

Universidade Federal do Rio Grande do Sul



Departamento de Informática Aplicada
Sistemas Operacionais I

Felipe de Souza Lahti - 170715
Germano de Mello Andersson - 137719

Documentação núcleo - Unife

Prof. Dr. Sérgio Luis Cechin

Porto Alegre, 03 de Maio de 2012

INTRODUÇÃO

O Unife é uma biblioteca da linguagem C para criação de threads em nível de usuário, ou seja, 1 processo N contextos (threads). Possui uma máquina de estados simples, com apenas três estados, uma primitiva de sincronização de término e uma primitiva de liberação voluntária de processador. Todas as especificações solicitadas na definição do trabalho estão funcionais.

Usando a biblioteca

Para utilização da biblioteca, inclua o header `unucleo.h` e siga os três passos a seguir no desenvolvimento do seu software:

- 1) Inicialização, através da função `libsisop_init`;
- 2) Criar as threads, através da função `mproc_create`;
- 3) Executar as threads, através da função `scheduler`.

Agora basta compilar seu programa ligando a biblioteca unife ao seu software:

```
>cd $unife_dir; gcc -o seu_programa.o seu_programa.c -I. -Llib -lsisop -Wall
```

UNIFE

Estruturas de Dados

a) `proc_struct`: estrutura utilizada para representar um processo no sistema. A fila de processos em cada estado é encadeada através de um atributo desta estrutura.

b) `proc_state`: estrutura utilizada para representar os possíveis estados do sistema. Neste trabalho, utilizada para implementar as filas `READY` e `BLOCKED`. A fila de prioridades em cada estado é encadeada através de um atributo desta estrutura.

c) `map_join`: estrutura utilizada para mapear que processo aguarda término de que processo.

d) `stats_unife`: estrutura utilizada para contabilidade e estatística do sistema.

Principais Primitivas

`libsisop_init()`

Aloca memória e inicializa todas estruturas de dados envolvidas.

`mproc_create (prioridade, ponteiro_da_função, parâmetro)`

Cria um novo processo sendo passado como parâmetros a prioridade que o processo terá, um ponteiro para a função que será executada por este processo e o parâmetro que essa função receberá. Por definição, só permite a passagem de um parâmetro. Internamente é testado se a prioridade é válida e se não atingiu o limite de processos simultâneos. É criada uma *“struct”* para o processo onde é armazenado seu pid, contexto e prioridade. Essa estrutura é inserida na fila de sua respectiva prioridade no estado `ready`, utilizando a função interna `__in_proc_state`. É criado o contexto para o processo. É retornado o pid do processo criado.

`mproc_yield()`

Reinsere o processo no fim da fila de processos `ready`, respeitando a sua prioridade. Retorna para o contexto de execução do `scheduler`.

`mproc_join(pid)`

Faz o processo aguardar o término do processo passado como parâmetro. Insere o processo que invocou a sua execução na fila que representa o estado `blocked`. Insere o processo

na lista de joins que estão aguardando um processo ser encerrado.

scheduler()

Fica em um loop while enquanto tiver processos a serem executados. Inicialmente olha na lista de joins para ver se tem algum processo a ser liberado. Após, executa o processo com mais alta prioridade que está na lista de ready e chaveia o contexto para o processo. Termina quando não houver mais processos na lista de ready ou blocked. Imprime as estatísticas da biblioteca.

AMBIENTE DE DESENVOLVIMENTO

Plataforma

Trabalhamos em ambientes distintos:

Ambiente1

Processador: Intel Core i5-2410M @2.30GHz (2 cores, 4 threads com HT)

GNU/Linux: Ubuntu 11.10

Kernel: 3.0.0-12

GCC: 4.6.1

Rodado em ambiente virtualizado utilizando: VirtualBox 4.1.10

Ambiente2

Processador: Intel Core2 Duo T8300, 2 cores com suporte HT

GNU/Linux: Debian 6.0.3 (squeeze)

Kernel: 2.6.32-5-686

GCC: 4.4.5

Repositório

Utilizamos o repositório público do google para hospedagem do nosso projeto:

<http://code.google.com/p/unife/>

TESTES

Utilizamos a ferramenta gdb (GNU Debugger) para depuração do programa, durante as fases de desenvolvimento. Para teste das funcionalidades do programa, criamos dois conjuntos de testes: o primeiro visando apontar falhas relacionadas a especificação e o segundo visando a utilização propriamente dita do sistema. Abaixo citamos os testes desenvolvidos e uma breve explicação do seu objetivo:

test1.c – Testa se a biblioteca foi inicializada corretamente.

test2.c – Testa erro de criação do processo ao inserir uma prioridade inválida.

test3.c – Testa erro de criação do processo ao tentar rodar mais de 128 processos simultaneamente. São criados 129 processos, o último deve apresentar um erro de criação.

test4.c – Testa se as prioridades estão sendo obedecidas. São criados dois processos um com prioridade baixa e outro com prioridade média.

test5.c – Testa a função `mproc_join` em que um processo deve aguardar outro processo terminar antes de continuar após a chamada a função `mproc_join`.

test6.c – Testa se o deadlock está ocorrendo. É criado dois processos e cada processo aguarda pelo término do outro.

test7.c – Teste completo do sistema. São utilizadas todas as funções da biblioteca. São criados vários processos com prioridades diferentes na main do programa. O processo “f2” cria outro processo com prioridade maior e chama `mproc_yield`. Valida a especificação que a primitiva `yield` só libera processador para processos de prioridade superior ou igual.

test8.c – Testa scheduler duplo. Segunda chamada da primitiva `scheduler` não executa nada, pois não há processos criados entre as chamadas de `scheduler()`.

test9.c – Testa criação de 10.000 processos, que não ultrapassam o limite de 128 processos simultâneos.

test10.c – Feito um mini-game utilizando a biblioteca. O game consiste em descobrir um número randômico adicionando ou subtraindo um número de um número inicial (0) no menor número de passos possíveis. (Boa sorte, meu melhor score foi 15 passos). A aplicação cria duas threads(processos), uma fica lendo a entrada do usuário (número inteiro positivo ou negativo) e a outra faz a lógica do mini-game imprimindo o status do jogo.

test11.c – Um programa para exemplificar como simular uma conversa entre duas pessoas, onde a fala de cada uma é uma thread.

Para compilação e execução dos testes, basta utilizar a diretiva 'install' do makefile. Eles serão salvos no diretório bin:

```
>cd $unife_dir; make install; bin/test1.o
```

DECISÕES TÉCNICAS / DIFICULDADES

Optamos por buscar uma biblioteca pronta para manipulação das filas. Encontramos no kernel do linux a biblioteca `list.h` (`kernel_source/include/linux/list.h`), porém tal biblioteca estava preparada para ser utilizada apenas pelo kernel do linux, não podendo ser reaproveitada em userspace. Decidimos que iríamos tentar adaptá-la para userspace, porém uma busca no google permitiu descobriremos que alguém já havia feito isso (<http://www.mcs.anl.gov/~kazutomo/list/index.html>). Fizemos uma pequena alteração para ela funcionar plenamente para nosso software.