Documentação do µnúcleo

INF01142 – Sistemas Operacionais I – Turma B – Prof. Alexandre Caríssimi

versão: 3 de maio de 2012

1. Identificação do Grupo

- Luiz Gustavo Frozi de Castro e Souza - Cartão 96957
- Pedro Henrique Frozi de Castro e Souza - Cartão 161502

2. Descrição da Plataforma de Desenvolvimento

Foram usados dois computadores com configurações diferentes para realizar o desenvolvimento, o µnúcleo foi compilado e testado e dois ambientes linux virtualizados. Foi também utilizado para desenvolvimento o versionador SVN, utilizando um repositório do Google Code, disponível em: http://code.google.com/p/unucleo/.
A tabela abaixo faz um detalhamento dos ambientes de desenvolvimento e execução:

	Computador 1	**Computador 2**
Tipo de Processador	Intel Core 2 Duo T5450 1,67GHz	AMD Athlon(tm) X2 Dual Core Processor L310 1.2 GHz
Número de Cores	2	2
Suporte a HT?	Sim	Sim
Memória RAM	4GB	3GB
Sistema Operacional (Host)	Windows 7 Professional de 64bits	Windows 7 Professional de 64bits
Foi usado Ambiente Virtualizado?	Sim	Sim
Máquina Virtual Utilizada	Oracle VM VirtualBox 4.1.6 r74713	Oracle VM VirtualBox 4.1.12 r77245
Sistema Operacional (Guest, VM)	Linux de 32bits	Linux 64bits
Distribuição Utilizada	Debian 6.0.3 com LXDE	Ubuntu 12.04 com Xubuntu
Versão do Kernel	2.6.32-5-686	3.0.0-17
Versão o GCC	4.4.5	4.6.1

3. Programas de Teste Desenvolvidos

Os programas de Teste estão listados abaixo, todas as informações necessárias para a execução dos mesmos já se encontram no próprio arquivo fonte.

Programa	Objetivo	Funcionamento
teste1.c	Verificar a execução de múltiplos processos diferentes.	Cria os processos p1, p2, p3. O processo p1 cria p4. O processo p2 aguarda p4.
teste2.c	Verificar a execução de múltiplos processos diferentes.	Cria os processos p1, p2, p3, p7, p8, p9, p10. * p1 cria p4 * p1 aguarda p3 * p2 faz cedência voluntária * p3 cria p5 * p5 cria p6
teste3.c	Verificar a execução correta dos processos de acordo com as prioridades.	Cria MAX_PROC_I processos com a mproc_create(), processos com PID ímpar recebem prioridade 1 (devem ser executados primeiro), processos com PID par recebem prioridade 2 (devem ser executados depois).
teste4.c	Verificar se os processos não estão sendo criados com prioridades indevidas.	Cria 1 processo com prioridade < 1 e outro processo com prioridade > 2. Deve retornar erro.
teste5.c	Verficar o bloqueio de processos e a cedencia voluntária.	Cria os processos p1, p2, p3, p4, p5. * p1 executa normalmente. * p2 deve aguardar p3 que deve aguardar p4. * p4 e p5 fazem uma cedência voluntária.
teste6.c	Verficar o bloqueio de vários processos aguardando a execução de um.	Cria os processos p1, p2, p3, p4, p5, p6, p7. * p1, p2, p3, p4 deve aguardar p5. * p6 deve aguardar p1 (já executou). * p7 deve agaurdar um processo inexistente.
teste7.c	Verficar um deadlock com prioridades diferentes.	Cria os processos p1, p2, p3. * p1 deve aguardar p2. * p2 deve aguardar p3. * p3 deve aguardar p1 gerando o deadlock.

4. Funcionamento da primitiva mproc_create()

Estruturas de Dados envolvidas:
FIFO_DESC - Estrutura de dados do tipo fila criada para fila de processos;
PCB - Estrutura de dados que representa o PCB do processo;
ucontext_t

Funções chamadas:
ini_make() - Inicializa os parametros do contexto;
insere_fifo() - Função utilizada para iserir um novo PCB na estrutura FIFO_DESC;
malloc() - Usada para alocar uma pilha e um contexto para a estrutura PCB;
getcontext()
makecontext()

A primitiva cria uma nova estrutura PCB, alocando memória para o campo ‘Contexto’ e para pilha do contexto(‘uc_stack.ss_sp’). No retorno do contexto (uc_link) colocamos o endereço para o contexto ‘scheduler’ (contexto principal do escalonador), assim, sempre que um processo terminar, o controle de execução volta para o escalonador. O novo

contexto é inicializado através da primitiva makecontext() e sua estrutura PCB é inserida na fila de aptos da sua prioridade, que é um como parâmetro da primitiva.

5. Funcionamento da primitiva mproc_yield()

Estruturas de Dados envolvidas:
FIFO_DESC
PCB
ucontext_t

Funções chamadas:
insere_fifo()
swapcontext()

Faz com que o processo que está executando realize cedência voluntária, trocando o estado do PCB para 'APTO' e colocando de volta na fila da sua prioridade. Após isso, executa a primitiva swapcontext() para o contexto 'scheduler' (contexto principal do escalonador).

6. Funcionamento da primitiva mproc_join()

Estruturas de Dados envolvidas:
FIFO_DESC
PCB
ucontext_t

Funções chamadas:
existe_pcb() - Função que verifica se um processo está em alguma fila do escalonador, verificando pelo parametro 'pid' da estrutura PCB;
insere_fifo()
swapcontext()

Faz com que o processo que está executando fique bloqueado aguardando um outro processo. O pid do processo que ele irá aguardar é passado como parâmetro desta primitiva. Caso o pid não exista a primitiva retorna o valor -1. Se existir, o processo em execução é colocado no estado 'BLOQ' e inserido na lista de processos bloqueados, neste momento ocorre uma troca de contexto através da primitiva swapcontext() para o contexto 'scheduler' (contexto principal do escalonador).

7. Descrição sobre o que está funcionando e o que não está

Inicialmente estávamos tendo problemas ao executar os testes 1 e 2. Dependendo do ambiente, o arquivo compilado executava na maioria das vezes, em outras dava vários erros como "Segmentation Fault" e até "Floating Point Error", este último chegava a impressionar, pois em nenhum lugar da biblioteca e dos programas de teste são usados números de ponto flutuante. Após várias verificações descobrimos que o erro acontecia devido à um problema no ponteiro usado como argumento das funções. As funções de teste dentro dos arquivos *teste1.c* e *teste2.c* estavam declaradas como `void funcao(void)`, pois não usavam argumentos. Após corrigir todas as declarações para `void funcao(void* arg)`, os problemas foram eliminados, tanto no ambiente do **Computador 1** (onde o funcionamento era instável, com os erros descritos), quanto no ambiente do **Computador 2** (onde os testes executavam normalmente).

8. Metodologia de Teste Utlizada

Utilizamos a metodologia de teste com valores inválidos, que consiste em executar as funções com os valores limites (valores de entrada e quantidade de processos). Não foi utilizado nenhum “debugger” específico, verificamos o retorno das funções no próprio terminal.

9. Dificuldades encontradas e soluções

Tivemos algumas dificuldades para fazermos a correta troca de contexto, que na maioria das vezes gerava o erro “Segmentation Fault”. Após conversarmos com o professor e fazer algumas pesquisas resolvemos reescrever a estrutura PCB que armazena o contexto do processo, colocando o campo ‘contexto’ como ponteiro e alocando este ponteiro quando um novo PCB é adicionado. Esta técnica nos possibilitou fazer as trocas de contexto de maneira mais limpa e organizada. Por exemplo, quando um processo entra no estado BLOQUEADO, ele é removido da estrutura em execução e colocado na lista de processos bloqueados e, como o ‘contexto’ agora é um ponteiro, podemos simplesmente utilizar o swapcontext() para a troca, que salvará o contexto alocado inicialmente, não importando se ele se encontra na fila de aptos ou na lista de bloqueados.