UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS DE TELECOMUNICAÇÕES

JORGE LUIS GONÇALVES DA SILVA

IMPLEMENTAÇÃO DO AMBIENTE DE DESENVOLVIMENTO KAYA EM UM MICROCONTROLADOR PIC DE 32 BITS”

TRABALHO DE CONCLUSÃO DE CURSO

CURITIBA
2011

JORGE LUIS GONÇALVES DA SILVA

IMPLEMENTAÇÃO DO AMBIENTE DE DESENVOLVIMENTO KAYA EM UM MICROCONTROLADOR PIC DE 32 BITS

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de TCC, do Curso Superior de Tecnologia em Sistemas de Telecomunicações do Departamento Acadêmico de Eletrônica – DAELN – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Dr. Luís Alberto Lucas.

CURITIBA

2011

JORGE LUIS GONÇALVES DA SILVA

IMPLEMENTAÇÃO DO AMBIENTE DE DESENVOLVIMENTO KAYA EM UM MICROCONTROLADOR PIC DE 32 BITS

Este Trabalho de Conclusão de Curso foi julgado e aprovado como requisito parcial para a obtenção do Título de Tecnólogo em Sistemas de Telecomunicações do Curso Superior de Tecnologia em Sistemas de Telecomunicações da Universidade Tecnológica Federal do Paraná.

Curitiba, 01 de julho de 2011.

_____

Prof. César Janeczko
Coordenador do Curso
Departamento Acadêmico de Eletrônica

_____

Prof. Décio Estevão do Nascimento
Professor Responsável pela Atividade do Trabalho de Conclusão de Curso
Departamento Acadêmico de Eletrônica

BANCA EXAMINADORA

_____

Prof. Dr. Luís Alberto Lucas
Departamento Acadêmico de Eletrônica

_____

Prof. Luiz Fernando Copetti
Departamento Acadêmico de Eletrônica

_____

Prof. Nelson Garcia de Paula
Departamento Acadêmico de Eletrônica

A Folha de Aprovação assinada encontra-se na Coordenação do Curso

AGRADECIMENTOS

Ao professor Luís Alberto Lucas, pela orientação indispensável para o desenvolvimento do trabalho e para a elaboração da monografia.

Aos professores do curso de Tecnologia em Sistemas de Telecomunicações pelos conhecimentos que foram repassados.

Aos professores Décio Estevão do Nascimento e Denise Elisabeth Hey David pela orientação no projeto.

Aos meus familiares e amigos, pela força e estímulo para a conclusão deste trabalho.

RESUMO

SILVA, Jorge Luis Gonçalves da. Implementação do ambiente de desenvolvimento Kaya em um microcontrolador PIC de 32 bits. 2011. 35 f. Trabalho de Conclusão de Curso (Curso Superior de Tecnologia em Sistemas de Telecomunicações), Universidade Tecnológica Federal do Paraná. Curitiba, 2011.

Este trabalho visa mostrar como implementar um sistema operacional em modo-texto, através do Projeto Kaya, dentro de um microcontrolador, executando diversas funções, utilizando a linguagem de programação C, demonstrando como os sistemas operacionais fazem a interface entre o usuário e o *hardware* em um sistema embarcado. Foi utilizado o kit do microcontrolador PIC de 32 bits fabricado pela *Microchip*, que possui 03 *leds* e 03 microchaves, e através destes recursos oferecidos pelo kit foi possível "simular" o que acontece quando se tenta "produzir" e "consumir" um determinado número de informações dentro de uma memória, e ao mesmo tempo descobrir uma finalidade para o uso do microcontrolador de 32 bits.

Palavras chaves: Sistema Operacional, Projeto Kaya, Microcontrolador, Linguagem C, *Hardware,* Sistema embarcado.

ABSTRACT

SILVA, Jorge Luis Gonçalves da. Implementation of development environment Kaya in a 32-bit PIC microcontroller. 2011. 35 f. Trabalho de Conclusão de Curso (Curso Superior de Tecnologia em Sistemas de Telecomunicações), Universidade Tecnológica Federal do Paraná. Curitiba, 2011.

This work aims to show how to implement an operating system in text-mode, by Kaya Project, within a microcontroller, performing various functions, using the C programming language, demonstrating how operating systems are the interface between the user and the hardware in a embedded system. We used the kit 32-bit PIC microcontroller manufactured by Microchip, which has 03 LEDs and microswitches 03, and through these resources offered by the kit was possible to "simulate" what happens when you try to "produce" and "consume" a certain number of information within a memory, and at the same time discover a purpose for the use of 32-bit microcontroller.

Key words: Operating System, Kaya Project, Microcontroller, C language, Hardware, Embedded System.

SUMÁRIO

LISTA DE SIGLAS E ABREVIATURAS

CAN	Controller Area Network
DIP	Dual In-line Package
EEPROM	Electrically Erasable Programmable ROM
HW	Hardware
kB	kilo Bytes
LCD	Liquid Crystal Display
MCU	Multipoint Control Unit
MIPS	Microprocessor without interlocked pipeline stages ou milhões de instrução por segundo
OTP	One Time Programmable
PWM	Pulse Width Modulation
RAM	Random Access Memory
ROM	Read-Only Memory
SO	Sistema Operacional
SOIC	Small Outline Integrated Circuit
TQFP	Thin-profile Quad FlatPack
USB	Universal Serial Bus

CAPÍTULO 1 – INTRODUÇÃO

O Projeto Kaya é a implementação de um sistema operacional (SO) completo, sem sofisticação (somente modo-texto), a partir de uma especificação dada (Especificação para o Projeto do Sistema Operacional FutKaya, 2010). O sistema operacional deverá rodar sobre um simulador, por exemplo, o µMPS, que foi desenvolvido pela Universidade de Bolonha na Itália, onde também se utilizam especificações de sistemas operacionais semelhantes (GOLDWEBER; DAVOLI; MORSIANI, 2005).

Mas o µMPS nada mais é do que um "*hardware* implementado em um *software*". Ou seja, um sistema operacional pode "rodar" dentro de um *hardware*, como por exemplo, um microcontrolador. Para que os sistemas operacionais possam fazer a interface entre os usuários e o *hardware*, os sistemas embarcados possuem um microcontrolador que comanda as ações executadas pelo usuário. Como o Projeto Kaya é apenas uma especificação de como "fazer" um sistema operacional, foi implementado em outro equipamento, por exemplo, no microcontrolador PIC de 32 bits, fabricado pela *Microchip* (PIC32, 2009).

Lançada em 2007, a família de microcontroladores PIC32 possui mais recursos de memória (tanto de *flash* quanto de RAM) do que as famílias anteriores, embora mantenha compatibilidade de pinagem e periféricos com as séries de 16 pinos e aproveite as ferramentas de desenvolvimento existentes (como por exemplo, o MPLAB IDE).

1.1 PROBLEMA

Na especificação Kaya não está clara a convenção de codificação, ou seja, muitas vezes não é possível dizer como foram escolhidos os nomes para as funções e para as variáveis. Nota-se, por exemplo, que, às vezes, as funções aparecem em letras maiúsculas e outras vezes, não.

O Kaya rodando sobre o simulador µMPS, da forma como é feito atualmente, apresenta uma série de limitações:

1) o simulador está distante da realidade, porque a máquina simulada não existe realmente (foi criada a partir da mescla de equipamentos antigos, tais como, o processador MIPS R2000 e o computador IBM 360);

2) como a máquina simulada não existe de fato, há um certo trabalho em transferir o código desenvolvido (a implementação do Kaya) para rodar num *hardware* (HW) tal como o PIC32;

3) os alunos da disciplina de Sistemas Operacionais não se sentem suficientemente motivados a estudar um HW fictício, já que este conhecimento não será útil posteriormente.

1.2 JUSTIFICATIVA

O aprendizado do simulador µMPS não é tão motivante quanto o de um *hardware* de verdade, embora seja mais fácil de aprender do que o próprio *hardware*. Implementando um sistema operacional dentro de um microcontrolador, já é possível ter uma ideia de como o computador faz a interação com o usuário, de acordo com as ações que o usuário executa e, ao mesmo tempo, já se ganha conhecimento sobre a família de microcontroladores PIC 32, que foi desenvolvida há pouco tempo, e que oferece uma série de novos recursos, se comparada às famílias de microcontroladores de outros fabricantes ou mesmo às outras famílias fabricadas pela *Microchip*. Alguns recursos disponíveis no PIC 32 são: mais memória (tanto *flash* como RAM) do que os modelos anteriores e maior velocidade de processamento (PIC32, 2009).

1.3 OBJETIVOS

1.3.1 Objetivo Geral

Implementar um ambiente de desenvolvimento que permita a codificação de aplicações em um sistema operacional conforme as especificações do Projeto Kaya (GOLDWEBER; DAVOLI, 2005) em um microcontrolador PIC de 32 bits.

Da mesma forma que o Projeto Kaya simula um sistema operacional no µMPS, foi necessário desenvolver um código utilizando a linguagem C para programar o microcontrolador para que o sistema operacional “rode” dentro dele.

Em suma, foi adaptado o ambiente de desenvolvimento do PIC 32 (MPLAB IDE) para que seja possível implementar a especificação Kaya neste, ou seja, foi desenvolvido uma camada de *software* que permita implementar o Kaya em um microcontrolador PIC de 32 bits (implementar-se-á o nível L1, a *bios*, apresentado na Figura 1).

1.3.2 Objetivos Específicos

A partir de “orientações de codificação” (*coding guidelines*) previamente estabelecidas, renomear as funções definidas na especificação Kaya e reimplementar estas funções em um microcontrolador, utilizando a linguagem de programação C.

A reimplementação foi necessária porque o novo *hardware* (PIC 32), embora tenha similaridade com o antigo (µMPS), apresenta particularidades.

1.4 PROCEDIMENTOS METODOLÓGICOS

Para a realização deste projeto foram utilizados, como metodologia de pesquisa, livros e apostilas sobre sistemas operacionais e Linguagem C, sobretudo os materiais de apoio que foram utilizados nas disciplinas do curso que envolveram estes dois tópicos. Também foram utilizadas as referências disponíveis nos *sites* de fornecedores de componentes eletrônicos, especialmente de *kits* de desenvolvimento compostos por microcontroladores, assim como os *sites* que contam um pouco da história dos microcontroladores PIC fabricados pela *Microchip*, verificando a disponibilidade de compra para poder, assim, elaborar um orçamento. Foram estudados os principais recursos oferecidos pelo microcontrolador PIC de 32 bits, em comparação com as famílias anteriores da *Microchip* e com as famílias de outros fabricantes.

O código que foi elaborado, em linguagem C, para programar o microcontrolador, foi baseado naquele utilizado no simulador µMPS.

A versão final do código foi testada no ambiente de desenvolvimento MPLAB IDE, fazendo os ajustes necessários e corrigindo os erros que surgirem. A comunicação entre o *software* e o microcontrolador foi feita através de uma porta USB.

CAPÍTULO 2 - EMBASAMENTO TEÓRICO

2.1 SISTEMAS OPERACIONAIS

Os sistemas operacionais são programas que fazem a interface entre os usuários e o *hardware*. São os ambientes mais adequados (amigáveis) para que os usuários possam executar programas e são os sistemas operacionais que tornam eficiente o uso dos computadores. Os Sistemas Operacionais mais conhecidos são o Windows e o Linux, mas podem ser citados também o Mac OS X, Solaris e Haiku (TANENBAUM, 2003).

2.1.2 Processos

Um processo é basicamente um programa em execução. A cada processo está associado o seu espaço de endereçamento, que é uma lista de posições de memória que este processo pode ler e escrever. Este espaço de endereçamento contém o código do programa executável, os dados do programa e sua pilha. Também associado a cada processo está um conjunto de recursos, normalmente incluindo registradores, uma lista dos arquivos abertos, alarmes pendentes, listas de processos relacionados e todas as demais informações necessárias para executar um programa. Em outras palavras, um processo é um container que armazena todas as informações necessárias para executar um programa (TANENBAUM, 2003).

2.1.3 *Threads*

Threads são “processos leves” (*lightweight processes*) criados para melhorar o desempenho de um sistema. Os *threads* existem para que múltiplas atividades possam ocorrer ao mesmo tempo em uma determinada aplicação, sendo que algumas destas atividades podem ser bloqueadas de tempos em tempos. O modelo de programação fica mais simples quando uma aplicação é decomposta em múltiplos *threads* sequenciais que executam em quase paralelo.

Outro fator importante para a existência de *threads* é que eles são mais rápidos de criar e destruir do que os processos, pois os primeiros não têm quaisquer recursos associados a eles (TANENBAUM, 2003).

2.1.4 Comunicação entre processos/ o problema do produtor-consumidor

Em alguns sistemas operacionais, processos podem trabalhar juntos e compartilhar algum armazenamento comum, a partir do qual cada um seja capaz de ler e escrever. O armazenamento compartilhado pode estar na memória principal ou em um arquivo compartilhado. No entanto, pode acontecer de vários processos tentarem acessar um mesmo local de memória ao mesmo tempo. Situações como essa – onde dois ou mais processos estão lendo ou escrevendo algum dado compartilhado e cujo resultado final depende de quem e de quando executa precisamente – são chamadas de **condições de corrida** (*race conditions*). Os resultados da maioria dos testes não apresentam problemas, mas uma hora, em um momento raro, algo estranho e inexplicável pode acontecer (TANENBAUM, 2010).

Para resolver este problema, é necessário encontrar algum modo de impedir que mais de um processo leia e escreva ao mesmo tempo em uma memória compartilhada. Vejamos um exemplo, o problema do produtor-consumidor, descrito a seguir.

Dois processos compartilham um *buffer* comum e de tamanho fixo. Um deles, o produtor, coloca informação dentro do *buffer* e o outro processo, o consumidor, a retira.

O problema se origina quando o produtor quer armazenar alguma informação nova no *buffer*, mas este já está cheio. A solução é colocar o produtor para dormir e só despertá-lo quando o consumidor retirar um ou mais itens do *buffer*. Da mesma forma, se o consumidor quiser retirar algum item do *buffer* e este estiver vazio, ele dormirá até que o produtor coloque algo no *buffer* e o desperte.

A condição de corrida (ou disputa) ocorrerá neste caso porque ambos (produtor e consumidor) utilizam uma mesma variável para saber se o *buffer* está cheio ou vazio. Num dado instante (por exemplo), o *buffer* está vazio e o consumidor acabou de ler a variável para verificar se o seu valor é zero. O escalonador pára temporariamente de executar o consumidor e começa a executar o produtor. Este por sua vez insere um item no *buffer*, incrementa a variável e “acorda” o consumidor. Porém, como o consumidor não está logicamente adormecido, este sinal de acordar é perdido e quando o consumidor for “ler” o valor da

variável, lido anteriormente por ele, verificará que o valor é zero e adormecerá. O produtor, por sua vez, preencherá todo o *buffer* e também adormecerá. Os dois dormirão para sempre.

2.1.5 Semáforos

Semáforos, em sistemas operacionais, são tipos de variáveis que possuem a função de despertar ou fazer adormecer um determinado processo utilizando valores pré-determinados (binários ou não) e funções como *down* e *up* (ou então *sleep* – "dormir"- e *wakeup* – "acordar", respectivamente). Uma vez iniciada uma operação de semáforo, nenhum outro processo pode ter acesso ao semáforo até que a operação tenha terminado ou sido bloqueada.

A operação *down* sobre um semáforo verifica se o seu valor é 0, e se for, decrementará o valor e prosseguirá. Caso contrário, o processo será posto para dormir.

A operação *up* incrementa o valor de um dado semáforo. Depois de um *up* em um semáforo, com processos dormindo nele, o semáforo permanecerá 0, mas haverá um processo a menos dormindo nele.

2.2 O PROJETO KAYA

O Projeto Kaya é a implementação de um sistema operacional a partir de uma especificação dada. É dividida em cinco camadas (Figura 1), descritas a seguir:

-Nível 0 (L0): O *hardware*, descrito no manual do µMPS (GOLDWEBER; DAVOLI, 2005A).

-Nível 1 (L1): Os serviços complementares em ROM (*bios*).

-Nível 2 (L2): O Gerente de filas. Com base no conceito-chave de sistemas operacionais em camadas, as entidades utilizadas em uma camada são apenas estruturas de dados em camadas inferiores.

-Nível 3 (L3): O *kernel*. Este nível executa oito primitivas de gestão de processos novos no modo *kernel* e sua sincronização. Executa, também multiprogramação, agendador de processos, dispositivos manipuladores de interrupção e detecção de impasse.

-Nível 4 (L4): Nível de Suporte. Este é o nível de usuário, cujos processos rodam em seu próprio espaço de endereçamento e memória virtual ativada.

L4	Modo usuário
L3	Núcleo
L2	Gerente de filas
L1	*Bios*
L0	HW

Figura 1 – Visão arquitetural do Kaya.
Fonte: Autoria Própria

2.3 O SIMULADOR µMPS

O µMPS é um simulador de sistema operacional que possui uma arquitetura baseada no processador MIPS RISC R2/3000 e no computador IBM 360. O gerenciamento de memória e capacidades de manipulação de exceção do µMPS é baseado no MIPS. Finalmente, um conjunto completo de dispositivos de Entrada/Saída (por exemplo, discos, impressoras e terminais) é fornecido (GOLDWEBER; DAVOLI, 2005A).

O MIPS (sigla para *Microprocessor without interlocked pipeline stages* - Microprocessador sem estágios interligados de *pipeline*) é uma arquitetura de processadores RISC desenvolvida pela *MIPS Computer Systems*. Foi criado na década de 80 com o intuito de tornar mais simples o projeto dos microprocessadores.

Os processadores MIPS são usados em aplicações tais como:

- Computadores da *Silicon Graphics* (Arquitetura MIPS, 2010);
- Sistemas embarcados (Arquitetura MIPS, 2010);
- Dispositivos com Windows CE (Arquitetura MIPS, 2010);
- Roteadores da Cisco (Arquitetura MIPS, 2010);
- *Videogames* como *Nintendo* 64 e *PlayStation* (Arquitetura MIPS, 2010).

As primeiras versões das CPU's MIPS eram de 32 bits, mas as mais recentes são de 64 bits. O primeiro modelo comercial do processador MIPS, o R2000, foi anunciado em 1985 (ARQUITETURA MIPS, 2010).

2.4 LINGUAGEM C

A linguagem C é uma linguagem de programação compilada de propósito geral, criada em 1972, por Dennis Ritchie, no *AT&T Bell Labs*, para desenvolver o sistema operacional UNIX (que foi originalmente escrito em *Assembly*). A programação em linguagens de alto nível tem como característica não ser necessário conhecer o processador, ao contrário das linguagens de baixo nível. A linguagem C permite acesso de baixo nível com a inserção de código *assembly* no código-fonte, ou seja, o baixo nível é realizado por *assembly* e não C. Desde a sua criação, C espalhou-se por outros sistemas computacionais, tornando-se uma das linguagens de programação mais usadas, influenciando muitas outras linguagens, especialmente C++, que foi originalmente desenvolvida como uma extensão para C.

C foi criada com o objetivo principal de facilitar a criação de programas extensos com menos erros, recorrendo ao paradigma da programação algorítmica ou procedimental. As principais características da linguagem são:

- o núcleo é simples, não possuindo funcionalidades que não sejam fundamentais, tais como funções matemáticas ou de manuseio de arquivos. Tais funções extras são fornecidas por um conjunto de bibliotecas padronizadas;
- sistema de tipos simples;
- uso de uma linguagem de pré-processamento para tarefas como a definição de macros e a inclusão de múltiplos arquivos de código-fonte;
- acesso de baixo-nível, através de inclusões de código *assembly* no meio do programa C;
- estruturas de variáveis (*structs*) que permitem que dados relacionados sejam combinados e manipulados como um todo (C, 2010).

2.5 MICROCONTROLADORES PIC

Os PICs constituem uma família de microcontroladores fabricados pela *Microchip Technology*, que processam dados de 8, 16 e mais recentemente 32 bits com extensa variedade de modelos e periféricos internos, com arquitetura *Harvard* e conjunto de instruções RISC (conjuntos de 35 e de 76 instruções), com recursos de programação por memória *flash*, EEPROM e OTP. Os microcontroladores PIC possuem núcleos de

processamento de 12, 14 e 16 bits e trabalham em velocidades de até 80 MHz, usando, em média 1 ciclo de clock por instrução, o que permite uma velocidade de até 10 MIPS (10 milhões de instrução por segundo). Há o reconhecimento de interrupções tanto externas como de periféricos internos. Funcionam com tensões de alimentação de 2 a 6V, possuindo encapsulamento de 6 a 100 pinos em diversos formatos (SOT23, DIP, SOIC, TQFP, etc).

Alguns de seus principais periféricos internos (a disponibilidade varia conforme o modelo) são:

- Conversores Analógico-Digitais de 8 a 12 bits;
- Contadores e *timers* de 8 e 16 bits;
- Comparadores Analógicos;
- Controladores PWM;
- Controladores de LCD;
- Controladores de motores;
- Controladores *Ethernet*;
- *Watchdog timer*;
- Detectores de falha na alimentação;
- Portas digitais com capacidade de 25mA (fornecimento e dreno) para acionar circuitos externos;
- Osciladores internos.

Os PICs podem ser programados em linguagem mnemônica (*assembly*) ou usando-se compiladores de linguagem de alto nível que geram um código em formato hexadecimal, usado para ser gravado na memória de programa desses microcontroladores. Para tal procedimento, utiliza-se um *hardware* especial (gravador) acoplado a um PC. Como ferramentas de desenvolvimento, encontram-se disponíveis: gravadores, depuradores, emuladores, placas de protótipos, etc (MICROCONTROLADOR PIC, 2010).

A linha de microcontroladores PIC é comumente utilizada em:

- Eletrônicos de consumo;
- Automação;
- Robótica;
- Instrumentação;
- Eletrônica embarcada;
- Periféricos de informática.

A família PIC de 32 bits inclui um núcleo de processamento MIPS32 M4K, com *pipeline* de cinco estágios, *hardware* dedicado para multiplicação e acumulação, e oito bancos

com 32 registros cada. Além disso, esta família oferece uma arquitetura altamente eficiente de barramento interno, e *cache* de instrução. As 5 famílias de PIC32 MCUs oferecem *Ethernet*, CAN, e USB. A memória *flash* integrada varia de 32 kB a 512 kB e a RAM interna varia de 8 kB a 128 kB.

Os clientes da *Microchip* se beneficiam da oferta de bibliotecas de *software* livre, geralmente disponíveis em forma de código-fonte. O ambiente de desenvolvimento MPLAB atende todos os 600 microcontroladores da *Microchip*, ou seja, quando se decide trabalhar com um microcontrolador diferente (16 → 32 bits, por exemplo), não há a necessidade de se estudar novas ferramentas de desenvolvimento (MICROCONTROLADOR DE 32 BITS, 2010).

O kit de desenvolvimento PIC (Figura 2) de 32 bits é composto de:

- 1 *Starter Kit Microchip* PIC32;
- 1 Cabo USB;
- 1 CD-ROM com softwares e guia do desenvolvedor.

Figura 2 – Kit de desenvolvimento PIC de 32 bits da *Microchip*.
Fonte: PIC32 STARTER KIT, 2011.

CAPÍTULO 3 – METODOLOGIA

Para o desenvolvimento deste trabalho foi utilizado o kit PIC32 fabricado pela *Microchip*, sendo que a placa é composta por 03 chaves (SW1, SW2 e SW3) do tipo *"Dip-switch"* e por 03 *leds* (L1, L2 e L3) e o método utilizado para demonstrar o funcionamento e também a resolução dos problemas de sincronização envolvendo sistemas operacionais é do tipo produtor-consumidor, da seguinte maneira:

- As chaves SW1 e SW2 codificam o n° (em binário) a ser armazenado no *buffer*. Como são 02 chaves, será possível codificar 04 números conforme quadro 1:

Número em decimal	Codificação em binário
0	00
1	01
2	10
3	11

Quadro 1: algarismos decimais e seus correspondentes em binário
Fonte: Autoria própria

- A chave SW3 insere o n° no *buffer*.

No kit, as chaves SW1 e SW2 serão o "produtor" e os *leds* L1 e L2 serão o "consumidor". A chave SW3 vai "armazenar" a informação no *buffer* e o *led* L3 irá acender para demonstrar que a informação foi "armazenada" com sucesso. As informações serão armazenadas da seguinte forma:

a) ao pressionar SW3 (sem pressionar SW1 e SW2), insere-se 00 no *buffer* e acende-se o LED3, e os outros *leds* permanecem apagados;

b) Ao pressionar SW2 e depois SW3, insere-se 01 no *buffer*, acende-se o LED3 e depois o LED2 (o kit "mostra" a informação que foi armazenada no *buffer*);

c) Ao pressionar SW1 e depois SW3, insere-se 10 no *buffer*, acende-se o LED3 e depois o LED1 (o kit "mostra" a informação que foi armazenada no *buffer*);

d) Ao pressionar SW1 e SW2 (as duas ao mesmo tempo) e depois SW3, insere-se 11 no *buffer*, acende-se o LED3 e depois os *leds* 1 e 2 (acende os dois juntos, "mostrando" a informação que foi armazenada).

Para se obter os resultados necessários foi preciso editar um dos arquivos do projeto Kaya e substituí-lo por um código em C que fizesse tudo o que foi especificado e que solucionasse o problema do produtor-consumidor. Neste código foi necessário utilizar:

1) *Threads*: t1 (processo raiz, pai de todos, que não precisou ser utilizado mas que precisa constar no código para que o mesmo funcionasse), produtor e consumidor;

2) Dois *buffers*, sendo um para a primeira chave de leitura (SW1) e outro para a segunda chave de leitura (SW2);

3) Funções para ler as chaves e mostrar as devidas informações nos *leds*;

4) Semáforos, para evitar os conflitos (condições de corrida);

5) *Delays* (tempo de atraso), para que seja possível visualizar as informações que estão sendo armazenadas (produtor) e removidas (consumidor).

O Projeto Kaya utiliza um recurso denominado "*syscall*"(chamadas de sistema) para que se possa criar e terminar uma *thread* (*createThread* e *terminateThread*, respectivamente). Foram criadas 03 *threads*: t1, produtor e consumidor. A *thread* t1 foi criada apenas para dar origem às demais, visto que ela é o processo raiz do sistema, e se ela não estiver presente, não será possível criar as demais. As outras *threads* (produtor e consumidor) são criadas a partir da *thread* t1 e são executadas conforme as ações do usuário. Como a *thread* t1 não é utilizada, ela é posta para dormir, chamando as demais *threads* para serem executadas. A função *terminateThread* não é utilizada porque em nenhum momento, uma *thread* é terminada porque as mesmas ficam rodando em laço infinito, ou seja, uma chamando a outra.

Será utilizado um *buffer* de 04 posições e o produtor não poderá produzir mais do que cabe no *buffer*. Quando o *buffer* estiver cheio (ou seja, com todas as 04 posições preenchidas), o produtor será posto para "dormir" e fará com que o consumidor "acorde" para que este possa remover um ou mais itens de dentro do *buffer*.

Da mesma forma, o consumidor não poderá retirar um item de dentro do *buffer* quando este estiver vazio. Quando isto acontecer, o consumidor é que será posto para dormir, acordando o produtor para que este possa inserir um ou mais itens de dentro do *buffer*.

Serão utilizados 04 semáforos no código: *xyz*, vazio, cheio e *mutex*. Os semáforos "cheio" e "vazio" serão utilizados para demonstrar que um determinado item foi inserido ou removido do *buffer*. O semáforo "*mutex*" será utilizado para evitar conflito entre o produtor e o consumidor, visto que os dois "compartilham" o mesmo *buffer* e este *buffer* é a região crítica do "sistema", ou seja, se não existisse este semáforo, o programa poderia travar em um determinado momento, exatamente como ocorre nos sistemas operacionais quando mais de um processo é posto para rodar simultaneamente sem que seja definida uma "prioridade".

Quando o produtor ou o consumidor entrar em ação, aquele que estará atuando fará um *down* no *mutex* (entra na região crítica) e ninguém poderá utilizar o *buffer* naquele momento. O *buffer* só poderá ser utilizado depois que a *thread* correspondente (produtor ou consumidor) fazer um *up* no *mutex* (sai da região crítica) e também no semáforo correspondente (*up* no "cheio" se for o produtor ou *up* no "vazio" se for o consumidor), para demonstrar que um ou mais itens foram produzidos ou removidos de dentro do *buffer*. Depois que a *thread* já utilizou o *buffer*, a mesma faz um *up* no *mutex*, saindo da região crítica e liberando o *buffer* para que a outra *thread* possa utilizá-la. Já o semáforo *xyz* será utilizado para "adormecer" a *thread* t1, visto que ela não é utilizada no código, mas precisa estar presente porque é a responsável por gerar as *threads* produtor e consumidor.

O *buffer* é a região crítica do sistema porque ele é compartilhado pelo produtor e pelo consumidor, ou seja, as duas *threads* têm acesso a ele. É utilizado um semáforo para que haja um controle de acesso ao *buffer*. Quando uma *thread* acessa o *buffer*, a mesma faz um *down* no semáforo e nenhuma outra *thread* poderá acessá-lo. O *buffer* só poderá ser acessado quando a *thread* que estiver utilizando fizer um *up* no semáforo, liberando o *buffer* para que outra *thread* possa acessá-lo. Se não tivesse este semáforo, o *buffer* seria acessado pelo produtor e pelo consumidor a qualquer momento sem nenhum tipo de controle, fazendo que o *buffer* se comporte de maneira inconsistente, como por exemplo, o consumidor tentar remover uma informação que ainda não foi produzida ou então quando o produtor tenta sobrescrever uma informação que ainda não foi consumida.

CAPÍTULO 4 - TESTES E RESULTADOS

4.1 PASSO A PASSO

Para testar o código passo a passo no MPLAB foram necessários colocar 03 *breakpoints* (pontos de parada) no mesmo, simulando as informações que são produzidas e consumidas, mostrando-as nos *buffers* (*Watch*) do MPLAB. O código é posto para rodar e o programa aguarda o usuário pressionar (ou não) as chaves SW1 e SW2 na combinação que desejar e em seguida pressionar a chave SW3 para armazenar as combinações escolhidas. Cada vez que a chave SW3 é pressionada, acende-se o LED3 para mostrar que a(s) informação(ões) foi (foram) armazenada(s) no *buffer*. Como nos *buffers* cabem somente 04 informações binárias, quando se tenta colocar uma quinta informação, o produtor é posto para dormir e o consumidor entra em ação, removendo as informações do *buffer* e mostrando através dos *leds* quais as informações que foram produzidas e que estão sendo removidas, na ordem em que foram colocadas, através das combinações escolhidas pelo usuário. Da mesma forma, nos *buffers* do MPLAB mostram-se as informações sendo removidas pelo consumidor. Quando o *buffer* é esvaziado e o consumidor tenta remover uma informação que não existe, o mesmo é posto para dormir e acorda o produtor para que este possa "encher" o *buffer* novamente.

Os *breakpoints* são colocados na linha (ver anexos) onde a chave SW3 é lida no produtor (`ch3 = lerChave(3);`) e na linha onde o consumidor faz um *down* no semáforo "cheio" (`down(&cheio);`). Depois que o *buffer* é esvaziado, é colocado um *breakpoint* na linha onde o produtor entra na região crítica, ou seja, faz um *down* no semáforo *"mutex"* (`down(&mutex);`) para que novas informações sejam produzidas e colocadas no *buffer*.

4.2 COM *DELAY*

Rodando-se o código ininterruptamente, é possível visualizar quais as informações que o produtor produziu e que serão removidas pelo consumidor depois que o *buffer* estiver cheio. Mas para isso foi necessário colocar um "*delay*" (atraso) para que esta informação não seja

consumida imediatamente após ser produzida (ou seja, o LED1 e o LED2 não podem acender depois que o usuário pressiona a chave SW3, após ter escolhido as combinações desejadas através das chaves SW1 e SW2). Porém, quando o *buffer* está cheio, logicamente a próxima informação que o usuário tentar inserir no *buffer* não será armazenada, mas será "gravada" na memória do PIC. Vejamos um exemplo:

- o usuário quer gravar as seguintes combinações no *buffer*: 11, 10, 01 e 00, nesta ordem. Então, ele pressiona as chaves SW1 e SW2 (as duas ao mesmo tempo) e depois a chave SW3. Depois, pressiona a chave SW1 e em seguida SW3. Depois, pressiona SW2 e em seguida SW3. E por último pressiona somente SW3. Supondo que ele "tente" armazenar depois que o *buffer* está cheio a combinação 01 (pressionando SW2), quando SW3 for pressionada, o consumidor entra em ação removendo as informações na ordem em que foram colocadas, acendendo os *leds* correspondentes. Mas depois que a última informação for removida, o produtor automaticamente coloca na primeira posição do *buffer* a combinação 01, que foi memorizada quando o usuário tentou colocar uma informação que não cabia no *buffer*. Como esta informação já está presente no *buffer*, o produtor poderá colocar somente mais 03 informações no *buffer* e quando se tenta colocar mais uma combinação quando o *buffer* já está cheio, o consumidor remove as 04 informações contidas no *buffer* e quando este for esvaziado, o produtor coloca no *buffer* a informação que não coube e assim sucessivamente.

4.3 SEM *DELAY* (CONSUMO INSTANTÂNEO)

Caso o código seja executado sem os *delays* (atrasos), ocorre o chamado "consumo instantâneo". Quando isso acontece, o produtor insere uma única informação nas 04 posições do *buffer* e esta informação é imediatamente removida pelo consumidor, ou seja, quando o usuário pressionar a chave SW3, o LED3 irá acender para dizer que a informação foi armazenada no *buffer* e imediatamente os outros dois *leds* irão mostrar qual a informação que foi produzida e que está sendo consumida. Isto acontece devido à rápida velocidade de processamento, característico do PIC32.

CAPÍTULO 5 – CONCLUSÕES E SUGESTÕES PARA TRABALHOS FUTUROS

Este trabalho foi útil para se ter uma idéia de como funciona um sistema operacional, de como este faz a interação entre o *hardware* e o usuário, e o que pode acontecer quando se tenta executar vários processos ao mesmo tempo, estipulando métodos para que este sistema operacional funcione adequadamente sem causar transtornos para o usuário. Também foi importante para se estudar as principais funções e aplicações do microcontrolador PIC de 32 bits, bem como saber quais os recursos que ele proporciona.

As principais dificuldades encontradas foram: verificar a disponibilidade do kit para compra, adaptar o código do projeto Kaya para que se pudesse programar a placa de forma a atender os objetivos desejados e pesquisar os códigos necessários para que o kit funcionasse de acordo com o que foi proposto.

Como sugestão para trabalhos futuros, pode-se tentar utilizar o Projeto Kaya para solucionar outros problemas clássicos envolvendo sistemas operacionais tais como: barbeiro sonolento e jantar dos filósofos, e também pesquisar outras aplicações para o PIC32, aproveitando os recursos que o mesmo oferece.

REFERÊNCIAS

ARQUITETURA MIPS. Disponível em: <http://pt.wikipedia.org/wiki/Arquitetura_MIPS>. Acesso em: 01 mai. 2010.

C (linguagem de programação). Disponível em: <http://pt.wikipedia.org/wiki/C_(linguagem_de_programa%C3%A7%C3%A3o)>. Acesso em: 01 mai. 2010.

GOLDWEBER, Michael e DAVOLI, Renzo (2005). **Student Guide to the Kaya Operating System Project.** Disponível em: <http://www.daeln.ct.utfpr.edu.br/~lalucas/Disciplinas/Engenharia_Computacao/Sistemas_Operacionais/Kaya%20CD/doc/pdf/Kaya.pdf>. Acesso em: 01 mai. 2010.

GOLDWEBER, Michael, DAVOLI, Renzo e MORSIANI, Mauro (2005). **The Kaya OS project and the µMPS Hardware Simulator.** Disponível em: <http://www.cs.xu.edu/uMPS/Documentation/uMPSpaper.pdf>. Acesso em: 15 mai. 2010.

GOLDWEBER, Michael e DAVOLI, Renzo (2005). **µMPS Principles of Operation.** Disponível em: <http://www.daeln.ct.utfpr.edu.br/~lalucas/Disciplinas/Engenharia_Computacao/Sistemas_Operacionais/Kaya%20CD/doc/pdf/princOfOperations.pdf>. Acesso em: 01 mai. 2010.

MICROCONTROLADOR PIC. Disponível em: <http://pt.wikipedia.org/wiki/Microcontrolador_PIC>. Acesso em: 01 mai. 2010.

MICROCONTROLADOR DE 32 BITS. Disponível em: <http://www.nei.com.br/lancamentos/lancamento.aspx?i=12259>. Acesso em: 01 mai. 2010.

PIC32. Microcontroller Families (2009). Disponível em: <microchip.com/pic32>. Acesso em: 09 mai. 2010.

TANENBAUM, Andrew S. (2003). **Sistemas operacionais modernos.** 2ª edição. Prentice-Hall. São Paulo.

TANENBAUM, Andrew S. (2010). **Sistemas operacionais modernos.** 3ª edição. Prentice-Hall. São Paulo.

Especificação para o Projeto do Sistema Operacional FutKaya. Disponível em: <http://www.daeln.ct.utfpr.edu.br/~lalucas/Disciplinas/Engenharia_Computacao/Sistemas_Operacionais/FutKaya/manual/manual.pdf>. Acesso em: 09 mai. 2010.

PIC32 STARTER KIT. Disponível em: <http://www.digikey.com/us/en/ph/Microchip/pic32.html?WT.z_Tab_Cat=Featured%2520Products>. Acesso em: 06 ago. 2011.

APÊNDICE

APÊNDICE A – Código utilizado na programação do PIC32.

```
#include "types.h"
#include "stdio.h"
#include "syscalls.h"
#include "libumps.h"
#include <plib.h>


#define N   4
bool buf1 [N];
bool buf2 [N];
Semaphore mutex = 1;
Semaphore vazio = N;
Semaphore cheio = 0;

Semaphore xyz = 0;

// threads
private void produtor (void);
private void consumidor (void);
bool lerChave (uint ch);
void escreverLed (uint led, bool stat);
void delay (int n);

/*
 * Function: t1
 *
 * Purpose: Este , o processo raiz i.e. pai de todos
 *
 * Input:   none
 *
 * Output:  none
 *
 */
void t1(void)
{
      bool rc = !OK;

      const int LED1 = BIT_0;
      const int LED2 = BIT_1;
      const int LED3 = BIT_2;
      const int SW1  = BIT_6;
      const int SW2  = BIT_7;
      const int SW3  = BIT_13;

      mPORTDSetPinsDigitalIn(SW1 | SW2 | SW3 );
      mPORTDSetPinsDigitalOut(LED1 | LED2 | LED3);

      // apaga os leds
      mPORTDClearBits(LED1);
      mPORTDClearBits(LED2);
      mPORTDClearBits(LED3);

      rc = createThread (produtor);
```

```
	rc = createThread (consumidor);

	down(&xyz);
}

void produtor (void)
{
	uint i=0;
	while(true){
		bool ch3;
		ch3 = lerChave(3);
		if(ch3){
			escreverLed(3, false);
			bool ch1, ch2;
			ch1=lerChave(1);
			ch2=lerChave(2);  /* produzir item */
			down(&vazio);
			down(&mutex);
			/* insere item */
			buf1[i]=ch1;
			buf2[i]=ch2;
			i=(i+1)%N;
			escreverLed(3, true);
			delay(1); /* deixar esta linha como comentário  para
			utilizar o  código sem o delay */
			escreverLed(3, false);
			up(&mutex);
			up(&cheio);
		}
	}
}

void consumidor (void)
{
	uint j=0;
	while(true){
		down(&cheio);
		down(&mutex);
		escreverLed(3, true);
		bool ch1, ch2;
		ch1=buf1[j];
		ch2=buf2[j];
		j=(j+1)%N;
		/* remove item */
		up(&mutex);
		up(&vazio);
		escreverLed(1, ch1);
		escreverLed(2, ch2); /* consome item */
		delay(1); /* deixar esta linha como comentário para utilizar o
		código sem o delay */
		escreverLed(3, false);
	}
}

bool lerChave (uint ch)
{
	bool n;

	const int SW1  = BIT_6;
	const int SW2  = BIT_7;
	const int SW3  = BIT_13;
```

```
	switch (ch){
		case 1:
			n = mPORTDReadBits(SW1);
			break;
		case 2:
			n = mPORTDReadBits(SW2);
			break;
		case 3:
			n = mPORTDReadBits(SW3);
			break;
		default:
			break;
	}

	if (n) return false;
	else return true;
}

void escreverLed (uint led, bool stat)
{
	const int LED1 = BIT_0;
	const int LED2 = BIT_1;
	const int LED3 = BIT_2;

	switch (led){
		case 1:
			if (stat){
				mPORTDSetBits(LED1);
			} else {
				mPORTDClearBits(LED1);
			}
			break;
		case 2:
			if (stat){
				mPORTDSetBits(LED2);
			} else {
				mPORTDClearBits(LED2);
			}
			break;
		case 3:
			if (stat){
				mPORTDSetBits(LED3);
			} else {
				mPORTDClearBits(LED3);
			}
			break;
		default:
			break;
	}

}

void delay (int n)
 {
	int i,k,count;
	const int DLY = 60000;
	for (i=0; i<n; i++) {
	   count = 0;
	   for (k=0; k<DLY; k++)
			count++;
```

```
            }
    }
```

ANEXO

ANEXO A – Padrões de codificação:

/*
** Copyright 2010 Luís Alberto Lucas, luisalbertolucas@gmail.com.
** Distributed under the terms of the MIT License.
*/

PADRÕES DE CODIFICAÇÃO

Num trabalho colaborativo de desenvolvimento de software, o resultado é melhor se todos os programadores utilizarem os mesmos padrões de codificação. Portanto, para atingir a qualidade pretendida, seguir-se-ão as seguintes convenções:

FUNÇÕES (MÉTODOS)

- Numa chamada de função, não se usará espaços após o nome da função. Por exemplo, usar-se-á print("hello"); e não print ("hello");

- Os protótipos (arquivos .h) e as implementações (arquivos .c) de uma função serão precedidos por um cabeçalho:

/*
** Function:
**
** Description:
**
** Input:
**
** Output:
**
*/

onde "Function" é o nome da função, "Description" é a sua finalidade (descrição), "Input" são os parâmetros de entrada, que deverão ser relacionados e descritos, e "Output" é o parâmetro de saída que, da mesma forma, deverá ser descrito;
- os nomes das funções iniciar-se-ão em minúsculas e, se forem nomes compostos, serão separados por letras maiúsculas. Por exemplo: saveContext(). A exceção a essa regra são as funções já definidas pelo ambiente de desenvolvimento, e.g.: _general_exception_context().

- Funções que não recebem argumentos devem ser declaradas como f(void) e não como f();

PALAVRAS RESERVADAS

- Não se usará um espaço após as palavras reservadas "if", "for", "while" e "switch".
Por exemplo: usar-se-á if(x) ao invés de if (x);

- Haverá um espaço entre o fecha-parêntese da palavra reservada e o abre-chaves.
Por exemplo: "while(pcb) {" ao invés de "while(pcb){"

EDENTAÇÃO

- a edentação é feita com tabs e não com espaços e, para isso, configura-se o tamanho do tab como sendo equivalente a 4 espaços;

CONSTANTES

- constantes (pré-processador ou enum) serão, geralmente, em letras maiúsculas e.g. PAGESIZE mas, quando tais constantes tiverem grafia padronizada (por exemplo no datasheet do processador ou no ambiente de desenvolvimento), tal grafia será respeitada, como em AdES. Outra exceção são as constantes usualmente utilizadas em letra minúscula em outras linguagens, como C++ ou java, tais como: bool e null (não serão utilizadas as tradicionais BOOL e NULL da linguagem C);

- dar-se-á preferência às constantes criadas com enum ao invés daquelas criadas com #define (pré-processador), já que as primeiras fazem com que o compilador produza mensagens de erro mais compreensíveis. A exceção a essa regra são aquelas constantes que devem ser

utilizadas tanto em C quanto em assembler, quando enum não é possível (ver arquivo defines.h);

PONTEIROS

- ponteiros em declarações e em argumentos de função terão espaço apenas antes do asterisco:
int *pInt;
void funcao (int *pInt);

- ponteiros em retornos de função terão espaço depois do asterisco:
pInt* funcao (void);

DEREFERENCIAÇÕES

Dereferenciações de ponteiros e de regiões de memória não terão espaços i.e. Escreva:

*(Cpu *)TODLOADDR ao inv,s de * (Cpu *)TODLOADDR
*x ao invés de * x

TIPOS DEFINIDOS PELO USUÁRIO

- os tipos definidos pelo usuário serão criados com typedef, sendo escritos com inicial maiúscula e demais letras minúsculas (a exemplo das classes em Java e C++) sendo que, para nomes compostos, cada palavra terminaria com minúscula e começaria com maiúscula, como nas funções. Também não se utilizará _t (de tipo) ao final do nome, como é usual em C. Por exemplo: utilizar-se-á Semaphore ao invés de semaphore_t;

Veja mais dois exemplos: MemAddr, Context.

VARIÁVEIS (OBJETOS)
Os objetos serão criados com letras iniciais minúsculas, seguindo a mesma nomenclatura das funções.

CASTINGS

Castings de ponteiros e de regiões de memória não terão espaço. Veja alguns exemplos:

```
(State *)INTOLDAREA;
(MemAddr)p6;
```

MÓDULOS (ARQUIVOS)

- A modularização do código em C é conseguida através do uso de arquivos;

- Os arquivos serão nomeados com letra inicial minúscula, mesmo aqueles que implementam ADTs (tipos abstratos de dados). Por exemplo: para o ADT Semaphore, a interface estaria em semaphore.h e a implementação em semaphore.c. Nos nomes de arquivos compostos de várias palavras, utilizar-se-á a mesma convenção das funções;

- Todo arquivo deverá ser iniciado com uma nota sobre os direitos autorais, seguida da descrição do módulo. Veja um exemplo:

```
/*
** Copyright 2010 Luis Alberto Lucas, luisalbertolucas@gmail.com.
** Distributed under the terms of the MIT License.
*/

/************************** asl.c ********************************
**

**  Module: Active Semaphore List Module; This module creates
**       the abstraction for the ASL.  The head of this singly
**       linked list is semdH.  Elements for this list come
**       from the array semdTable.  Unused elements of the table
**       are kept on the semdFreeH list.  This module contains
**       the functions necessary to manipulate and maintain the
```

```
**        ASL, and its associated free list.

**

**  Implemented using an array of semaphore descriptors, whose
**        empty elements are kept on a free list.  The active
**       list is singly-linked kept in sorted order by its
**       semaphore address.  The active list has a dummy node as
**       its first element.  The free list is maintained via the
**       queue module.
**

**  Written by Lu¡s Alberto Lucas

**********************************************************************/
```

ENCAPSULAMENTO

- O código deverá ser adequadamente encapsulado com "private" (#define private static), para atributos e métodos privados, e "public" (#define public extern), para atributos públicos (estes públicos devem ser evitados ao máximo!) e para métodos públicos (estes públicos representam a interface do módulo).

Veja um exemplo de atributo privado:

```
private Semaphore mutex = 1;
```