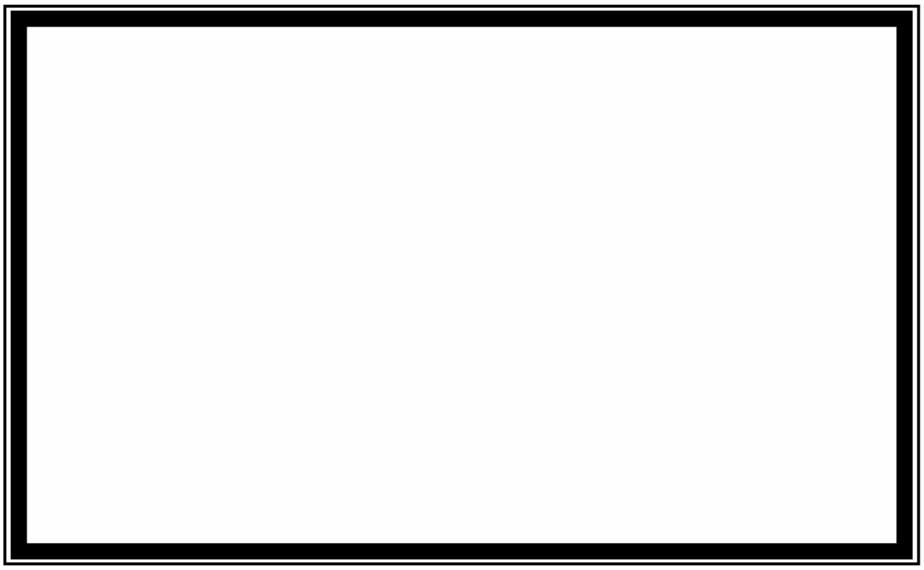
DISSERTAÇÃO DE MESTRADO

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM

CIÊNCIA DA COMPUTAÇÃO



CAIXA POSTAL 676
FONE/FAX (016) 260-8233
13565-905 - SÃO CARLOS - SÃO PAULO
BRASIL

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM

CIÊNCIA DA COMPUTAÇÃO

"CSK: Uma Abordagem para Estruturação de um Kernel de Tempo-Real em Componentes"

ORIENTADOR: Prof. Dr. Célio Estevan Moron
ALUNO: Lúcio José Herculano Correia

São Carlos
Abril/2004

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

C823cs

Correia, Lúcio José Herculano.
CSK: uma abordagem para estruturação de um kernel de tempo real em componentes / Lúcio José Herculano Correia. -- São Carlos : UFSCar, 2005.
81 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2004.

1. Sistemas de programação. 2. Programação em tempo real. 3. Sistemas operacionais (computadores). 4. Componentes de software. I. Título.

CDD: 005.42 (20ª)

Dedico esta dissertação a Deus, pelos pais que Ele me deu e pelas oportunidades a mim proporcionadas.

AGRADECIMENTOS

Ao Prof. Dr. Célio Estevan Morón, por orientar o desenvolvimento deste trabalho e pelo incentivo atribuído em todas as etapas de seu desenvolvimento.

Ao Prof. Dr. Antônio Francisco do Prado, pelo apoio essencial à realização deste trabalho.

Aos meus pais, os principais responsáveis por este trabalho.

Aos amigos do laboratório, pela agradável convivência e pela infinita quantidade de conhecimento transmitida.

À Universidade Federal de São Carlos, instituição que possibilitou a realização deste mestrado.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pela concessão de bolsa de estudo.

LISTA DE FIGURAS

Figura 1 Diagrama de estados de um processo 11

Figura 2 Necessidade de exclusão mútua 13

Figura 3 Definição clássica das operações de um semáforo 20

Figura 4 Solução do problema produtor-consumidor através de semáforos 22

Figura 5 Utilização da troca de mensagens para sincronização 23

Figura 6 Solução do problema produtor-consumidor por troca de mensagens 24

Figura 7 Esquema de funcionamento de um SOTR 25

Figura 8 Inversão de prioridades 27

Figura 9 Herança de prioridades 28

Figura 10 Representação de atores e casos de uso 35

Figura 11 Representação UML para uma classe 36

Figura 12 Relacionamentos em um modelo de classes UML 36

Figura 13 Representação de componentes através da liguagem UML 37

Figura 14 Fases da abordagem CSK 40

Figura 15 Arquitetura das classes dos componentes 45

Figura 16 Arquitetura típica de um componente 46

Figura 17 Arquitetura Virtuoso 51

Figura 18 Estados de uma tarefa Virtuoso 54

Figura 19 Esquema de funcionamento de uma fila FIFO no Virtuoso 56

Figura 20 Operações de um semáforo Virtuoso 58

Figura 21 Grafo cíclico inicial 62

Figura 22 Grafo acíclico resultante 64

Figura 23 Modelo de classes concretas do Virtuoso – Parte 1 66

Figura 24 Modelo de classes concretas do Virtuoso – Parte 2 67

Figura 25 (a) Componente `Semaphore` e (b) Projeto interno de sua classe concreta 68

Figura 26 Parte do código do componente `Semaphore` 69

Figura 27 Modelo de componentes do Virtuoso 70

Figura 28 Paleta e instanciação dos componentes implementados 71

LISTA DE TABELAS

Tabela 1 Primitivas de criação e controle de execução de tarefas 53

Tabela 2 Primitivas de controle de grupos de tarefas..... 55

Tabela 3 Primitivas de comunicação do Virtuoso..... 56

Tabela 4 Primitivas de sincronização do Virtuoso..... 58

Tabela 5 Primitivas de gerência de tempo do Virtuoso..... 59

Tabela 6 Primitivas de gerência de memória do Virtuoso 60

Tabela 7 Primitivas especiais do Virtuoso 60

Tabela 8 Serviços e primitivas identificados no Virtuoso..... 61

Tabela 9 Serviços e respectivas variáveis associadas..... 61

Tabela 10 Tempo médio de chamada a uma primitiva..... 72

SUMÁRIO

1. INTRODUÇÃO 1

2. CONCEITOS BÁSICOS 3

2.1. Categorias de sistemas computacionais 3

2.2. Sistemas de tempo-real 4

2.2.1. Definição 4

2.2.2. Necessidade de um sistema operacional de tempo-real 5

3. SISTEMAS OPERACIONAIS DE TEMPO-REAL 7

3.1. Definição 7

3.2. Conceitos básicos 8

3.2.1. Concorrência 8

3.2.2. Interrupções e chamadas de sistema 10

3.2.3. Processos 10

3.2.4. Comunicação e sincronização 12

3.2.5. Problemas clássicos de comunicação e sincronização 12

3.2.5.1. Exclusão mútua 12

3.2.5.1.1. Acesso único à memória 14

3.2.5.1.2. Inibição de interrupções 15

3.2.5.1.3. Instruções atômicas 16

3.2.5.1.4. Mecanismos para sistemas multiprocessados 17

3.2.5.2. O problema “produtor-consumidor” 17

3.2.5.3. O problema “leitores e escritores” 18

3.2.5.4. Compartilhamento de recursos 19

3.2.6. Semáforo 20

3.2.7. Troca de mensagens 22
3.3. Funcionamento de um kernel de tempo-real 25
3.3.1. Chamadas de sistema 25
3.3.2. Multiprogramação 26
3.3.3. Temporizadores e manipulação de interrupções 29
3.3.4. Gerência de memória 29
4. DESENVOLVIMENTO DE SOFTWARE BASEADO EM COMPONENTES 31
4.1. Introdução 31
4.2. Definição 32
4.3. Padrões e técnicas de desenvolvimento de componentes 33
4.3.1. UML 33
4.3.1.1. Características da UML 34
4.3.1.2. Modelo de casos de uso 35
4.3.1.3. Modelo de classes 35
4.3.1.4. Modelo de componentes 37
4.3.2. Catalysis 37
5. ABORDAGEM CSK 40
5.1. Construção dos componentes 41
5.1.1. Identificação dos serviços oferecidos pelo *kernel* 42
5.1.2. Associação das variáveis do *kernel* 42
5.1.3. Identificação das relações de “quem chama quem” 43
5.1.4. Projeto interno dos componentes 44
5.1.5. Implementação dos componentes 45
5.2. Reconstrução do *kernel* com reúso de componentes 46

5.2.1. Análise de requisitos 47
5.2.2. Projeto orientado a componentes 47
5.2.3. Implementação e testes 48
5.3. Considerações sobre a abordagem CSK 49
6. ESTUDO DE CASO 50
6.1. Construção dos componentes 52
6.1.1. Identificação dos serviços e primitivas do Virtuoso 52
6.1.1.1. Gerência de tarefas 53
6.1.1.2. Comunicação 55
6.1.1.3. Sincronização 57
6.1.1.4. Gerência de tempo 59
6.1.1.5. Gerência de memória 59
6.1.1.6. Primitivas especiais 60
6.1.1.7. Conclusão 60
6.1.2. Organização das variáveis do Virtuoso 61
6.1.3. Identificação das relações de “quem chama quem” 62
6.1.4. Projeto dos componentes 64
6.1.5. Implementação dos componentes 66
6.2. Reconstrução do *kernel* 69
6.2.1. Análise dos requisitos do Virtuoso 69
6.2.2. Projeto orientado a componentes do Virtuoso 70
6.2.3. Implementação e testes 71
6.3. Considerações de desempenho 72
7. TRABALHOS CORRELATOS 74

8. CONCLUSÃO 76

8.1. Contribuições 77

8.2. Trabalhos futuros 77

8.3. Considerações finais 78

REFERÊNCIAS BIBLIOGRÁFICAS 79

RESUMO

Sistemas operacionais em geral oferecem um conjunto de serviços através de primitivas, as quais são utilizadas sob demanda pelas aplicações. Similarmente, componentes de software implementam e disponibilizam serviços através de interfaces bem definidas, sendo reutilizados em diferentes projetos de software de um domínio de problema. Baseado nesta similaridade, é pesquisado o reuso de componentes de software na área de tempo-real, nos projetos de sistemas operacionais. É apresentada uma abordagem para a estruturação de *kernels* de tempo-real utilizando componentes de software, denominada CSK (*Component Structured Kernel*). Esta abordagem é dividida em duas grandes fases. Na primeira fase, constroem-se os componentes a partir do código legado e da documentação de um *kernel* já existente. Na segunda fase é feita a reestruturação do *kernel*, através do reúso de componentes. Um estudo de caso aplica a abordagem CSK ao *kernel* Virtuoso.

ABSTRACT

Generally, operating systems offer a set of services through primitives that are used on demand by applications. Similarly, software components implement and provide services through well-defined interfaces, which are reused in different software projects of a problem domain. Based upon this similarity, it is researched the reuse of software components in real time operating systems projects. It is presented CSK (Computer Structured Kernel) approach for structuring a real time kernel with software components. The approach is divided in two phases. In the first one software components are built from the legacy code and documentation of a pre-existing kernel. In the second one, that kernel is rebuilt by reusing the produced components. A case study applies CSK to real time kernel Virtuoso.

1. INTRODUÇÃO

O sistema operacional atua como uma camada de software intermediária entre a camada das aplicações e a camada de hardware de um sistema, disponibilizando uma interface de comunicação de alto nível aos engenheiros de aplicações, isentando-lhes da necessidade de conhecer as peculiaridades de cada dispositivo e tornando as aplicações menos dependentes do hardware no qual são executadas.

Esta interface de comunicação é oferecida na forma de chamadas de sistema, as quais implementam serviços úteis, como gerência de memória e controle de execução e comunicação entre os processos.

Do ponto de vista do engenheiro de aplicações, o sistema operacional pode ser visto como um conjunto de serviços. Tais serviços estão em constante atualização, e podem ser implementados de forma diferente de um sistema operacional para outro.

Trabalhos de pesquisa focados em algum destes serviços requerem o suporte de uma grande quantidade de código adicional sem relação com o objeto de estudo, o que torna as etapas de implementação e testes grandes consumidoras de tempo e recursos.

Analogamente, o próprio processo de desenvolvimento de software está em constante atualização, fato que leva ao desenvolvimento de novos paradigmas de programação, geralmente visando uma melhor estruturação e maior reutilização do código já desenvolvido e uma conseqüente economia de tempo e recursos. Uma destas abordagens é o desenvolvimento baseado em componentes de software.

Componentes de software são blocos de código previamente testados que facilitam o desenvolvimento através do reúso, por meio dos mecanismos de herança e instanciação de objetos. A principal diferença para o paradigma da orientação a objetos é o foco no

desenvolvimento. Enquanto a orientação a objetos busca uma alternativa mais realista ao desenvolvimento de software em relação ao desenvolvimento estruturado, a tecnologia de componentes visa o desenvolvimento de código que possa ser aproveitado em qualquer projeto de software, sem a necessidade de qualquer alteração.

Através de componentes, o conhecimento embutido no código legado de sistemas operacionais já consolidados poderia ser reutilizado na sua reestruturação ou na construção de novos sistemas operacionais, poupando esforços, tempo e custos de desenvolvimento.

Motivado por estas idéias, é apresentada a abordagem CSK para a estruturação do *kernel* de um sistema operacional utilizando componentes de software, desenvolvidos a partir de seu código legado e sua documentação.

Os principais objetivos da abordagem em questão são proporcionar um maior nível de encapsulamento aos serviços de um *kernel* e explorar o reúso de componentes obtidos de projetos de sistemas operacionais já consolidados na construção de novos sistemas operacionais.

Os Capítulos 2, 3 e 4 introduzem os conceitos básicos utilizados no decorrer do trabalho. O Capítulo 5 apresenta a abordagem CSK. O Capítulo 6 apresenta um estudo de caso, no qual a abordagem CSK é aplicada ao *kernel* Virtuoso [26]. Uma visão geral dos trabalhos relacionados à esfera do problema, comparada ao presente trabalho, é apresentada no Capítulo 7.

2. CONCEITOS BÁSICOS

Este capítulo apresenta os conceitos básicos relacionados à proposta de trabalho apresentada. Inicia-se com a localização do domínio do problema – os sistemas de tempo-real – na classificação de sistemas computacionais. Em seguida, definem-se os sistemas de tempo-real e discursa-se sobre a necessidade de um sistema operacional de tempo-real.

2.1. Categorias de sistemas computacionais

Sistemas computacionais podem ser classificados de diversas formas, dependendo do critério utilizado na classificação. Se o critério utilizado for o tempo de resposta do sistema, são identificadas três categorias principais: sistemas de processamento em lote, sistemas *on-line* e sistemas de tempo-real.

Um sistema de processamento em lote geralmente é utilizado em casos de escassez de recursos computacionais ou financeiros. As transações submetidas por seus usuários são pré-processadas e armazenadas localmente, sendo processadas quando houver disponibilidade de processador, tempo que pode variar de segundos a, até mesmo, horas. O resultado do processamento é armazenado e posteriormente recuperado pelo usuário.

Um sistema *on-line* se caracteriza por possuir um ou mais usuários conectados, esperando por uma resposta. Neste tipo de sistema, pequenos atrasos no tempo de resposta são toleráveis e não trazem danos significativos. A maioria dos sistemas atuais se enquadra nesta categoria. Um exemplo é o sistema de auto-atendimento bancário.

Já os sistemas de tempo-real são sistemas cujas restrições temporais são importantes a ponto de fazerem parte de sua especificação. Ou seja, o tempo de resposta é crucial para o funcionamento deste tipo de sistema.

Singhal [24] classifica os sistemas de tempo-real em dois tipos: crítico e não-crítico. Num sistema crítico, o não cumprimento dos prazos pré-estabelecidos pode até mesmo colocar em risco vidas humanas. Exemplos são sistemas de controle de tráfego aéreo, monitoramento de usinas nucleares, controle de automóveis, entre outros.

Já num sistema não crítico os requisitos temporais são importantes, mas não resulta em danos significativos o fato de um subconjunto deles não serem cumpridos. Um exemplo de sistema não crítico é um sistema multimídia.

Os sistemas de tempo-real consistem do assunto principal deste trabalho, sendo denotados também por STR.

2.2. Sistemas de tempo-real

As próximas seções definem um sistema de tempo-real e apresentam suas principais características. Em seguida é discutida a utilização de um sistema operacional de tempo-real na implementação dos STR.

2.2.1. Definição

Segundo Axford [2], "um sistema de tempo-real é um sistema no qual os requisitos temporais compreendem uma parte essencial de sua especificação". Um exemplo de requisito temporal pode ser o tempo de resposta do STR a um determinado evento.

Um sistema de tempo-real é visto como um conjunto de tarefas onde cada tarefa é associada a um conjunto de requisitos, não necessariamente temporais. Por tarefa entende-se uma unidade lógica do sistema, responsável por cumprir uma função específica [7].

As tarefas de um STR podem ser periódicas, aperiódicas ou esporádicas. Tarefas periódicas são tarefas que ocorrem em intervalos de tempo definidos. Tarefas aperiódicas

ocorrem em períodos de tempo variáveis. Tarefas esporádicas são tarefas que podem ocorrer em qualquer tempo, ou até mesmo não ocorrer.

As principais propriedades de uma tarefa são:

a) Tempo de início: tempo absoluto no qual a execução deve começar;
b) Deadline: tempo absoluto no qual a execução deve terminar;
c) Duração: tempo durante o qual a tarefa efetivamente executa;
d) Período: intervalo entre execuções sucessivas da tarefa;
e) Prioridade: valor que indica a importância da tarefa no sistema.

Alguns exemplos de sistemas de tempo-real são:

a) Sistemas operacionais, geralmente utilizados para gerenciar diversos dispositivos concomitantemente;
b) Sistemas de processamento de transações, os quais incluem os caixas automáticos dos bancos, o sistema de reserva de passagens aéreas e o sistema de leitura de código de barras nos caixas do supermercado;
c) Sistemas de controle de processos industriais, como os de refinarias de petróleo, de linhas de produção de automóveis e de usinas nucleares;
d) Sistemas embarcados, isto é, equipamentos onde o computador não realiza o papel principal, como fotocopiadoras, sistemas de aquecimento ou resfriamento, armas militares, entre outros.

2.2.2. Necessidade de um sistema operacional de tempo-real

Um STR pode ser implementado ou não sobre um sistema operacional de tempo-real. Esta seção discute a importância da utilização de um sistema operacional de tempo-real, ou SOTR, em sistemas de tempo-real.

Alguns sistemas de baixo custo são implementados através da metodologia *background*/*foreground*, ou executiva cíclica. Esta metodologia consiste da implementação de um laço principal infinito (*background*) que aciona rotinas de tratamento para eventos assíncronos ocorridos (*foreground*), tais como interrupções.

Além da vantagem de uma latência mínima na resposta a eventos, a metodologia *background*/*foreground* é de simples implementação, por possuir uma única área de armazenamento para sub rotinas, variáveis e interrupções.

Suas desvantagens são a falta de um mecanismo de prioridades e de serviços de gerenciamento de recursos, os quais, se necessários, devem ser implementados manualmente. Além disso, os programadores necessitam conhecer profundamente o hardware e o software utilizados, implementando uma versão do sistema para cada caso.

Por isso, em sistemas maiores e mais dotados de recursos o ideal é sempre utilizar um sistema operacional para o gerenciamento dos recursos do sistema. O próximo capítulo é dedicado ao estudo dos sistemas operacionais de tempo-real, trazendo sua definição, conceitos básicos, problemas encontrados e respectivas soluções.

3. SISTEMAS OPERACIONAIS DE TEMPO-REAL

Os primeiros SOTR surgiram nos anos 50. Desde então, eles evoluíram de simples escalonadores cíclicos a complexos ambientes operacionais. Entre os principais fatores que contribuíram para esta evolução está a padronização da POSIX.

POSIX [20], sigla para *Portable Operating System Interface for Computer Environments*, corresponde a uma especificação do IEEE (*Instituct of Electrical and Electronics Engineers* [13]) para facilitar o desenvolvimento de aplicações de tempo-real através do estabelecimento de padrões de implementação para os serviços básicos de um sistema operacional.

As próximas seções definem os principais conceitos relacionados aos sistemas operacionais de tempo-real, suas principais características e os problemas de concorrência enfrentados em sua implementação e suas respectivas soluções.

3.1. Definição

O sistema operacional é a primeira camada de software em um sistema computacional e sua função é a de gerenciar os recursos de hardware do sistema – placa-mãe, dispositivos de entrada e saída (E/S), disco rígido, memória, entre outros.

Desta forma, o sistema operacional atua como uma camada intermediária entre a camada das aplicações e a camada de hardware, estabelecendo uma interface de comunicação de alto nível aos programadores de aplicações, reduzindo a necessidade de conhecer as peculiaridades de cada dispositivo de hardware do sistema e tornando as aplicações menos dependentes do hardware no qual são executadas.

Um sistema operacional de tempo-real pode ser visto como um caso especial de sistema operacional que, além de desempenhar suas funções básicas, deve fazê- lo em um intervalo de tempo previsto.

3.2. Conceitos básicos

Para gerenciar todos os recursos de hardware de um sistema, o SOTR deve suportar a divisão das aplicações em processos (unidades seqüenciais com implementações lógicas separadas), permitir sua execução concorrente e lidar com a utilização periódica ou aleatória dos recursos do sistema [27].

O termo processo possui o mesmo significado que o termo tarefa, empregado anteriormente. O primeiro está relacionado à área de sistemas operacionais, enquanto o segundo se relaciona mais às aplicações, no caso, a sistemas de tempo-real.

3.2.1. Concorrência

Em um computador uniprocessado não é possível executar mais de um processo ao mesmo tempo. Neste tipo de sistema, pode ser obtido um pseudoparalelismo, isto é, um paralelismo aparente na execução dos processos.

O pseudoparalelismo é obtido através da execução concorrente de vários processos, chamada multiprogramação. Num sistema multiprogramado o processador é chaveado rapidamente entre os processos em execução, permitindo que cada um, em sua vez, seja executado por determinado intervalo de tempo.

Este intervalo de tempo, o *quantum*, é pequeno o suficiente de forma que, em uma fatia de tempo considerável, o usuário tenha a impressão de paralelismo de execução.

Um *quantum* demasiadamente pequeno pode comprometer o desempenho do sistema, pois a organização necessária a cada chaveamento de processo (troca de contexto) requer a utilização de um certo tempo de processamento.

O contexto, neste caso, corresponde ao estado momentâneo de execução do processo, e deve ser armazenado para que sua execução seja retomada posteriormente. Ele compreende:

a) O conteúdo de todos os registradores conhecidos pelo programador, incluindo o contador de programa e outros flags e registradores de estado;
b) O conteúdo de todas as áreas de memória reservadas para uso do processo;
c) O conteúdo de todos os arquivos abertos pelo processo.

Portanto, uma troca de contexto consiste em salvar o contexto do processo em execução e, em seguida, substituí-lo pelo contexto do processo que irá executar.

O engenheiro de aplicações não precisa se preocupar com a multiprogramação, pois todo o seu controle fica a cargo do sistema operacional. Aplicações paralelas podem ser executadas em sistemas multiprocessados, isto é, sistemas que possuem vários processadores. A implementação, neste caso, é chamada de multiprocessamento.

Outra modalidade de implementação de sistemas multiprocessados é o processamento distribuído, realizado sobre um conjunto de computadores interconectados por uma rede, onde cada um deles corresponde a uma unidade lógica de processamento do mesmo sistema.

3.2.2. Interrupções e chamadas de sistema

Uma interrupção é um sinal informando a um processo a ocorrência de determinado evento externo. Uma chamada de sistema funciona como uma interrupção por software, ou seja, um sinal de interrupção gerado a partir do código de um processo.

O tratamento de interrupções é sempre prioritário em relação à execução dos processos. Portanto, quando um sinal de interrupção é gerado, tal interrupção deve ser tratada imediatamente. Caso o tratamento de interrupções esteja inibido, suas informações são mantidas em uma lista para posterior tratamento.

Cada tipo de interrupção está associado a uma ISR (*Interrupt Service Routine*), ou rotina de serviço de interrupção, que é chamada na ocorrência da interrupção correspondente. Para isso é mantida na memória uma IDT (*Interrupt Descrition Table*), uma tabela que relaciona cada tipo de interrupção a um ponteiro para o endereço de execução de sua respectiva ISR.

3.2.3. Processos

Segundo Axford [2], um processo é uma parte de aplicação executada seqüencialmente, mas que pode executar de maneira concorrente ou pseudo-concorrente com outros processos da mesma ou de outra aplicação.

Em outras palavras, um processo pode ser considerado um programa em estado de execução: constitui se do código executável, dos dados referentes ao código, dos valores correntes de todos os registradores do hardware, de sua pilha de execução e de um conjunto de outras informações sobre sua execução.

A cada processo é atribuído um nível de prioridade, baseado em sua importância, geralmente representado por um número inteiro. Em SOTR baseados em prioridades, os processos com prioridade mais alta têm a preferência na utilização do processador.

Processos podem assumir vários estados de execução enquanto estão ativos. Após sua criação, são considerados “prontos” para execução. Quando são “escalonados” pelo sistema operacional, entram em execução, isto é, no estado “executando”. No momento de sua execução, um processo pode:

a) Ser bloqueado por uma interrupção, passando para o estado “interrompido” enquanto tal interrupção é manipulada pela sua respectiva ISR, ou rotina de manipulação de interrupção;
b) Entrar no “modo de espera”, aguardando pela ocorrência de determinado evento de comunicação ou sincronização;
c) Sofrer uma preempção (troca de contexto), ou seja, após seu quantum ser esgotado, dar lugar a um outro processo, voltando ao estado de “pronto”;
d) Terminar o seu trabalho, ter seu contexto desalocado da memória e ser desativado, entrando no estado “dormindo”.

Quando no modo de espera, o processo deve esperar pela ocorrência do evento desejado. Quando este ocorrer, ele volta a se tornar pronto para a execução. Em alguns casos, há um limite de tempo para esta espera, chamado *timeout*. Caso este limite seja esgotado e o evento não ocorreu, o processo é colocado no estado de pronto. A Figura 1 apresenta as possíveis transições entre estes estados.

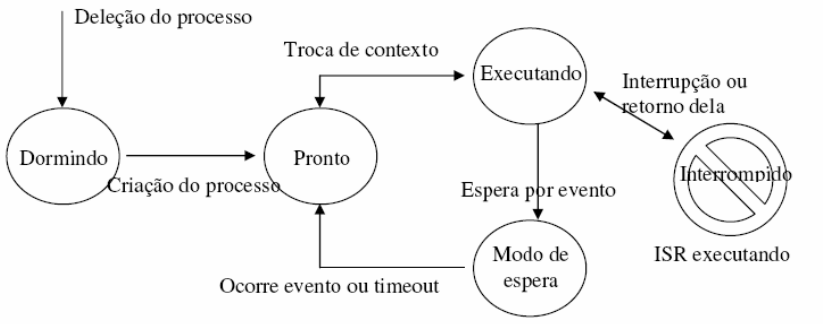


Figura 1 Diagrama de estados de um processo

3.2.4. Comunicação e sincronização

Se uma simples aplicação é composta por mais de um processo, executados concorrentemente, é comum a necessidade de cooperação entre tais processos. As duas principais categorias de cooperação entre processos concorrentes são a comunicação e a sincronização [2].

Na comunicação, dados são trocados entre processos através de mecanismos como memória compartilhada, filas de mensagens, *pipes*, FIFO (*First In First Out*) e *sockets*.

A sincronização em um SOTR é muito importante, pois controla o acesso dos processos a recursos compartilhados, como dispositivos, áreas de memória e *buffers*. A sincronização também é utilizada para ordenar a execução das instruções de processos concorrentes.

A necessidade de cooperação ocorre também entre processos de aplicações distintas, totalmente independentes, principalmente no acesso a um recurso compartilhado. Tal implementação pode ser feita através dos mesmos mecanismos anteriores.

Um exemplo desta necessidade é o caso da competição por uma impressora: um processo tem que esperar pelo término da impressão requerida pelo outro, antes de ter sua requisição processada.

3.2.5. Problemas clássicos de comunicação e sincronização

Esta seção introduz os problemas clássicos de comunicação e sincronização enfrentados pelos sistemas operacionais.

3.2.5.1. Exclusão mútua

A exclusão mútua é um dos mais importantes problemas de sincronização entre processos. Ela deve ser aplicada em circunstâncias nas quais é necessário garantir que partes de

dois processos concorrentes, que acessam um mesmo recurso compartilhado, não sejam executadas concorrentemente. Tais partes são chamadas regiões críticas.

Em situações mais complexas, pode haver várias regiões críticas, caso no qual a exclusão mútua não deve ser garantida entre todas as tarefas, mas somente entre aquelas que se relacionam, isto é, que disputam pelo mesmo recurso compartilhado, o qual pode ser uma estrutura de dados ou um dispositivo de hardware, por exemplo.

A Figura 2 exemplifica, através de um algoritmo, a necessidade de exclusão mútua, exibindo trechos de código de dois processos que estão sendo executados concorrentemente e compartilham o acesso à mesma variável, `indice`, e ao mesmo vetor, chamado `buffer`.

```
Processo1:                                   Processo2:
indice++;                                      indice++;
buffer[indice] = dado1;                  buffer[indice] = dado2;
```

Figura 2 Necessidade de exclusão mútua

Neste caso, a corrupção dos dados pode ocorrer, por exemplo, quando o processo 1 é interrompido logo após incrementar a variável `indice`. Se, em seguida, o processo 2 é executado, ele irá incrementá-la novamente, colocando `dado2` nesta posição do vetor. Quando o processo 1 volta a ser executado, ele simplesmente sobrescreve `dado1` na mesma posição escrita pelo processo 2.

Há vários mecanismos pelos quais é possível garantir a exclusão mútua, desde mecanismos de hardware até de software, dependendo das entidades envolvidas em cada caso.

A comunicação entre um processo executado no processador central e um controlador de periférico (o qual não é programável, mas executa um processo próprio)

geralmente se dá através de memória compartilhada. Neste caso, deve se garantir a exclusão mútua nos acessos à memória.

As próximas seções abordam mecanismos que representam tentativas de solucionar o problema da exclusão mútua. Porém, tais mecanismos não são utilizados na prática, ou por não funcionarem, ou por funcionarem somente em situações muito restritas.

3.2.5.1.1. Acesso único à memória

No âmbito do hardware, um simples acesso à memória, seja na forma de leitura ou escrita, é sempre indivisível. Ou seja, a operação, depois de iniciada, é concluída, não podendo ser interrompida. Operações indivisíveis também são chamadas de operações primitivas ou atômicas.

Isto nem sempre é verdadeiro no âmbito das linguagens de programação. Para que a exclusão mútua seja garantida no acesso a uma variável armazenada na memória, sua leitura e escrita devem ser feitas em um único acesso à memória.

Variáveis que ocupam mais de uma posição de memória, como vetores e estruturas de dados mais complexas, requerem mais de um acesso à memória e, portanto, sua exclusão mútua não é automaticamente garantida pelo hardware. Neste caso, deve ser garantida pelo próprio programador, através de mecanismos de software.

No desenvolvimento de programas concorrentes, o programador tem à sua disposição quatro caminhos a seguir:

a) Utilizar uma linguagem de programação de alto nível, que disponha de primitivas de concorrência. Neste caso, o programa é independente de arquitetura, com alto nível de confiabilidade e de fácil manutenção;

b) Utilizar um compilador de linguagem de alto nível que garanta a exclusão mútua na implementação dos tipos de dados utilizados no programa, o que é arriscado, pois a maioria dos compiladores não especifica os tipos de variáveis acessados atomicamente;

c) Escrever rotinas de acesso às variáveis compartilhadas em linguagem de máquina (*assembly*) e invocá-las a partir de um programa de alto nível, o que implica num produto final dependente da arquitetura utilizada;

d) Escrever todo o programa em *assembly*, o que é justificável apenas quando a eficiência obtida for considerável para a aplicação destino.

Conseqüentemente, o acesso único à memória não garante a exclusão mútua em situações gerais, não podendo ser considerado como uma real solução para o problema.

3.2.5.1.2. Inibição de interrupções

A mais evidente forma de se garantir a exclusão mútua é fazer com que cada processo desabilite a ocorrência de interrupções na entrada de sua região crítica e as reabilite novamente na saída.

Com as interrupções inibidas, a troca de contexto do processo em execução não acontece, pois ela depende de uma interrupção de tempo para ocorrer. Assim, está garantido que este processo não sofra preempção.

Enquanto estão desabilitadas, as interrupções requisitadas são mantidas em uma lista própria, sendo manipuladas somente após serem reabilitadas. Isto garante que nenhuma delas seja perdida.

Apesar de ser simples e confiável, esta solução possui fortes impedimentos: não funciona em sistemas multiprocessados e impede a troca de contexto para qualquer outro processo, inclusive aqueles que não possuem regiões críticas.

3.2.5.1.3. Instruções atômicas

O hardware é projetado de forma que uma interrupção não ocorra durante a execução de uma instrução, apenas entre o término de uma e o início da seguinte. Desta forma, uma vez iniciada, a instrução é executada até o fim.

Daí o fato de que programas escritos em *assembly* são livres do problema da exclusão mútua, pois nesta linguagem cada comando corresponde a uma instrução de máquina.

Por outro lado, comandos de linguagens de programação de alto nível são freqüentemente executados em várias instruções de máquina e, portanto, podem ser interrompidos por um sinal de interrupção. Desta forma, tais linguagens provêem instruções especiais de exclusão mútua, que são executadas atomicamente. As mais comuns são:

a) Instrução de troca: realiza uma operação de troca de conteúdo entre um endereço de memória e um registrador, ou entre dois endereços de memória;
b) Instrução '*test and set locked*', ou TSL: armazena o conteúdo de um endereço de memória num registrador e atribui um valor não nulo a tal endereço;
c) Variável de '*lock*', ou travamento: congela o valor de determinada palavra de memória, que, se alterado por outro processo, deve ser retornado imediatamente ao valor original;
d) Instruções de incremento e decremento: realiza uma operação de incremento ou decremento no valor de um endereço de memória e armazena o resultado em um registrador.

3.2.5.1.4. Mecanismos para sistemas multiprocessados

A indivisibilidade das instruções de máquina somente garante a exclusão mútua em sistemas uniprocessados. Cada instrução pode utilizar vários acessos à memória e não há maneira de saber se acessos consecutivos realizados por outro processador pertencem à mesma instrução ou não.

A inibição de interrupções também não possui efeito em sistemas multiprocessados, pois é implementada individualmente para cada processador. Conseqüentemente, novos mecanismos de exclusão mútua são necessários.

Alguns sistemas multiprocessados com memória compartilhada provêem uma instrução de travamento de memória, que funciona como um prefixo à instrução imediatamente posterior a ela, travando o acesso à memória durante sua execução.

Já em sistemas distribuídos, sem memória compartilhada, o único mecanismo disponível para implementar a exclusão mútua é a troca de mensagens.

3.2.5.2. O problema "produtor-consumidor"

O problema produtor-consumidor é uma abstração de uma classe de problemas muito comum na maioria dos STR. Nele, um processo, chamado produtor, produz informações e as envia a outro processo, chamado consumidor.

Assume se que a informação produzida consista de uma seqüência de itens, cada qual podendo constituir qualquer tipo de informação. Os itens produzidos são armazenados em um mecanismo de comunicação pelo produtor e, posteriormente, são retirados pelo consumidor.

Os requisitos essenciais para a solução satisfatória deste problema são:

a) Os itens de dados devem ser recebidos na mesma ordem em que foram enviados;

b) Nenhum item pode ser perdido durante o envio;

c) Nenhum item pode ser corrompido durante o envio, nem ser duplicado e enviado duas vezes.

Duas considerações inerentes ao problema e decorrentes da necessidade de sincronização são:

a) O processo consumidor não pode receber dados antes destes serem enviados pelo produtor;

b) Geralmente o mecanismo pelo qual os dados são enviados tem uma capacidade finita, o que implica na impossibilidade de envio caso não haja capacidade.

Um caso especial é o caso da inexistência deste mecanismo. Neste caso a sua capacidade é zero e o produtor somente pode enviar o item caso o consumidor esteja pronto para recebê-lo. Este caso especial é chamado de *rendezvous* [2].

3.2.5.3. O problema "leitores e escritores"

Este problema é decorrente de situação na qual vários processos acessam uma área de dados compartilhada. Em sua forma mais simples, é chamado de "problema leitores e escritor". Nele, um processo, o escritor, escreve na área compartilhada concorrentemente com um número arbitrário de processos que a lêem, os leitores.

O problema exige que a informação possa ler lida, ao mesmo tempo, por qualquer número de processos leitores e que, enquanto o processo escritor estiver escrevendo na área compartilhada, nenhum leitor poderá lê-la. Não é permitido que leitores escrevam na área nem que o escritor a leia.

Em sua forma mais geral, qualquer número de escritores é permitido. Obviamente, tal problema passa a se chamar “problema leitores e escritores”. Neste caso, além das duas condições anteriores, o problema exige que apenas um escritor esteja escrevendo na área compartilhada num determinado momento.

3.2.5.4. Compartilhamento de recursos

O problema de compartilhamento de recursos mais comum nos sistemas operacionais é o escalonamento de processos em um sistema uniprocessado. O requisito básico para este e os muitos outros problemas deste tipo é a garantia de exclusão mútua de acesso aos recursos.

A complexidade do problema pode aumentar caso os recursos sejam múltiplos ou possam ser alocados em parte, tal como a memória principal. Um problema representativo da categoria é o “jantar dos filósofos”.

Nele, cinco filósofos estão sentados a uma mesa circular. No centro da mesa, há uma tigela com macarronada reposta regularmente. Cada filósofo tem seu próprio prato e necessita de dois garfos para comer. Há apenas cinco garfos disponíveis, um deles entre cada par de pratos. Para comer, o filósofo deve pegar os garfos imediatamente à esquerda e à direita de seu prato, sendo os garfos, portanto, o recurso compartilhado.

A solução deve ser tal que impeça que um filósofo coma para sempre, e que nenhum deles seja impossibilitado de comer por um tempo infinito, o que é chamado de *starvation*.

3.2.6. Semáforo

Todos os problemas estudados requerem a adoção de um mecanismo que realmente garanta a exclusão mútua dos processos envolvidos. Como já mencionado, a inibição das interrupções garante a exclusão mútua, mas possui fortes impedimentos.

A utilização de variáveis simples para acesso às regiões críticas pode levar a condições de corrida, pois permite que um processo altere o valor da variável enquanto outro a está testando, como no caso da Figura 2.

A solução óbvia, então, é tornar o teste e alteração da variável de acesso uma operação indivisível, o que foi conseguido por E. W. Dijkstra, em 1965, com o conceito de semáforo [8].

Um semáforo é uma variável inteira protegida, cujo valor somente pode ser lido ou alterado através de duas operações atômicas especiais, P e V. Suas definições clássicas, através de algoritmo, são apresentadas pela Figura 3, onde S representa o semáforo sobre o qual a operação é realizada.

```
P (S):
            while (S <= 0) do continue;
            S--;
V (S):
            S++;
```

Figura 3 Definição clássica das operações de um semáforo

As operações P e V costumam ser encontradas na literatura na forma de, respectivamente, *wait* e *signal*, ou ainda, *up* e *down*. Tais nomenclaturas alternativas proporcionam uma melhor compreensão do funcionamento dos semáforos.

Há dois tipos de semáforo: o binário e o de contagem. Como o próprio nome diz, o valor de um semáforo binário pode ser 0 ou 1, indicando que no máximo um processo pode utilizar o recurso compartilhado em determinado instante.

O semáforo de contagem permite que um maior número de processos acessem ao mesmo tempo o recurso compartilhado. É utilizado para proteção de acesso a recursos mais complexos, que permitam o acesso múltiplo sem levar a condições de corrida.

No momento de iniciação, aos semáforos binários é atribuído o valor 1, enquanto que aos semáforos de contagem é atribuído o número máximo de processos que podem acessar o recurso ao mesmo tempo. O valor de um semáforo nunca pode ser negativo.

Cada processo concorrente, antes de entrar em sua região crítica, deve fazer uma chamada a P. Se, neste momento, o valor do semáforo for maior que zero, ele é decrementado e o processo pode executar sua região crítica. Caso seja zero, o processo é bloqueado à espera da liberação do semáforo.

Ao término de sua região crítica, o processo deve chamar V para liberar (incrementar) o semáforo para que outro processo, que nele esteja bloqueado ou venha a solicitá-lo, acesse sua região crítica.

A Figura 4 apresenta, em linguagem C, uma solução adequada para o problema produtor-consumidor utilizando três semáforos. Um deles, `mutex`, é utilizado para a obtenção da exclusão mútua; os outros dois, `empty` e `full`, para sincronização entre os processos, contando, respectivamente, o número de posições livres e ocupadas do *buffer*.

```
#include "prototypes.h"
#define N 100                          //número de posições do buffer

typedef int semaphore;
semaphore mutex = 1;                   //controla acesso à região crítica
semaphore empty = N;                   //conta posições vazias do buffer
semaphore full = 0;                    //conta posições ocupadas do buffer

void producer (void)
{
      int item;
      while (TRUE)                     //TRUE é a constante 1
      {
            produce_item(&item);       //produz um novo item
            down(&empty);
            down(&mutex);
            enter_item(item);          //coloca o novo item no buffer
            up(&mutex);
            up(&full);
      }
}

void consumer (void)
{
      int item;
      while (TRUE)
      {
            down(&empty);
            down(&mutex);
            remove_item(&item);        //retira um item do buffer
            up(&mutex);
            up(&empty);
            consume_item(item);        //trabalha com o item retirado
      }

}
```

Figura 4 Solução do problema produtor-consumidor através de semáforos

3.2.7. Troca de mensagens

Os semáforos são adequados a sistemas multiprocessados com memória compartilhada. No caso dos sistemas distribuídos, nos quais vários processadores, cada um com sua própria memória, estão conectados através de uma rede de comunicação, a utilização do semáforo não tem efeito.

Neste tipo de sistema a exclusão mútua pode ser obtida através da troca de mensagens entre os processos envolvidos. Além de possibilitar sincronização entre processos remotos, a troca de mensagens ainda permite a comunicação entre eles.

O mecanismo de troca de mensagens está fundamentado em duas primitivas: *send* e *receive*. A primeira envia uma mensagem para determinado destino e a outra recebe uma mensagem de determinada fonte.

Quando um processo envia uma mensagem e nenhum outro processo corrente já estiver esperando por ela, o processo remetente é bloqueado, sendo acordado somente quando o processo destino executar uma chamada a `receive`. A Figura 5(a) ilustra tal situação.

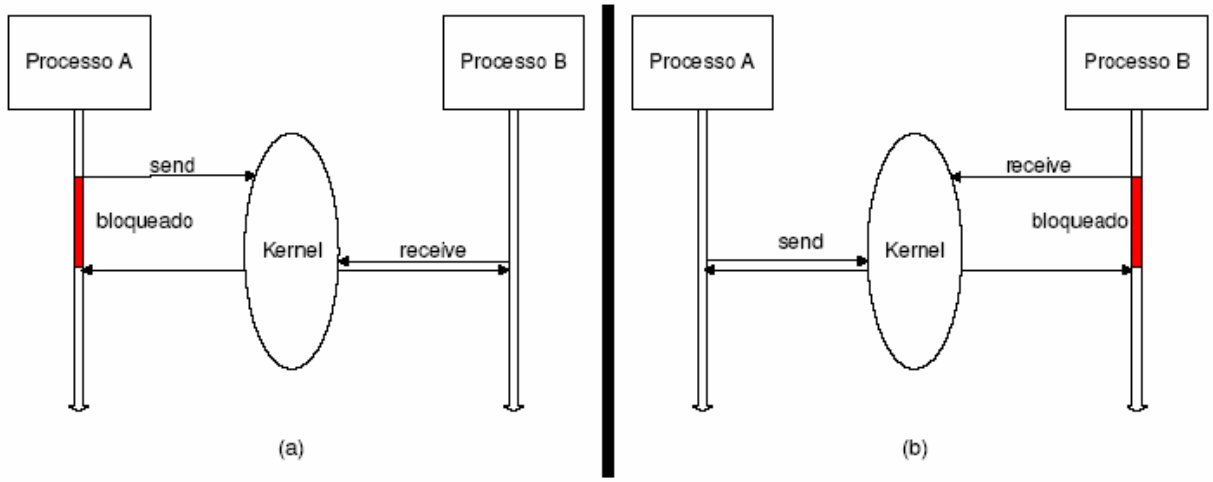


Figura 5 Utilização da troca de mensagens para sincronização

Por outro lado, se o processo destino chamar `receive` antes de a mensagem ser enviada pelo processo fonte, o primeiro fica bloqueado até o recebimento da mensagem. O remetente não é bloqueado, pois sua mensagem é recebida diretamente pelo processo destino. A Figura 5(b) ilustra tal situação. Com o advento do mecanismo de troca de mensagens surgem novos problemas de comunicação.

Por exemplo, uma mensagem pode ser perdida durante a transmissão. Para contornar este problema, todo processo que receba uma mensagem deve responder ao remetente com uma mensagem de reconhecimento, certificando sua chegada. Assim, se uma das mensagens é perdida, sua mensagem de reconhecimento não chegará (ou por não ter sido enviada ou por ter sido perdida), e a mensagem original deve ser reenviada. Os sistemas baseados em troca de mensagens devem gerenciar também a questão de identificação dos processos, para que estes não sejam confundidos. A Figura 6 apresenta a solução do problema produtor-consumidor através de troca de mensagens, utilizando linguagem C.

```
#include "prototypes.h"
#define N 100                       //número de posições do buffer
#define MSIZE 4                     //tamanho da mensagem

typedef int message[MSIZE];

void producer (void) {
      int item;
      message m;                    //buffer de mensagens

      while (TRUE)
      {                             //TRUE é a constante 1
            produce_item(&item);    //produz um novo item
            receive(consumer, &m);  //aguarda chegada de mensagem vazia
            build_message(&m, item);//cria uma nova mensagem para envio
            send(consumer, &m);     //envia um item ao consumidor
      }
}

void consumer (void) {
      int item, i;
      message m;
      for (i = 0; i < N; i++)       //envia N mensagens vazias
      send (producer, &m);

      while (TRUE)
      {
            receive(producer, &m);  //recebe mensagem com o item
            extract_item(&m, &item);//retira item da mensagem
            send(producer, &m);     //envia de volta mensagem vazia
            consume_item(item);     //trabalha com o item retirado
      }

}
```

Figura 6 Solução do problema produtor-consumidor por troca de mensagens

Na Figura 6, o produtor produz um item e o envia ao consumidor, através de uma mensagem. Este, após consumir o item da mensagem recebida, responde ao produtor com uma mensagem vazia.

3.3. Funcionamento de um kernel de tempo-real

A Figura 7 mostra o esquema de funcionamento de um SOTR. Nela o sistema operacional interage diretamente com o hardware, servindo os processos, os quais compõem as aplicações de tempo-real que estão sendo executadas [27].

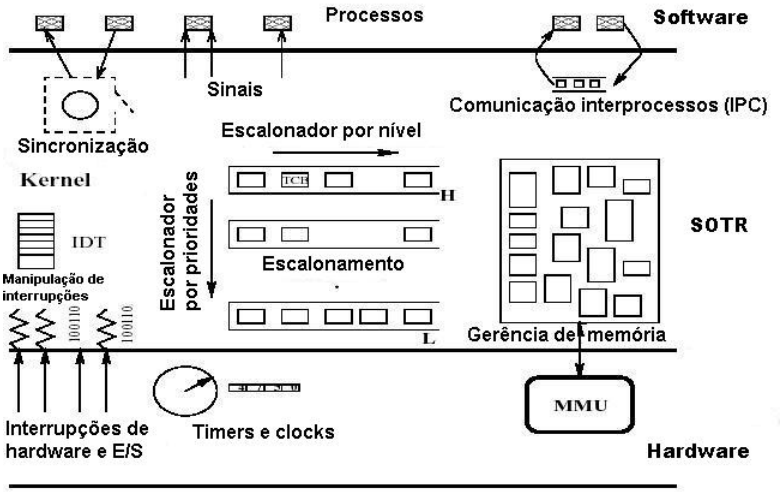


Figura 7 Esquema de funcionamento de um SOTR

É função do *kernel* (núcleo) do SOTR controlar todos os recursos físicos (hardware) da máquina, além de gerenciar os processos (criar, destruir, escalonar e sincronizar), as interrupções (monitorar e manipular a IDT), a alocação de memória, a memória virtual, os sinais de *clock* e temporizadores (*timers*), e estabelecer a comunicação entre as tarefas.

3.3.1. Chamadas de sistema

Há diversos tipos de chamadas de sistema, também conhecidas como primitivas ou, ainda, serviços. Elas podem ser bloqueantes ou não-bloqueantes, com *timeout* ou assíncronas.

Primitivas bloqueantes são usadas quando um processo deve esperar pela ocorrência de um evento antes de prosseguir. Caso este evento ainda não tenha ocorrido no momento de chamada a uma primitiva bloqueante, o processo que a chamou é bloqueado, entrando no modo de espera até a ocorrência deste evento.

Algumas primitivas bloqueantes trabalham com *timeout*, ou seja, um limite de tempo para a espera. Depois de esgotado tal limite, sem a ocorrência do evento esperado, a chamada à primitiva é perdida e o processo torna-se pronto para execução novamente. O *timeout* é utilizado para que o processo não fique no modo de espera infinitamente, caso o evento não ocorra.

Primitivas não-bloqueantes são usadas em situações contrárias, isto é, quando o processo chamador pode continuar sua execução normalmente, sem esperar pela ocorrência de um evento. Já as primitivas assíncronas provocam retorno imediato do processo chamador, sem esperar pelo resultado da chamada.

3.3.2. Multiprogramação

A utilização de multiprogramação permite que vários processos estejam em estado de execução ao mesmo tempo (um em cada processador). Esta é uma característica essencial para que os SOTR cumpram suas *deadlines*.

Para que os processos sejam multiprogramados, são utilizadas primitivas para sua iniciação, bloqueio e finalização. Além disso, o mecanismo de multiprogramação deve ser:

a) Baseado em níveis de prioridade, atribuídos aos processos no momento de sua criação, podendo ser alterados posteriormente. É necessário que o SOTR forneça um número mínimo de níveis de prioridade (32 é considerado plausível [14]) para que tal sistema seja utilizável na prática;

b) Preemptível, tanto no nível dos processos quanto no nível das interrupções, isto é, deve permitir que interrupções ou processos mais prioritários tomem a posse do processador (ou de outro recurso) de outros processos em qualquer instante.

A utilização de prioridades pode acarretar no problema da inversão de prioridades, o qual deve ser evitado por um SOTR. Este problema ocorre quando um processo de alta prioridade (H) espera um processo de baixa prioridade (L) liberar um recurso, cuja posse pertence a um terceiro processo (M), de prioridade intermediária entre os dois.

A Figura 8 ilustra tal problema em seis passos. Inicialmente, o processo L adquire o semáforo (1). Em seguida, o processo L dá lugar ao processo H para executar (2). O processo H tenta adquirir o semáforo e é bloqueado (3). O processo L volta à execução e novamente é interrompido, dando lugar ao processo M (4), que executa até o fim, dando lugar (5) ao processo L continuar até liberar o semáforo (6).

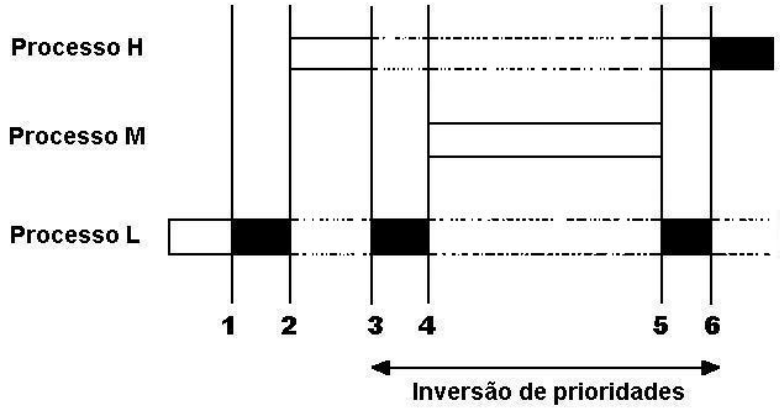


Figura 8 Inversão de prioridades

Portanto, por um longo intervalo de tempo, indicado na Figura 8, a inversão de prioridades é mantida, atrasando a execução de um processo mais prioritário, o processo H.

Este problema pode ser resolvido através do mecanismo de herança de prioridades: no momento em que o processo H tenta adquirir o semáforo, de posse do processo L, de menor prioridade, a sua prioridade é atribuída a ele, fazendo com que este não mais sofra uma preempção do processo M. Para que ocorra esta atribuição, o SOTR em questão deve oferecer um mecanismo de atribuição dinâmica de prioridades [14].

A Figura 9 ilustra a minimização deste problema através da herança de prioridades em seis passos. Inicialmente, o processo L adquire o semáforo (1). Em seguida o processo L dá lugar ao processo H para executar (2). O processo H tenta adquirir o semáforo, é bloqueado (3). Sua prioridade é atribuída ao processo L, que volta a ser executado, não sofrendo mais preempção do processo M. Após o processo L liberar o semáforo, ele é bloqueado (4). O processo H assume o processador e utiliza o recurso protegido pelo semáforo, liberando o em seguida (5). Após o término do processo H, o processo M entra em execução (6).

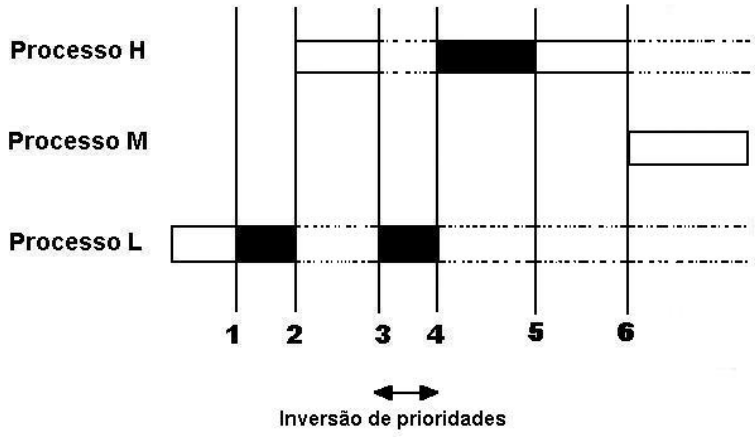


Figura 9 Herança de prioridades

Um SOTR deve distinguir com clareza entre suas entidades escalonáveis (as quais podem sofrer escalonamento) e não-escalonáveis. Tipicamente, entidades escalonáveis se caracterizam por um contexto (bloco de dados) e podem requisitar recursos explicitamente;

portanto devem ser escalonadas por um escalonador. Escalonadores, manipuladores de interrupção e chamadas de sistema são não-escalonáveis por natureza, pois sua utilização da CPU é incerta, isto é, pode ser contínua, periódica ou ainda em conseqüência da ocorrência de eventos.

Além disso, um SOTR deve possuir tempos de latência pequenos e conhecidos. Em outras palavras, os tempos gastos com trocas de contexto entre processos e interrupções devem ser curtos e previsíveis (não podem sofrer variações bruscas).

3.3.3. Temporizadores e manipulação de interrupções

SOTR devem dar um bom suporte a temporizadores e *clocks*, com resolução de nanossegundos. Isto permite uma execução mais rápida dos processos gatilhados por interrupções de tempo.

Para um melhor desempenho dos sistemas de tempo-real, é importante que o SOTR permita aos programadores implementar ISR específicas para cada tipo de interrupção.

3.3.4. Gerência de memória

Em quase todas as aplicações de tempo-real tarefas competem por memória, a qual é alocada dinamicamente a partir de *pools* de memória.

Os dois principais problemas no esquema de alocação e liberação de memória são o indeterminismo e a fragmentação. O indeterminismo está ligado a tempos desconhecidos e imprevisíveis de execução, os quais são decorrentes do tempo gasto com a busca por um bloco de memória que seja suficientemente grande para armazenar as informações desejadas.

A fragmentação ocorre quando são feitas requisições de memória de diferentes tamanhos num mesmo *pool*. Uma requisição posterior pode falhar por não haver um bloco

contínuo do tamanho desejado, apesar de haver memória suficiente disponível no sistema, mas em fragmentos menores.

As técnicas de *swapping* e paginação, as quais permitem que seja utilizada uma memória maior que a disponível no sistema através da utilização do disco, geram tempos de execução imprevisíveis e, portanto, não são apropriadas aos SOTR. Por outro lado, a especificação POSIX [20] cobre mecanismos de *swapping* e paginação. Mas, ao mesmo tempo em que faz isso, fornece mecanismos para travá-los nas porções de código desejadas [27].

A maioria dos processadores atuais vem com uma MMU (*Memory Management Unit*), a qual corresponde a uma unidade especial para gerenciar a memória. Se, por um lado, a alocação de memória dinâmica dá uma maior flexibilidade de programação, por outro ela acrescenta o tempo gasto com a coleta de lixo.

Fornecer ou não suporte a memória virtual é sempre uma difícil escolha no desenvolvimento de um SOTR. Se o processador possui uma MMU e o SOTR não suporta memória virtual, as funções da MMU são desperdiçadas; se a memória virtual é suportada ela pode levar a um indeterminismo das aplicações.

A maioria dos SOTR de tempo-real restringe a utilização da alocação de memória dinâmica. Por exemplo, quase todos eles impedem ISRs de realizarem tais alocações.

4. DESENVOLVIMENTO DE SOFTWARE BASEADO EM COMPONENTES

Este capítulo introduz os conceitos relacionados à tecnologia de componentes de software, apresentando suas principais características. Em seguida, é apresentado o padrão UML e o método Catalysis de desenvolvimento de componentes, cujas idéias formam a base da abordagem CSK.

4.1. Introdução

A tecnologia de programação orientada a objetos vem sendo amplamente adotada nos processos de desenvolvimento de software. Sua utilização proporciona uma visão mais natural no desenvolvimento de software, através da especificação de classes cujas características são abstraídas dos objetos do mundo real.

Apesar disso, seus benefícios não estão sendo totalmente consolidados em resultados. Conseqüentemente, os ganhos de produtividade esperados não estão sendo obtidos. Isto acontece porque os resultados não dependem somente da aplicação da tecnologia, mas de uma aplicação correta, o que não é conseguido por todos os engenheiros de software, pelos mais variados motivos. Desta forma, a utilização da tecnologia de orientação a objetos por si só não resultou em sistemas totalmente reutilizáveis.

A tecnologia de componentes de software preenche estas necessidades, pois está focada exclusivamente no reúso de software. Por reúso de software entende se a utilização de produtos de software – por exemplo, componentes – em uma situação diferente daquela para a qual tais produtos foram desenvolvidos [9].

O grande diferencial desta tecnologia está em desenvolver os componentes como projetos à parte, focados na solução completa de problemas de menor escala, funcionando como caixas pretas e tendo como objetivo o reúso de software. Desta forma, tais componentes não são atrelados a projetos específicos de software. Ao contrário, correspondem a serviços que podem ser utilizados em vários projetos conforme a demanda.

Além da reutilização automática, outras vantagens da tecnologia de componentes são os maiores níveis de flexibilidade e facilidade de manutenção, melhor aproveitamento do legado (sistemas já existentes), maior facilidade na integração de sistemas e um menor tempo de desenvolvimento.

4.2. Definição

O desenvolvimento de software baseado em componentes está relacionado com a especificação, desenvolvimento, manutenção e reutilização de componentes de software.

Segundo Parrish [17] [19], um componente é uma unidade de software projetada e construída independentemente de uma aplicação e posteriormente integrada a uma ou mais aplicações.

D'Souza [9] apresenta uma definição mais apropriada ao que chama de componente de implementação:

"Um pacote coerente de implementação de software que pode ser desenvolvido independentemente e entregue como unidade; tem interfaces explícitas e bem definidas para os serviços que oferece; tem interfaces explícitas e bem definidas para os serviços que requer; e pode ser composto com outros componentes, talvez após a personalização de algumas propriedades, mas sem modificar os componentes em si".

A partir destas definições, pode se notar que a principal diferença de componentes para outros artefatos de software, como bibliotecas, é a construção de aplicações por montagem, sem o conhecimento do código fonte do componente. O componente pode até ter sido implementado em uma linguagem de programação diferente, desde que possua interfaces compatíveis.

4.3. Padrões e técnicas de desenvolvimento de componentes

A linguagem UML (*Unified Modeling Language* [23], ou Linguagem de Modelagem Unificada) se tornou uma notação comum para a especificação, visualização, construção e documentação de sistemas orientados a objetos.

Com isso, várias técnicas adicionais surgiram com o intuito de auxiliar nas mais variadas tarefas de desenvolvimento. Na área de desenvolvimento de componentes, destaca-se o método Catalysis [9], cuja linguagem oficial adotada para a representação e documentação de sistemas é a UML.

Nas próximas seções são apresentados o padrão UML e o método Catalysis, os quais são utilizados no desenvolvimento de componentes na abordagem CSK.

4.3.1. UML

O padrão UML, estabelecido pelo OMG (*Object Management Group* [16]), auxilia na especificação, visualização e documentação de modelos de sistemas, facilitando o cumprimento dos requisitos do produto.

Através da linguagem UML é possível documentar qualquer tipo de aplicação, escrita em qualquer linguagem de programação, operando em qualquer plataforma de hardware, sistema operacional e rede.

A modelagem e a documentação são aspectos essenciais no desenvolvimento de um bom projeto de software, especialmente no caso de sistemas mais complexos, que envolvam uma grande equipe de engenheiros.

Através de notação predominantemente gráfica, de fácil entendimento, a linguagem UML passou a ser amplamente adotada entre os engenheiros de software, tornando-se a linguagem oficial de troca de informações entre os membros da grande maioria das equipes de desenvolvimento.

4.3.1.1. Características da UML

A UML faz uso de um conjunto de modelos para a documentação e representação das características dos sistemas. Cada modelo apresenta uma diferente perspectiva do sistema, em variados graus de abstração. Os modelos comumente criados num projeto de software são:

a) **Modelo de casos de uso,** que descreve as situações possíveis entre usuários e sistema;

b) **Modelo de classes,** que apresentam a modelagem de classes, pacotes, objetos, com seus respectivos conteúdos, estruturas e relacionamentos;

c) **Modelo de seqüência,** que exibe a seqüência temporal de participação dos objetos nas interações;

d) **Modelo de colaborações,** que exibem interações organizadas entre os objetos;

e) **Modelo de estados,** que apresenta a seqüência de estados de um objeto em uma interação;

f) **Modelo de atividades,** que exibe os fluxos de processamento interno;

g) **Modelo de componentes,** que mostra a estrutura em alto nível dos pacotes de código, incluindo as dependências entre seus componentes;

h) **Modelo de organização,** que expõe a configuração em tempo de execução dos componentes, processos e objetos do sistema.

As próximas seções apresentam os modelos mais comumente adotados no desenvolvimento de software: modelo de casos de uso, modelo de classes e modelo de componentes.

4.3.1.2. Modelo de casos de uso

Modelos de casos de uso são úteis principalmente na exposição dos requisitos de um sistema, nas fases de análise e projeto. Para representar os requisitos, o modelo de casos de uso utiliza os conceitos de atores e casos de uso em suas modelagens.

Um ator representa um usuário ou outro sistema que interage com o sistema que está sendo modelado. Um caso de uso é uma visão externa ao sistema que representa uma ação executada por um ator. A representação UML para atores e casos de uso é exibida pela Figura 10. Na figura, a seta indica qual ator executa qual caso de uso. O modelo de casos de uso reúne representações para todos os casos de uso do sistema modelado.

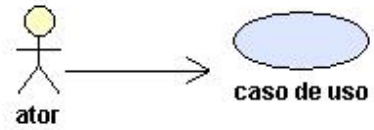


Figura 10 Representação de atores e casos de uso

4.3.1.3. Modelo de classes

Modelos de classes são utilizados em praticamente todos os projetos de software orientado a objetos. Eles exibem uma visão lógica das classes de objetos do sistema, através da descrição de sua estrutura, conteúdo e relacionamentos.

As classes são compostas de três partes: um nome, um conjunto de atributos e um conjunto de métodos (operações). A representação UML para as classes é apresentada pela Figura 11.

Figura 11 Representação UML para uma classe

O modelo de classes também representa relacionamentos, tais como associação e generalização. A Figura 12 exibe um exemplo com quatro classes, `Pedido`, `Cliente`, `PessoaFísica` e `PessoaJurídica`.

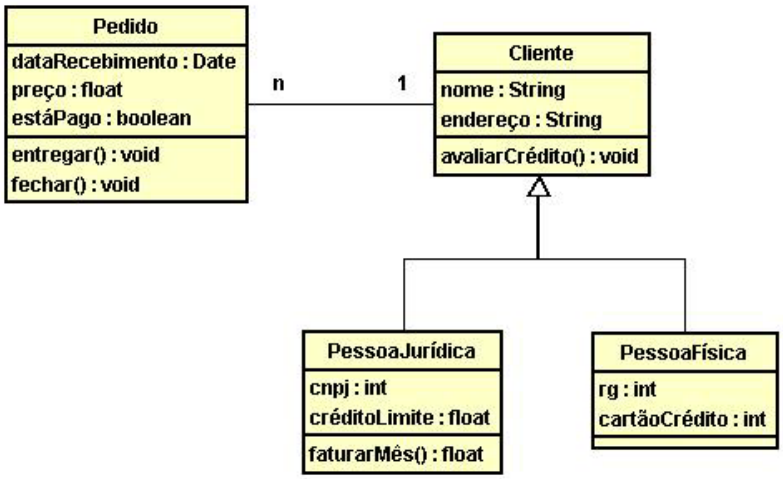


Figura 12 Relacionamentos em um modelo de classes UML

O exemplo exibe um relacionamento de associação entre as classes `Pedido` e `Cliente`, onde os valores n e 1 indicam a multiplicidade da relação, e um relacionamento de generalização entre as classes `Cliente`, `PessoaJurídica` e `PessoaFísica`.

4.3.1.4. Modelo de componentes

O modelo de componentes é utilizado após a implementação completa do sistema e exibe os componentes de software envolvidos e as dependências entre eles. As dependências são representadas através de conexões entre as interfaces dos componentes. A Figura 13 apresenta um exemplo de implementação da classe `Cliente` através de um componente.

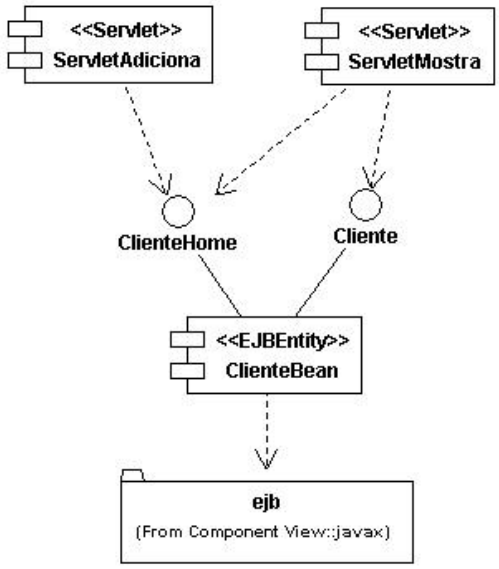


Figura 13 Representação de componentes através da liguagem UML

No exemplo, a classe `Cliente` está implementada pelo componente `ClienteBean`, que depende do pacote `ejb`, da biblioteca Java. O componente `ClienteBean` possui duas interfaces, `ClienteHome` e `Cliente`, das quais dependem os componentes `ServletAdiciona`, cuja função é adicionar um cliente na base de dados, e `ServletMostra`, cuja função é exibir os clientes que constam na base de dados.

4.3.2. Catalysis

Catalysis [6] [9] é um método não-proprietário utilizado para o desenvolvimento sistemático de sistemas baseados em componentes, utilizando o padrão UML.

O método Catalysis é aplicado com sucesso desde 1992 por empresas das mais variadas áreas de negócio, como telecomunicações, seguros, sistemas embarcados, finanças, viagens, transportes, entre outras.

O método é divido em três níveis:

a) Definição do domínio do problema: entendimento do problema e especificação do que o sistema deve fazer para resolver o problema, isto é, abstração de suas funcionalidades;
b) Especificação dos componentes: descrição do comportamento do sistema em termos de componentes, de maneira não ambígua;
c) Implementação dos componentes: definição de como os requisitos especificados serão implementados.

Nestes três níveis são utilizados os princípios da abstração, refinamento e conexão de componentes. A abstração guia o engenheiro na busca dos aspectos essenciais ao sistema. O refinamento tem por objetivo a detecção de erros e inconsistências na fase de modelagem. A conexão de componentes é responsável por prover uma forma de reutilização em outras aplicações, através das interfaces.

O primeiro nível consiste, inicialmente, em identificar, a partir da descrição do problema em linguagem natural, os objetos e ações envolvidos, descrevendo-os através de técnicas como os *storyboards* ou *mind maps* [21].

Em seguida, tais informações são expandidas em modelos de colaboração. Finalmente, os modelos de colaboração são refinados num modelo de casos de uso (*use case model* [23]).

O segundo nível consiste na descrição do comportamento do sistema. O primeiro passo é a criação do modelo de tipos, o qual define os atributos e operações dos tipos de objeto, e

do dicionário de dados, que contém a definição de cada um destes tipos. A este dicionário são acrescentadas informações durante todo o processo.

Em seguida, o comportamento dos objetos é detalhado, sem ambigüidade, por uma linguagem descritiva, a qual pode ser linguagem natural, OCL (*Object Constraint Language* [9]) ou qualquer outra que possibilite uma descrição sem ambigüidades.

O modelo de casos de uso, produzido no primeiro nível, é refinado em modelos de interação, representados por modelos de seqüência [23], os quais detalham o cenário de utilização dos componentes.

O terceiro nível é responsável pela implementação dos componentes. A partir deste momento detalhes como linguagem de programação, arquitetura, segurança, persistência, entre outros, tornam se significantes.

O primeiro passo é o refinamento do modelo de tipos produzido no nível anterior em um modelo de classes para cada componente [23]. Neles as classes são modeladas com seus relacionamentos, levando em consideração as definições dos componentes e de suas interfaces.

Os modelos de seqüência do modelo de interações são refinados, exibindo detalhes do comportamento dos métodos de cada classe. Finalmente, o modelo de componentes é desenvolvido, exibindo a organização dos componentes e suas relações de dependência (através das interfaces). O modelo de componentes pode utilizar componentes já existentes.

Ao final da aplicação do método Catalysis, portanto, dispõe-se de toda a documentação dos componentes, os quais devem ser implementados coerentemente.

5. ABORDAGEM CSK

A abordagem CSK (*Component Structured Kernel*) é dividida em duas fases, Construção de Componentes e Reconstrução do *Kernel*, conforme mostra a Figura 14, usando notação SADT [22]. Cada fase contém etapas que são executadas segundo o modelo espiral [21] de ciclo de vida do software, gerando protótipos a cada ciclo. As evoluções dos protótipos geram versões cada vez mais refinadas dos componentes que permitem um maior reúso das aplicações.

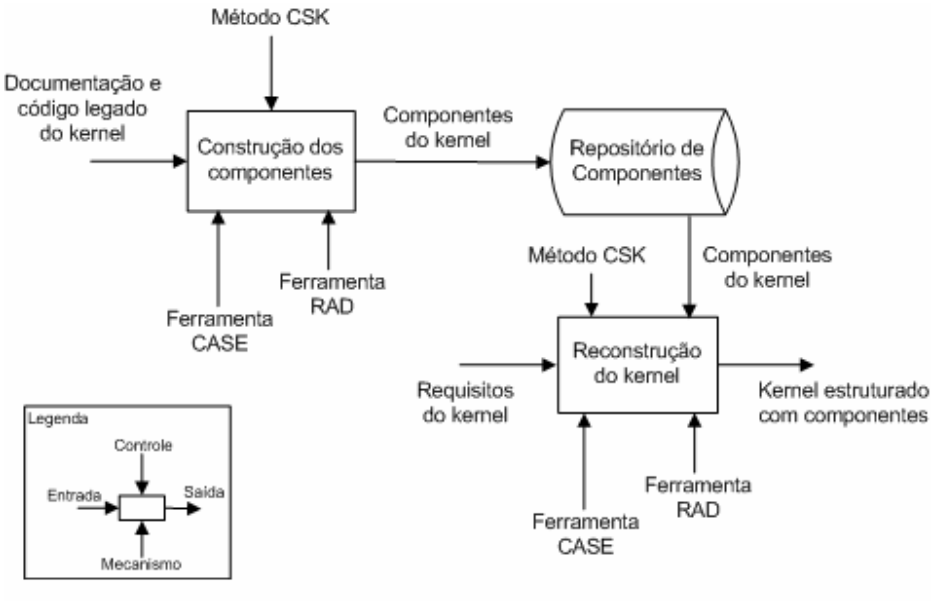


Figura 14 Fases da abordagem CSK

Os principais mecanismos que auxiliam na aplicação da CSK são uma ferramenta CASE (*Computer-Aided Software Design*) e uma ferramenta de desenvolvimento visual RAD (*Rapid Application Development*).

A idéia básica da abordagem é separar o código legado do *kernel* segundo suas funcionalidades e implementar cada uma delas através de um componente ou conjunto de componentes. A abordagem CSK pressupõe que o *kernel* legado que será reconstruído está operacional, ou seja, pode ser executado e seus requisitos estão corretamente implementados.

A idéia de separação em funcionalidades se baseia no fato de que sistemas operacionais são muito mais orientados a funções do que a dados. Busca-se, ainda, uma generalização destas funcionalidades, de forma a possibilitar um maior reúso dos componentes na construção de diferentes *kernels*.

O resultado da primeira fase da abordagem é a produção de um conjunto de componentes organizados conforme suas funcionalidades. Assim, podem-se ter componentes específicos para gerência de memória, comunicação, sincronização, escalonamento de tarefas, gerência do tempo, e assim por diante. Na segunda fase, o *kernel* original é reestruturado a partir destes componentes.

As próximas seções apresentam mais detalhadamente cada uma das fases da abordagem CSK.

5.1. Construção dos componentes

Sistemas operacionais reúnem em seu *kernel* um conjunto de serviços que provêem interfaces de alto nível às aplicações. Por serviço entende-se um conjunto de primitivas (funções ou procedimentos) e variáveis responsáveis pela implementação de uma funcionalidade específica.

Os serviços mais comuns oferecidos por sistemas operacionais são escalonamento de processos, gerência de memória, gerência de recursos, implementação de semáforos e comunicação entre processos.

O objetivo da primeira fase é construir os componentes de software para serem reutilizados na segunda fase. Inicia-se esta fase a partir do código legado e da documentação do sistema operacional de tempo-real alvo da estruturação.

Numa primeira atividade desta fase, são identificados os serviços oferecidos pelo *kernel* alvo. Após identificados, os serviços são associados às primitivas e variáveis envolvidas em sua implementação. Com base nas relações de comunicação entre estas primitivas, são definidas as interfaces dos componentes. Em seguida, os componentes são projetados e implementados. As próximas seções apresentam estas etapas mais detalhadamente.

5.1.1. Identificação dos serviços oferecidos pelo *kernel*

A primeira etapa da abordagem CSK consiste em identificar os serviços do *kernel* a partir da documentação disponível, da análise do código legado e de sua execução.

A documentação do *kernel* é um importante guia nesta tarefa, diminuindo consideravelmente o tempo necessário para a identificação dos serviços. A indisponibilidade desta documentação implica numa minuciosa análise do código legado do *kernel*, o que pode consumir muito tempo no levantamento dos serviços oferecidos.

Após a identificação dos serviços do *kernel*, uma lista de serviços é elaborada, contendo cada um dos serviços identificados e as respectivas primitivas envolvidas em sua implementação.

As primitivas que não possuem relação com nenhum serviço entre os identificados são, temporariamente, associadas a um serviço global, denominado *Main*.

5.1.2. Associação das variáveis do *kernel*

Após identificar os serviços oferecidos pelo *kernel*, faz-se a associação de suas variáveis a estes serviços. Esta associação baseia-se nas seguintes diretrizes:

a) Se a variável é exclusivamente manipulada por primitivas associadas a um determinado serviço da lista, ela deve ser associada a este serviço;

b) Se a variável não é global, mas é manipulada por mais de um serviço da lista, ela deve ser associada ao serviço que mais freqüentemente a manipula. Ou seja, o acoplamento entre os serviços deve ser diminuído ao máximo;

c) Se a variável é global, ela deve ser associada ao serviço global *Main*.

Uma característica comum a sistemas não orientados a objetos é a existência de manipulações diretas a variáveis. Neste caso, para todas as variáveis devem ser criadas funções de acesso, dos tipos *get* e *set*, em caso de inexistência de funções alternativas com o mesmo propósito.

Ao final desta etapa, a lista de serviços é incrementada com as novas funções de acesso às variáveis e com as variáveis associadas a seus respectivos serviços.

5.1.3. Identificação das relações de "quem chama quem"

Esta etapa consiste em identificar as conexões de mensagens entre as primitivas da lista de serviços resultante das etapas anteriores. Estas relações são representadas através de um grafo direcionado, construído segundo as seguintes diretrizes:

a) Para cada serviço da lista deve ser criado um vértice no grafo;

b) Para cada serviço da lista que possuir ao menos uma primitiva que acesse uma variável ou primitiva associada a outro serviço da lista, é traçada uma aresta, partindo do vértice correspondente ao primeiro serviço e chegando ao vértice correspondente ao segundo serviço.

O grafo direcionado resultante da execução destas diretrizes não deve conter ciclos ou ligações de duplo sentido. Esta ocorrência indica uma divisão incorreta dos serviços do *kernel*, ou seja, erros cometidos anteriormente.

Caso o grafo contenha este tipo de anomalia, tenta-se inicialmente uma reorganização dos serviços através da inclusão ou ainda da quebra de serviços, mantendo uma coesão para cada serviço sem mesclar funcionalidades não relacionadas. Se a anomalia não puder ser retirada apenas com a reorganização dos serviços, volta-se à primeira etapa e tenta-se realocar as variáveis e primitivas causadoras do problema.

Após estas alterações na composição dos serviços, o grafo é reconstruído e novamente analisado. Novas ocorrências de ciclos implicam na necessidade de novas alterações.

O grafo dirigido acíclico resultante desta análise representa os relacionamentos entre os serviços do *kernel*. Neste grafo, cada vértice sugere uma futura classe, cujos métodos correspondem às primitivas e cujos atributos correspondem às variáveis associadas ao serviço representado.

5.1.4. Projeto interno dos componentes

A partir desta etapa, o código legado do *kernel* passa a ser transformado segundo as idéias do paradigma orientado a componentes. Isto se dá através da definição de classes abstratas e concretas para cada um dos serviços identificados, representados pelos vértices do grafo direcionado produzido na etapa anterior.

A classe abstrata corresponde à interface que cada classe concreta deve implementar. Numa classe abstrata não são incluídos atributos, apenas os métodos da interface de exportação do serviço, ou seja, aqueles acessados por outros serviços do *kernel*. Eles são declarados como métodos virtuais puros, o que significa que sua real implementação se encontrará na respectiva classe concreta.

A classe concreta representa a própria implementação do componente. Cada classe concreta deve estender sua respectiva classe abstrata, através do mecanismo de herança. Numa

classe concreta são incluídos todos os atributos e métodos associados ao serviço, com suas respectivas implementações.

A forma de implementação destas classes varia conforme o ambiente no qual os componentes serão disponibilizados. A abordagem CSK é flexível e trata de características inerentes aos componentes, não se restringindo a um ambiente de implementação em particular.

Os métodos da interface de exportação de uma classe concreta, ou seja, aqueles que também fazem parte da respectiva classe abstrata, são agrupados em duas interfaces distintas, segundo a arquitetura da Figura 15.

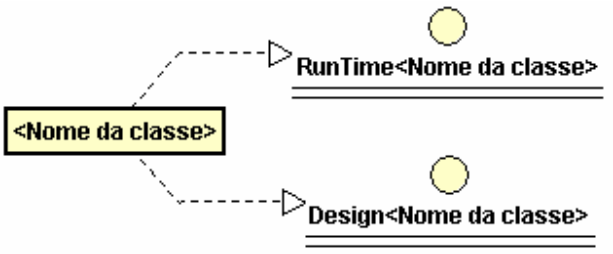


Figura 15 Arquitetura das classes dos componentes

Estas interfaces têm um padrão de nomes baseado no nome da classe concreta, conforme a seguinte nomenclatura: `Design`<Nome da classe> para a interface com propriedades e eventos que estarão disponíveis em tempo de *design*, e `RunTime`<Nome da classe> para a interface com os atributos e métodos disponíveis apenas em tempo de execução. Por exemplo, a classe `Timer` tem as interfaces `DesignTimer` e `RunTimeTimer`.

O resultado desta etapa do processo é um modelo de classes contendo as classes concretas para cada serviço da lista de serviços.

5.1.5. Implementação dos componentes

A partir do modelo de classes concretas os componentes do *kernel* são implementados. Esta tarefa consiste em adaptar o código-fonte do *kernel* segundo o paradigma

orientado a componentes e segundo as novas plataformas de hardware e software. Esta atualização inclui modificações como a redefinição de tipos e estruturas de dados, a eliminação de características obsoletas, entre outras.

Para facilitar as conexões entre os componentes e seu reúso, devem ser fornecidas formas de conexão, de preferência do tipo "*plug-in*". A forma de conexão depende da plataforma de hardware e software na qual os componentes serão implementados.

Na abordagem CSK, as classes concretas dão origem aos componentes extraídos do *kernel*. Desta forma, os componentes compartilham a mesma arquitetura, com duas interfaces, dos tipos `Design` e `RunTime`, conforme mostra o modelo da Figura 16.

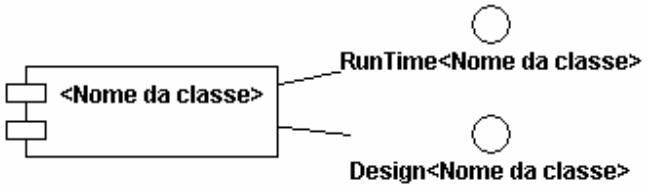


Figura 16 Arquitetura típica de um componente

O produto final desta etapa é um repositório de componentes que implementam os serviços do *kernel*.

5.2. Reconstrução do *kernel* com reúso de componentes

A segunda fase da abordagem CSK consiste na estruturação do *kernel* legado através da reutilização dos componentes construídos na fase anterior. As próximas seções detalham cada uma das etapas desta segunda fase do processo, que correspondem às etapas normais do ciclo de vida do software, desde a análise dos requisitos e projeto até a implementação e testes do novo *kernel* estruturado com componentes.

5.2.1. Análise de requisitos

Como em todo projeto de software, a criação de um novo *kernel* de tempo-real inicialmente requer uma análise do problema para identificar seus requisitos. Na fase de análise de requisitos são definidos os serviços a serem disponibilizados pelo *kernel*, sem contudo considerar as restrições de implementação. Como a idéia é reconstruir o *kernel* original através dos componentes, para que estes sejam testados, os requisitos originais devem ser mantidos.

Os artefatos produzidos nesta fase são modelos de casos de uso, de seqüência, de estados e de classes, usando a notação UML [17]. Em se tratando de sistemas de tempo-real, as restrições de tempo são consideradas, principalmente na especificação dos modelos de seqüência e de estados dos diferentes serviços do *kernel*. A utilização de uma ferramenta CASE (*Computer Aided Software Engineering*), como a MVCASE [1], facilita a especificação de requisitos.

5.2.2. Projeto orientado a componentes

Nesta etapa, refinam-se os modelos produzidos na etapa anterior, agora considerando os requisitos não-funcionais, como, por exemplo, a plataforma de hardware e software, a linguagem de implementação, entre outros.

A partir da análise de requisitos do novo *kernel*, realizada na etapa anterior, deve-se investigar quais componentes do repositório serão reutilizados. Para a reconstrução do *kernel* original, orientado a componentes, podem ser reutilizados todos os componentes produzidos na primeira fase do processo. Contudo, os componentes construídos na primeira fase podem não cobrir todos os serviços requeridos pelo novo *kernel*. Desta forma, pode ser necessário adicionar novos componentes para realizar serviços específicos do novo *kernel*.

O reúso dos componentes em diferentes projetos de *kernels* depende de quão genéricos eles são. Em suas primeiras versões, os componentes muitas vezes ainda não são

suficientemente genéricos, podendo inclusive ter algumas dependências em relação à arquitetura do *kernel* original. Para que os componentes produzidos possam se tornar mais genéricos, eles devem passar por um processo de refinamento, que exige do engenheiro de software um bom conhecimento do domínio do problema. Na abordagem proposta parte-se do princípio que através da reconstrução de diferentes *kernels* pode-se refinar os componentes até a obtenção de um grau de reúso satisfatório.

5.2.3. Implementação e testes

Nesta etapa o *kernel* é implementado conforme as especificações do projeto, definidas na etapa anterior. A utilização de uma ferramenta RAD (*Rapid Application Development*), como o C++ Builder [4], na qual os componentes possam ser disponibilizados em paletas, pode facilitar esta etapa do processo.

Depois de implementado com a reutilização de componentes e adição de código específico do novo *kernel*, este deve ser testado com as mesmas aplicações executadas pelo *kernel* original. Considerando que a abordagem CSK é aplicada a *kernels* de tempo-real, são realizados testes lógicos e temporais do tipo “caixa-preta”.

Os testes “caixa-preta”, embora não garantam formalmente que o novo *kernel* funcione exatamente como o antigo, servem para verificar se os requisitos previstos foram atendidos e o *overhead* gerado pela mudança de paradigma prejudicou o atendimento de suas restrições temporais. Desta forma, procura-se manter as mesmas funcionalidades do *kernel* original.

Caso ocorram erros, os quais podem ser dos componentes construídos ou de suas reutilizações, deve-se retornar às etapas anteriores para corrigi-los, gerando novas versões dos protótipos dos componentes e de suas aplicações.

5.3. Considerações sobre a abordagem CSK

A abordagem proposta não considera a adição de novos requisitos não-funcionais, ou seja, novos serviços, na reconstrução do *kernel*.

A produção de componentes genéricos depende do conhecimento do domínio de sistemas operacionais. Contudo, dada a maturidade deste domínio, a reengenharia de sistemas operacionais já existentes pode ser uma boa solução para recuperar e reutilizar todo o conhecimento de anos embutido no código destes sistemas.

Assim, mesmo não sendo suficientemente genéricos em suas primeiras versões, os componentes já possibilitam a realização de estudos com funcionalidades específicas de sistemas operacionais de tempo-real. Além disso, servem de partida para a generalização e padronização na obtenção de outras versões com maiores graus de reutilização.

6. ESTUDO DE CASO

Este capítulo apresenta um estudo de caso no qual a abordagem CSK é aplicada ao sistema operacional de tempo-real Virtuoso [26], um sistema operacional não orientado a objetos desenvolvido em linguagem C.

Virtuoso é um sistema operacional baseado em *microkernel* que oferece suporte a multiprogramação e comunicação entre tarefas e uma eficiente manipulação de interrupções, com mínima perda de tempo.

Virtuoso está disponível em duas versões: SP (*Single Processor*), para sistemas monoprocessados, e VSP (*Virtual Single Processor*), para sistemas multiprocessados. A versão utilizada neste estudo de caso é a SP, executada sobre o sistema operacional Windows XP com o auxílio de um emulador baseado na implementação de *threads* do ambiente C++ Builder. Após a estruturação, o novo *kernel* é executado sobre a mesma plataforma.

Como mostra a Figura 17, Virtuoso possui uma arquitetura baseada em quatro níveis, a saber: nível das tarefas ou *microkernel*, nível dos processos ou *nanokernel*, ISR1 e ISR0. Os níveis mais baixos são prioritários, podendo fazer a preempção de qualquer atividade de um nível mais alto.

Na nomenclatura do Virtuoso há diferenciação entre tarefas e processos segundo o nível onde são executados. Tarefas, geralmente escritas em C ou C++, são escalonadas pelo *microkernel* e possuem código portável a outras configurações de hardware. Processos, geralmente escritos em *assembly* e com um contexto reduzido, são escalonados pelo *nanokernel*, com um menor tempo de latência durante as trocas de contexto. Porém, seu código não é portável.

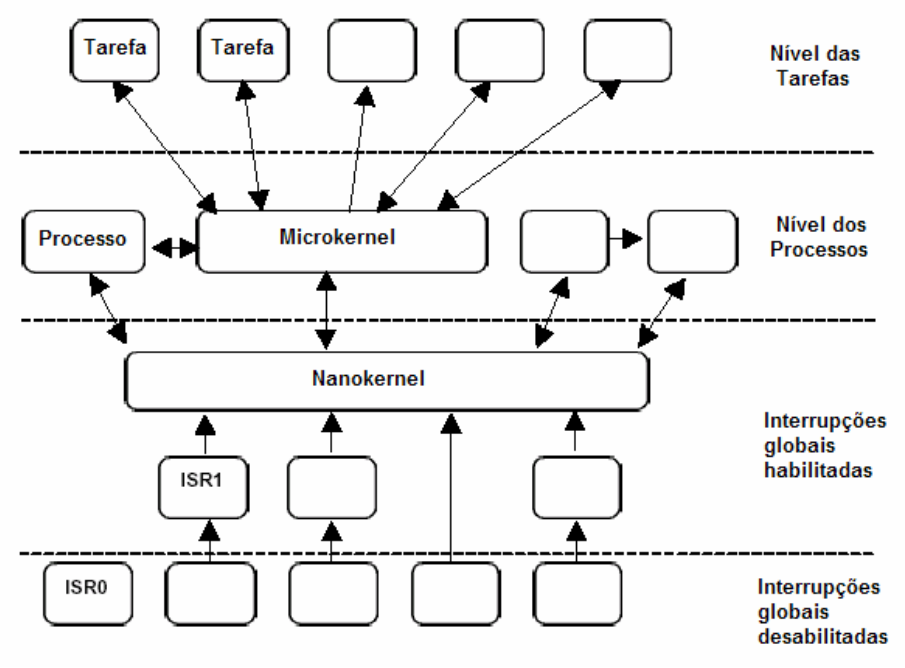


Figura 17 Arquitetura Virtuoso

Acima do *nanokernel* encontra-se o *microkernel*, responsável por gerenciar as tarefas que compõem a aplicação. É o *microkernel* quem realiza a multiprogramação preemptível, com escalonamento baseado em prioridades. O nível de prioridade associado a cada tarefa é baseado na importância desta para o sistema. Uma política é associar níveis de prioridade mais altos a tarefas que requerem respostas mais rápidas a eventos externos. Alternativamente, é possível atribuir níveis de prioridade mais altos a tarefas mais freqüentes.

Abaixo do *nanokernel* situam-se os níveis ISR (*Interrupt Service Routines*), onde são executados os processos manipuladores de interrupções. Eles são divididos em dois níveis: ISR0 e ISR1. Esta divisão diminui o tempo de latência gasto na manipulação das interrupções.

As interrupções manipuladas no nível ISR0 se originam diretamente dos dispositivos de hardware. Durante a manipulação destas interrupções, todas as demais interrupções globais são desabilitadas, enquanto que as interrupções nos níveis superiores são habilitadas. Por isso, recomenda-se o reconhecimento das interrupções no nível ISR0 e seu

processamento no nível ISR1, para a minimização do tempo em que as interrupções globais ficam desabilitadas.

O nível ISR0 é utilizado somente em processadores que permitem a inibição de interrupções e não é necessário em sistemas que suportam o tratamento de interrupções no próprio hardware, como o processador ADI SHARC, por exemplo. O nível ISR1 manipula a interrupção do mesmo modo que o ISR0, mas trabalha com as interrupções globais habilitadas e permite fazer chamadas de sistema para o nível ISR0.

As próximas seções apresentam cada fase abordagem CSK, com suas etapas, para a reestruturação do Kernel Virtuoso.

6.1. Construção dos componentes

Nesta primeira fase de aplicação da abordagem CSK, o *kernel* Virtuoso é dividido segundo suas funcionalidades e remodelado com componentes que implementam cada uma das partes. As ferramentas MVCASE [1], para modelagem dos componentes e do novo k*ernel*, e C++ Builder 6 [4], para implementação e reúso dos componentes, foram os principais mecanismos utilizados para apoiar o desenvolvimento da aplicação da abordagem CSK.

As próximas seções apresentam mais detalhadamente cada etapa desta reconstrução.

6.1.1. Identificação dos serviços e primitivas do Virtuoso

O *microkernel* Virtuoso oferece vários serviços às tarefas, representados por primitivas cujo nome se inicia por `KS_`. A partir da análise da documentação [26] e do código-fonte do *kernel*, tais primitivas foram divididas em seis categorias: gerência de tarefas,

comunicação, sincronização, temporizadores, gerência de memória e primitivas especiais. Tais categorias são apresentadas nas próximas seções.

6.1.1.1. Gerência de tarefas

Como as aplicações do Virtuoso são implementadas através de um conjunto de tarefas, são necessários serviços de criação e controle de execução destas tarefas. A Tabela 1 apresenta uma lista com os serviços relacionados à criação e controle de execução de tarefas, indicando suas respectivas funções.

Tabela 1 Primitivas de criação e controle de execução de tarefas

PRIMITIVA	FUNÇÃO
`KS_Start`	Inicia uma tarefa
`KS_Abort`	Aborta a execução de uma tarefa
`KS_Suspend`	Pausa a execução de uma tarefa
`KS_Resume`	Continua a execução de uma tarefa bloqueada ou pausada
`KS_Sleep`	Atrasa uma tarefa em *ticks* de tempo
`KS_Yield`	Entrega o processador a uma tarefa de mesma prioridade
`KS_SetEntry`	Muda o ponto de entrada de uma tarefa
`KS_TaskId`	Retorna o ID da tarefa
`KS_NodeId`	Retorna o ID do nó de execução da tarefa
`KS_SetPrio`	Modifica o nível de prioridade de uma tarefa
`KS_Aborted`	Instala ou remove uma função manipuladora (*handler*) que é executada quando a tarefa é abortada

A execução de uma tarefa inicia quando o escalonador determina que os recursos necessários estão disponíveis e ela é a tarefa com maior prioridade. Ela continuará executando até seu término, ou será interrompida caso tenha que esperar por um evento ou dado, ou pela disponibilidade de um recurso. Ela ainda pode ser interrompida por uma ISR.

A Figura 18 apresenta um modelo de transição entre os possíveis estados de uma tarefa, indicando a primitiva responsável por cada uma das transições.

Ao ser iniciada com a primitiva `KS_Start`, a tarefa é colocada no estado PRONTO. Ao ser escalonada para execução, passa ao estado EXECUTANDO. Sua execução pode ser abortada ou pausada, através das primitivas `KS_Abort` e `KS_Suspend`, respectivamente. Ou ainda, a tarefa pode entrar em MODO DE ESPERA, caso faça uma chamada cujo recurso, evento ou dado requisitado não esteja disponível. Ao final da espera pelo recurso, a tarefa pode tanto ser suspensa quanto ir para o estado PRONTO.

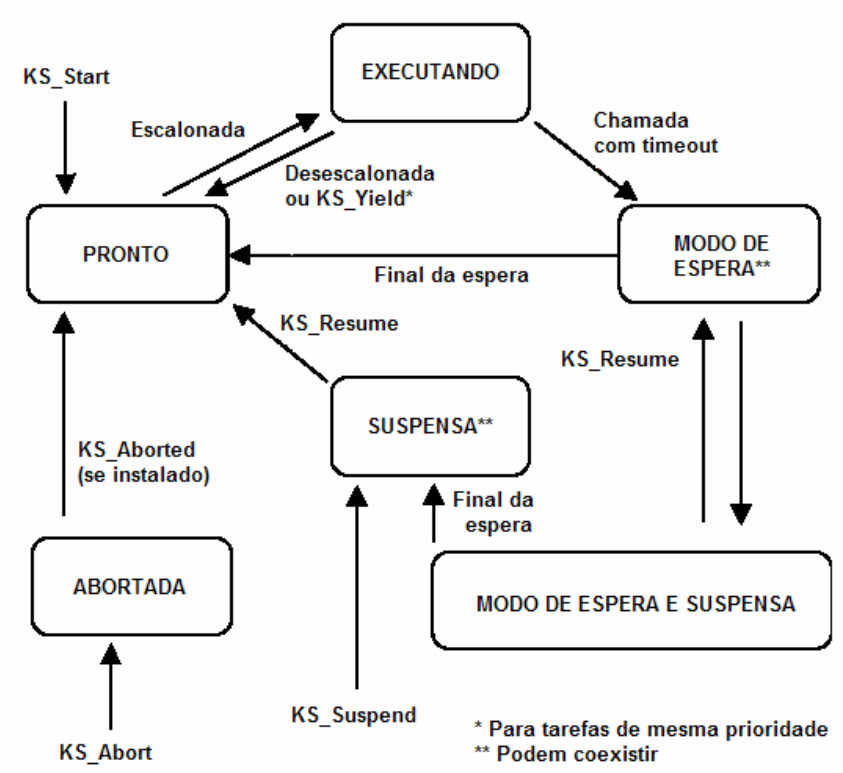


Figura 18 Estados de uma tarefa Virtuoso

Como mencionado, o escalonamento é baseado em prioridades. Virtuoso suporta 63 níveis de prioridade, que variam entre os valores inteiros zero e 62. Tais níveis podem ser modificados dinamicamente, através da primitiva `KS_SetPrio`, útil na prevenção do problema da inversão de prioridades.

Virtuoso suporta a reunião de tarefas em grupos. Uma ação aplicada a um grupo de tarefas é na verdade aplicada a cada uma das tarefas do grupo. As primitivas relacionadas à manipulação de grupos de tarefas são exibidas pela Tabela 2.

Tabela 2 Primitivas de controle de grupos de tarefas

PRIMITIVA	FUNÇÃO
`KS_StartG`	Inicia todas as tarefas de um grupo
`KS_SuspendG`	Suspende todas as tarefas de um grupo
`KS_ResumeG`	Continua a execução de todas as tarefas de um grupo
`KS_AbortG`	Aborta a execução de todas as tarefas de um grupo
`KS_JoinG`	Adiciona uma tarefa a um grupo
`KS_LeaveG`	Remove uma tarefa de um grupo

6.1.1.2. Comunicação

A comunicação entre tarefas no Virtuoso é feita através de *mailbox* ou filas FIFO. O *mailbox* é uma estrutura versátil e adequada à maioria das necessidades. Pode manipular grandes ou pequenas quantidades de dados, possui comunicação síncrona ou assíncrona e permite a identificação das tarefas remetente e destino.

A cada *mailbox* são vinculadas duas filas, uma para enviar e outra para receber mensagens. Na comunicação síncrona via *mailbox*, as tarefas remetente e destinatária permanecem bloqueadas até a mensagem ter sido copiada. Na comunicação assíncrona, a tarefa remetente é desbloqueada imediatamente após o envio.

Quando uma tarefa envia uma mensagem a um *mailbox*, somente o cabeçalho da mensagem é transmitido. Quando a tarefa destinatária requisita tal mensagem, recebe o corpo da mensagem diretamente do remetente.

FIFO é uma fila simples que manipula pequenas quantidades de dados (mensagens). Diferentemente de *mailboxes*, os itens de dados são transferidos ordenadamente (*First In First Out*) de forma assíncrona e anônima.

As filas FIFO do Virtuoso não possuem proprietários, o que significa que qualquer tarefa pode escrever ou ler as mensagens nelas escritas. Elas podem ser usadas para coleta de dados de uma porta de entrada e saída ou como portas de comunicação. A Figura 19 mostra o esquema de funcionamento de um FIFO.

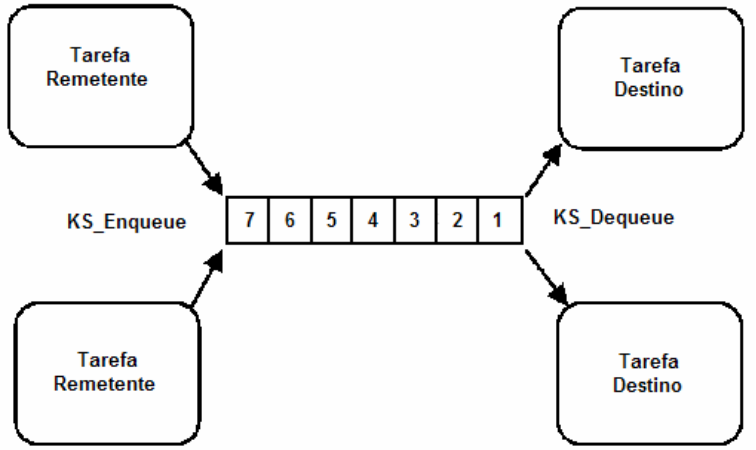


Figura 19 Esquema de funcionamento de uma fila FIFO no Virtuoso

As primitivas relativas à comunicação entre as tarefas do Virtuoso são mostradas pela Tabela 3.

Tabela 3 Primitivas de comunicação do Virtuoso

PRIMITIVA	FUNÇÃO
`KS_Send`	Envia uma mensagem a um *mailbox*
`KS_SendW`	Envia uma mensagem a um *mailbox* (bloqueante)
`KS_SendWT`	Envia uma mensagem a um *mailbox* (com *timeout*)
`KS_Receive`	Recebe uma mensagem de um *mailbox*
`KS_ReceiveW`	Recebe uma mensagem de um *mailbox* (bloqueante)
`KS_ReceiveWT`	Recebe uma mensagem de um *mailbox* (com *timeout*)
`KS_ReceiveData`	Coleta dados de uma mensagem
`KS_Enqueue`	Inclui uma entrada na fila FIFO especificado
`KS_EnqueueW`	Inclui uma entrada na fila FIFO especificado (bloqueante)
`KS_EnqueueWT`	Inclui uma entrada na fila FIFO especificada (com *timeout*)
`KS_Dequeue`	Lê uma entrada da fila FIFO especificada
`KS_DequeueW`	Lê uma entrada da fila FIFO especificada (bloqueante)
`KS_DequeueWT`	Lê uma entrada da fila FIFO especificada (com *timeout*)
`KS_PurgeQueue`	Limpa as entradas da fila FIFO especificada
`KS_InqQueue`	Retorna o número de entradas da fila FIFO especificada

6.1.1.3. Sincronização

Tarefas e manipuladores de interrupções (ISR) freqüentemente necessitam informar a outras tarefas a ocorrência de determinado evento. Esta informação pode ser transmitida no Virtuoso através de semáforos. A tarefa ou ISR termina sua parte do serviço e sinaliza um semáforo. Uma outra tarefa testa o mesmo semáforo para saber se o serviço já foi concluído.

As operações de sinalização e teste em semáforos são assíncronas: a tarefa que sinaliza não sabe se existe uma tarefa esperando pela sinalização do semáforo. Por outro lado, a tarefa que testa um semáforo ainda não sinalizado pode decidir se continua sua execução, utilizando `KS_Wait`, ou espera pela chegada do sinal, utilizando `KS_WaitW` ou `KS_WaitWT`.

Quando um semáforo é sinalizado, seu valor é incrementado, e quando testado, seu valor é decrementado. O valor de contagem pode ser reiniciado a qualquer momento, através de `KS_ResetSema`.

Quando há mais de uma tarefa esperando pela sinalização de um semáforo, elas são mantidas em uma fila com prioridades. A cada sinalização do semáforo, a tarefa de maior prioridade é liberada para utilizar o recurso.

As operações sobre um semáforo e as primitivas utilizadas em cada caso são ilustradas pela Figura 20.

Recursos (*resources*) também são chamados semáforos de exclusão mútua. Um recurso consiste de uma variação de semáforo responsável por proteger um recurso do sistema, que pode ser um dispositivo de entrada e saída, bloco de memória, estrutura de dados compartilhada, fila de mensagens, entre outros.

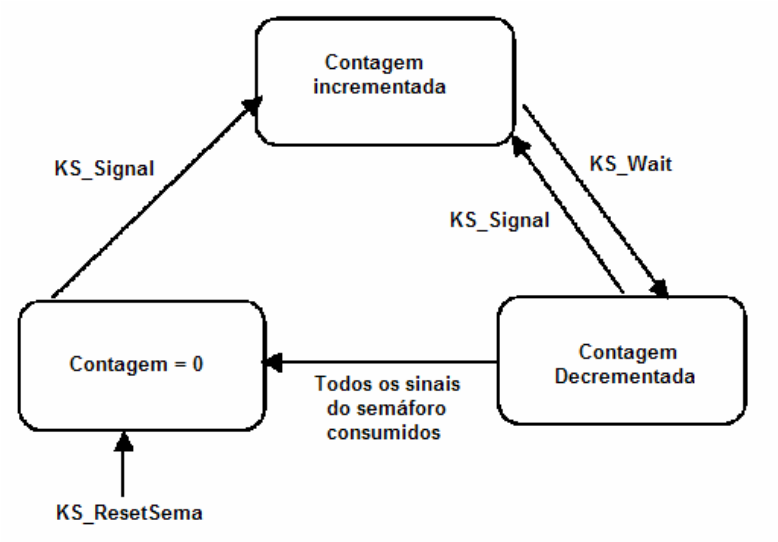


Figura 20 Operações de um semáforo Virtuoso

Toda tarefa que necessite acessar um recurso protegido, deve travá-lo, através de `KS_Lock`, `KS_LockW` ou `KS_LockWT`. Isto só é conseguido se nenhuma outra tarefa estiver utilizando o recurso. Ao terminar seu trabalho, a tarefa destrava o recurso, através de `KS_Unlock`, liberando-o para outros acessos. Quando utilizada uma primitiva bloqueante para o travamento, a tarefa que espera por um recurso já travado é mantida em uma fila de prioridades.

As primitivas de manipulação de semáforos e recursos do Virtuoso são exibidas pela Tabela 4.

Tabela 4 Primitivas de sincronização do Virtuoso

PRIMITIVA	FUNÇÃO
`KS_Signal`	Sinaliza o semáforo especificado
`KS_Wait`	Testa o semáforo especificado
`KS_WaitW`	Testa o semáforo especificado (bloqueante)
`KS_WaitWT`	Testa o semáforo especificado (bloqueante com *timeout*)
`KS_ResetSema`	Reinicia a contagem do semáforo especificado
`KS_InqSema`	Retorna o valor de contagem do semáforo especificado
`KS_Lock`	Trava o recurso especificado
`KS_LockW`	Trava o recurso especificado (bloqueante)
`KS_LockWT`	Trava o recurso especificado (bloqueante com *timeout*)
`KS_Unlock`	Destrava o recurso especificado

6.1.1.4. Gerência de tempo

Virtuoso suporta temporizadores (*timers*) de baixa resolução, implementados via software. Os temporizadores podem ser criados para a geração de um evento em um momento específico ou ciclicamente. Além disso, os temporizadores controlam os *timeouts* das tarefas que estão à espera de determinado evento.

As primitivas relacionadas à utilização de temporizadores no Virtuoso são exibidas pela Tabela 5.

Tabela 5 Primitivas de gerência de tempo do Virtuoso

PRIMITIVA	FUNÇÃO
`KS_AllocTimer`	Desaloca um temporizador
`KS_DeallocTimer`	Aloca um temporizador
`KS_StartTimer`	Inicia um temporizador
`KS_StopTimer`	Pára um temporizador
`KS_Elapse`	Retorna os *ticks* passados desde a última chamada à função
`KS_LowTimer`	Lê o valor do *tick* atual de um temporizador
`KS_RestartTimer`	Reinicia um temporizador

6.1.1.5. Gerência de memória

No Virtuoso, a memória disponível no sistema é dividida em um ou mais mapas de memória, que são subdivididos em blocos lógicos de tamanho único. Blocos de memória de um mapa só podem ser alocados e liberados localmente.

Quando a aplicação faz uma requisição por um bloco de memória, através de `KS_Alloc`, `KS_AllocW` ou `KS_AllocWT`, um endereço de um bloco lógico do mapa especificado é retornado. Se não houver bloco disponível, `NULL` é retornado.

As primitivas de gerência de memória do Virtuoso são exibidas pela Tabela 6.

Tabela 6 Primitivas de gerência de memória do Virtuoso

PRIMITIVA	FUNÇÃO
`KS_Alloc`	Aloca bloco de um mapa de memória
`KS_AllocW`	Aloca bloco de um mapa de memória (bloqueante)
`KS_AllocWT`	Aloca bloco de um mapa de memória (com *timeout*)
`KS_Dealloc`	Desaloca bloco de um mapa de memória
`KS_InqMap`	Informa blocos em uso no mapa de memória

6.1.1.6. Primitivas especiais

Virtuoso ainda oferece algumas primitivas com funções especiais, as quais são mostradas pela Tabela 7.

Tabela 7 Primitivas especiais do Virtuoso

PRIMITIVA	FUNÇÃO
`KS_Nop`	Não executa instrução alguma (usada em *benchmarks*)
`KS_User`	Define função a ser executada com as interrupções desabilitadas

6.1.1.7. Conclusão

A partir das categorias indicadas pelas seções anteriores, foram identificados sete possíveis serviços oferecidos pelo Virtuoso. A Tabela 8 apresenta todos estes serviços, incluindo o serviço global *Main*, com suas primitivas associadas.

Como já foram apresentadas nas seções anteriores, as primitivas do tipo `KS_` foram omitidas desta relação. Estas funções são responsáveis por preparar parâmetros e chamar a função `K_` correspondente dos demais componentes, a qual realmente implementa o serviço chamado.

As primitivas da categoria especial, apresentadas na seção anterior, foram associadas ao serviço global *Main*, já que não possuem coesão para serem implementadas através de componentes.

Tabela 8 Serviços e primitivas identificados no Virtuoso

Main	Semaphore	Timer	Task
K_nop	K_swaittimo	K_alloc_timer	K_taskop
K_user	K_signal	K_dealloc_timer	K_groupop
K_event	K_signalm	K_start_timer	K_set_prio
kernel_init	K_sreset	K_stop_timer	K_yield
kernel	K_sresetm	K_elapse	set_bit
kernel_call_entry	K_swait	enlist_timer	res_bit
kernel_sigevent	K_swaitm	delist_timer	start_task
kernel_idle	unwaitm	ticker	abort_task
init_event	sigsema	enlist_timeout	init_task
	init_sema	delist_timeout	
		init_ticks	
Queue	**Memory**	**Resource**	**Mailbox**
K_queuetimo	K_alloctimo	K_locktimo	K_recvdata
K_enqueue	K_alloc	K_lock	K_messtimo
K_dequeue	K_dealloc	K_unlock	K_send
K_purgequeue	init_map	init_res	K_recv
init_queue			init_mail

6.1.2. Organização das variáveis do Virtuoso

Após a identificação dos serviços do Virtuoso, com suas primitivas, foram analisadas suas variáveis. A Tabela 9 exibe todas as variáveis associadas a cada serviço. As variáveis globais identificadas são `TaskPtrs`, `t_curr`, `t_next` e `action`.

Tabela 9 Serviços e respectivas variáveis associadas

SERVIÇO	VARIÁVEIS
Main	`TaskPtrs, t_curr, t_next, action, syserror, virtuosofun, IrqWaits,`
Semaphore	`SEM_ASIZE, SEM_ARRAY, WTM_ASIZE, WTM_ARRAY, Wfree`
Timer	`TIM_ASIZE, TIM_ARRAY, KCP_ASIZE, KCP_ARRAY, lowtime, Thead, Ttail, Tfree, Afree`
Task	`ntasks, tasklist`
Queue	`QUE_ASIZE, QUE_ARRAY`
Memory	`MAP_ASIZE, MAP_ARRAY`
Resource	`RES_ASIZE, RES_ARRAY`
Mailbox	`MBX_ASIZE, MBX_ARRAY`

6.1.3. Identificação das relações de “quem chama quem”

Analisando as primitivas e variáveis de cada serviço identificado e conforme as heurísticas fornecidas pela abordagem CSK, construiu-se um grafo dirigido que representa as relações de conexão entre os serviços do Virtuoso.

O grafo, exibido pela Figura 21, apresenta vários ciclos, formados pelas ligações de duplo sentido entre o serviço `Main` e os demais serviços, indicando uma divisão inadequada dos serviços do *kernel*.

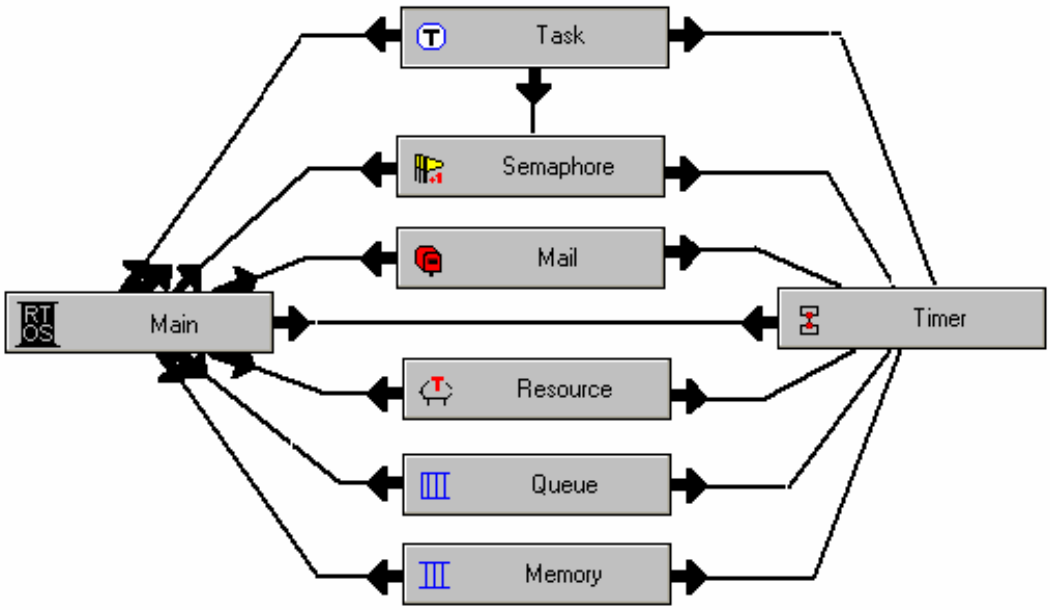


Figura 21 Grafo cíclico inicial

Dado que o serviço `Main` precisa chamar primitivas de todos os demais serviços, tais conexões não podem ser removidas. Mas, analisando a partir dos demais serviços, observou-se que todos dependem da primitiva `kernel_call_entry` e das variáveis globais do serviço `Main`.

Como a primitiva `kernel_call_entry` é acessada exclusivamente através das primitivas `KS` de cada serviço, estas podem ser transferidas para o serviço `Main`, já que

correspondem à API oferecida pelo kernel às aplicações e apenas preparam parâmetros para a execução da primitiva `K`, a qual realmente implementa o serviço.

A dependência das variáveis globais é um problema de solução mais complexa. Uma solução simples, mas deselegante, seria transferi-las ao serviço `Timer`, de onde poderiam ser acessadas por todos os serviços.

Analisando mais a fundo o funcionamento do *kernel*, nota-se que a execução dos serviços depende apenas da variável global `t_curr`. As demais, `t_next`, `TaskPtrs` e `action`, são apenas setadas ao final da execução de cada serviço para informar ao kernel o estado de execução.

Desta forma, a solução para o problema foi manter a variável `t_curr` no serviço `Main` e passá-la como parâmetro a todas as chamadas dos demais serviços. As variáveis `t_next`, `TaskPtrs` e `action` foram removidas do esquema. Para substituir o código de atualização destas variáveis, em cada serviço foi criado uma nova função, a qual se comportará como um evento nos futuros componentes.

Além disso, o serviço `Task` não pôde ser implementado através de um componente, pois possui dependências em relação ao serviço `Semaphore`. Assim, as variáveis e primitivas associadas inicialmente ao serviço `Task` foram transferidas para o serviço `Main`. O grafo resultante destas modificações é apresentado pela Figura 22, onde é possível observar a ausência de ciclos.

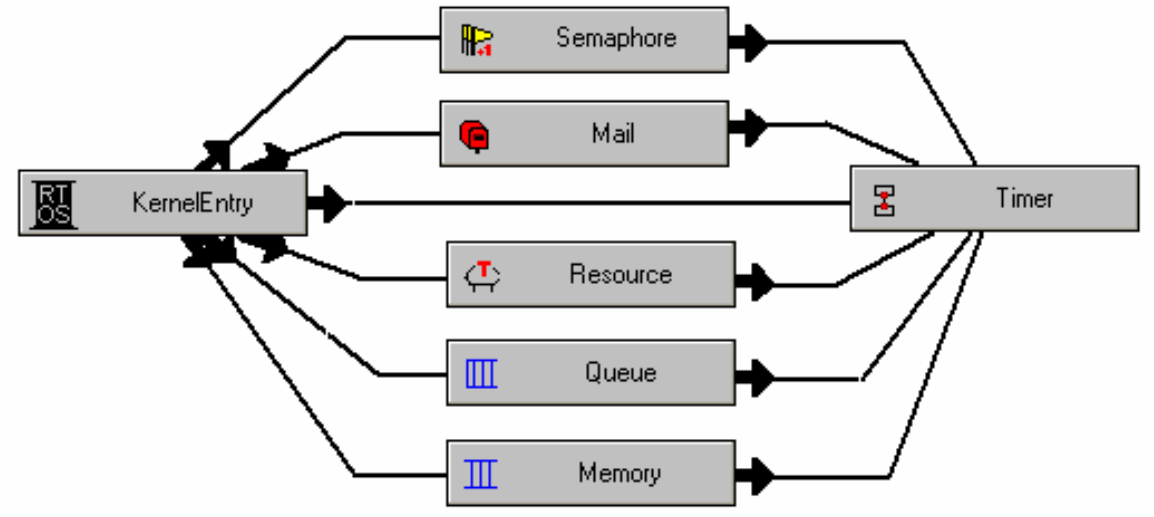


Figura 22 Grafo acíclico resultante

O serviço global `Main` não corresponde a um componente do domínio de sistemas operacionais de tempo-real, por não implementar uma funcionalidade específica do *kernel*. Na verdade, suas primitivas implementam o código de suporte para a execução do *kernel*. Apesar disso, este serviço também será implementado como um componente. Seu nome foi alterado para `KernelEntry`, pois ele contém a API oferecida pelo *kernel* à camada de aplicações.

Em princípio, o serviço `KernelEntry` será responsável por complementar a implementação quando o Virtuoso for reconstruído a partir dos componentes obtidos nesta primeira fase da abordagem CSK.

6.1.4. Projeto dos componentes

Nesta etapa, inicialmente são criadas classes abstratas para cada um dos serviços representados pelos vértices do grafo acíclico produzido na etapa anterior. No caso do Virtuoso, estas classes foram nomeadas `AbstractKernelEntry`, `AbstractSemaphore`, `AbstractTimer`, `AbstractQueue`, `AbstractMailbox`, `AbstractMemory` e `AbstractResource`.

Em seguida, foram criadas as classes concretas – derivadas das classes abstratas – que completam a implementação dos serviços do *kernel*. Nas classes concretas, cada primitiva associada ao serviço dá origem a um método e cada variável dá origem a um atributo.

Conforme a arquitetura proposta pela abordagem CSK, cada classe concreta disponibiliza seus métodos através de duas interfaces, conforme o modelo de classes concretas mostrado na Figura 23 e na Figura 24, onde as classes `KernelEntry` e `Timer` fazem as conexões entre estes modelos.

Na interface `RunTimeKernelEntry`, provida pela classe `KernelEntry`, estão todas as primitivas `KS`, as quais correspondem aos serviços oferecidos pelo *kernel* às aplicações nele executadas.

Na Figura 23 encontram-se as classes concretas para os serviços `Semaphore`, `Queue` e `Resource`. Na Figura 24 encontram-se as classes concretas para os serviços `Memory` e `Mailbox`, com suas respectivas interfaces.

Todas as classes do modelo utilizam a interface `RunTimeTimer`, que implementa as características de tempo-real necessárias para a execução dos demais serviços, além de fornecer os métodos de acesso às variáveis globais, neste caso específico.

A interface de tempo de *design* de cada classe contém os eventos criados em virtude da remoção das variáveis globais `TaskPtrs`, `t_next` e `action`, realizada na etapa anterior.

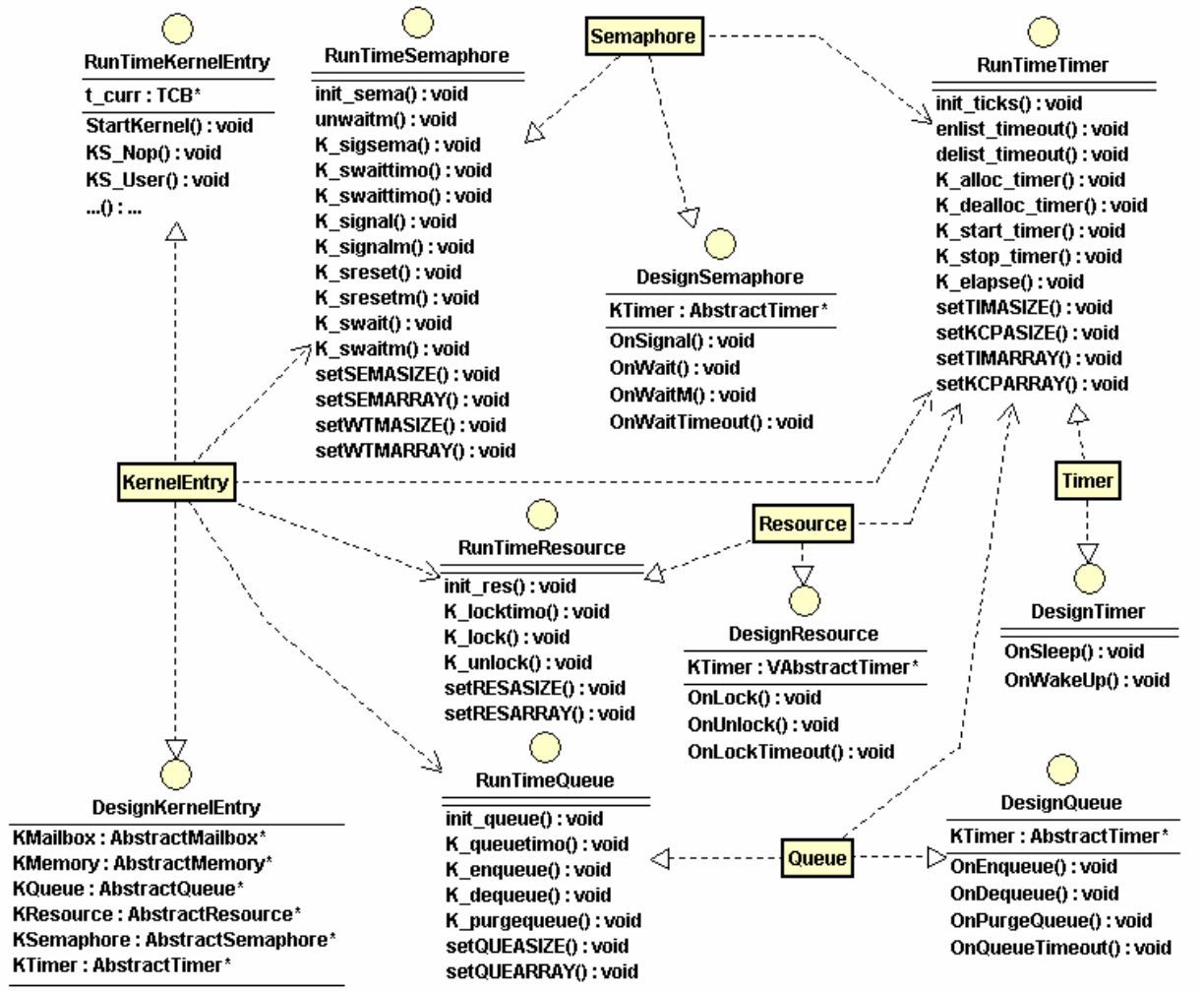


Figura 23 Modelo de classes concretas do Virtuoso – Parte 1

6.1.5. Implementação dos componentes

Para finalizar a primeira fase da abordagem CSK, foram implementados os componentes do *kernel* Virtuoso, com base nos modelos de classes concretas, produzidos na etapa anterior.

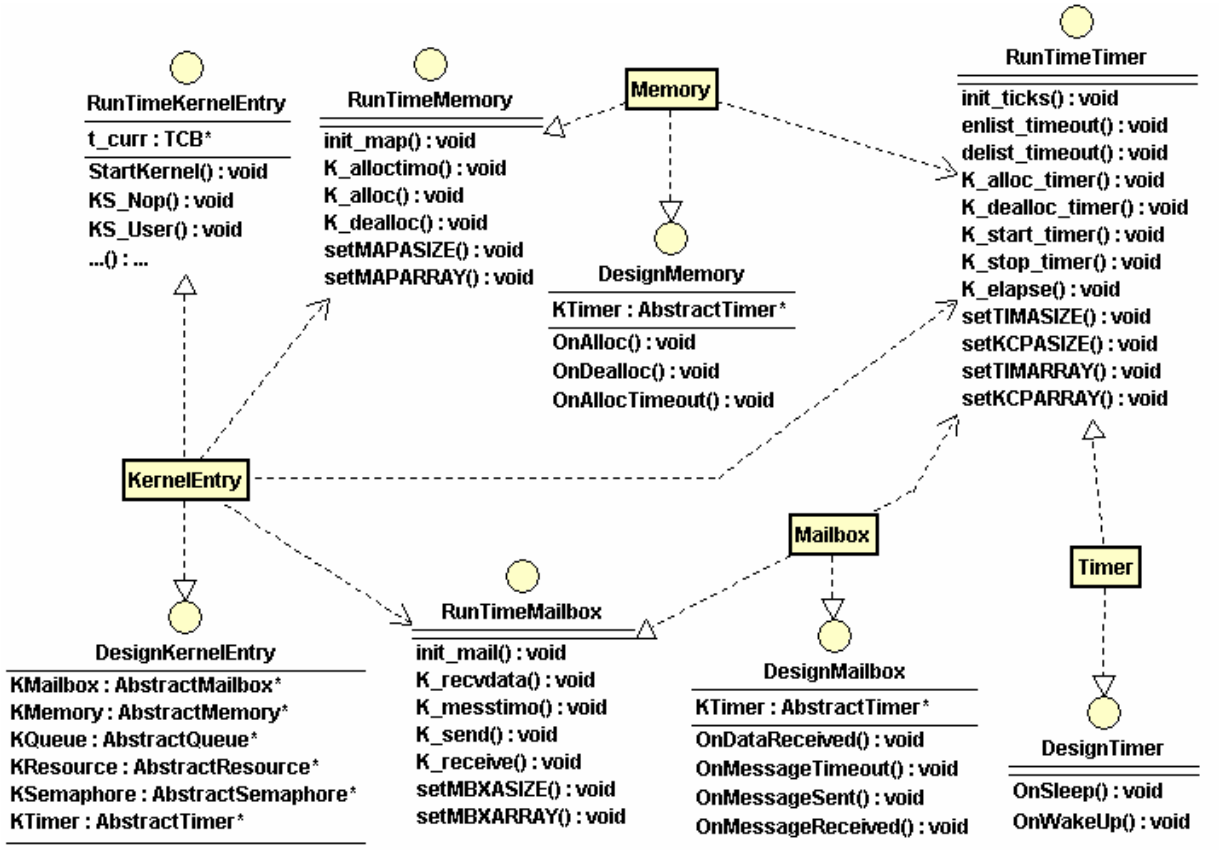


Figura 24 Modelo de classes concretas do Virtuoso – Parte 2

A implementação dos componentes foi baseada nos padrões do ambiente C++ Builder. Os atributos e métodos da interface de tempo de *design* de cada componente foram disponibilizados através do ObjectInspector, um gerenciador de interfaces em tempo de *design*. Os componentes foram disponibilizados na paleta de componentes, na categoria KernelDev.

Para que se tenha uma idéia do projeto interno de cada classe concreta que implementa o componente, a Figura 25 mostra o componente `Semaphore` (a) e o projeto interno de sua classe concreta (b), com suas interfaces `DesignSemaphore` e `RunTimeSemaphore`.

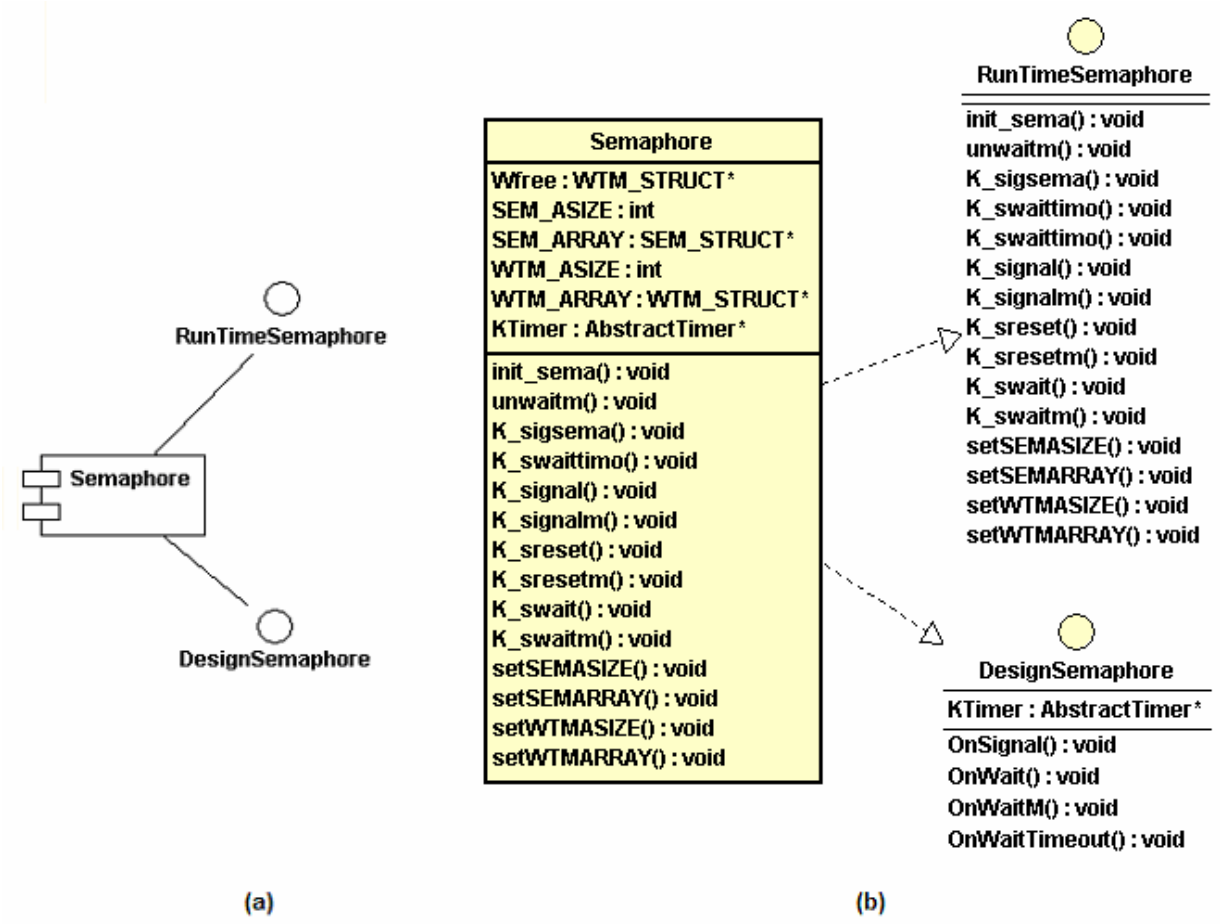


Figura 25 (a) Componente `Semaphore` e (b) Projeto interno de sua classe concreta

Em seguida, os componentes são refinados com mais detalhes de implementação. A Figura 26 mostra, por exemplo, detalhes do código do componente `Semaphore`. Em (1) tem-se o atributo que armazena o ponteiro para o *plug-in* com o componente `Timer`. Em (2) encontram-se as funções de acesso aos atributos da classe. Em (3) tem-se a propriedade disponível em tempo de design para o *plug-in* com o componente `Timer`.

6.2. Reconstrução do *kernel*

Nsta segunda fase da abordagem CSK, faz-se a reconstrução do *kernel* Virtuoso utilizando os componentes produzidos. As próximas seções apresentam cada etapa realizada na estruturação do Virtuoso.

```
class PACKAGE VSemaphore : public AbstractSemaphore
{
    private:
        AbstractTimer* FKTimer;   (1)
        int SEM_ASIZE;
        int WTM_ASIZE;
        SEM_STRUCT* SEM_ARRAY;
        WTM_STRUCT* WTM_ARRAY;
        WTM_STRUCT *Wfree;

    public:
        __fastcall VSemaphore(TComponent* Owner);
        ...
        void setSEMASIZE(int newSEMASIZE) {SEM_ASIZE = newSEMASIZE;}
        void setWTMASIZE(int newWTMASIZE) {WTM_ASIZE = newWTMASIZE;}
        void setSEMARRAY (SEM_STRUCT* newSEMARRAY) {SEM_ARRAY = newSEMARRAY;}
        void setWTMARRAY (WTM_STRUCT* newWTMARRAY) {WTM_ARRAY = newWTMARRAY;}
        void init_sema ();
        void unwaitm (_S1ARG *Args);
        void sigsema (K_SEMA sema);
        void K_swaittimo (_S1ARG *Args);
        void K_signal (_S1ARG *Args);
        void K_signalm (_S1ARG *Args);
        void K_sreset (_S1ARG *Args);
        void K_sresetm (_S1ARG *Args);
        void K_swait (_S1ARG *Args);
        void K_swaitm (_S1ARG *Args);

    __published:
        __property AbstractTimer* KTimer = {read=FKTimer, write=FKTimer};
};
```

(2)

(3)

Figura 26 Parte do código do componente `Semaphore`

6.2.1. Análise dos requisitos do Virtuoso

Esta etapa se torna facilitada considerando que grande parte da análise já foi realizada na fase de construção dos componentes. No entanto, é nesta etapa que podem ser

introduzidos novos requisitos caso se deseje incluir novos serviços no novo *kernel*. No caso do Virtuoso, foram mantidos os mesmos serviços do *kernel* original.

6.2.2. Projeto orientado a componentes do Virtuoso

O reúso dos componentes produzidos reimplementa grande parte dos serviços do *kernel*. O novo *kernel* foi executado com a adição de outros componentes ao projeto, além dos construídos na primeira fase da CSK, para o fornecimento de código de iniciação do *kernel*.

A Figura 27 mostra o modelo de componentes do novo *kernel*, no qual podem ser vistos sombreados os novos componentes adicionados: `VParameters`, `VStart` e `VApplication`, que implementam os serviços de iniciação de parâmetros e variáveis do *kernel* e o código das tarefas que compõem a aplicação. Tais componentes acessam as primitivas do *kernel* através da interface fornecida pelo componente `KernelEntry`.

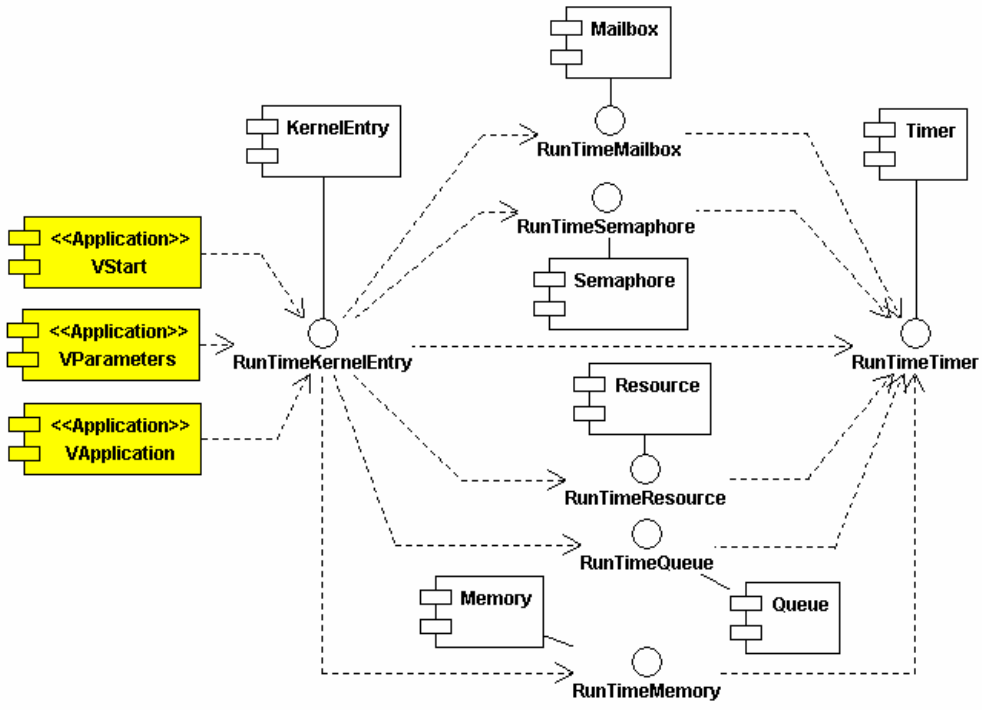


Figura 27 Modelo de componentes do Virtuoso

6.2.3. Implementação e testes

Nesta etapa, utilizando o C++ Builder, foi feita a implementação do novo *kernel*. A Figura 28 exibe parte da tela do ambiente de desenvolvimento, incluindo a paleta de componentes e a instanciação dos mesmos.

Durante os testes "caixa-preta" com diferentes aplicações que funcionavam com o a implementação original, foram encontrados erros relacionados com a fase da estruturação do *kernel* e com os componentes construídos na primeira fase. Depois de identificada a origem dos erros, retornou-se às etapas anteriores até a obtenção de uma versão correta do protótipo do *kernel*.

Após as correções, novos testes foram realizados com as mesmas aplicações, incluindo chamadas a todos os serviços implementados pelo *kernel*, com todos eles funcionando corretamente.

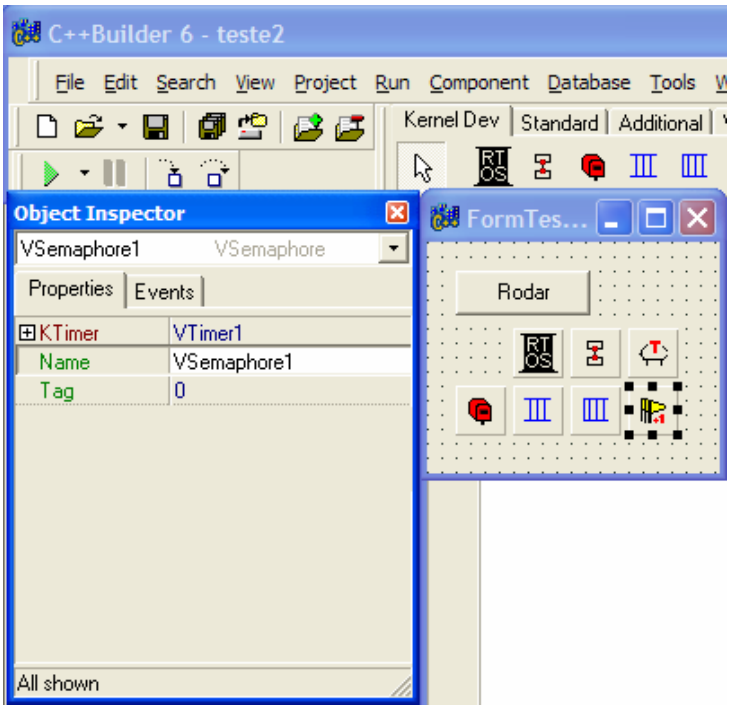


Figura 28 Paleta e instanciação dos componentes implementados

6.3. Considerações de desempenho

Após a comprovação de funcionamento lógico da nova implementação, submeteu-se os componentes do novo *kernel* a testes temporais. Uma nova aplicação Virtuoso foi desenvolvida para o teste do tempo de acesso a uma primitiva do novo *kernel*, utilizando a primitiva `KS_Nop`, a qual não realiza operação alguma e existe exatamente para este propósito.

A Tabela 10 compara os tempos médios de acesso a uma primitiva, em nanossegundos, observados tanto para a implementação original do *kernel*, quanto para sua versão orientada a componentes construída pela CSK.

Tabela 10 Tempo médio de chamada a uma primitiva

Implementação	Tempo Mínimo	Tempo Médio	Tempo Máximo
Original	111 ns	115 ns	121 ns
CSK	142 ns	147 ns	153 ns

Obsevando o tempo médio obtido pelo novo kernel e comparando-o ao tempo médio observado pela implementação original, conclui-se que tal transformação produziu um *overhead* de 28%. Um *overhead* entre 20% e 30% já era esperado devido à conversão do kernel para o paradigma orientado a objetos.

Apesar de esperado, não se pode concluir que a conversão seja responsável por todo o *overhead* gerado, havendo grande possibilidade de diminuição deste overhead através de otimizações no código dos componentes.

Um *overhead* de apenas 28% não compromete as restrições de tempo-real do *kernel* na maioria dos casos, já que em um sistema de tempo-real, apesar da escasses de recursos dos dispositivos embarcados, é rara a existência de várias tarefas de tempo-real.

A possibilidade de aplicação da abordagem CSK a sistemas operacionais utilizados em sistemas de tempo-real mais complexos depende de viabilidade de redução deste *overhead*.

Desta forma, baseando-se nos números observados, conclui-se que a abordagem CSK é realmente viável, o que encoraja a sua aplicação a outros sistemas operacionais de tempo-real.

7. TRABALHOS CORRELATOS

A necessidade de reúso sempre existiu na área de sistemas operacionais. São minoria os casos de sistemas operacionais totalmente implementados sem reutilização de código de outros sistemas. Isto se deve ao alto custo de desenvolvimento de um sistema operacional completo.

Por exemplo, VINO [25], um sistema operacional otimizado para a implementação de sistemas gerenciadores de bancos de dados, aproveitou o mecanismo de memória virtual de baixo nível do NetBSD. Esta forma de "pseudo-reúso" geralmente depende de adaptações no código aproveitado.

A partir desta necessidade, alguns trabalhos focados em *kernels* extensíveis foram desenvolvidos. Entre eles estão SPIN [3], Choices [4] e Exokernel [10]. SPIN oferece um núcleo de funcionalidades básicas, permitindo às aplicações modificar a interface e a implementação do sistema operacional. Choices oferece um *kernel* extensível, também baseado num conjunto fixo de serviços. Exokernel é uma arquitetura que permite que aplicações personalizem o sistema operacional conforme suas necessidades.

Outros trabalhos foram focados em *kernels* orientados a componentes. O principal exemplo é OSKit [12], que consiste de um conjunto completo de componentes que implementam funcionalidades típicas de um sistema operacional e foram desenvolvidos especificamente para o reaproveitamento em outros sistemas operacionais. Apesar de seus componentes constituírem um sistema operacional completo e funcional, o OSKit não almeja ser, por si só, um sistema operacional útil.

Um exemplo mais recente, THINK [11], propõe um *framework* que, aliado a uma biblioteca de componentes próprios, chamada KORTEX, permite o desenvolvimento de *kernels* para sistemas operacionais.

A abordagem CSK se diferencia no fato de propiciar a criação de componentes a partir do conhecimento embutido nos sistemas operacionais legados e sua reutilização no desenvolvimento de novos trabalhos de pesquisa com *kernels*.

Portanto, a abordagem CSK é mais poderosa ao proporcionar uma maior flexibilidade ao desenvolvedor de sistemas operacionais, como acontece em SPIN, Exokernel e Choices, além de permitir a escolha da implementação mais adequada para o serviço desejado através de seu conhecimento prévio de outros sistemas operacionais, ao invés de simplesmente oferecer uma implementação única para cada serviço, como fazem OSKit e THINK.

8. CONCLUSÃO

A criação de um novo sistema operacional consome muitos recursos de desenvolvimento. Estudos focados em um determinado serviço do *kernel* requerem todo o aparato um sistema operacional completo para que ele seja implementado e testado. A reestruturação de um *kernel* em componentes contribui na solução deste problema, pois reduz o acoplamento entre estes serviços, permitindo que estudos específicos em um determinado serviço sejam realizados independentemente dos demais.

Considerando os problemas descritos acima, foi apresentada neste trabalho a abordagem CSK para a estruturação de um *kernel* de tempo-real em componentes de software. A abordagem utiliza componentes para implementar separadamente cada serviço oferecido do *kernel*, reaproveitando o conhecimento embutido em seu código legado.

Como estudo de caso, a abordagem CSK foi aplicada ao *kernel* Virtuoso [26] com sucesso. Desta aplicação resultaram seis componentes, os quais implementam serviços de comunicação, sincronização, escalonamento de tarefas, gerência de tempo e gerência de memória.

Em seguida, o *kernel* Virtuoso foi reconstruído a partir destes componentes. A nova implementação passou por testes do tipo “caixa-preta”, nos âmbitos lógico e temporal, utilizando algumas aplicações que funcionavam com o *kernel* original. Os testes lógicos mostraram que os mesmos serviços do *kernel* original foram reimplementados corretamente pelo novo *kernel*. Os testes de desempenho mostraram a aplicabilidade da abordagem CSK a *kernels* de tempo-real sem substanciais prejuízos aos seus requisitos temporais.

8.1. Contribuições

Visando o atendimento dos objetivos propostos, o projeto trouxe as seguintes contribuições:

a) Criação da abordagem CSK para a reutilização do código legado de sistemas operacionais, facilitando a manutenção e atualização do *kernel* e incentivando pesquisas específicas com os serviços do *kernel*;
b) Desenvolvimento de um conjunto de componentes que reimplementam os serviços oferecidos pelo *kernel* do sistema operacional de tempo-real Virtuoso. Tais componentes podem ser reutilizados em projetos de novos *kernels* de tempo-real, com pequenas dependências em relação ao Virtuoso;
c) Evidência da possibilidade de utilização do paradigma orientado a componentes na implementação de sistemas operacionais de tempo-real.

8.2. Trabalhos futuros

Embora a abordagem CSK tenha atingido seus objetivos, podem ser vislumbrados alguns trabalhos futuros. Algumas sugestões são:

a) Inclusão de uma etapa na abordagem CSK para a realização de um melhor refinamento nos componentes construídos, visando a generalização e padronização dos componentes para serem reutilizados por diferentes sistemas operacionais, além do Virtuoso;
b) Integração do *kernel* do Virtuoso orientado a componentes produzidos no estudo de caso com o TEV (*Teaching Environment for Virtuoso*) [15], ambiente visual para desenvolvimento de aplicações para o Virtuoso;

c) Desenvolvimento de componentes para a implementação de outros serviços do domínio de sistemas operacionais de tempo-real, o que pode ser obtido com a aplicação da abordagem CSK a outros *kernels* já existentes.

8.3. Considerações finais

Através da reutilização dos componentes produzidos, é eximida do pesquisador a necessidade de desenvolvimento de código de suporte em pesquisas específicas sobre determinada funcionalidade do sistema operacional. Este código de suporte é fornecido pelos demais componentes, o que reduz esforços, tempo e custos de desenvolvimento, além de facilitar a manutenção e atualização dos serviços implementados.

Embora ainda não sejam suficientemente genéricos para serem reutilizados na construção de qualquer *kernel*, estes componentes constituem uma primeira versão para o domínio de *kernels* de tempo-real. Acredita-se que, com o seu refinamento, podem ser obtidos componentes mais genéricos, com maiores possibilidades de reúso.

Apesar da abordagem CSK tratar apenas da reconstrução de *kernels* já existentes, com algumas adaptações suas diretrizes podem servir como parâmetro na construção de novos *kernels* orientados a componentes.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] ALMEIDA, E. S. et. al. MVCase: An Integrating Technologies Tool for Distributed Component-Based Software Development. In: APNOMS'2002, The Asia-Pacific Network Operations and Management Symposium, Poster Session. **Proceedings** of IEEE.

[2] AXFORD, T. **Concurrent Programming**: Fundamental Techniques for Real Time and Parallel Software Design. Chichester: Ed. John Wiley & Sons Inc., 1989.

[3] BERSHAD, B. N. et. al. Extensibility, safety and performance in the SPIN operating system. In: SOSP'95, pp 267–283.

[4] BORLAND SOFTWARE CORPORATION. **C++ Builder Personal**. URL: http://www.borland.com/cbuilder/personal. Consultado em 10/05/2003.

[5] CAMPBELL, R. et. al. Designing and implementing Choices: an object-oriented system in C++. In: Communications of the ACM, Vol.36, No. 9, Setembro de 1993.

[6] CATALYSIS.ORG. **CATALYSIS:** Enterprise Components With UML. URL: <http://kinetium.com/catalysis-org/>. Consultado em 13/04/2003.

[7] CHENG, S.; STANKOVIC, J. Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey. IBM Watson Research Center, University of Massachusetts, 1987.

[8] DIJKSTRA, E. W. Co-operating Sequential Processes. In: Programming Languages, Genuys, London: Academic Press, 1965.

[9] D'SOUZA, D.; WILLS, A. C. **Objects, Components and Frameworks with UML:** The Catalysis Approach. Reading, Massachusetts: Ed. Addison Wesley, 1998.

[10] ENGLER, D.R.; KAASHOEK, M.F.; O'TOOLE, J.W. Exokernel: An operating system architecture for application level resource management. In: SOSP'95, pp 251–266.

[11] FASSINO, J.-P. et. al. THINK: A software framework for component-based operating system kernels. In: USENIX Annual Technical Conference, Monterey (EUA), Junho de 2002.

[12] FORD, B. et. al. The Flux OSKit: A substrate for kernel and language research. In SOSP'97, pp 38–51.

[13] INSTITUCT OF ELECTRICAL AND ELETRONICS ENGINEERS. **IEEE Homepage.** URL: http://www.ieee.org. Consultado em 17/05/2003.

[14] LABROSSE, J. J. Designing with Real Time Kernels. In: SysComms'98.

[15] MORÓN, C. E.; RIBEIRO, J. R. P.; SILVA, N. C. da. A Teaching Environment for the Development of Parallel Real Time Programs. In: Frontiers in Education'98, 1., 1998, Tempe, Arizona - EUA. EP Innovations, 1998.

[16] OBJECT MANAGEMENT GROUP INC. **OMG Homepage**. URL: http://www.omg.org. Consultado em: 18/05/2003.

[17] OBJECT MANAGEMENT GROUP INC. **UML™ Resource Page**. URL: http://www.omg.org/uml. Consultado em: 18/05/2003.

[18] PARRISH, A.; DIXON, B.; CORDES, D. A Conceptual Foundation for Component-Based Software Deployment. Journal of Systems and Software, 2001.

[19] PARRISH, A.; DIXON, B.; HALE D. Component Based Software Engineering: A Broad Based Model is Needed. In: 2nd International Workshop on Component-Based Software Development, in conjunction with ICSE '99.

[20] POSIX 1003.1. International Standard for Real-Time: POSIX 1003.1;ISO/IEC-9945-1. IEEE Standards Publication.

[21] PRESSMAN, R. S. **Software Engineering:** A Practitioner's Approach. 5a Edição, 2001.

[22] ROSS, D. T. Structured Analysis (SA): A language for communicating Ideas. IEEE Transaction on Software Engineering, January 1977.

[23] RUMBAUGH, J.; et al. **The Unified Modeling Language Reference Manual.** Ed. Addison-Wesley, EUA, 1998.

[24] SINGHAL, A. Real Time Systems: A Survey. 1996. URL: http://citeseer.nj.nec.com/singhal96real.html. Consultado em 05/10/2002.

[25] SMALL, C.; SELTZER, M. VINO: An Integrated platform for operating system and database research. Technical Report TR-30-94, Harvard University, 1994.

[26] **Virtuoso - The Virtual Single Processor Programming System:** User Manual, Version 3.11, EONIC SYSTEMS, 1996.

[27] YERRABALLI, R. Real Time Operating Systems: An Ongoing Review. In: 21st. Real Time Systems Symposium, WIP Section, Orlando, EUA. 2000. URL: http://www.cse.ucsc.edu/~sbrandt/rtss2000/proceedings/16.pdf. Consultado em: 02/11/2002.