INF1413 Teste de Software
Período: 2018-1
Profs. Arndt von Staa

3o. Trabalho

Data de divulgação: 16 de maio (terça-feira)

Data de entrega: 28 de maio (segunda-feira)

1. Descrição do trabalho

O objetivo do terceiro trabalho é:

1. exercitar a criação de especificações baseadas em técnicas formais leves.
2. exercitar a leitura de código complexo redigido por terceiros;
3. exercitar a criação de máquina de estados a partir de código existente;
4. exercitar a redação de assertivas executáveis capazes de atuarem como oráculos ao testar um método;
5. exercitar a criação casos de teste úteis (valorados e com resultado esperado definido) para testar um método e os métodos privados por ele chamados segundo critérios de teste estruturais.

1.1. Contexto

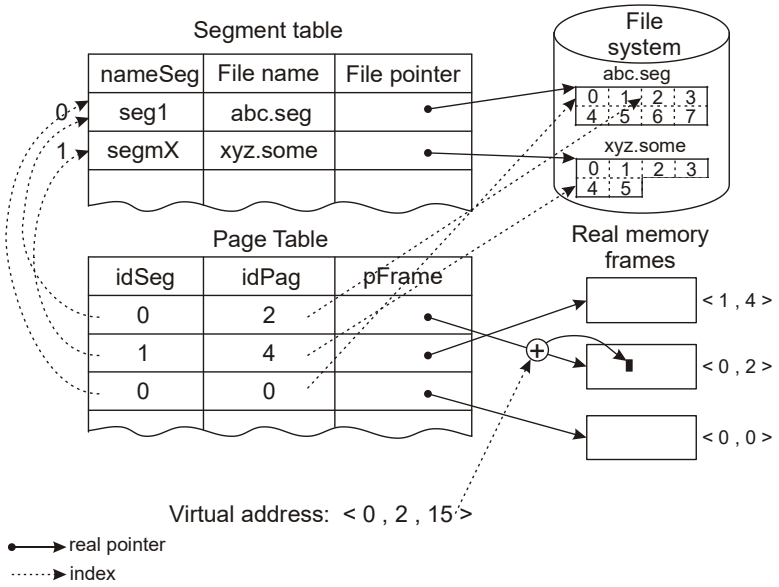

Figura 1. Exemplo de um estado instantâneo do controle de memória virtual segmentada.

Em sistemas de memória virtual o espaço de endereçamento é particionado em *páginas*. Todas as páginas possuem um mesmo tamanho. As páginas virtuais residem em *arquivos de paginação* e precisam ser trazidas para memória real para que o seu conteúdo possa ser manipulado. Em *memória real* as páginas estarão armazenadas em um dos vários *portadores*. Cada portador tem um *endereço real* conhecido. As páginas são persistidas em um arquivo de paginação e lá um *endereço virtual* referencia um espaço de dados, identificando a página contida no *arquivo de paginação* e o *deslocamento* (*offset*)

relativo à origem dessa página a partir de onde se encontra o espaço de dados. Através de uma *função de busca* a página é trazida para um dos *portadores de páginas* residente em *memória real*.

Em sistemas de *memória virtual não segmentada*, tais como Windows e Linux, existe um único grande arquivo de paginação compartilhado por todos os processos ativos em determinado momento. Este arquivo e os processos que com ele interagem estão sempre em uma mesma máquina. Nestes sistemas o endereço virtual de um dado contido em uma dada página é formado por **<idPagina, offset>**. Esse tipo de memória virtual obriga que os programas e os dados reais sejam copiados para esse arquivo de paginação, o que torna o endereço virtual variável entre diferentes usos de um mesmo programa. A consequência é que não se pode gravar ponteiros virtuais, além do custo da cópia, consequentemente estruturas de dados e estruturas formadas por objetos interconectados precisam ser persistidos (serialização) de modo que possam reconstruídos no espaço virtual (deserialização).

Em sistemas de *memória virtual segmentada* os arquivos de paginação podem ser arquivos quaisquer. Podem, inclusive, residir em diferentes máquinas. Processos ativos podem acessar páginas contidas em diversos segmentos. Nestes sistemas o endereço virtual de um espaço de dados é formado por **<idSegmento, idPagina, offset>** ou, mais sucinto, **<idSeg, idPag, offset>**, onde **idSeg** é o índice do descritor de segmento na tabela de segmentos. O descritor contém um campo que referencia o descritor de arquivo de segmento aberto. **idPag** é o índice da página no corpo do segmento e **offset** é o índice do caractere inicial do espaço de dados a ser acessado dentro da página.

Nesse tipo de memória virtual as páginas virtuais contidas no segmento são acessadas diretamente e não precisam ser copiadas para um arquivo de paginação central. Consequentemente a parte **<idPagina, offset>** do endereço virtual não varia de uma execução para outra. A consequência disso é que ponteiros virtuais relativos a objetos contidos no segmento podem ser gravados, uma vez que o endereço da página é sempre relativo à origem do segmento. Elimina-se assim a necessidade de serialização e deserialização.

Na figura 1, aparece uma coluna **nameSeg**, que vem a ser o nome simbólico do segmento local ao programa em execução. Os nomes simbólicos que um dado programa utiliza são sempre os mesmos, o que varia de uma instância de uso para outra é o **idSeg**, ou seja, o índice na tabela de *segmentos ativos* do sistema. Dessa forma o endereço virtual **<nameSeg, idPag, offset>** passa a ser constante para todas as instâncias de uso de um programa. Portanto, esse endereço pode ser gravado, permitindo criar programas que operam usando uma variedade de segmentos possivelmente distribuídos sobre diferentes máquinas. O nome do arquivo associado a um **nameSeg** pode ser fornecido durante a execução de um programa, permitindo que se possa, ao usar um programa, estabelecer e até variar o contexto a ser usado. Finalmente, como os nomes dos arquivos podem referenciar diferentes máquinas, é possível desenvolver programas que operem em ambientes de persistência distribuída. Para resolver o acesso, procura-se o **nameSeg** na tabela de segmentos. Caso seja encontrado, tem-se o **idSeg** desejado. Caso não seja encontrado, busca-se o nome do arquivo segmento associado ao **nameSeg** na tabela de segmentos do segmento origem e abre-se esse arquivo. Ao fazer isso, é criado o **idSeg** utilizado no processamento. O **idSeg** indexa um descritor na tabela de segmentos que contém o descritor de arquivo aberto (**FILE ***) correspondente ao segmento.

Cabe mencionar que ambos os tipos de memória virtual podem ser tratados por hardware, sendo a substituição de páginas tratado por software contido no kernel do sistema operacional. Quando realizada por hardware, desaparece a restrição dos espaços de dados não poderem ultrapassar as fronteiras das páginas. A memória virtual segmentada foi criada no sistema MULTICS nos anos 1960. Apesar de toda a flexibilidade, a memória virtual segmentada não obteve grande sucesso em virtude da complexidade do hardware necessário.

Simuladores de memória virtual segmentada são opções interessantes ao implementar sistemas de persistência orientados a objetos (*object stores*) e bases de dados *NoSql*. O lado negativo é o complexo problema de sincronização caso um mesmo segmento possa ser alterado simultaneamente por diferentes programas e a limitação de tamanho dos espaços de dados, que não podem ultrapassar as fronteiras das páginas. Outro lado negativo evidente é o custo computacional despendido para acessar e manipular as páginas, além da necessidade de obedecer estritamente a uma disciplina de manipulação da memória virtual.

A memória virtual segmentada tem várias vantagens. Uma das mais interessantes é que as páginas virtuais são persistentes por construção. Portanto, pode-se processar e alterar diretamente o conteúdo dessas páginas. Como ponteiros podem ser gravados torna-se desnecessário utilizar operações de serialização e deserialização, ou seja, não é necessário, reconstruir estruturas de dados e/ou objetos em memória real, ou no arquivo de paginação centralizado, tampouco é necessária a operação inversa. Outra vantagem é que o endereço virtual (i.e. ponteiro virtual), dado por **<idSeg**, **idPag**, **offset>**, relativo a um determinado segmento é independente de onde as páginas se encontram em memória real. Como os ponteiros virtuais são sempre relativos à origem do correspondente segmento e esta é sempre igual a 0, o endereço virtual do primeiro byte em um determinado segmento é sempre **<idSeg, 0, 0>**. Consequentemente, segmentos podem conter estruturas encadeadas, desde que se utilizem endereços virtuais como "ponteiros".

O presente trabalho gravita em torno de um simulador de memória virtual segmentada. Ao invés de utilizar hardware, no nosso caso o endereço virtual de uma página é transformado em endereço real da cópia da página via o método:

```
VMC_PageFrame VMC_VirtualMemoryRoot::GetPageFrame(
    int idSeg , int idPag , bool inMemory = false )
```

Como o **offset** não varia ao tornar real um endereço virtual, esse valor não é necessário para esse método. O método cria um objeto **VMC_PageFrame** que estará amarrado (*bound*) ao portador que contém a cópia em memória real da página virtual identificada pelo endereço virtual **<idSeg**, **idPag>**. Caso, ao buscar, a página virtual procurada já se encontre em algum portador, o objeto **VMC_PageFrame** criado será amarrado a esse portador. Poderão existir vários objetos **VMC_PageFrame** amarrados ao mesmo portador, porém cada página virtual terá no máximo uma cópia em memória real. Potencialmente, isso cria problemas de sincronização de acesso a portadores em sistemas multi-programados. Caso a página virtual não esteja copiada para algum portador, será procurado um portador livre. Se encontrado, a página virtual será lida para ele e o objeto **VMC_PageFrame** criado estará amarrado a ele. Caso não exista um portador livre, será procurado um portador a ser liberado, que receberá a cópia da página virtual solicitada. A leitura da página virtual somente ocorrerá caso **inMemory** seja **false**. Caso **inMemory** seja **true**, o objeto **VMC_PageFrame** será criado somente se a página virtual solicitada já se encontre em algum portador. Se a página não estiver em memória real o método retornará **NULL**. Isso permite otimizar o processamento caso não seja estritamente necessário acessar a página virtual. Por exemplo, verificadores estruturais de estruturas em memória virtual podem usar essa opção para que não ocorra mudança de página, evitando interferir no processamento produtivo. Como as últimas páginas acessadas se encontram em memória real, os itens recentemente modificados estarão muito provavelmente em memória real, permitindo que o verificador estrutural confirme a corretude da estrutura sem necessita percorrer toda ela.

Como mencionado, para poder trazer uma página para memória real, é necessário que exista um portador livre. Caso não exista, um dos portadores existentes é selecionado e seu conteúdo é esvaziado. Portanto, no decorrer do processamento, uma mesma página virtual poderá estar contida em diferentes portadores, consequentemente o endereço real a ser acessado ao manipular uma página virtual pode ser diferente em diferentes tentativas de acesso, ou seja, se não forem tomados cuidados, os ponteiros para os dados contidos nas páginas em memória real podem deixar de valer (ponteiros efêmeros). Para evitar que uma página referenciada por um objeto **VMC_PageFrame** seja removida e, assim, invalide os ponteiros para espações nela contidos, pode-se "piná-la", ou seja, marcá-la como *fixa*. Quando fixa, o método que procura um portador a ser usado não poderá libertar o portador que a contém. Sempre que um objeto **VMC_PageFrame** estiver amarrado a um portador, esse portador estará pinado. Como vários objetos podem estar amarrados a um mesmo portador, torna-se necessário contar o número de objetos amarrados a ele. O portador será liberado quando o último objeto amarrado a ele for excluído. Ao liberar o portador a página continua em memória real. Em um programa mal feito é possível ocorrer que todos os portadores estejam "pinados", tornando impossível trazer a página virtual. Neste caso é gerada a exceção de programa **VMC_NoFreeFrame**.

A Figura 1 ilustra a organização das tabelas utilizadas pelo simulador. Na tabela de segmentos vinculam-se o **idSeg**, o índice na tabela de segmentos (um **array**), com um descritor contendo a

identificação simbólica do segmento (**nameSeg**), o nome do arquivo (ou, se desejar, a sua URI) contendo o segmento, e o descritor de acesso ao arquivo aberto (**FILE ***). Como não se sabe, a priori, qual será a ordem de abertura dos arquivos de segmentação, os valores **idSeg** valem somente durante uma sessão de uso do programa. Portanto, diferentes sessões de uso podem levar a diferentes valores de **idSeg** para um mesmo arquivo. Como já mencionado, resolve-se esse problema com o nome simbólico **nameSeg**.

A tabela de páginas associa o endereço virtual **<idSeg, idPag>** da página com um o endereço real do portador que a contém. É através dessa tabela que se assegura que exista uma única cópia de uma página virtual em um portador contido em memória real. Todos os fragmentos de código que porventura acessem uma determinada página virtual acessarão o mesmo portador.

Ao desenvolver um simulador de memória virtual segmentada (ver esboço do diagrama de classes na Figura 2) é conveniente separar o processamento dos segmentos do processamento dos portadores (Page Frames) e do acesso a páginas (Page) residentes em memória real.

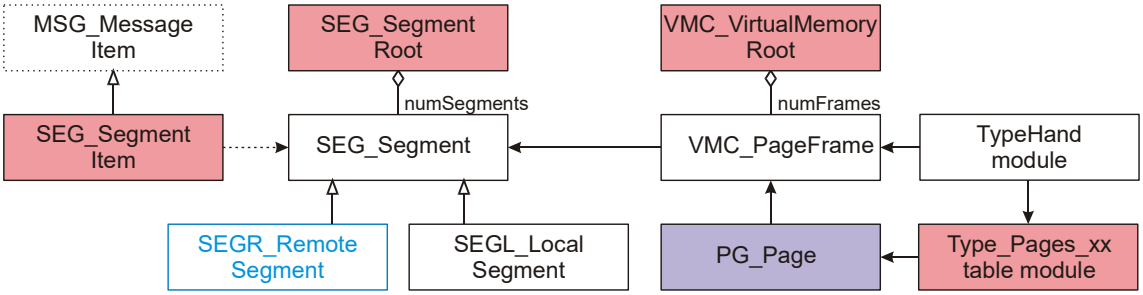

Figura 2. Estrutura de classes de um simulador de memória virtual segmentada.

O conjunto de classes **SEG...** realiza todas as operações de *criar*, *abrir* e *fechar* segmentos e, também as operações de *ler*, *gravar* e *adicionar* páginas inteiras. Ao gravar uma página, o local dela contido no arquivo é sobre-escrito pelo valor da página a ser gravada. O endereço inicial da página no segmento (arquivo) é dado pelo valor de **idPag * DIM_PAGINA** contido no endereço virtual.

Páginas virtuais contidas em um segmento não podem ser removidas. Portanto, as páginas que porventura não estejam mais em uso devem ser registradas em uma *lista de páginas livres*. Ao requisitar uma nova página, primeiro são utilizadas as páginas livres, somente se a lista de páginas livres estiver vazia é que se adiciona uma nova página ao final do segmento. Para aumentar o número de páginas utiliza-se o método **AddPage(pPagina)**. Esta função incrementa a dimensão do segmento e grava o conteúdo da página real **pPagina** ao final do segmento.

Na Figura 2 a classe **SEG_SegmentRoot** é um *singleton* (pode existir no máximo um objeto dessa classe) e contém a âncora de acesso ao conjunto de segmentos em uso em determinado momento. A estrutura de segmentos contém uma tabela de símbolos (**array**) que estabelece uma relação 1 para 1 entre o identificador de segmento (**idSeg**, um número inteiro), o identificador simbólico do segmento (**nameSeg**), e o nome do arquivo do segmento. O identificador de segmento é calculado durante o processamento de abertura do arquivo. Para tornar o processamento mais eficiente e dar mais flexibilidade aos programas que manipulam segmentos, estes são identificados pelo correspondente inteiro **idSeg**.

A classe **SEG_Segment** é uma classe abstrata que define os dados relativos a um segmento. A classe **SEGL_LocalSegment** especializa a classe **SEG_Segment** possibilitando o acesso a arquivos Windows. Nesse sistema operacional os endereços de arquivos são dados por **letra:\diretórios\arquivo.extensão**. O item **letra** identifica um dispositivo local, remoto, ou removível. Portanto, esses arquivos podem estar em diferentes máquinas possivelmente controladas por diferentes sistemas operacionais, desde que existam conversores que simulem o sistema de arquivos Windows nessas máquinas remotas. Por exemplo, o pacote *Samba* torna possível acessar arquivos em máquinas sob LINUX como se estivessem operando com um sistema de arquivos Windows. A classe **SEGR_RemoteSegment** especializa a classe **SEG_Segment** de modo que defina arquivos acessíveis por **URIs** (ainda não está implementada).

A classe **VMC_VirtualMemoryRoot** também é um *singleton*. Ela ancora o conjunto de portadores (*frames*). Cada portador pode conter a cópia de uma página de um determinado segmento. As páginas são endereçadas pelo endereço virtual **<idSeg, idPag>**.

O conjunto de portadores é controlado por duas estruturas de dados: uma lista LRU (*least recently used*) e uma memória associativa* (tabela de símbolos). A lista LRU encadeia os portadores em ordem crescente do tempo decorrido desde o último acesso realizado através do método **GetPageFrame(idSeg**, **idPag**, **inMemory)**. A memória associativa utilizada pelo simulador é uma tabela de randomização (*hash*) que resolve colisões através de listas de colisão. Ela é utilizada para associar o endereço virtual **<idSeg, idPag>** ao correspondente portador caso a página se encontre em algum portador.

Os portadores disponíveis são encadeados em uma lista LRU (*least recently used*). Como já foi dito, essa lista é ordenada segundo o momento de uso. As mais recentemente acessadas ficam na frente e as menos frequentemente acessadas ficam atrás na lista. Quando uma página é acessada usando **GetPageFrame** ela é movida para a frente da lista. O conteúdo das páginas em memória real pode ser acessado diretamente por ponteiros, porém estes acessos não reordenam a lista LRU. Quando for necessário remover uma página para dar lugar a outra a ser trazida para memória real, as candidatas estarão entre as mais antigas segundo a ordenação LRU. Para evitar que páginas em uso possam acidentalmente ser removidas e, dessa forma, invalidar os ponteiros reais que apontam para o seu conteúdo, portadores devem ser "pinados". Cabe à aplicação controlar a colocação e a remoção de pinos (**PinFrame**, **UnpinFrame**).

Quando uma página for alterada o correspondente portador é marcado **sujo** usando o método **SetFrameDirty()**. De tempos em tempos as páginas contidas em portadores sujos são gravadas. Ao remover a página contida em um portador esta será gravada, caso o portador esteja marcado sujo. Cabe à aplicação informar sempre que uma dada pagina for alterada.

Para assegurar a consistência do sistema, as operações sobre páginas de segmentos devem ser realizadas somente através da classe **VMC_VirtualMemoryRoot**. As operações que envolvam páginas virtuais específicas em memória real devem ser realizadas através da classe **VMC_PageFrame**.

2. Descrição do terceiro trabalho

Observação: os arquivos não contêm todo o código. Sempre que encontrarem um método que não faça parte do conjunto definido nos arquivos fornecidos, considerem-nas como sendo pseudo instruções.

1. No componente de controle de acesso os direitos de uso são tratados como se fossem vetores de bytes. Isso tem diversos problemas, um deles é a facilidade com que hackers podem vir a adulterar direitos de uso. Uma alternativa é passar um objeto da classe **DireitoDeUso** que encapsula e encripta os direitos que determinado usuário tem. Evidentemente, esse objeto é criado ao fazer um **Login** bem sucedido. O objeto pode conter ainda outros dados, tais como o apelido e o nome do usuário e o e-mail dele. Isso permite personalizar a sessão de uso do sistema **Sis**.

   Entre os métodos dessa classe figura o método **bool TemDireito(char idDireito)** que retorna **true** se e somente se **idDireito** é um dos direitos que o usuário autorizado tem. A lista dos possíveis **idDireito** é estabelecida ao projetar o sistema **Sis** que utilizará o componente. O construtor da classe **DireitoDeUso** recebe como parâmetro um vetor de caracteres que corresponde ao conjunto dos direitos que o usuário autorizado tem e que está contido no cadastro de usuários.

   Especifique as assertivas de entrada e saída de cada um dos métodos públicos que você imagina serem necessários na classe **DireitoDeUso**.

* Uma memória associativa endereça o seu conteúdo por um valor, ou melhor, uma chave. Uma tabela de símbolos é uma das formas de implementar uma memória associativa.

2. Com base na descrição do método **`GetPageFrame(...)`** construa e verifique a tabela de decisão a ser usada para criar a suíte de teste. O que é mais fácil entender, a tabela ou o texto?
3. Com base no código fornecido, construa uma máquina de estados similar ao grafo de chamadas correspondente a todo processamento com início no método **`GetPageFrame( ... )`**, levando em conta os métodos chamados contidos no módulo **`VRTMEM`**. Chamadas a métodos que não estão nesse módulo devem ser considerados como pseudo-instruções. Nessa máquina de estados defina as assertivas de entrada em cada um dos estados.
4. Escreva o módulo de instrumentação com as assertivas executáveis do método **`ReplacePage`**.
5. Desenhe os grafos de estrutura do método **`GetPageFrame(...)`** e do método **`VMC_VirtualMemoryRoot::ReplacePage( )`**. Com base nesses grafos produza expressões sintáticas que descrevam os caminhos existentes nos dois grafos. Escreva três casos de teste abstratos criados a partir da combinação dos caminhos existentes nos dois métodos. Transforme esses casos de teste abstratos em caso de teste semânticos. Finalmente, transforme os casos de teste semânticos em casos de teste úteis. Para poder criar os casos de teste úteis, esboce o conteúdo de um segmento a ser usado para fins de teste. O resultado de cada passo deve estar no material entregue. **OBS. para viabilizar a correção, devolva o arquivo .cpp com as linhas numeradas na forma como você usou.**

3. Entrega do Trabalho

O trabalho deve ser realizado em grupos de 2 ou 3 alunos. Neste trabalho isso é mais eficiente trabalhar em grupo do que distribuir tarefas a serem feitas isoladamente por cada um.

Além dos documentos relacionados acima, cada membro da equipe deve entregar um relatório descrevendo o tempo despendido no trabalho. O relatório consta de uma tabela Excel de registro de trabalho organizada como a seguir:

Data Horas Trabalhadas Tipo Tarefa Descrição da Tarefa Realizada

Na descrição da tarefa redija uma explicação breve sobre o que foi feito. Esta descrição deve estar de acordo com o Tipo Tarefa. Cada Tipo Tarefa identifica a natureza do esforço realizado (ex. estudar, projetar máquina de estados,...) e deverá ser discriminada explicitamente, mesmo que durante uma mesma sessão de trabalho tenham sido realizadas diversas tarefas da mesma natureza.

4. Critérios de correção básicos

- A correção considera o trabalho como um todo, ou seja, problemas possivelmente encontrados levam a perda de pontos, sem considerar qualquer limite por item descrito acima.
- As avaliações dos itens resultam em { OK, +/- (-0,5), fraco (-1), ruim (-2), não fez ou não relacionado com o enunciado (-3) }. Note que esse critério pode resultar em nota negativa, tratada como zero.
- Corretude e completeza das assertivas.
- Coerência dos diversos grafos.
- Coerência dos casos de teste com o artefato a ser testado.
- Qualidade do texto – o texto deve estar correto do ponto de vista ortográfico e sintático. O texto deve ser sucinto, fácil de ler e de compreender.