos do tipo `boost::any`, ou seja, dado um "Alias"/Nome `std::string`, ela armazena qualquer tipo de variável nesse nome.

Ela possui métodos para salvar novas variáveis, recuperar como diversos tipos através do `boost::any_cast` (`int, long, double, string, ...`), sendo essas funções de recuperar atualizadas para o uso de templates.

Essa era uma das classes que "engolia" erros de forma invisível, sendo atualizada para o uso de exceptions.

EVENT

É uma classe extremamente simples *(41 linhas)*, puramente abstrata (isto é, ela não pode ser instanciada, ela deve ser herdada e ter um de seus métodos overloaded). Ela apresenta uma interface simples para que o motor de eventos possa chama-lo e para que o evento possa gerenciar a lista de eventos em tempo de execução. Possui método para que o evento possa ser reinstanciável caso o evento necessite que seu construtor seja chamado toda vez que for ser executado, com os devidos cuidados para que tudo seja desconstruído antes de reinstanciar e evitar vazamento de memória.

EVENTCATALOG

Mais uma classe simples que registra os eventos do usuário com um nome (`std::string`) e chama os eventos dado um nome.

EVENTLIST

Classe responsável por listar os eventos, detectar fim da lista, achar e pular para labels (através do comando GOTO) e manter controle do índice do evento em execução.

EVENT

É uma classe extremamente simples *(41 linhas)*, puramente abstrata (isto é, ela não pode ser instanciada, ela deve ser herdada e ter um de seus métodos overloaded). Ela apresenta uma interface simples para que o motor de eventos possa chama-lo e para que o evento possa gerenciar a lista de eventos em tempo de execução.

SYSTEMEVENTS

É uma compilação de três classes herdadas de Event que implementam os eventos específicos do motor (HALT, GOTO e LABEL).

SYSTEMMANAGER

Responsável por armazenar configurações do motor, status do motor (em execução, pausado, finalizado pelo usuário, finalizado por fim da lista, finalizado por erro...), detecção de fim de execução, funções globais de uso constante (`Boost2String` e `PrintSysMsg`) e armazenar controladores que são passados para todos os eventos, fornecendo a eles acesso à lista de eventos, configurações, variáveis do usuário, etc.

TEXT PARSER

Essa classe não foi alterada, seu funcionamento está ótimo e bem documentado em relatório anterior. O TextParser.cpp tem o objetivo de abrir a lista de eventos codificado em texto (.txt por exemplo) e processar esse arquivo, removendo comentários, identificando variáveis do usuário, identificando strings, parâmetros e nome de eventos e com isso construir uma lista ligada de eventos e seus respectivos parâmetros no tipo de variável certo `(string, int, ...)`

DESENGINE

Com a modernização do motor de eventos, essa classe se tornou muito mais simples sendo muitas de suas funcionalidades antigas implementadas nas outras classes auxiliares. Agora é basicamente um laço que chama o próximo evento e verifica o fim da execução, ao atingir o fim, libera memória e publica um resumo da execução.

PRÓXIMOS PASSOS

Tem-se, do EP anterior de Sistemas de Programação, uma Máquina de Von Neumann (MVN) implementada para o motor de eventos antigo, atualizá-la para o novo motor de eventos é uma tarefa rápida e simples de ser feita, porém, antes de fazer isso, tenho desprendido tempo para compreender os novos requisitos do novo EP e planejar qual a melhor forma de implementá-las, sendo que esses requisitos irão alterar substancialmente o funcionamento da MVN antiga.

A MVN antiga era simulada em uma única etapa simples de "Fetch and Decode". Agora, com a implementação de interrupções, multiprogramação, memória virtual e entrada e saída realística esse ciclo deve ser melhor elaborado e muito bem pensado para evitar que futuras implementações levem a alterações substanciais de código já implementado.

A meu ver, o elemento mais crucial e difícil de implementar é a virtualização e gerenciamento de memória que implica em modificação do "hardware" simulado e do "software" simulado por esse "hardware" (sistema operacional simulado). Se a implementação de memória virtual for mal executada acredito que haverá muitas complicações posteriores.

Assim, a solução a qual estou tendendo a implementar é da seguinte forma:

Podem-se executar, simultaneamente, 16 programas, cada um com 64 kbytes de tamanho máximo, totalizando 1 mb de memória necessária, porém o sistema terá apenas 64 kbytes de memória, assim, através da virtualização de memória será possível executar todos os programas mesmo que eles não caibam simultaneamente na memória física disponível.

Cada programa poderá ter até 16 páginas de 256 bytes (16*256 = 64 kbytes). Os operandos das instruções são de 1 byte e então podem endereçar até 256, assim, eles endereçam dentro da **própria página**. Com o uso da instrução IND, que torna a próxima instrução com um operando de 2 bytes (16 bits), é possível endereçar o resto do programa.

A Memory Management Unit (MMU) irá obter o endereço físico com base no número do processo (registrador definido pelo SO), pela página e pelo endereço dado pelo programa. Dessa forma, o programador endereça todas as instruções com relação aos endereços dentro de seu próprio programa, sem se preocupar com o endereço físico efetivo.

Exemplo: programa X (4 bits, 0 a 15) na página Y (4 bits, 0 a 15):

```
JP 3A // pula para a posição FÍSICA da memória XY3A (2 bytes, 16 bits, o prefixo XY
é resolvido pelo MMU
IND ADD  ABCD // o MMU verifica se a posição AB está carregada. Se estiver, ele
encontra o Z (página). O X é dado pelo número do programa. Então soma-se ao
acumulador o conteúdo da memória XZCD.
```

A MMU ao receber o endereço ABCD acima, procura por AB na tabela de mapeamento que pertence àquele programa (byte X). Ao encontrar, ele tem o endereço da página (byte Z) e forma o endereço XZCD. Se não encontrar, ele carrega do disco a página AB00 na memória usando algoritmo First In First Out (FIFO) de substituição de páginas.

Dessa forma, a resolução de endereços é transparente ao programador e todos os endereços que ele usa são absolutos em relação ao seu próprio programa, apenas se preocupando ao chegar em uma nova página.

Com o funcionamento da memória virtual já planejado, acredito que o remanescente das funcionalidades não devem ser muito complicadas.