Universidade Federal do Paraná

Vanessa Burkot Alves de Oliveira

Porte do XINU para cMIPS

Curitiba PR
2016

Vanessa Burkot Alves de Oliveira

Porte do XINU para cMIPS

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação.*

Orientador: Prof. Dr. Roberto André Hexsel.

Curitiba PR
2016

A Ana e Antonio, meus pais.

Agradecimentos

Agradeço ao meu orientador pela oportunidade de compartilhamento de seu conhecimento e experiência. Agradeço ao David pela compreensão e carinho em momentos de dificuldades com a administração do meu tempo e com meu nervosismo.

Resumo

O XINU é um sistema operacional que foi projetado para o ensino dos conceitos de sistemas operacionais, é utilizado em várias universidades e tem sido portado para várias plataformas de *hardware*. Este trabalho descreve o porte e a adaptação do sistema operacional XINU para o cMIPS - *classical*MIPS, que trata-se de um modelo VHDL sintetizável do processador MIPS clássico de cinco estágios. Este modelo foi desenvolvido para ser um ambiente de experimentação realista para disciplinas de Arquitetura de Computadores, deste modo o cMIPS contém toda a funcionalidade para suportar um sistema *Unix-like*. A verificação de corretude do porte é feita através das simulações das principais funções do sistema operacional, que incluem criação e destruíção de processos, sincronização com semáforos e comunicação através de '*mailboxes*'.

Palavras-chave: porte, xinu, cmips.

Abstract

XINU is operational system projected to teach operational system concepts, used in several Universities and has been ported to many platforms of *hardware*. This text describes the port and adaptation of the XINU operational system into the cMIPS - *classical*MIPS, which is a synthesizable VHDL model of classical MIPS processor of five stages. This model was developed to be an enviroment of realist experimentation for Computer Architecture subjects, in this way cMIPS has all the functionality to suport an Unix-like system. The correctness verification of this port is made through main functions simulation of the operational system, which includes creation and destruction of process, synchronization with semaphores and comunication through mailboxes.

Keywords: porte, xinu, cmips.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

cMIPS	*classical*MIPS, uma implementação clássica do MIPS
CPU	*Central Processing Unit*
FIFO	*First in First out*
GCC	*GNU Compiler Collection*
MIPS	*Microprocessor without Interlocked Pipe Stages*
MMU	*Memory Management Unit*
RAM	*Random Access Memories*
RISC	*Reduced Instruction Set Computers*
ROM	*Read-Only Memory*
RR	*Round Robin*
SDRAM	*Synchronous Dynamic Random Access Memory*
TLB	*Translation Lookaside Buffer*
UART	*Universal Assynchronus Receiver Transmitter*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*
XINU	*Xinu is Not Unix*

Capítulo 1

Introdução

O cMIPS é um modelo sintetizável em VHDL do processador MIPS clássico de cinco estágios, e implementa o conjunto de instruções do MIPS32r2 como descrito na literatura de Patterson & Hennessy [10]. Seu desenvolvimento, iniciado em 2012, foi motivado pela necessidade de ambiente de experimentação para disciplinas de Arquitetura de Computadores do curso de Ciência da Computação no Departamento de Informática da Universidade Federal do Paraná.

Este modelo de processador executa código em linguagem C, compilado com GCC. O ambiente de simulação no modelo cMIPS consiste de memórias RAM e ROM, e dispositivos de entrada e saída. Este ambiente pode ser executado em simuladores ou em dispositivos lógicos programáveis.

Com o avanço do desenvolvimento do cMIPS, tornou-se evidente a necessidade de verificar o funcionamento de um sistema operacional sob esta plataforma, e o porte de um sistema operacional no cMIPS permite efetuar ajustes e melhorias, além de constatar que a sua implementação está correta e que comporta um sistema operacional com todas as suas funcionalidades. Motivados por essa necessidade, escolhemos um sistema operacional a ser portado para o cMIPS, um sistema operacional que segue um padrão de desenvolvimento consistente e regular, que prioriza o ensino das suas funcionalidades e não necessariamente presando por eficiência. Selecionamos o sistema operacional XINU para ser portado no cMIPS, além dos motivos citados anteriormente, este sistema operacional foi bem projetado (subdividindo suas abstrações) e o seu objetivo inicial era ser desenvolvido passo a passo para o ensinamento de sistemas operacionais.

O sistema operacional XINU segue um padrão para se projetar sistemas operacionais, ele inclui todos os componentes de um sistema operacional de categoria UNIX, por exemplo, gerenciamento de memória e de processos, coordenação, sincronização e comunicação de processos. A organização desses componentes é em camadas, e o sistema operacional é particionado em níveis, cada um deles é construído sobre os níveis mais baixos. A principal vantagem desta abordagem que destaca-o em comparação a outros sistemas operacionais é a modularidade, que simplifica a depuração e verificação do sistema.

Cada camada do sistema operacional XINU é implementada focando apenas nas operações supridas pelas camadas de nível mais baixo. O projetista de determinada camada não precisa saber como as operações de níveis mais baixos estão implementadas, necessitando apenas saber o que estas operações fazem. As funções do sistema foram particionadas em, aproximadamente, oito grandes categorias (*hardware, memory, process, real-time clock, device, intermachine communication, file system, application programs*) e os componentes foram organizados numa hierarquia multicamadas (dez camadas). Desta

forma, seu projeto é simples e modular, e o porte para outras plataformas de *hardware* é menos custoso. O sistema operacional XINU, além de proporcionar aprendizado sobre sistemas operacionais, também é utilizado em produtos comerciais de companhias como a Mitsubishi, Lexmark, HP e IBM, por exemplo. O código do sistema operacional XINU é executado em muitas plataformas de *hardware* e neste trabalho utilizamos a plataforma cMIPS.

Neste trabalho, descrevemos o esforço de portar o sistema operacional XINU para o cMIPS e, principalmente, verificamos se o porte foi completo e correto. Para isso, foram realizados testes em um conjunto de funcionalidades principais que o sistema operacional dispõe, como criação de processos e a troca de contextos, por exemplo. A motivação inicial para este trabalho era constatar que a plataforma de *hardware* cMIPS suportaria um sistema operacional da classe UNIX. O sistema operacional XINU foi escolhido para ser portado por se tratar de um sistema pequeno, com boa organização e por fundamentar-se nos sistemas de tempo compartilhado UNIX.

Veremos neste trabalho que vários ajustes foram realizados no XINU-MIPS disponibilizado pelo autor [7], para que a plataforma comportasse o sistema operacional, pois a plataforma de *hardware* onde o XINU-MIPS executa é diferente do XINU-cMIPS que estamos utilizando a partir deste momento. O mapa de memória é distinto e os seus periféricos são diferentes.

Ressalta-se que todos os testes funcionais do XINU-cMIPS foram realizados utilizando simulador (código VHDL sintetizável) porque não temos um kit de testes e a utilização de um FPGA (*Field Programmable Gate Array*) será considerada futuramente, pois a memória interna do FPGA disponível no laboratório não comporta o XINU.

A partir dos testes funcionais realizados no simulador, pode-se constatar que a plataforma cMIPS é capaz de comportar um sistema operacional da categoria UNIX, tendo em vista que os princípios que o sistema operacional XINU segue foram retirados do UNIX e, com a verificação e confirmação dos resultados esperados, consegue-se comprovar que o porte do XINU é realizado com sucesso no cMIPS.

O texto está organizado como segue. O capitulo dois aborda o que é o XINU e quais as funcionalidades que este sistema operacional fornece aos usuários e o porquê de sua escolha. Em sequência, no mesmo capitulo é mostrado como os componentes do XINU estão organizados nas camadas e depois são descritos os principais conceitos e componentes relacionados a esse sistema operacional. O capitulo três aborda a plataforma de *hardware*. Como o cMIPS é uma implementação do conjunto de instruções MIPS, inicia-se este capitulo com a explicação deste último processador, para em seguida apresentar o cMIPS. São descritas as principais informações do MIPS inicialmente, apresentando seu formato e conjunto de instruções e os registradores do coprocessador que são fundamentais para utilização no sistema operacional. Após a apresentação do MIPS, seguimos com a explicação da sua implementação cMIPS, exibindo o diagrama de blocos que ele forma.

O capitulo segue abordando as adaptações realizadas no cMIPS para acomodar o sistema XINU, depois expomos uma explicação para cada teste que foi realizado no simulador e qual o seu objetivo, estes testes comprovam o funcionamento correto das funções disponíveis no sistema operacional. Em seguida apresentamos os pseudos-programas dos testes implementados com os resultados de suas simulações. Terminamos o trabalho com a indicação de trabalhos e uma conclusão. Um anexo com os programas de testes realizados é adicionado ao final.

Capítulo 2

Sistema Operacional XINU

2.1 XINU

Um sistema operacional (SO) é um conjunto de funções que suporta a execução de programas. Quando o processador está executando um programa do usuário o SO está inativo. O SO controla e coordena o uso do *hardware* pelos diversos programas aplicativos para os diversos usuários [18].

O XINU é um SO simples e pequeno, foi projetado para ensinar os conceitos de sistemas operacionais tais como gerenciamento de memória, gerenciamento de processos, coordenação e sincronização de processos, comunicação entre processos, gerenciamento de tempo real, acionadores de dispositivos, comunicação entre máquinas (rede), e um sistema de arquivos [7].

Esse SO foi desenvolvido por Douglas Comer, em 1984, e o código foi escrito para o computador LSI-11 [4]. O significado do nome XINU é tanto UNIX escrito de trás para frente quanto um acrônimo *Xinu Is Not Unix* o que, segundo o autor, refere-se a facilidade de modificação, simplicidade e organização, diferentemente do UNIX. Apesar de ele ser internamente diferente dos sistemas operacionais existentes, as ideias fundamentais não são novas, e várias delas tem origem no sistema de tempo compartilhado UNIX [7]. Os programas escritos para um sistema normalmente não podem ser executados no outro sem (grandes) alterações.

Ao longo dos anos desde seu desenvolvimento, o XINU foi portado para várias plataformas, também é utilizado em universidades no auxílio do ensino de Sistemas Operacionais, e existem versões para sistemas embarcados [1].

O código do sistema operacional XINU é escrito na linguagem C, com algumas funções dependentes de máquina escritas em *assembly*.

Para projetar o sistema operacional XINU, utiliza-se a hierarquia em camadas, também denominada modelo estratificado [6, 3]. Neste modelo, os componentes do SO são organizados em níveis hierárquicos, facilitando a implementação e depuração [9]. Desta forma, foi possível a identificação dos serviços abstratos comuns aos SOs, e uma definição de componentes básicos que executam essas abstrações independentemente de *hardware*.

Os níveis, ou camadas, são construídos a partir do centro do modelo e cada nível fornece um serviço abstrato que é executado em termos dos serviços abstratos fornecidos pelas camadas de níveis inferiores. Cada camada precisa ser ordenada, de forma que os serviços necessários para uma determinada camada estejam definidos em camadas mais baixas. A abordagem estratificada é interessante também porque permite, por exemplo,

que subconjuntos sucessivamente maiores formem sistemas sucessivamente mais poderosos [7].

A essência de um SO está nos serviços que ele fornece aos programas de usuário e, em sistemas bem projetados, só é possível acessar esses serviços através de chamadas de sistema, (as *syscalls*). Essas, quando aparecem em um programa, se parecem com chamadas de procedimentos, mas durante a execução o controle é passado para rotinas do SO. São as *syscalls* que estabelecem uma fronteira bem definida entre o programa em execução e o sistema operacional. Elas são API (*Application Programming Interface*) do SO e definem os serviços que o sistema fornece. As *syscalls* também protegem o sistema contra acessos ilegais, elas gerenciam três aspectos importantes: conferem todos os argumentos; certificam-se de que as alterações deixam um estado consistente e reportam falha ou sucesso ao chamador.

O XINU é executado em um microprocessador juntamente com o código dos programas de usuário. Ele oferece serviços como:

- ler caracteres de um teclado;
- exibir caracteres em um terminal;
- gerenciar múltiplas computações concorrentes;
- operar temporizadores;
- armazenar arquivos em disco; e
- trocar mensagem entre programas.

A configuração mínima do XINU opera em um computador pequeno e com poucos recursos, sem a utilização de outros dispositivos de armazenamento adicionais [7], o que também influenciou para que fosse escolhido como SO a ser portado para o cMIPS.

Para a execução de um programa, o usuário o compila no hospedeiro, utilizando um compilador cruzado (*cross-compiler*), e o combina com o código da biblioteca XINU, usando um carregador cruzado (*cross-loader*). O compilador cruzado produz uma imagem de memória exata para o microprocessador, combinando a saída do compilador com programas compilados. Neste trabalho foram seguidas as instruções do próprio GCC para construir *cross*-compiladores para obter o compilador cruzado utilizado, o mesmo é uma versão do GCC compilada para gerar código para MIPS. (Detalhes em cMIPS/docs/install/Crosscompiler).

No sistema XINU, como em outros sistemas operacionais, admite-se processamento concorrente, de modo que sejam várias computações independentes realizadas em um processador, que é capaz de executar somente uma instrução de cada vez. O SO multiplexa um único processador entre vários programas, permitindo que cada um seja executado por um tempo determinado antes de mudar para outro.

2.1.1 Modelo Estratificado

Na arquitetura do XINU, as funções do sistema são divididas em oito[1] componentes e são organizados em camadas. A Figura 2.1 exibe a visão geral dos componentes e sua organização.

[1]Para o componente Processo as funções são subdivididas separadamente em três camadas (funções de gerenciamento, coordenação e comunicação)

Figura 2.1: A organização multi-nível dos componentes no XINU.

No nível zero, representado no centro do modelo estratificado, encontra-se o *hardware* do computador. Apesar de não fazer parte do próprio SO, em sistemas modernos são incluídos recursos que permitem integração com o SO. Os componentes de *hardware* incluem um processador, interface de memória, dispositivos de interfaces I/O, interface de rede Ethernet para uma conexão de rede local ou de internet.

A partir do *hardware*, cada nível superior de *software* do SO fornece primitivas mais potentes que os mecanismos de proteção do *hardware*. No nível um, está um gerenciador de memória, que controla e aloca memória. No nível dois está o gerenciador de processos que fornece primitivas para criar, encerrar, suspender e reiniciar processos, e inclui um escalonador e troca de contexto.

Logo acima da camada que gerencia os processos, vem a camada de coordenação de processos, que implementa os semáforos. Em seguida, no nível quatro, estão as funcionalidades para a comunicação entre processos, como as caixas postais. No nível cinco estão as funções do gerenciamento do relógio de tempo real que permitem, entre outras coisas, que processos sejam suspensos por um intervalo especificado de tempo.

Acima da camada do relógio está o nível de rotinas de I/O independente de dispositivo que fornece serviços como ler e escrever. Acima das rotinas de dispositivo, está o nível que implementa a comunicação entre máquinas, e um nível acima desta implementa um sistema de arquivos. Programas de aplicação interagem com o nível conceitual mais alto da hierarquia, mas eles podem chamar diretamente rotinas das camadas inferiores. Essa estruturação em níveis descreve apenas a implementação e não o fluxo de controle [7].

Evidentemente, os componentes são organizados em níveis para que o *design* e a implementação interna do sistema torne-se mais simples. A organização interna do sistema não deve ser confundida com os serviços que o sistema oferece pois, após a construção do sistema, as estruturas de todos os níveis da hierarquia são expostas aos programas aplicativos.

2.1.2 Memória

O sistema XINU inclui gerenciador de memória de alto e baixo nível. O gerenciador de baixo nível é utilizado para alocar e liberar a região do *heap* ou da pilha de processos, e de alto nível para criar *buffer pools*, sendo que cada *pool* (ou espaço) contém um conjunto de *buffers* de tamanhos fixos.

Um dos principais recursos que o SO gerencia é a memória principal. Para tanto são mantidas informações sobre tamanho e local de memória disponível e alocada. A memória alocada para um processo é recuperada e torna-se disponível quando o processo que a utilizava termina. São duas as categorias de memória, *Stable storage* (armazenamento estável) e *Random Access Memory* (RAM) que são alocadas em regiões de memória separadas [7].

O sistema operacional XINU inicia um único processo e em seguida permite que esse processo crie segmentos de memória adicionais para que todos os demais processos criados executem no mesmo espaço de endereço. O compilador C que utilizamos, emprega quatro regiões contínuas de memória: *Text segment, Data segment, Block Started by Symbol segment* (BSS) e *Free space*.

O segmento de texto inicia no endereço de memória mais baixo, contém código compilado para cada uma das funções que fazem parte da imagem de memória, incluindo a função *main*. O segmento de dados, que situa-se em endereço acima do segmento de texto, armazena todas as variáveis globais que têm atribuídas à elas um valor inicial. O segmento BSS, que está no endereço acima da região de dados, contém variáveis globais que não foram inicializadas explicitamente.

O sistema XINU grava o valor zero em cada posição antes de começar a execução. Acima da região do BSS, os endereços restantes são considerados livres *Free space*.

O carregador define três símbolos externos, *etext, edata, end*, que correspondem ao primeiro local de memória após o segmento de texto, primeiro endereço de memória após o segmento de dados e a primeira localização de memória além do segmento BSS, respectivamente [7].

O gerenciador de memória de baixo nível utiliza um conjunto de funções e estruturas de dados para gerenciar a memória livre. São cinco as funções: *getstk* que aloca espaço de pilha quando um processo é criado; *freestk* que libera pilha quando um processo termina; *getmem* que aloca armazenamento no *heap* sob demanda; *freemem* que libera armazenamento *heap* conforme solicitado; e *meminit* que inicializa a lista de espaços livres.

O texto do programa e variáveis globais ainda podem ser alocados dinamicamente para *stack* e *heap*.

Stack Cada processo precisa de espaço para a sua pilha que contém registro de ativação associado a cada função invocada pelo processo. Um registro de ativação possui armazenamento para variáveis locais.

Heap Um ou mais processos podem utilizar armazenamento na *heap*. Os itens na *heap* são alocados dinamicamente e persistem independentemente de chamadas de funções específicas.

O sistema XINU comporta os dois tipos de memória dinâmica descritas acima. Primeiro ao criar um novo processo, a pilha é alocadas a partir do endereço livre mais alto no espaço livre de memória que acomoda o pedido, e segundo, quando um processo requer 'armazenamento *heap*', o XINU aloca uma quantidade de espaço necessário do bloco livre de endereço mais baixo [8].

2.1.3 Processos

Um programa carregado na memória e em execução é um processo [18]. No sistema operacional XINU, um processo inicia a execução no começo do programa principal do usuário, quando o sistema é inicializado. Esse processo inicial pode continuar a execução por si mesmo, ou pode criar processos novos e independentes. Quando um processo cria um novo, o original continua a executar, e os dois processos passam a executar concorrentemente.

Duas funções do SO são utilizadas para criar e recomeçar processos:
*create(*funcao, tamPilha, priori, nomeDoProces, numDeArgs, arg1, arg2, ...);*
resume(id);

Cada chamada de *create* cria um novo processo, que começará a executar as instruções no endereço especificado por seu primeiro argumento. A *create* estabelece o processo, deixando-o pronto para executar, mas temporariamente suspenso, e devolve o identificador (**id**) do novo processo. O **id** é um número inteiro que identifica o processo criado, para que seu criador possa referenciá-lo posteriormente.

A chamada *resume* começa (ativa) o processo, de forma que ele começa a ser executado, seu argumento é um identificador de processo.

No sistema operacional XINU, mais de um processo pode executar um mesmo código pois o sistema aloca um conjunto independente de variáveis locais para cada processo.

Como exemplo, seja *main* o programa principal do SO XINU. Utilizamos duas chamadas *create* para começar dois novos processos. Ambos vão executar a função **enviaCh**, que recebe um caractere e o emite utilizando *kprintf*. No primeiro processo é passado o valor 'A' como argumento, e no segundo processo o valor 'B'. Embora executem o mesmo código, os dois processos seguem concorrentemente, sem interferir um com o outro, já que cada um possui suas cópias privadas das variáveis locais. Assim, um processo emite As enquanto o outro emite Bs.

Implementação do exemplo da criação de dois processos na função *main*:

```
/************************************************/
* main - Criação de processos com create e resume. *
************************************************/
#include <xinu.h>

void enviaCh(char);

void main(int argc, char **argv) {
      resume(create(enviaCh, 4096, 50, "Envio A", 1, 'A'));
      resume(create(envioCh, 4096, 50, "Envio B", 1, 'B'));
}
```

```
/************************************************/
* enviaCh - Emite caractere recebido indefinidamente. *
************************************************/
void enviaCh( char ch) {
      while (TRUE)
            kprintf(ch);
}
```

No sistema XINU há uma grande economia de memória e de tempo de processamento para a criação de processos, porque esses processos são *threads* (ou processos leves). O término de uma *thread* ocorre quando é alcançado o final de um procedimento, ou com uma rotina de sistema chamada *kill*. Essa rotina tem um argumento, que é o **id** do processo a ser terminado imediatamente.

Estritamente falado, em termos de implementação, cada processo leve no XINU tem sua própria pilha com suas variáveis locais, parâmetros e eventuais registros de ativação de chamadas de procedimentos. Todos compartilham as variáveis globais e cada processo tem seu próprio contexto de trabalho. Desta forma, na passagem de um processo para outro, é necessário salvar as variáveis que não são compartilhadas e que pertencem apenas àquele processo. A essa alternância entre processos, com proteção dos elementos de cada processo, se dá o nome de comutação (ou troca) de contexto de processos [8].

A troca de contexto consiste em interromper a execução do processo corrente, guardar informação suficiente para que esse processo interrompido agora possa ser reiniciado depois, e recomeçar um outro processo que estava aguardando para utilizar a CPU.

2.1.4 Tabela de processos

Para manter todas as informações sobre processos para poder recuperá-las futuramente, o SO utiliza uma estrutura de dados denominada tabela de processos (*proctab*). O índice da *proctab* é o próprio identificador do processo, porque a localização de um processo se torna mais eficiente desta forma. A tabela contém uma entrada para cada processo existente, e essa entrada é preenchida no momento da criação do processo, e invalidada quando o processo termina. Cada entrada em *proctab* consiste de uma estrutura do tipo *procent* que define que informação será mantida por processo [8].

As informações encontradas em uma entrada na *proctab* é mostrada na tabela 2.1. O SO deve salvar todas as variáveis que são destruídas na execução de outro processo. Como o conteúdo do registrador da pilha e registradores de propósito geral podem ser alterados, esses devem ser armazenados. A tabela de processos do sistema operacional XINU é um vetor com espaço para até NPROC, que é número limite de processos ativos [8].

Além dos valores a partir do hardware, o SO ainda mantém meta-informações de cada processo, na tabela de processos. Elas são utilizadas por algumas funções do sistema operacional, como para a contabilidade dos recursos, prevenção de erros e demais tarefas administrativas [7].

Muitas rotinas que manipulam os processos se baseiam no estado em que o processo se encontra. O estado (*prstate*) é uma informação do processo armazenada na tabela de processos e é através do estado do processo que o sistema consegue saber o que o processo está fazendo, bem como valida a semântica das operações executadas pelo processo naquele estado.

Campo	Propósito
prstate	Estado atual do processo.
prprio	Prioridade de escalonamento do processo.
prstkptr	Valor salvo do ponteiro de pilha do processo quando o processo não está executando.
prstkbase	Endereço da posição de memória mais elevada na região de memória utilizada como pilha do processo.
prstklen	Limite para o tamanho máximo que a pilha do processo pode crescer.
prname	Nome atribuído ao processo.

Tabela 2.1: Informações mantidas na tabela de processos do XINU.

2.1.5 Estados do processo

O sistema XINU define sete estados válidos, atribui uma constante simbólica para cada um deles, e define uma constante adicional, que serve para informar se a entrada na tabela de processos não está sendo utilizada [8]. As constantes simbólicas e seus significados são descritos na tabela 2.2.

Constante	Significado
PR_FREE	Entrada na tabela de processos livre. (Não representa um estado)
PR_CURR	O processo está atualmente executando.
PR_READY	O processo está elegível (pronto) para executar.
PR_RECV	O processo está esperando por uma mensagem.
PR_SLEEP	O processo está esperando por um determinado tempo.
PR_SUSP	O processo está suspenso.
PR_WAIT	O processo está esperando em um semáforo.
PR_RECTIM	O processo está esperando por um tempo ou uma mensagem, o que ocorrer primeiro.

Tabela 2.2: Estados de um processo e indicação de uso de entrada na *tabproc*.

Para haver a troca de um processo para outro processo, seleciona-se um processo entre os que estão no estado PR_READY, e dá-se o controle do processador para o processo selecionado. A política para a escolha do processo, é feita por *software*, e recebe o nome de Escalonamento de Processos [8].

A política de escalonamento estabelece que o processo de maior prioridade será escolhido para executar, e entre processos com mesmo valor de prioridade é utilizada a política *Round-Robin*[2]. No sistema XINU, a função *resched* faz a escolha de acordo com a política de escalonamento [8].

2.1.6 Prioridade de processos

A prioridade de um processo é um número inteiro positivo. Além de ser armazenada na tabela de processos, a prioridade é também utilizada para ordenar os processos que serão armazenados na lista de processos prontos para executar. Como a ordem de armazenamento

[2]Mais de um processo com mesma prioridade. RR garante que todos os processos irão receber, um após o outro, o serviço solicitado, para que todos tenham uma oportunidade de executar. Leva em conta preempção para realizar comutação entre processos.

nesta lista é descendente, o processo de maior prioridade estará na primeira posição (cabeça) dessa lista.

O projeto Xinu implementa uma outra política para o acesso ao processo corrente, porque ele não aparece na lista de prontos [8]. Assim, para fornecer acesso rápido ao processo corrente, o seu identificador é armazenado em *currpid*, que é uma variável inteira global. O escalonador não recebe um argumento explícito que especifica a disposição do processo corrente, em vez disso, as funções do sistema usam um argumento implícito: se o processo corrente não deve permanecer no estado pronto, antes de chamar a função *resched*, o campo *prstate* do processo atual deve ser atualizado para o próximo estado desejado [8].

Sempre que se prepara para trocar um processo, *resched* verifica o campo *prstate* do processo corrente. Se o estado for a constante PR_CURR e a prioridade desse processo corrente for a maior prioridade do sistema, *resched* retorna imediatamente e esse processo corrente permanece em execução. Porém, se a prioridade não for a maior, *resched* considera que o processo deve estar no estado pronto, e o move para a lista de processos prontos. *Resched* em seguida, remove um processo do início da lista de processos prontos (o processo com maior prioridade), e faz a troca de contexto.

2.1.7 Preempção

O sistema pode conter muitos processos com mesmo valor de prioridade, assim, enquanto um processo está em execução, outro processo de igual prioridade está na lista de processos prontos, elegível para o uso do processador. Com a chamada de *resched*, o processo corrente vai para a lista de prontos, na posição logo após os processos de mesma prioridade, e a execução reinicia no primeiro processo da lista de prontos. Dessa forma, todos os processos de igual prioridade, que estão prontos para usar o processador, executarão por uma 'fatia de tempo', um após o outro. Utilizado pelo gerenciador de processos, o mecanismo de preempção serve para implementar o *time slicing* ('fatia' de tempo), que garante que processos com igual prioridade recebam serviço (Round-Robin), conforme especificado pela política de escalonamento no sistema [8].

Resched seleciona um novo processo para executar, atualiza a entrada da tabela de processos com o novo processo, remove esse processo da lista de prontos, marca-o como corrente, e atualiza o *currpid*. Essa função ainda redefine o contador de preempção e finalmente chama a função *assembly ctxsw*, para salvar os registradores do *hardware* do processo corrente e restaurar os registradores do novo processo que irá executar.

Para a preempção é definido uma constante QUANTUM, que especifica o número de tiques do relógio para uma fatia de tempo em que o processo executa [18]. Sempre que há a troca de processos, *resched* reinicializa a variável global *preempt* para QUANTUM [8]. A cada tique do relógio, o gerenciador de interrupção do relógio decrementa o valor de *preempt*, quando chega a zero, o gerenciador recompõe *preempt* para QUATUM, chama *resched* e retorna da interrupção. A preempção garante que um processo não utilize o processador infinitamente e ainda força a política Round-Robin de serviços permitindo que todos os processos tenham chances de recebê-los.

2.1.8 Gerenciamento de tempo - *clock*

A maior parte dos sistemas operacionais fornecem mecanismos que permitem que uma aplicação crie ou gerencie eventos programados (*timed events*). Os eventos sob o paradigma assíncrono são de responsabilidade do programador, ele é quem define uma série

de manipuladores de evento e o SO invoca o manipulador apropriado quando um evento ocorre. Para eventos sob o paradigma síncrono, nos quais o sistema operacional somente fornece *delay*, o programador cria processos extras para agendar eventos. O sistema XINU utiliza a abordagem síncrona.

Relacionados com tempo, há quatro tipos de dispositivos de *hardware* [8]: *processor clock*, *time-of-day clock*, *interval timer* e *real-time clock*.

Processor clock: O *clock* do processador refere-se a um dispositivo de *hardware* que emite ondas quadradas (ou pulsos) em intervalos regulares com alta precisão. O relógio do processador controla a taxa na qual o processador executa as instruções.

Real-time clock: O relógio de tempo real opera independentemente do processador, e seu pulso é um número inteiro de vezes a cada segundo (por exemplo 1000 ciclos por segundo). O relógio de tempo real, a cada pulso, força a CPU a processar uma interrupção. Este relógio não contém contador e não acumula interrupções. A responsabilidade de contar os pulsos é do sistema. Quando a CPU leva muito tempo para atender a uma interrupção do relógio de tempo real, ou se a CPU opera com as interrupções desabilitadas, por mais de um ciclo do relógio, ela perderá a interrupção. Por esse motivo, os sistemas devem ser projetados para atender rapidamente às interrupções do relógio. Atribuir maior prioridade às interrupções do relógio é uma solução para não haver essa perda de interrupção.

Time-of-day clock: Como um relógio digital de pulso, um relógio de hora do dia é um cronômetro que calcula o tempo decorrido desde que foi configurado. O seu *hardware* consiste de um relógio de tempo real conectado a um contador que acumula os pulsos. Uma vez definido, o mecanismo funciona independente do processador, enquanto receber energia não há necessidade de programas privilegiados escreverem no contador para acertar a hora. O relógio de hora do dia não gera interrupção, e os programas leem o contador desse relógio para determinar a hora e data correntes.

Interval timer: Um temporizador de intervalo (ou contador decrescente) consiste de um relógio de tempo real e um contador. Para utilizar o temporizador de intervalo, o sistema inicializa o contador com um valor positivo. A cada pulso do relógio de tempo real, o contador do temporizador é decrementado e quando chega ao valor zero, é gerado uma interrupção. Pode-se utilizar esse temporizador como um contador crescente, dessa forma o contador é inicializado com o valor zero e a cada pulso do relógio de tempo real é incrementado o valor do contador. Quando atingir um valor configurado como limite para o contador, é gerada uma interrupção.

Os sistemas operacionais utilizam o relógio de tempo real para limitar a quantidade de tempo que um processo pode executar e para fornecer serviços como intervalo temporizado e hora e data atuais. O sistema mantém uma lista de eventos, ordenados de acordo com a hora em que devem ocorrer. Quando o relógio de tempo real interrompe, o sistema examina essa lista de eventos, e inicia o evento para o qual a espera acabou. O sistema XINU, permite que dois tipos de eventos sejam programados para o futuro. Quando transfere o serviço do processador para um processo, o sistema programa um evento de preempção para evitar que o processo execute infinitamente. Quando um processo solicita um intervalo temporizado, o sistema XINU remove esse processo do estado corrente, e programa um evento de *wake up*, para reiniciar o processo depois do número apropriado de pulsos do relógio. O sistema insere esse processo, que está com evento programado, na lista de processos 'adormecidos'.

Para os processos 'adormecidos', o sistema mantém uma lista denominada lista *Delta*. A lista Delta é ordenada pela diferença do intervalo temporizado que os processos precisam aguardar, de acordo com a hora que precisam ser 'acordados'. Dessa forma, o

primeiro processo na lista Delta é o processo com menor tempo de espera. O estado dos processos que estão na lista Delta é PR_SLEEP [8].

2.1.9 Sincronização

O mecanismo de sincronização de processos deve ser projetado com muito cuidado em qualquer SO porque, em um sistema com um único processador, nenhum processo, enquanto estiver esperando por outro, deve utilizar a CPU. Quando um processo está executando instruções enquanto espera por outro ele é dito estar em espera ativa, ou espera ocupada [19].

O XINU evita espera ocupada, fornecendo primitivas de coordenação de processos conhecidas como **semáforos**, e chamadas de sistema que operam sobre eles. Cada semáforo é constituído por um valor inteiro, inicializado quando o semáforo é criado. Existem duas chamadas que atuam sobre os semáforos, *wait* que decrementa um semáforo, causando bloqueio do processo, se o resultado for negativo, e *signal* que incrementa o semáforo permitindo que um processo que estava em espera continue. A atomicidade é implementada ao desabilitar as interrupções.

Os semáforos são criados com a chamada de sistema *semcreate*, que toma o valor inicial desejado como argumento e retorna um inteiro através do qual o semáforo pode ser manipulado.

Semáforos servem também para realizar exclusão mútua no acesso a um recurso por mais de um processo. Estes são conhecidos como (*mutexes*) e são usados em programação concorrente para evitar que dois processos (ou *threads*) tenham acesso simultâneo a uma região crítica [18]. Processos participam de exclusão mútua quando cooperam de forma que somente um dos processos tenha acesso a um recurso, por um determinado intervalo de tempo. Deve-se utilizar *mutexes*, por exemplo, quando vários processos tentam inserir itens em uma lista encadeada simultaneamente.

Para se excluírem mutuamente, no uso de um recurso, os processos devem criar um semáforo (com valor inicial 1) e antes de ter acesso ao recurso, cada processo deve executar uma chamada *wait* com o número do semáforo, e após utilizar o recurso, o processo deve executar *signal*, para liberar outros processos que estejam a esperar pelo recurso.

2.1.10 Transferência de mensagens

A 'Transferência de mensagens' (*message passing*) é uma forma de comunicação entre processos na qual um processo solicita ao SO que envie dados. O XINU suporta duas formas de transferência de mensagens, mensagens entregues de um processo diretamente para outro, e mensagens que são colocadas e retiradas em 'pontos de captação'. Nessa segunda forma, os processos depositam e recuperam mensagens nas *mailboxes* (ou caixas postais) [8].

Muitos sistemas operacionais utilizam a transferência de mensagem como base de toda a comunicação e coordenação entre processos. Por exemplo, operações como mandar dados para um terminal ou, através de uma rede para outra máquina, podem ser projetadas sobre as primitivas de transferência de mensagens.

As mensagens fornecem uma forma de coordenar processos porque o receptor pode esperar até a mensagem chegar. Dessa forma, a transferência de mensagens pode substituir a suspensão e o recomeço de processos. Dependendo da implementação, a transferência de mensagem pode substituir até primitivas de sincronização tais como semáforos.

A transferência de mensagem pode ser implementada de duas formas, síncrona e assíncrona. Na síncrona, se um receptor tenta receber uma mensagem antes que essa mensagem chegue, ele se bloqueia. Se um emissor tenta enviar uma mensagem antes do receptor estar preparado, o emissor se bloqueia. Os processos de envio e recepção devem ser coordenados, ou um pode ser bloqueado enquanto espera pelo outro, o que causaria um bloqueio permanente (*deadlock*) [8].

Na implementação assíncrona, uma mensagem pode chegar a qualquer momento, e o receptor é notificado. Um receptor não precisa saber com antecedência quantas mensagens irão chegar ou quantos emissores irão enviar mensagens.

Quando um processo manda mensagem diretamente para outro processo, ele precisa conhecer o destino da mensagem. No sistema operacional XINU, esse modo de transferência de mensagem foi cuidadosamente desenvolvido para que os processos não bloqueiem enquanto mandam mensagens, e que as mensagens em espera não ocupem toda a memória disponível. Assim, as mensagens têm o tamanho de um inteiro (32 bits), e as caixas postais, além do inteiro, possuem um semáforo para controle da retirada da mensagem da caixa (recepção) e outro semáforo para controle do envio.

As mensagens no XINU são manipuladas por três chamada de sistema, *send, receive* e *recvclr*. *Send* possui uma mensagem e o id de um processo como argumentos, e envia a mensagem para o processo especificado. *Receive*, sem argumentos, espera pela chegada de uma mensagem, e então devolve a mensagem para o processo que a chamou. *Recvclr* nunca espera pela chegada de uma mensagem; se o processo corrente tem uma mensagem quando chama *recvclr*, a chamada devolve exatamente como o *receive*. Caso nenhuma mensagem seja esperada, *recvclr* devolve 'OK' para o processo que a chamou, sem esperar que chegue uma mensagem. Como seu nome *recvclr* induz, essa chamada é usada para limpar as mensagens antigas que podem estar esperando. Os processo esperam por mensagens no estado *receiving*, representado pela constante PR_RECTIM [8].

2.1.11 Sistema de arquivos

O sistema de arquivos fornece operações para criar ou excluir um arquivo, abrir um arquivo usando seu nome, ler um arquivo aberto, gravar um objeto de dados em um arquivo ou fechar um arquivo [8]. A interface do sistema de arquivos também pode fornecer uma forma de um processo procurar uma posição específica em um arquivo caso o sistema de arquivos permita um acesso aleatório. Além de uma interface de programação, o sistema de arquivos pode oferecer um espaço abstrato de nomes e de alto nível para manipular objetos dentro desse espaço. Esse, denominado *namespace* de arquivos, consiste de um conjunto de nomes válidos para arquivos.

A semântica de arquivos do XINU é similar à do UNIX, de acordo com o princípio de que cada arquivo é uma sequência de *bytes*. Isso oferece muitas vantagens, dentre elas a de que não é necessário distinguir tipos de arquivos, e o conteúdo de um arquivo é independente de processador ou da tecnologia de memória (disco/*FLASH*) em que é mantido [8].

No sistema XINU, o sistema de arquivos suporta as funções *init, open, getc, read, putc, write, seek, close* e *control* para trabalhar com arquivos [8]. *Init* inicializa estruturas de dados na inicialização. *Open* é utilizada para abrir um arquivo com nome e conectar um processo de execução com os dados no disco e estabelecer o primeiro *byte*. As operações *getc* e *read* recuperam dados do arquivo e avançam o ponteiro; *getc* recupera um *byte* e *read* pode recuperar vários *bytes*. *Putc* e *write* alteram *bytes* no arquivo e movem o

ponteiro estendendo o tamanho desse arquivo a medida que novo dado é inserido; *putc* altera um *byte* e *write* pode alterar vários *bytes*. A operação *seek* move o ponteiro para uma posição (de *byte*) especificada do arquivo, o primeiro *byte* está na posição zero. E por final, a função *close* remove 'a ligação' do processo em execução com o arquivo, deixando os dados no arquivo em um armazenamento permanente [8].

O sistema de arquivos não é testado neste trabalho pois o cMIPS ainda não o possui. Está entre as indicações de trabalhos futuros.

2.1.12 Inicialização do sistema XINU

O *hardware* trata de algumas tarefas básicas de inicialização e os programas de *bootstrap* tratam das mais complexas. O programa de inicialização cria uma lista de blocos de memória disponíveis e coloca a lista na memória, tornando-a disponível para o SO XINU. Embora haja a inicialização de baixo nível antes do SO inicializar, é necessário ainda haver uma função para a inicialização de linguagem de montagem. O sistema XINU precisa estabelecer o ambiente de tempo de execução adequado para linguagem C, assim a execução começa no início do rótulo *start* que está no arquivo start.S do diretório XINU. Neste arquivo de código, define-se um inicial *stack pointer* (ponteiro de pilha), habilita-se a *cache* de instruções e faz-se, por último, uma chamada para a função *nulluser*. A partir deste ponto, um único programa está em execução [8].

A função *nulluser* é que inicializa o SO: cria um processo para executar a função principal e 'torna-se' o processo nulo.

Como *nulluser* se tornou o processo 0, esse não pode terminar, dormir, esperar um semáforo ou suspender-se. Uma vez que a inicialização é concluída e um processo foi criado para executar o programa principal, o processo 0 permanece em um loop infinito, chamando *resched* quando nenhum processo do usuário estiver pronto para ser executado. E por final, após preenchimento da entrada na tabela de processos com processo 0, é definido a variável *currpid* para zero, que transforma o programa em um processo [8].

Capítulo 3

Plataforma de hardware

O cMIPS é uma implementação do MIPS, desta forma abordaremos o processador MIPS inicialmente para então entrarmos no cMIPS.

3.1 MIPS

O processador MIPS *(Microprocessor without Interlocked Pipe Stages)* foi projetado na década de 80, na Universidade de Standford, pelo grupo de John L. Hennessy [16, 11]. Sua fabricação começou quatro anos mais tarde, pela *Silicon Graphics Inc.* e atualmente é fabricado pela *Imagination Technologies* [2] além de ser fabricado sob licença por várias outras empresas.

A arquitetura MIPS segue a filosofia RISC *(Reduced Instruction Set Computers)* e seu modelo de execução, juntamente com o conjunto de instruções, foi gerado a partir de um processador segmentado de cinco estágios como mostrado na Figura 3.1, no qual cada instrução somente efetua uma operação simples em cada segmento.

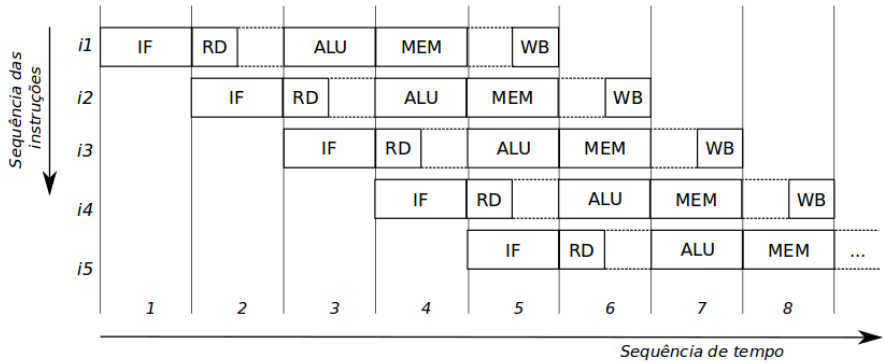


Figura 3.1: *Pipeline* de cinco estágios do MIPS.

No *pipeline* da Figura 3.1, no estágio **IF** - *Instruction Fetch* - o processador busca a próxima instrução na memória de instruções; em **RD** - *Read Registers* decodifica a instrução e busca operandos no banco de registradores; em **ALU** - *Arithmetic and Logic Unit* executa as operações aritméticas e lógicas sobre os operandos; em **MEM** - *Memory* acessa operandos na memória *cache* de dados; e em **WB** - *Write Back* grava o resultado no banco de registradores.

3.1.1 Registradores

No banco de registradores há trinta e dois os registradores de uso geral *(General-Purpose Register)* chamados de $0, $1, $2, ..., $31 [10].

Todas as versões de 32 bits existentes da arquitetura MIPS possuem o mesmo conjunto básico de instruções de 32 bits [10], definido em [15, 5]. Todas as instruções têm tamanhos iguais. Emprega-se uma convenção de *software* para a nome e funcionalidade de cada registrador, e estas são mostrados na Tabela 3.1.

Tabela 3.1: Registradores do MIPS.

Reg.	**Nome**	**Descrição**
0	zero	Sempre contém o valor zero.
1	at	*Assembler Temporary*: usado pelo montador.
2..3	v0..v1	Valor de retorno de uma chamada de função.
4..7	a0..a3	Primeiros argumentos para uma chamada de função.
8..15	t0..t7	Variáveis temporárias; não precisam ser preservadas.
16..23	s0..s7	Variáveis salvadas; precisam ser preservadas.
24..25	t8..t9	Variáveis temporárias.
26..27	k0..k1	Para uso do sistema operacional.
28	gp	*Global Pointer*.
29	sp	*Stack Pointer*.
30	fp	Apontador para registro de ativação ou variáveis de sub-rotina.
31	ra	Endereço de retorno da última chamada de sub-rotina.

São três os registradores especiais:

- *Multiply and Divide Register Higher result (HI)*;
- *Multiply and Divide Register Lower result (LO)*; e
- *Program Counter (PC)*.

As instruções de multiplicação e divisão produzem um resultado em 64 bits e armazenam seus resultados nos registradores especiais **HI** e **LO**.

Na multiplicação é gerado um produto de tamanho duas vezes o tamanho dos operandos de entrada; a metade menos significativa do resultado é colocada em **LO** e a mais significativa em **HI**. Na divisão é gerado um quociente que é armazenado em **LO** e o resto, armazenado em **HI**. Os resultados da multiplicação e divisão são acessados através de instruções de transferência de dados, entre **HI**/**LO** e os registradores de uso geral. Para tanto são utilizadas as instruções **mfhi** e **mflo**.

O registrador especial **PC** indica o endereço da próxima instrução a ser executada.

O processador MIPS suporta *bytes* (8 bits), *halfwords* (16 bits), *words* (32 bits) e *doublewords* (64 bits, em ponto flutuante).

Para instruções de memória, os tipos de dados inteiros nas instruções *load* e *store* são *bytes*, meia palavra e palavra. Nas operações aritméticas e lógicas, os tipos inteiros são meia palavra (estendida para 32 bits) e palavra.

Os modos de endereçamento são cinco:

- Endereçamento a registrador, no qual os operandos estão em registradores;

- endereçamento imediato, em que um operando na própria instrução é uma constante;
- endereçamento base-deslocamento, em que o endereço efetivo é a soma do conteúdo de um registrador (base) com de um imediato (deslocamento) de 16 bits;
- endereçamento relativo ao **PC**, no qual o endereço de destino é a soma de um imediato de 16 bits (multiplicado por 4) com o **PC**; e
- endereçamento pseudo-absoluto, em que o endereço efetivo é formado pela concatenação de um imediato de 26 bits (multiplicado por 4) com os 4 bits mais significativos do **PC**.

3.1.2 Instruções

Os formatos de instrução são três: **I**, **R** e **J**, e são apresentados na Figura 3.2. O tipo da instrução é codificado no primeiro campo, *opcode*, em 6 bits e todas as instruções têm 32 bits [10].

O formato do tipo **I** reserva 16 bits para o imediato, e campos para registradores fonte (**rs**) e destino (**rt**).

O formato do tipo **R** emprega dois registradores fonte (**rs** e **rt**) e um destino (**rd**), um campo (**shamt**) de 5 bits para instruções de deslocamento, e 6 bits que estendem o **opcode** para definir a função (**funct**).

O formato do tipo **J** contém um campo para o endereço pseudo-absoluto de 26 bits.

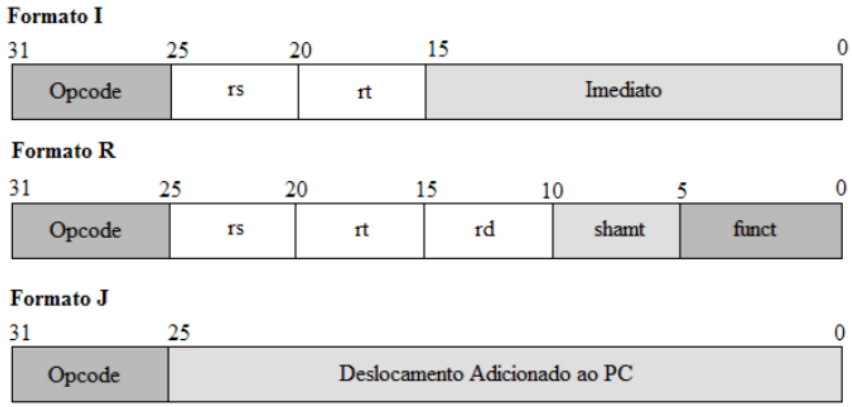


Figura 3.2: Formato das instruções do MIPS.

As instruções podem ser:

1. Instruções de memória, com variações *load* e *store* para:
   - *Words*: **lw/sw**
   - *Halfwords*: **lh/sh**
   - *Bytes*: **lb/sb**
2. Instruções de operações de ULA, que usam um imediato de 16 bits estendido com sinal ou com zeros completando 32 bits, o imediato faz parte da instrução;

3. instruções de desvios e saltos. Os saltos incondicionais são ditos pseudo-absolutos porque o destino é especificado em 26 bits, que apontam para uma instrução numa faixa de 256MB, ou 64M instruções. Esses 26 bits do destino são deslocados 2 bits para a esquerda e então concatenados com os 4 bits mais significativos do **PC**, para então produzir o endereço absoluto de 32 bits; e

4. instruções de operações com ponto flutuante, que não são discutidas neste trabalho.

Para as instruções de controle do processador ou de gerenciamento de memória são usados outros formatos.

O MIPS também possui instruções que permitem transferência de dados de diversos tamanhos, tratamento de dados carregados como inteiros com ou sem sinal, acesso a campos desalinhados e a atualização atômica da memória (*read-modify-write*) [17].

A instrução **jal** (*jump-and-link*) salta para uma função e armazena no registrador **r31** o endereço de retorno (**PC+4**). A instrução **jr** (*jump-register*) salta para um endereço apontado por um registrador, utilizada para o retorno de função (**jr r31**).

Os desvios condicionais são relativos ao **PC**. Não há registrador de *status* e as decisões são tomadas pela comparação de igualdade entre dois registradores, utilizando (**beq, bne, beqz, bnez**) (*branch if equal, not equal, equal zero, not equal zero*).

Desvios por comparação de magnitude necessitam duas instruções: **slt** (*set-less-than*), **bne** e equivalem a instrução *branch-less-than* (**ble**) [10].

A instrução ***syscall*** é para chamadas ao sistema operacional.

O processador de controle, co-processador 0 (CP0) executa as instruções que controlam o ambiente de *hardware* onde os programas de usuário são executados.

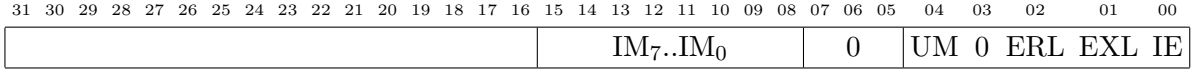
Para acessar os registradores do CP0, as instruções são **mfc0** (*move from coprocessor 0*) e **mtc0** (*move to coprocessor 0*). Ambas possuem dois argumentos, um dos 32 registradores de uso geral, e o número de um registrador do CP0.

3.1.3 Registradores do co-processador (CP0)

Principais registradores e suas funcionalidades no CP0.

COUNT & COMPARE O registrador COUNT é utilizado para para contar os ciclos do relógio e o registrador COMPARE armazena um número de 32 bits. Da comparação entre esses dois registradores é gerada uma interrupção periódica. O XINU utiliza COUNT e COMPARE para gerar as interrupções do relógio do sistema.

STATUS O registrador STATUS é utilizado para definir o modo de execução (usuário ou sistema), se as interrupções estão habilitadas, e ainda controla outras funcionalidades do processador. Pode ser alterado por *software*, é utilizado no XINU. Os bits *UM, ERL, EXL, IE* são alterados por *hardware* na ocorrência de uma exceção.



$IM_7..IM_0$ *interrupt mask*; se bit IM_i=1 então a interrupção de nível i está habilitada;

UM *user mode*, UM=1 indica modo usuário, UM=0 indica modo sistema;

ERL *at error level*, ERL=1 quando ocorre um dentre Reset, SoftReset, NMI ou CacheError;

EXL *at exception level*, EXL=1 quando ocorre uma exceção que não seja Reset, SoftReset, NMI ou CacheError. Se STATUS.EXL=1, as interrupções são ignoradas;

IE *interrupt enable*, IE=1 indica que as interrupções estão habilitadas.

CAUSE O registrador CAUSE indica a causa da última exceção ou interrupção. Os campos *TI, DC, IV* podem ser alterados por *software*, e os demais são atualizados, a cada ciclo do relógio por *hardware*.

Quando ocorre uma exceção, o campo *ExcCode* só é atualizado após uma leitura ocorrer, com `mfc0 r, c0_cause` ou `eret`(*return from exception*).

31	30	29	28	27	26	25	24	23	22 21 20 19 18 17 16	15 14 13 12 11 10 09 08	07	06 05 04 03 02	01 00
BD	TI	0	0	DC	0	0	0	IV		$IP_7..IP_0$	0	ExcCode	0

BD *in branch delay*, BD=1 indica que a última exceção ocorreu num *branch delay slot*;

TI *timer interrupt*, TI=1 indica que há uma interrupção do temporizador pendente;

DC *Disable Count register*, DC=1 desabilita o contador interno; DC=0 habilita o contador. Usado para reduzir dissipação de energia em aplicações de baixo consumo;

IV *interrupt uses general exception vector*, IV=0 para tratamento de interrupções como se fossem exceções (`0x0180`); IV=1 para o tratamento das interrupções ser desviado para o vetor de interrupções (`0x0200`) – este é o modo recomendado para o cMIPS;

$IP_7..IP_0$ interrupção pendente; se bit IP_j=1 então a interrupção de nível j está pendente;

ExcCode *exception code*, identifica a exceção, conforme a Tabela 3.2.

Binário	Dec.	Mnemônico	Causa da exceção
00000	0	Int	Interrupção
00001	1	Mod	Modificação de página (*store* em página protegida)
00010	2	TLBL	exceção da TLB (*load* ou busca)
00011	3	TLBS	exceção da TLB (*store*)
00100	4	AdEL	Erro de endereçamento (*load* ou busca)
00101	5	AdES	Erro de endereçamento (*store*)
00110	6	IBE	Erro no barramento de instruções
00111	7	DBE	Erro no barramento de dados
01000	8	Sys	*Syscall*
01001	9	Bp	*Breakpoint*
01010	10	RI	Instrução reservada (*opcode* inválido)
01100	12	Ov	*Overflow*
01101	13	Tr	*Trap*
11111	31	–	Nenhuma exceção pendente

Tabela 3.2: Códigos de exceção, bits 6 a 2 do registrador CAUSE.

EPC O registrador EPC contém o endereço da instrução que seria executada se não houvesse uma interrupção ou exceção.

CONFIG O registrador CONFIG mantém parâmetros de configuração das caches e processador.

LLaddr O registrador LLaddr mantém o endereço efetivo do último *load-linked* executado.

Uma implementação da arquitetura MIPS, seguindo essas características é o *classical* MIPS que será descrito adiante.

3.2 cMIPS

O *classical*MIPS [13] é um modelo sintetizável do processador MIPS clássico de cinco estágios, e implementa o conjunto de instruções do MIPS32r2 como descrito em [11]. O modelo foi sintetizado para um FPGA *Altera EP4CE30F23*, executa com 50MHz e utiliza 15% dos blocos combinacionais e 5% dos registradores do FPGA [13]. O cMIPS foi desenvolvido para ser um ambiente de experimentação realista para a disciplina Arquitetura de Computadores do Departamento de Informática da UFPR.

O desenvolvimento do cMIPS foi iniciado em 2012, seus módulos são escritos em VHDL e o modelo do processador executa códigos C compilado com o GCC. O código fonte completo está disponível em [12].

Para simulá-lo, emprega-se um arquivo de teste (*testbench*). A Figura 3.3 mostra um computador com o processador, *caches* de instruções e de dados, memória ROM e RAM e mais quatro periféricos. As caixas pontilhadas representam os arquivos de dados, usados ou gerados, nas simulações. Modelados no *testbench* estão o processador, RAM, ROM e arquivo de entrada e saída.

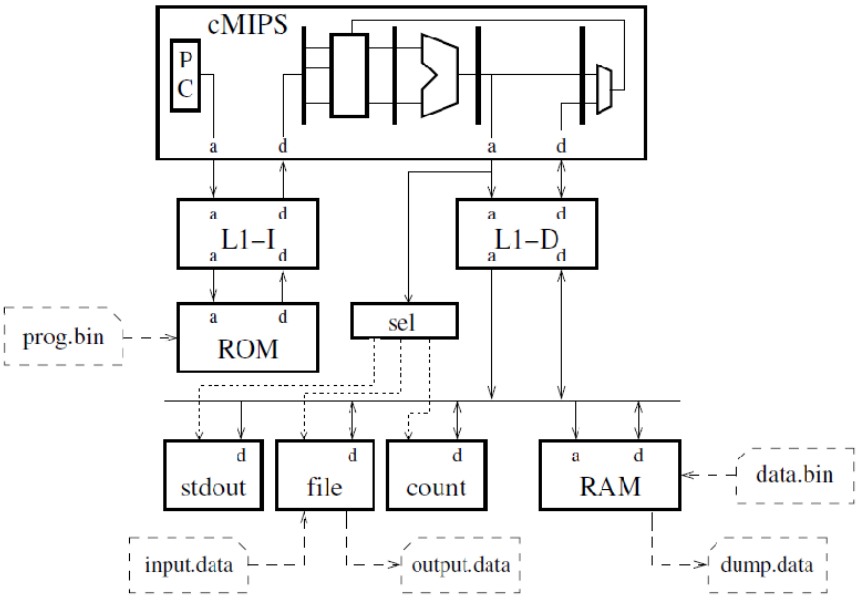


Figura 3.3: Diagrama de blocos do arquivo de testes do cMIPS.

O processador contém o circuito de dados e o CP0. A implementação de sistemas operacionais é possível, pois esse modelo contém toda a funcionalidade para suportar um sistema *Unix-like*. Possui instruções de controle que, utilizadas adequadamente,

possibilitam a execução no modo usuário ou sistema, e também dispõe de tratativas para exeções.

CP0 O CP0 implementa, no *Interrupt Compatibility Mode*, seis interrupções de *hardware*, duas interrupções de *software* e interrupções não-mascaráveis. No modelo do cMIPS ainda há uma unidade de gerenciamento de memória, *Memory Management Unit* (MMU), que permite implementar paginação sob demanda. Estes recursos são implementados conforme especificado em [14].

MMU A MMU permite a implementação de paginação sob demanda, ela compreende oito elementos de TLB totalmente associativa, e suporta páginas de quatro *Kbytes*.

Simulador cMIPS As memórias ROM e RAM são inicializadas a partir de arquivos gerados pelos *scripts* de compilação. A memória ROM é inicializada com **prog.bin** que contém o código de máquina gerado ao compilar o código de um programa escrito em C ou assembly. A memória RAM é inicializada com **data.bin**, e ambos são carregados na inicialização do processador, com o sinal de *reset* ativo. [1]

Em termos de dispositivos, o *testbench* do cMIPS contém três periféricos que possibilitam interação entre o sistema simulado e o mundo externo. O periférico no diagrama da Figura 3.3 representado por *stdout*, permite a escrita de um inteiro (quatro *bytes*) na saída padrão do simulador, formato hexadecimal, ou um caractere. Também é possível a leitura de um caractere da entrada padrão. O dispositivo *file* possibilita ler um inteiro do arquivo binário *input.data*, ou ainda escrevê-lo no arquivo *output.data*. O periférico denotado *count* pode ser utilizado para geração de interrupção após determinado número de ciclos do relógio.

Para compilar um programa escrito em C é necessário incluir o cabeçalho cMIPS.h, porque é neste arquivo que estão os endereços dos dispositivos e as funções necessárias para utilização dos dispositivos do arquivo de teste do cMIPS.

As interfaces para o controlador de SDRAM (*Synchronous Dynamic Random Access Memory*) e conexão *Ethernet* estão em desenvolvimento. Após aquisição de um kit de desenvolvimeto para testes, a UART *Universal Assynchronus Receiver Transmitter* será utilizada. Todos os testes funcionais do XINU-cMIPS foram realizados no simulador. A utilização de um FPGA será considerada futuramente, neste momento a memória interna do FPGA disponível não comporta o XINU.

3.3 Adaptações do XINU ao cMIPS

O sistema XINU foi adaptado e modificado para funcionar na plataforma do cMIPS. Para isso, excluímos componentes cujos correspondentes em *hardware* não existem no cMIPS e/ou que não atende às necessidades do simulador utilizado para testes neste trabalho, com a finalidade de simplificar a depuração das funcionalidades que o sistema operacional oferece dentro do limite da plataforma de *hardware*.

Foram removidos e/ou adaptados os testes dependentes de endereços do sistema XINU, o módulo TLB e suas exceções, algumas das variáveis não utilizadas, o driver *Ethernet* e de acesso TCP/UDP, e não contemplamos o *file system* porque não há modelo

[1] A versão de RAM e ROM além da simulada é a que suporta escrita de dados em display LCD, leitura de dados através de um teclado e uma interface serial UART.

para *hardware* que suporte um sistema de arquivos, tal como um disco rígido ou memória *FLASH* no cMIPS.

Foi modificado o tamanho dos índices em *qid* e as estruturas de dados foram alinhadas em 4xN *bytes*. Todas as variáveis globais foram inicializadas explicitamente, principalmente por motivo de redução de tempo de simulação. Os intervalos de valores para *clkhandler* e *sleep* foram definidos com valores pequenos para acelerar as simulações. A fila de processos prontos sempre é acessada pela primeira posição, a variável *newqueue* é inicializada com o valor NPROC.

Foi removido a inicialização do *heap*, o laço que inicializa a memória com zeros foi comentado para diminuir o tempo de simulação. Com isso, o tempo de simulação foi reduzido em 4 minutos o que equivale a uma redução de 66%.

Para verificação da corretude dos semáforos e suas filas, foi utilizado o gtkwave, que possibilitou a verificação de que as filas estavam sendo trocadas trazendo erradamente um processo. Também foi utilizado essa ferramenta para verificação da interrupção, onde estava acontecendo e se haveria a troca de conexto.

Com relação a códigos em assembly, modificamos a inicialização dos registradores internos ao processador.

Para que as interrupções de relógio do sistema XINU funcionassem foram empregados os registradores COUNT e COMPARE do MIPS que são implementadas no cMIPS.

o executável gerado é exposto na figura 3.4.

3.3.1 Memória

A versão original do sistema operacional XINU não possui memória virtual. A memória *cache* não está sendo utilizada porque não é implementada no cMIPS. Com relação à capacidade da memória, reduzimos o número de processos e o número de semáforos criados, já que o tempo de simulação aumenta proporcionalmente à capacidade da memória simulada. Número de processos é 10 e número de semáforos é 10.

3.3.2 Relógio de tempo real

O relógio de tempo real implementado está utilizando intervalos curtos no XINU-cMIPS. Para o escalonamento, o processo com maior prioridade no estado pronto é sempre o escolhido para utilizar o processador, e processos com valores iguais de prioridade são escalonados seguindo o critério de FIFO.

Clktime fornece a 'data' no XINU.

3.4 Testes

Os testes realizados servem para verificar as funcionalidades do sistema operacional XINU-cMPIS. Utilizamos *log* em tela para avaliar os resultados dos testes, e comparamos com testes de mesa a corretude da saída. É através do resultado de cada teste exibidos em tela que foi possível avaliar a corretude do porte e comprovar que o mesmo foi realizado corretamente. As seguintes funcionalidades são testadas e analisadas:

- Criação de processos, semáforos e mutexes;

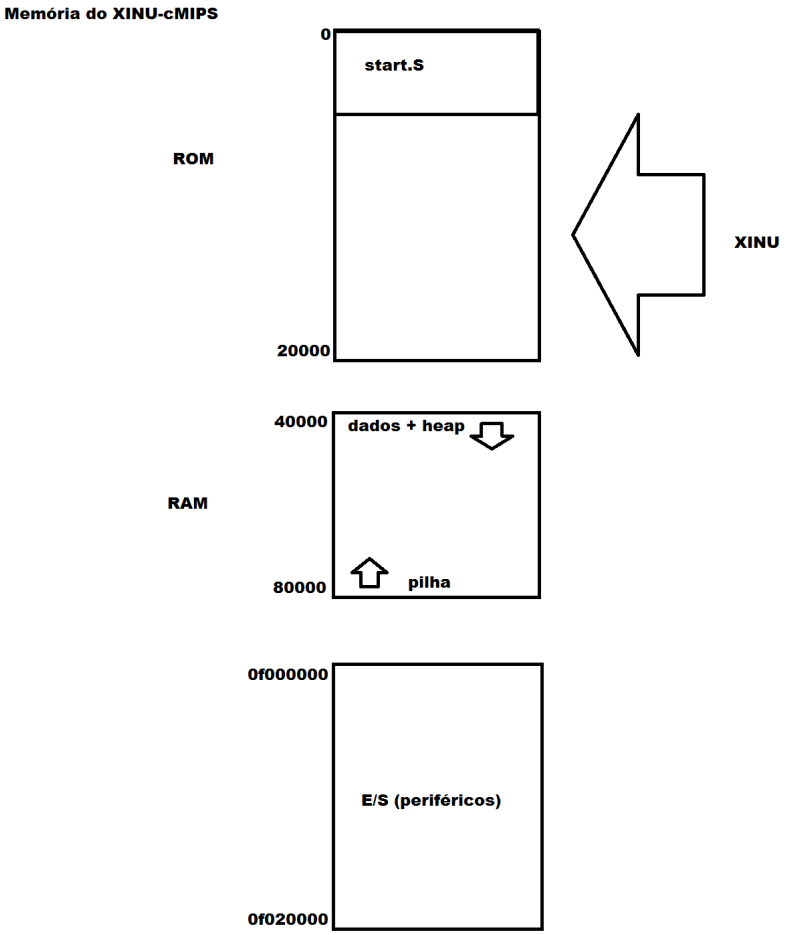


Figura 3.4: Memória do XINU-cMIPS.

- Alternância entre processos, para verificação do funcionamento do relógio e análise da troca de contexto entre processos;
- Utilização de semáforo binário, para testar os semáforos, as filas de espera e também se as prioridades dos processos são respeitadas;
- Semáforos contadores;
- Caixas postais, para verificar sincronização e comunicação entre processos.

São testadas as funções *receive, send, rcvclr, rcvtime, userret, yield, xdone, kill, semreset, semdelete, getmem, getstk, getprio, wakeup. Send, receive, recvclr* são as três chamadas de sistema utilizadas para a manipulação de mensagens, e são testadas com caixas postais.

Userret.c - esta função é chamada de um processo do usuário, retorna quando o processo termina.

Xdone - esta função envia uma mensagem de sistema de completude para o último processo que terminou. A *system call kill* termina o processo e chama esta função.

Semreset - é utilizada para inicializar um semáforo contador e libera um processo esperando.

Semdelete - é utilizada para desalocar um semáforo criado, apagando o semáforo criado.

Getmem e *freemem* - aloca espaço *heap* para o mais baixo endereço de espaço livre de memória.

Getstk e *freestk* - a função *create* chama *getstk* para alocar a pilha, utilizando o endereço mais alto do espaço livre de memória, retornando um ponteiro para esse bloco livre. *Freestk* libera a pilha do processo e retorna o bloco para a lista de espaços disponíveis.

Wakeup e *sleepq* - a função *wakeup* é chamada pelo gerenciador de interrupção de relógio para acordar os processos. Essa função utiliza a *sleepq* para remover o processo da lista de processos "dormindo"e prontos e deixa-o elegível para usar o processador.

Em seguida discutiremos os testes funcionais.

3.5 Programas de teste

Os programas que testam as funções *kill, getprio* são simples. Para o teste da primeira, um processo encerra ele mesmo, passando como parâmetro seu próprio identificador. A segunda função, retorna o valor da prioridade do processo, sendo esse o valor que foi atribuído na criação do processo.

Criação de processos e criação de semáforos A criação de processos, criação e inicialização de semáforos e mutexes são implementadas em todos os programas de teste.

Alternância entre processos O teste de alternar a execução de processos é para verificar se a troca de contexto ocorre corretamente quando há interrupção. A corretude da troca de contexto é evidenciada com este teste que utiliza três processos criados, um processo que utiliza uma função para calcular a sequência de Fibonacci para o valor n = 45 (limite para o cálculo da sequência, o valor do resultado que é comportado pelo inteiro) e imprime a sequência. O outro processo imprime o Fatorial para valores pares até o valor n = 12 (limitante do valor do resultado que o inteiro comporta) e o terceiro processo imprime infinitamente a palavra 'PrC'. Além da interação com a função *main*, é verificado que os processos executam corretamente, eles utilizam o mesmo código da função *kprintf* imprimindo seus valores e na troca de contexto os valores estão consistentes. É respeitado a prioridade de cada processo e o tempo que os processos estão adormecidos. A interrupção ocorre corretamente.

Pseudocódigo

```
main() {

      //Criação dos processos
      create(prA, 4096, 50, "Processo a", 0);
      resume(prA);
      create(prB, 4096, 50, "Processo a", 0);
      resume(prB);
      create(prC, 4096, 20, "Processo c", 0);
      resume(prC);
}

//Implementação dos processos
processo_A() {
      while(TRUE) {
```

```
            for(i<45)
                  kprintf("Fibonacci(i)");
      sleep(3); // Espera
      }
}

processo_B() {
      while(TRUE) {
            for( i<=12)
                  kprintf("Fatorial(par(i))");
      sleep(3); //Espera
      }
}

processo_C() {
      while(TRUE)
            kprintf("Processo c");
      sleep(0);
}
```

Resultado da alternância entre processos

```
            Processo A iniciou
            Fib(1) = 1
            Fib(2) = 1
            Fib(3) = 2
                  ... continua ...
            Fib(44) = 701408733
            Fat(0) = 1
            Fat(2) = 2
            Fat(4) = 24
            Fat(6) = 720
            Fat(8) = 40320
            Fat(10) = 3628800
            Fat(12) = 479001600
                  PrC.
                  PrC.
                  ... continua ...

                  PrC.
                   Main.
                  PrC.
                  PrC.
                  PrC.
                  ... continua ...

                  PrC.
                  Processo A iniciou
            Fib(1) = 1
            Fib(2) = 1
            Fib(3) = 2
                  ... continua ...

            Fib(44) = 701408733
            Fat(0) = 1
            Fat(2) = 2
            Fat(4) = 24
            Fat(6) = 720
```

```
            Fat(8) = 40320
            Fat(10) = 3628800
            Fat(12) = 479001600
                  PrC.
                  PrC.
                  ... continua ...

                  PrC.
                   Main.
                  PrC.
                  PrC.
                  PrC.
                  ... continua ...

                  PrC.
```

Semáforo binário Para o teste com semáforo binário são criados dois processos, dois semáforos inicializados com 0 e 1, respectivamente, e são passados como parâmetros para cada um dos processos criados. O processo denominado produtor é responsável por gerar um inteiro e informar que produziu o inteiro (imprimindo mensagem). O outro processo é o consumidor que é responsável por 'consumir' um valor inteiro, se já foi produzido. O problema neste teste consiste de que ambos não devem sobrepor-se um ao outro, de forma que o consumidor não tente consumir um valor que não foi produzido. Com os semáforos neste teste é possível controlar a produção e o consumo dos valores inteiros, confirmando a corretude.

Pseudocódigo

```
main() {

      //Criação e inicialização dos semáforos
      consumed = semcreate(0);
      produced = semcreate(1);

      //Criação dos processos
      resume(create(produtor, 4096, 20, "prod", 2, consumed, produced));
      resume(create(consumidor, 4096, 20, "cons", 2, consumed, produced));
}

//produtor
produtor(consumed, produced) {
      for(i<50) {
            wait(consumed);
            n++;
            signal(produced);
      }
}

//consumidor
consumidor(consumed, produced) {
      for(i<50) {
            wait(produced);
            signal(consumed);
      }
}
```

Resultado com semáforo binário

```
            Produtor iniciou...
            Consumidor iniciou...
                  Produzi n = 1
                  Consumi n = 1
                  Produzi n = 2
                  Consumi n = 2
                  ... continua ...

                  Produzi n = 49
                  Consumi n = 49
                  produzi n = 50
            Produtor terminou!
                  Consumi n = 50
            Consumidor terminou!
      All user processes have completed.
```

Semáforo contador Para o teste com os semáforos contadores, são criados os processos produtor e consumidor. São criados dois semáforos, um denominado 'cheio' e inicializado com o valor 0, e outro semáforo denominado 'vazio' e iniciado com o valor inicial N, com o valor 8 para esse teste. Na criação dos processsos esses semáforos são passados como parâmetros.

Pseudocódigo

```
main() {

      n = 0;
      N = 8;

      //Criação e inicialização dos semáforos
      cheio = semcreate(0);
      vazio = semcreate(N);

      //Criação de processos
      resume(create(prod2, 4096, 20, "prod", 2, cheio, vazio));
      resume(create(cons2, 4096, 20, "cons", 2, cheio, vazio));

}

//Implementação do processo produtor
prod(cheio, vazio) {
      for(i=0 ; i<50 ; i++) {
            wait(cheio);
            n++;
            signal(vazio);
      }
}

//Implementação do processo consumidor
cons(cheio, vazio) {
      for(i=0 ; i<50 ; i++) {
            wait(vazio);
            signal(cheio);
      }
}
```

Resultado do teste do semáforo contador O resultado da simulação para processos que utilizam o semáforo contator é idêntico ao resultado dos que utilizam semáforo binário.

Caixas postais São dois testes com as caixas postais, um que utiliza duas filas - consumo e produção, para os processos aguardarem até poderem utilizar o processador e consequentemente a caixa postal e um outro teste que utiliza apenas dois semáforos para o controle de acesso à *mailbox*.

O primeiro teste possui quatro produtores, dois consumidores e dois semáforos. Cada um destes utiliza uma fila para organizar os acessos à caixa postal. Para entrar na fila, o processo que solicita produzir, ou consumir, recebe uma 'senha' (inteiro) e vai esperar na fila de seu procedimento (produzir ou consumir). Quando a 'senha' é atualizada para poder utilizar a caixa postal, o processo é retirado da sua fila e utilizará a caixa postal - depositando ou retirando a sua mensagem. Após utilização da caixa postal atualiza-se a senha selecionada da outra fila e o processo termina o envio dessa mensagem.

O segundo teste com *mailboxes* é realizado de forma semelhante ao teste de processos produtor-consumidor, com processos que enviam e processos que recebem mensagens depositadas em um caixa postal e dois semáforos. O tamanho da mensagem é de um inteiro de 32 bits. Um dos semáforos controla a retirada de mensagens depositadas na caixa, impedindo que um processo tente receber mensagem de uma caixa vazia (bloqueando-o até que outro processo deposite alguma mensagem na caixa). O segundo semáforo controla o envio da mensagem, sincronizando o produtor com o consumidor. Se há mensagem na caixa, o processo que tentar deixar uma nova mensagem é bloqueado, evitando que haja somente produção de mensagens e assim permitindo que o consumidor processe mensagens pendentes. Essa estratégia é frequentemente conhecida como *rendezvous*. O teste utilizando caixa postal consiste de quatro processos que produzem valores inteiros e dois processos que consomem os valores inteiros produzidos, dois semáforos controlam o acesso à caixa postal. Neste teste, verificamos a sincronização e comunicação entre os processos através de *mailbox*.

Pseudocódigos Primeiro teste.

```
#define N 1
#define nprods 10
// 2x mais produtores que consumidores
#define ncons 20

int cxpostal[N]; //Um inteiro

//Estrutura para semafóro com senha
struct semaforo_fila {
      semaforo;
      int senha, prox_senha;
} semaforo_fila;

main() {
      semaforo_fila produz, consome;

      inicializa_semaforo_fila(&consome, 0); //trancado
      inicializa_semaforo_fila(&produz, 1);

      resume(create(prod1, 4096, 20, "prod1", 2, &consome, &produz));
```

```
      resume(create(prod2, 4096, 20, "prod2", 2, &consome, &produz));
      resume(create(prod3, 4096, 20, "prod3", 2, &consome, &produz));
      resume(create(prod4, 4096, 20, "prod4", 2, &consome, &produz));
      resume(create(cons1, 4096, 20, "cons1", 2, &consome, &produz));
      resume(create(cons2, 4096, 20, "cons2", 2, &consome, &produz));
      return OK;
}

inicializa_semaforo_fila(semaforo_fila *sf, int init) {
      sf->prox_senha = 1;
      if (init == 0) // inicia trancado
            sf->senha = 0;
      else  // inicia aberto
            sf->senha = 1;
      sf->semaforo = semcreate(1); // aberto para poder pegar senha
}

wait_f(semaforo_fila *sf) {
      int minha_senha;

      wait(sf->semaforo);
      minha_senha = sf->prox_senha;
      sf->prox_senha = sf->prox_senha + 1;
      signal(sf->semaforo);

      while (sf->senha != minha_senha)
            sleep(0);
}

signal_f(semaforo_fila *sf) {
      sf->senha = sf->senha + 1;
}

//produtor1
void prod1(semaforo_fila *consome, semaforo_fila *produz) {
      int32 i, msg=1;
      imprime("Produtor1 iniciou");
      for (i = 0; i < nprods; i++) {
            kprintf("produtor 1 vai produzir");
            wait_f(produz);
            cxpostal[0] = msg; //mensagem na caixa postal
            signal_f(consome);
            sleep(1);
      }
}

//produtor2
void prod2(semaforo_fila *consome, semaforo_fila *produz) {
      int32 i, msg=2;
      imprime("Produtor2 iniciou");
      for (i = 0; i < nprods; i++) {
            kprintf("Produtor 2 vai produzir");
            wait_f(produz);
            cxpostal[0] = msg; //mensagem na caixa postal
            signal_f(consome);
            sleep(1);
      }
}
```

```
//produtor3
void prod3(semaforo_fila *consome, semaforo_fila *produz) {
      int32 i, msg=3;
      imprime("Produtor 3 iniciou");
      for (i = 0; i < nprods; i++) {
            kprintf("produtor 3 vai produzir\n");
            wait_f(produz);
            cxpostal[0] = msg; //mensagem na caixa postal
            signal_f(consome);
            sleep(1);
      }
}
//produtor4
void prod4(semaforo_fila *consome, semaforo_fila *produz) {
      int32 i, msg=4;
      imprime("Produtor4 iniciou");
      for (i = 0; i < nprods; i++) {
            kprintf("produtor 4 vai produzir\n");
            wait_f(produz);
            cxpostal[0] = msg; //mensagem na caixa postal
            signal_f(consome);
            sleep(1);
      }
}

//consumidor1
void cons1(semaforo_fila *consome, semaforo_fila *produz) {
      int32 i, msg;
      imprime("Consumidor1 iniciou");
      for (i = 0; i < ncons; i++) {
            kprintf("consumidor 1 vai consumir");
            wait_f(consome);
            msg = cxpostal[0]; //caixa postal fica vazia
            signal_f(produz);
            sleep(1);
      }
}

//consumidor2
void cons2(semaforo_fila *consome, semaforo_fila *produz) {
      int32 i, msg;
      imprime("Consumidor2 iniciou");
      for (i = 0; i < ncons; i++) {
            kprintf("consumidor 2 vai consumir");
            wait_f(consome);
            msg = cxpostal[0]; //caixa postal fica vazia
            signal_f(produz);
            sleep(1);
      }
}
```

Segundo teste.

```
#define nprods 10
#define ncons 20  // 2x mais produtores que consumidores

int cxpostal;  // Caixa postal para Mensagem = Um inteiro
```

```
main() {

      consome = semcreate(0);  // Trancado
      produz = semcreate(1);

      //Criação dos processos
      resume( create(produtor1, 4096, 40, "prod1", 2, consome, produz) );
      resume( create(produtor2, 4096, 30, "prod2", 2, consome, produz) );
      resume( create(produtor3, 4096, 20, "prod3", 2, consome, produz) );
      resume( create(produtor4, 4096, 10, "prod4", 2, consome, produz) );
      resume( create(consumidor1, 4096, 10, "cons1", 2, consome, produz) );
      resume( create(consumidor2, 4096, 10, "cons2", 2, consome, produz) );
}

// Processo produtor1
void produtor1(consome, produz) {
      msg = 1;

      Imprime("Produtor1 iniciou");
      for (i = 0; i < nprods; i++) {
            wait(produz);
            cxpostal[0] = msg;      // Produtor 1 depositou sua msg
            Imprime na tela msg produzida;
            signal(consome);
            sleep(1);
      }
}

//Processo produtor2
void produtor2(consome, produz) {
      msg = 2;

      Imprime("Produtor2 iniciou");
      for (i = 0; i < nprods; i++) {
            wait(produz);
            cxpostal[0] = msg;      // Produtor 2 depositou sua msg
            Imprime na tela msg produzida;
            signal(consome);
            sleep(1);
      }
}
//Processo produtor3
void produtor3(consome, produz) {
      msg = 3;

      Imrprime("Produtor3 iniciou");
      for (i = 0; i < nprods; i++) {
            wait(produz);
            cxpostal[0] = msg;      // Produtor 3 depositou sua msg
            Imprime na tela msg produzida;
            signal(consome);
            sleep(1);
      }
}
//Processo produtor4
void produtor4(consome, produz) {
      msg = 4;
```

```
      kprintf("Produtor4 iniciou");
      for (i = 0; i < nprods; i++) {
            wait(produz);
            cxpostal[0] = msg;      // Produtor 4 depositou sua msg
            Imprime na tela msg produzida;
            signal(consome);
            sleep(1);
      }
}

//Processo consumidor1
void consumidor1(consome, produz) {
      int msg;

      kprintf("Consumidor1 iniciou");
      for (i = 0; i < ncons; i++) {
            wait(consome);
            msg = cxpostal[0]; // Consumidor 1 retirou uma msg
            Imprime na tela msg Pega na caixa postal;
            signal(produz);
            sleep(1);
      }
}

//Processo consumidor2
void consumidor2(consome, produz) {
      int msg;

      kprintf("Consumidor2 iniciou");
      for (i = 0; i < ncons; i++) {
            wait(consome);
            msg = cxpostal[0]; // Consumidor 2 retirou uma msg
            Imprime na tela msg Pega na caixa postal;
            signal(produz);
            sleep(1);
      }
}
```

Resultado do teste da caixa postal

```
      Produtor1 iniciou
       msg1 = 1   // Caixa postal recebeu mensagem

      Produtor2 iniciou
      Produtor3 iniciou
      Produtor4 iniciou

      Consumidor1 iniciou
       consumi1 a msg = 1 // Mensagem foi consumida e
                          // Caixa postal voltou a ficar vazia.
       msg2 = 2   // Caixa postal recebeu mensagem nova

      Consumidor2 iniciou
       consumi2 msg = 2

       msg3 = 3

       consumi1 msg = 3
```

```
msg4 = 4

consumi2 msg = 4

msg1 = 1

consumi1 msg = 1

msg2 = 2

consumi2 msg = 2

msg3 = 3

consumi1 msg = 3

msg4 = 4

consumi2 msg = 4

msg1 = 1

(continua ...)
```

Aplicação Para a utilização da maioria das funcionalidades que o SO suporta criamos uma aplicação conhecida como o 'Jantar dos filósofos'. Este é um problema clássico na literatura de sistemas operacionais e representa a interação entre cinco processos (filósofos) que utilizam cinco recursos (garfos) de forma compartilhada, sem que qualquer processo permaneça sem receber recursos ficando em um estado de inanição.

Os filósofos inicialmente estão pensando, depois ficam com fome e, quando conseguem o garfo da direita e o da esquerda, comem. O tempo que um filósofo permanece 'pensando, comendo ou com fome' é definido por uma função aleatória. Há três estados possíveis para cada filósofo esperar, *THINKING, HUNGRY* e *EATING*. Cada filósofo pode fazer as seguintes ações: *think, take_forks, eat, put_forks* (pensa, pega_garfo, come, devolve_garfo). Nesta implementação, dois filósofos podem comer simultâneamente, se estiverem em posições alternadas para conseguir a obtenção do recurso (garfo) para comer.

Pseudocódigo

```
estado[5];              //Um estado para cada filósofo
filosofos[5] = {"Aristóteles", "Blaise", "Carl", "Descartes", "Euclides"};
letra[3] = {'P', 'F', 'C'};  //Representa o estado: Pensa, Fome, Come
semaforos[5];     //Um semaforo para cada filósofo

main() {

      // Criação e inicialização da mutex
      mutex = semcreate(1);

      // Criação e inicialização dos semáforos, um para cada filósofo
      for (index = 0; index < N; index ++){
            sem[index] = semcreate(0);
      }
```

```
      // Criação dos processos (Filósofos)
      for (index = 0; index < N; index++){
            resume(create(philosopher, 4096, 20,
                                    filosofos[index], 1, index));
      }

      // Estado inicial dos processos (filósofos)
      for (index = 0; index < N; index++){
            estado[index] = THINKING; // Estado inicial
      }
}

//o que o filósofo faz?
philosopher(id){
      while(TRUE){
            think(id);        //Pensa
            take_forks(id); //Pega o garfo
            eat(id);          //come
            put_forks(id);  //Devolve o garfo
      }
}

//Ações possíveis para cada filósofo
take_forks(i) {
      wait(mutex);
      state[i] = HUNGRY;

      test(i);
      signal(mutex);
      if (state[i] == HUNGRY) {
            wait(sem[i]);
      }
}

put_forks(i) { //Devolve o garfo
      wait(mutex);
      state[i] = THINKING;

      if (test(LEFT) == 1) {
            signal(sem[LEFT]); //Acorda o da esquerda
      }
      if (test(RIGHT) == 1 ) {
            signal(sem[RIGHT]); //Acorda o da direita
      }
      signal(mutex);
}

//Testa para ver se consegue o recurso e come
test(i) {
      if (estado[i] == HUNGRY && state[LEFT] != EATING
                              && state[RIGHT] != EATING) {
            estado[i] = EATING;

            return 1;
      } else {
            return 0;
      }
}
```

```
think(i) {
      tempo;
      tempo = abs(Random) // Tempo para pensar
      sleep(tempo);
}

eat(i) {
      tempo;
      tempo = abs(Random) // Tempo para comer
      sleep(tempo);
}
```

Resultado do da aplicação - jantar dos filósofos

```
      Legenda:
            ---------------
            A = Aristóteles
            B = Blaise
            C = Carl
            D = Descartes
            E = Euclides
            ---------------
            P = Pensando
            F = com Fome
            C = Comendo
            ---------------

      Jantar:
      |A|B|C|D|E|
      -----------
      |P|P|P|P|P|             // Todos pensando
      |F|P|P|P|P|             // Aristóteles com fome
      |C|P|P|P|P|             // Aristóteles comendo
      |C|F|P|P|P|             // Blaise com fome
      |C|F|F|P|P|             // Carl com fome
      |C|F|C|P|P|             // Carl comendo
      |C|F|C|P|F|             // Euclides com fome
      |C|F|P|P|F|
      |P|F|P|P|F|             // Aristóteles devolve garfo, volta a pensar
      |P|F|P|P|C|             // Euclides comendo
      |P|C|P|P|C|             // Blaise comendo
      |P|C|P|F|C|             // Descartes com fome
      |P|C|F|F|C|
      |P|C|F|F|P|             // Euclides devolve garfo, volta a pensar
      |P|C|F|C|P|             // Descartes comendo
      |P|P|F|C|P|
      |F|P|F|C|P|
      |C|P|F|C|P|
      |C|P|F|P|P|
      |C|P|C|P|P|
      |P|P|C|P|P|
      |P|F|C|P|P|
      |P|F|P|P|P|
      |P|C|P|P|P|
      |P|P|P|P|P|
      |P|F|P|P|P|
      |P|C|P|P|P|
```

```
|P|P|P|P|P|
|P|F|P|P|P|
|P|C|P|P|P|
|P|C|P|P|F|
|P|C|P|P|C|
|F|C|P|P|C|
|F|P|P|P|C|
|F|P|P|F|C|
|F|P|F|F|C|
|F|P|C|F|C|
|F|P|C|F|P|
|C|P|C|F|P|
|C|F|C|F|P|
|C|F|P|F|P|
|C|F|P|C|P|
|C|P|P|C|P|
|C|P|P|P|P|
```

3.6 Trabalhos futuros

Trabalhos futuros a serem realizados com o XINU-cMIPS, incluem como testar o porte em *drivers* que estiverem disponíveis futuramente. A implementação de interface *Ethernet* no cMIPS para verificação de uso no XINU-cMIPS também pode ser realizada futuramente para reforçar os objetivos de comprovar que a plataforma comporta um sistema operacional completo.

O desenvolvimento de um *driver* para *display* de sete segmentos duplo para indicação do estado de processamento também fica como sugestão para trabalhos futuros, seria interessante por exemplo em caso de pane no sistema, para indicar nesse *display* a causa dessa pane.

Implementar e utilizar o sistema de arquivos no cMIPS, para testar as funções suportadas pelo sistema de arquivos do XINU. Utilizar a UART e dispositivos TTY.

3.7 Conclusão

Este trabalho descreve o porte do sistema operacional XINU para o cMIPS e aborda os testes funcionais empregados no XINU-cMIPS. Através dos testes realizados no simulador, é possível verificar que porte para a plataforma de *hardware* é realizado com sucesso. As funcionalidades do sistema operacional XINU que são testadas vão desde a criação de processos, utilização de semáforos até a comunicação entre processos utilizando caixas postais. A verificação das funções disponíveis no Sistema operacional XINU, com resultados corretos e esperados, possibilita comprovar que o cMIPS é capaz de comportar um sistema operacional. É possível expandir a utilização do cMIPS, essa ferramenta de exploração de conceitos da área de Arquitetura de Computadores, para conceitos envolvendo sistemas operacionais agora no XINU-cMIPS, como semáforos por exemplo, que são importantes e necessários para se trabalhar com processos.

Desta forma, é mostrado (simulado) como o cMIPS é capaz de comportar um SO da categoria do UNIX, visto que o sistema operacional escolhido a ser portado para o cMIPS segue príncipios dessa categoria e ainda, por ter seu projeto focado em ser uma ferramenta para o ensino de sistemas operacionais, possuir seu design simples e modular o

que facilita a depuração das funções durante o porte. É Utilizado o XINU-MIPS fornecido pelo autor como ponto de partida neste porte. Como as plataformas são diferentes são necessárias várias adaptações e alterações, por exemplo com relação ao mapeamento de memória e os dispositivos utilizados por elas.

Mesmo um processador simples, com registradores padronizados e pouco variados em suas formas, como é o caso da implementação do MIPS, há dificuldades com relação aos serviços ofertados pelo sistema operacional, como troca de contextos, sincronização de processos, entre outros. Pôde-se utilizar as chamadas de sistema principais do sistema operacional e testar as interrupções neste porte com a ressalva de que o porte não foi completo por falta da SDRAM, e também porque a memória interna disponível não foi suficiente para acomodar o XINU.

Apesar desses detalhes com relação ao *hardware* disponível, é possível verificar o comportamento e a resposta do sistema operacional XINU através do dispositivo *stdout* no simulador, consegue-se verificar as funcionalidades que o sistema operacional deve ter sob o processador e as interações entre os mesmos. Os testes que podem gerar problemas quando o sistema operacional não interage corretamente, e que foram analisados, são os que utilizam as chamadas de sistema, desde a criação dos processos, o gerenciamento de memória e de recursos. Todos esses obtiveram resultados esperados. A completude do porte foi constatada nas simulações com os testes isoladamente realizados e, principalmente através da implementação da aplicação do jantar dos filósofos, a qual utiliza a maior parte das funções do sistema operacional em conjunto, permitindo o compartilhamento de um recurso corretamente entre os processos, fazendo com que a espera pela obtenção do recurso seja limitada e que nenhum processo não receba o recurso em algum determinado tempo. Neste trabalho é constatado, através do simulador, que o XINU-cMIPS está operando corretamente e na possibilidade de DRAM suficiente, será possível verificar o porte completo em um sistema embarcado. O código XINU-cMIPS e os testes encontram-se em https://gitlab.c3sl.ufpr.br/vbao07/XINU-cMIPS e https://gitlab.c3sl.ufpr.br/roberto/xinu-cMIPS.

Referências Bibliográficas

[1] Embedded XINU wiki. http://xinu.mscs.mu.edu/Main_Page. Acessado em 01/12/2015.

[2] Imagination Technologies. https://imgtec.com/mips/. Acessado em 05/12/2015.

[3] Information about XINU software. https://www.cs.purdue.edu/homes/dec/xsoft.html. Acessado em 18/11/2015.

[4] LSI-11. http://simh.trailing-edge.com/semi/lsi11.html. Acessado em 03/01/2017.

[5] The MIPS32 instruction set. https://engineering.purdue.edu/~ece437l/labs/lab3/files/MIPS_ISA.pdf. Acessado em 05/12/2015.

[6] The XINU page. http://www.xinu.cs.purdue.edu/. Acessado em 18/11/2015.

[7] Douglas Comer. *Projeto de Sistema Operacional - O Enfoque XINU*. Editora Campus, 1988.

[8] Douglas Comer. *Operating System Design - The XINU Approach*. CRC Press, 2015.

[9] Edsger W Dijkstra. The structure of the "THE" multiprogramming system. In *Classic operating systems*, pages 223–236. Springer, 2001.

[10] David A. Patterson e John L. Hennessy. *Computer Organization and Design - The Hardware/Software Interface*. 2008.

[11] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. MIPS: A microprocessor architecture. In *ACM SIGMICRO Newsletter*, volume 13, pages 17–22. IEEE Press, 1982.

[12] Roberto A. Hexsel. CMIPS - an FPGA ready VHDL model for 5-stage pipeline, MIPS32r2 core. https://gitlab.c3sl.ufpr.br/roberto/cmips. Acessado em 01/10/2016.

[13] Roberto A. Hexsel. The cMIPS page. http://www.inf.ufpr.br/roberto/cMIPS.html. Acessado em 18/06/2016.

[14] MIPS. MIPS32 architecture for programmers - MIPS32 privileged resourse achitecture. volume 3, 2005.

[15] MIPS. MIPS32 architecture for programmers - the MIPS32 instruction set. volume 2, 2005.

[16] Sunil Mirapuri, Michael Woodacre, and Nader Vasseghi. The MIPS R 4000 processor. *IEEE Micro*, 12(2):10–22, 1992.

[17] Charles Price. MIPS IV instruction set, revision 3.2. *MIPS Technologies, Inc. Relatório Técnico*, 1995.

[18] Silberschatz, Galvin, and Gagne. *Fundamentos de Sistemas Operacionais*. Editora LTC, 2011.

[19] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating Systems Concepts*. 2003.

Apêndice A

Códigos fonte

A.1 Alternância entre processos

```
/************************************************************/
/* Troca de contextos - alterna processos                   */
/* main - main program that Xinu runs as the first user process */
/*
/************************************************************/

#include <xinu.h>
#include <ramdisk.h>

extern process shell(void);

int fibonacci(int n);
int fatorial(int32 n);

void prA(void);
void prB(void);
void prC(void);
int32 n;

int main(int argc, char **argv) {

      int i;
      n = 0;

      (pri16)resume(create(prA, 4096, 50, "Processo a", 0));
      (pri16)resume(create(prB, 4096, 50, "Processo b", 0));
      (pri16)resume(create(prC, 4096, 20, "Processo c", 0));

      kprintf("\n\t Main.\n");

      return OK;
}

int fibonacci(int32 n) {
      int32 i;
      int32 f1 = 0;
      int32 f2 = 1;
      int32 fi = 0;;

      if (n == 0)
```

```
            return 0;
      if(n == 1)
            return 1;

      for(i = 2 ; i <= n ; i++ ) {
            fi = f1 + f2;
            f1 = f2;
            f2 = fi;
      }
      return fi;
}

int fatorial(int32 n) {
      int32 fat, i;
      fat = 1;
      for (i = 1; i <= n; i++) {
            fat = fat * i;
      }
      return fat;
}


void prA(){
      int i;
      n = 0;
      while (TRUE){
      kprintf("\n\tProcesso a iniciou\n\n");
            for ( i=1 ; i < 45 ; i++ ) {//45
                  kprintf("Fib(%d) = %d\n", i, fibonacci(i));
            }
      sleep(2);
      }
}

void prB(){
      int i;
  while (TRUE){
      for ( i = 0 ; i <= 16 ; i+=2 )
            kprintf("Fat(%d) = %d\n", i, fatorial(i));
   sleep(3);
  }
}

void prC(){
      while (TRUE)
            kprintf("PrC.\n");
   sleep(0);
}
```

A.2 Semáforo binário

```
/************************************************************/
/* Semáforo binário - produtor consumidor.                  */
/* main - main program that Xinu runs as the first user process */
/*
/************************************************************/
```

```
#include <xinu.h>
#include <ramdisk.h>

extern process shell(void);
int32 n;

void prod2(sid32, sid32);
void cons2(sid32, sid32);

int main(int argc, char **argv) {
      sid32 produced, consumed;
      int i;
      n = 0;

      consumed = semcreate(0); //semaphore
      produced = semcreate(1);

      //create(end, espaco pilha, prio, nome processo, total args do proc)
      (pri16)resume(create(prod2, 4096, 20, "prod", 2, consumed, produced));
      (pri16)resume(create(cons2, 4096, 20, "cons", 2, consumed, produced));

      return OK;
}

//produtor2
void prod2(sid32 consumed, sid32 produced) {
      int32 i;
      kprintf("\n\tProdutor iniciou:\n*********\n");
      for(i=0 ; i<50 ; i++) {
            wait(consumed);
            n++;
            kprintf("\n\tproduzi n= %d\n", n);
            signal(produced);
      }
      kprintf("\n\tprodutor terminou!\n\n");
}

//consumidor2
void cons2(sid32 consumed, sid32 produced) {
      int32 i;
      kprintf("\n\tConsumidor iniciou\n*********\n");
      for(i=0 ; i<50 ; i++) {
            wait(produced);
            kprintf("\n\tconsumi n= %d\n", n);
            signal(consumed);
      }
      kprintf("\n\tconsumidor terminou!\n\n");
}
```

A.3 Semáforo contador

```
/************************************************************/
/* Semáforo contador - produtor consumidor.                 */
/* main - main program that Xinu runs as the first user process */
/*
```

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

#include <xinu.h>
#include <ramdisk.h>

extern process shell(void);
int32 n, N;

void prod2(sid32, sid32);
void cons2(sid32, sid32);

int main(int argc, char **argv) {
      sid32 vazio, cheio;
      int i;
      n = 0;
      N = 8;

      cheio = semcreate(0);
      vazio = semcreate(N);

      (pri16)resume(create(prod2, 4096, 20, "prod", 2, cheio, vazio));
      (pri16)resume(create(cons2, 4096, 20, "cons", 2, cheio, vazio));

      return OK;
}

//produtor2
void prod2(sid32 cheio, sid32 vazio) {
      int32 i;
      kprintf("\n\tProdutor iniciou:\n\n");
      for(i=0 ; i<50 ; i++) {
            wait(cheio);
            n++;
            kprintf("\n\tproduzi n= %d\n", n);
            signal(vazio);
      }
      kprintf("\n\tprodutor terminou!\n\n");
}

//consumidor2
void cons2(sid32 cheio, sid32 vazio) {
      int32 i;
      kprintf("\n\tConsumidor iniciou\n\n");
      for(i=0 ; i<50 ; i++) {
            wait(vazio);
            kprintf("\n\tconsumi n= %d\n", n);
            signal(cheio);
      }
      kprintf("\n\tconsumidor terminou!\n\n");
}
```

A.4 Caixa postal

Com fila.

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

```
/* Caixa postal com fila - produtor consumidor.       */
/* main - main program that Xinu runs as the first user process */
/*
/************************************************************/

#include <xinu.h>
#include <ramdisk.h>
#define N 1
#define nprods 10
// 2x mais produtores que consumidores
#define ncons 20

int cxpostal[N]; //Um inteiro
extern process shell(void);
int32 n;
int32 idcons2;

typedef struct semaforo_fila {
      sid32 semaforo;
      volatile unsigned int senha, prox_senha;
} semaforo_fila;

void prod1(semaforo_fila*, semaforo_fila*);
void prod2(semaforo_fila*, semaforo_fila*);
void prod3(semaforo_fila*, semaforo_fila*);
void prod4(semaforo_fila*, semaforo_fila*);

void cons1(semaforo_fila*, semaforo_fila*);
void cons2(semaforo_fila*, semaforo_fila*);

void inicializa_semaforo_fila(semaforo_fila *sf, int init);
void wait_f(semaforo_fila *sf);
void signal_f(semaforo_fila *sf);

int main(int argc, char **argv) {
      semaforo_fila produz, consome;
      n = 0;

      inicializa_semaforo_fila(&consome, 0); //trancado
      inicializa_semaforo_fila(&produz, 1);

      (pri16)resume(create(prod1, 4096, 20, "prod1", 2, &consome, &produz));
      (pri16)resume(create(prod2, 4096, 20, "prod2", 2, &consome, &produz));
      (pri16)resume(create(prod3, 4096, 20, "prod3", 2, &consome, &produz));
      (pri16)resume(create(prod4, 4096, 20, "prod4", 2, &consome, &produz));
      (pri16)resume(create(cons1, 4096, 20, "cons1", 2, &consome, &produz));
      (pri16)resume(create(cons2, 4096, 20, "cons2", 2, &consome, &produz));
      return OK;
}

void inicializa_semaforo_fila(semaforo_fila *sf, int init) {
      sf->prox_senha = 1;
      if (init == 0) // inicia trancado
            sf->senha = 0;
      else // inicia aberto
            sf->senha = 1;
      sf->semaforo = semcreate(1); // aberto para poder pegar senha
}
```

```
void wait_f(semaforo_fila *sf) {
      volatile unsigned int minha_senha;

      wait(sf->semaforo);
      minha_senha = sf->prox_senha;
      sf->prox_senha = sf->prox_senha + 1;
      signal(sf->semaforo);

      while (sf->senha != minha_senha)
            sleep(0);
}

void signal_f(semaforo_fila *sf) {
      sf->senha = sf->senha + 1;
}

//produtor1
void prod1(semaforo_fila *consome, semaforo_fila *produz) {
      int32 i, msg=1;
      kprintf("\n\tProdutor1 iniciou\n");
      for (i = 0; i < nprods; i++) {
            kprintf("produtor 1 vai produzir\n");
            wait_f(produz);
            cxpostal[0] = msg;
            kprintf("\n\t msg1 = %d\n", msg);
            signal_f(consome);
            sleep(1);
      }
}
//produtor2
void prod2(semaforo_fila *consome, semaforo_fila *produz) {
      int32 i, msg=2;
      kprintf("\n\tProdutor2 iniciou\n");
      for (i = 0; i < nprods; i++) {
            kprintf("produtor 2 vai produzir\n");
            wait_f(produz);
            cxpostal[0] = msg;
            kprintf("\n\t msg2 = %d\n", msg);
            signal_f(consome);
            sleep(1);
      }
}
//produtor3
void prod3(semaforo_fila *consome, semaforo_fila *produz) {
      int32 i, msg=3;
      kprintf("\n\tProdutor3 iniciou\n");
      for (i = 0; i < nprods; i++) {
            kprintf("produtor 3 vai produzir\n");
            wait_f(produz);
            cxpostal[0] = msg;
            kprintf("\n\t msg3 = %d\n", msg);
            signal_f(consome);
            sleep(1);
      }
}
//produtor4
void prod4(semaforo_fila *consome, semaforo_fila *produz) {
```

```
      int32 i, msg=4;
      kprintf("\n\tProdutor4 iniciou\n");
      for (i = 0; i < nprods; i++) {
            kprintf("produtor 4 vai produzir\n");
            wait_f(produz);
            cxpostal[0] = msg;
            kprintf("\n\tmsg4 = %d\n", msg);
            signal_f(consome);
            sleep(1);
      }
}

//consumidor1
void cons1(semaforo_fila *consome, semaforo_fila *produz) {
      int32 i, msg;
      kprintf("\n\tConsumidor1 iniciou");
      for (i = 0; i < ncons; i++) {
            kprintf("consumidor 1 vai consumir");
            wait_f(consome);
            msg = cxpostal[0];
            kprintf("\n\tconsumi1 msg = %d\n", msg);
            signal_f(produz);
            sleep(1);
      }
      kprintf("\n\tconsumidor1 terminou!\n\n");
}

//consumidor2
void cons2(semaforo_fila *consome, semaforo_fila *produz) {
      int32 i, msg;
      kprintf("\n\tConsumidor2 iniciou");
      for (i = 0; i < ncons; i++) {
            kprintf("consumidor 2 vai consumir");
            wait_f(consome);
            msg = cxpostal[0];
            kprintf("\n\tconsumi2 msg = %d\n", msg);
            signal_f(produz);
            sleep(1);
      }
      kprintf("\n\tconsumidor2 terminou!\n\n");
}
```

Com semáforo.

```
/*****************************************************************/
/* Caixa postal - produtor consumidor.                           */
/* main - main program that Xinu runs as the first user process */
/*
/*****************************************************************/

#include <xinu.h>
#include <ramdisk.h>
#define N 1
#define nprods 10
#define ncons 20 // 2x mais produtores que consumidores

int cxpostal[N]; //Um inteiro
extern process shell(void);
int32 n;
```

```
int32 idcons2;
void prod1(sid32, sid32);
void prod2(sid32, sid32);
void prod3(sid32, sid32);
void prod4(sid32, sid32);
void cons1(sid32, sid32);
void cons2(sid32, sid32);

int main(int argc, char **argv) {
      sid32 produz, consome;
      int i;
      n = 0;

      consome = semcreate(0); //trancado
      produz = semcreate(1);

      //create(end, espaco pilha, prio, nome processo, total args do proc)
      (pri16)resume(create(prod1, 4096, 20, "prod1", 2, consome, produz));
      (pri16)resume(create(prod2, 4096, 20, "prod2", 2, consome, produz));
      (pri16)resume(create(prod3, 4096, 20, "prod3", 2, consome, produz));
      (pri16)resume(create(prod4, 4096, 20, "prod4", 2, consome, produz));
      (pri16)resume(create(cons1, 4096, 20, "cons1", 2, consome, produz));
      (pri16)resume(create(cons2, 4096, 20, "cons2", 2, consome, produz));
      return OK;
}

//produtor1
void prod1(sid32 consome, sid32 produz) {
      int32 i, msg=1;
      kprintf("\n\tProdutor1 iniciou");
      for (i = 0; i < nprods; i++) {
            wait(produz);
            cxpostal[0] = msg;
            kprintf("\n\t msg1 = %d\n", msg);
            signal(consome);
            sleep(1);
      }
}

//produtor2
void prod2(sid32 consome, sid32 produz) {
      int32 i, msg=2;
      kprintf("\n\tProdutor2 iniciou");
      for (i = 0; i < nprods; i++) {
            wait(produz);
            cxpostal[0] = msg;
            kprintf("\n\t msg2 = %d\n", msg);
            signal(consome);
            sleep(1);
      }
}

//produtor3
void prod3(sid32 consome, sid32 produz) {
      int32 i, msg=3;
      kprintf("\n\tProdutor3 iniciou");
      for (i = 0; i < nprods; i++) {
            wait(produz);
```

```
            cxpostal[0] = msg;
            kprintf("\n\t msg3 = %d\n", msg);
            signal(consome);
            sleep(1);
      }
}

//produtor4
void prod4(sid32 consome, sid32 produz) {
      int32 i, msg=4;
      kprintf("\n\tProdutor4 iniciou");
      for (i = 0; i < nprods; i++) {
            wait(produz);
            cxpostal[0] = msg;
            kprintf("\n\tmsg4 = %d\n", msg);
            signal(consome);
            sleep(1);
      }
}

//consumidor1
void cons1(sid32 consome, sid32 produz) {
      int32 i, msg;
      kprintf("\n\tConsumidor1 iniciou");
      for (i = 0; i < ncons; i++) {
            wait(consome);
            msg = cxpostal[0];
            kprintf("\n\tconsumi1 msg = %d\n", msg);
            signal(produz);
            sleep(1);
      }
      kprintf("\n\tconsumidor1 terminou!\n\n");
}

//consumidor2
void cons2(sid32 consome, sid32 produz) {
      int32 i, msg;
      kprintf("\n\tConsumidor2 iniciou");
      for (i = 0; i < ncons; i++) {
            wait(consome);
            msg = cxpostal[0];
            kprintf("\n\tconsumi2 msg = %d\n", msg);
            signal(produz);
            sleep(1);
      }
      kprintf("\n\tconsumidor2 terminou!\n\n");
}
```

A.5 Jantar dos filósofos

```
/*------------------------------------------------------------------*/
/* Aplicação - Jantar dos Filósofos                                 */
/* main - main program that Xinu runs as the first user process */
/*
/*------------------------------------------------------------------*/
```

```
#include <xinu.h>
#include <ramdisk.h>

#define MAX_THINK_TIME 4
#define MAX_EAT_TIME 2
#define MAX_TAM_NOME 13
#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2

extern process shell(void);
int m_w;    /* must not be zero, nor 0x464fffff */
int m_z;    /* must not be zero, nor 0x9068ffff */
int state[N];  //Um estado para cada filósofo
int pronto;  //Quando o filósofo inicializados
char *friends[N]={"Aristóteles", "Blaise", "Carl", "Descartes", "Euclides"};
char letra[3] = {'P', 'F', 'C'}; //Representa o estado: Pensa, Fome, Come
sid32 mutex;
sid32 sem[N];  //Um semaforo para cada filósofo

void print_estado(void);
void *philosopher(int id);
void take_forks(int i);
void put_forks(int i);
void think(int i);
void eat(int i);
int test(int i);
int Random( int *m_z, int *m_w); // from wikipedia
int abs(int x);

int main(int argc, char **argv) {
      int res;
      int index;

      m_w = 177;  //Valor primo
      m_z = 311;
      pronto = 0;

      kprintf("\nJantar dos Filósofos\n\n");

      // Iniciar mutex
      mutex = semcreate(1);

      //kprintf("Aristóteles | Blaise | Carl | Descartes | Euclides\n");
      kprintf("|A|B|C|D|E|\n");
      kprintf("-----------\n");

      // Iniciar semaforos
      for (index = 0; index < N; index ++){
            sem[index] = semcreate(0);
      }

      // Criar filósofos
      for (index = 0; index < N; index++){
            res = (pri16)resume(
```

```
                create(philosopher, 4096, 20, friends[index], 1, index ));
            if (res == SYSERR){
                kprintf("Thread creation failed");
            }
        }

        // Estado inicial dos filosofos
        for (index = 0; index < N; index++){
            state[index]=THINKING; // Estado inicial pensando
        }

        pronto = 1;
        print_estado();

        return 1;
}

//O que o filósofo está fazendo?
void *philosopher(int id){

        while (pronto == 0) {
            sleep(0); // Inicia assim que a CPU estiver disponível
        }

        while(TRUE){
            think(id);
            take_forks(id);
            eat(id);
            put_forks(id);
        }
}

void take_forks(int i) {
        wait(mutex);
        state[i] = HUNGRY;
        print_estado();
        test(i);
        signal(mutex);
        if (state[i] == HUNGRY) {
            wait(sem[i]);
        }
}

void put_forks(int i) {
        wait(mutex);
        state[i] = THINKING;
        print_estado();
        if (test(LEFT) == 1) {
            signal(sem[LEFT]);
        }
        if (test(RIGHT) == 1 ) {
            signal(sem[RIGHT]);
        }
        signal(mutex);
}

int test(int i) {
        if (state[i] == HUNGRY && state[LEFT] != EATING
```

```
                && state[RIGHT] != EATING) {
            state[i] = EATING;
            print_estado();
            return 1;
      } else {
            return 0;
      }
}

void think(int i) {
      int tempo;
      tempo = abs(Random(&m_z, &m_w)) % MAX_THINK_TIME;
      sleep(tempo);
}

void eat(int i) {
      int tempo;
      tempo = abs(Random(&m_z, &m_w)) % MAX_EAT_TIME;
      sleep(tempo);
}

int Random( int *m_z, int *m_w) { // from wikipedia
  *m_z = 36969 * (*m_z & 65535) + (*m_z >> 16);
  *m_w = 18000 * (*m_w & 65535) + (*m_w >> 16);
  return ((*m_z << 16) + *m_w); /* 32-bit result */
}

int abs(int x) {
      if (x < 0) {
            return 0 - x;
      }
      else {
            return x;
      }
}

void print_estado() {
      int i;

      kprintf("|");
      for (i = 0; i < N; i++) {
            kprintf("%c|", letra[state[i]]);
      }
      kprintf("\n");
}
```