Trabalho de Sistemas Operacionais I

micro-núcleo FIFO não preemptivo, com prioridades

Questionário

1. Nome dos componentes do grupo e número do cartão.

Gabriel Visconti - 162754
João Gross - 180171

2. Descrição da plataforma utilizada para desenvolvimento. Qual o tipo de processador (número de cores, com ou sem suporte HT)? Qual a distribuição GNU/Linux utilizada e a versão do núcleo? Qual a versão do gcc? Se o trabalho foi feito ou não em ambientes virtualizados? Em caso afirmativo, qual a máquina virtual utilizada (versão)?

Durante o desenvolvimento o código foi rodado nos laboratórios da inf com processador Core 2 VPro, e na máquina dos alunos: uma com processador Core 2 Quad e outra com Intel Core 2 Duo CPU E7400 @ 2.80GHz, ambas sem suporte a HT. A versão do GNU/Linux foi o Ubuntu 11.04 e Gnome Versão: 2.32.1. A versão do gcc nos laboratórios da inf é a 4.4.3 e do computador dos componentes grupo é 4.5.2-1ubuntu3.

3. Para cada programa de teste elaborado pelo grupo: descrever o que programa faz; indicar claramente quais os parâmetros a serem passados para sua execução e qual a saída esperada.

- teste t01_queue_FIFO: utilizado para verificar se os processos são corretamente criados com seu PCB, se a criação da fila está sendo feita e na ordem esperada e se a estrutura de dados da fila simplesmente encadeada está bem implementada. Nenhum parâmetro de entrada é esperado.
- teste t02_mproc_create: utilizado para verificar se a implementação da mproc_create está certa. Criamos vários processos corretos e dois processos com prioridades que não existem (0 e 3, por exemplo), nesses esperamos erro. Imprimimos a fila dos processos ready para mais uma vez testar a fila (readyQueue).
- t03_mproc_create_and_scheduler: utilizado para verificar se a função scheduler está escalonando os processos da fila. Faz os testes se o processo é Ready, Blocked ou Executing. Cada vez que executa um processo, esse é retirado da fila.
- t04_mproc_create_and_scheduler: ídem ao t03, com a diferença que cada processo está associado a uma função diferente na sua criação, diferente do t03 que tem a apenas uma função.
- t05_mproc_yield: testa a primitiva mproc_yield pela primeira vez. Quatro processos são criados e todos possuem a mesma prioridade.
- t06_mproc_yield_multi_priority: ídem ao t05, porém agora com múltiplas prioridades, forçando a escolha pelo processo com maior prioridade assim que houver um yield.
- t07_mproc_join: testa a primitiva mproc_join pela primeira vez, fazendo apenas um

processo realizar join.
- t08_double_join: também testa a primitiva join. Neste teste dois processos realizam join para o mesmo pid.
- t09_create_200_processes: testa o limite de criação de processos, criando no máximo os 128 da definição do projeto. Após, roda todos os processos criados.

4. Explique o funcionamento da primitiva mproc_create desenvolvida pelo grupo, citando as principais estruturas de dados envolvidas e funções chamadas.

A primitiva mproc_create recebe como parâmetro a prioridade, a rotina para executar e o argumento dessa rotina. Primeiro é testado se a prioridade passada está dentro daquelas aceitas (média ou baixa), caso contrário retorna erro. O PCB é inicializado, sendo que o PID é escolhido dentre um vetor de 128 posições, ou seja, será alocado para o PID o primeiro índice do vetor cujo valor for zero (significa que não foi usado). Por fim o processo é inserido na fila dos prontos para executar (readyQueue). O retorno da primitiva é o PID alocado.

5. Explique o funcionamento da primitiva mproc_yield desenvolvida pelo grupo, citando as principais estruturas de dados envolvidas e funções chamadas.

A primitiva mproc_yield lida apenas com um estrutura global, que corresponde ao PCB do processo que está executando no momento e a primitiva não requer nenhum parâmetro. Quando chamado, o mproc_yield atualiza o estado do processo corrente (que está executando no momento) como ready, ou seja, ele está disponível para entrar na lista de processos readys. Após a alteração do estado para ready, é realizada uma troca de contexto deste processo com o scheduler. No scheduler é ativada uma lógica que busca na lista de processos ready um processo com prioridade igual ou maior ao do processo que realizou o yield. Se a busca retornar um processo, aquele que fez yield entra no final da lista de readys. Caso contrário, volta novamente a executar.

6. Explique o funcionamento da primitiva mproc_join desenvolvida pelo grupo, citando as principais estruturas de dados envolvidas e funções chamadas.

A primitiva mproc_join quando chamada, faz com que o processo que a chamou fique bloqueado até que o pid passado como argumento à primitiva mproc_join finalize sua execução. Caso o pid passado como parâmetro seja um pid que não possui processo a ele associado, então o join é simplesmente ignorado, e o processo que o chamou continua sua execução. Se o pid pasado como parâmetro for um pid válido, ou seja, há um processo associado àquele pid, então o processo que realizou o join pára sua execução e entra em uma lista de processos bloqueados.

Quando um processo termina sua execução, é pesquisado dentre os processos bloqueados quais deles possuem alguma relação de join com o processo recém finalizado. Se houver algum processo bloqueado que realizou join com o processo finalizado, então este processo bloqueado vai para a fila de readys. O scheduler continua o seu gerenciamento normalmente.

Vale lembrar que se dois processos realizam join entre si, ou seja, o processo 1 fez join com o processo 2 e o processo 2 fez join com o processo 1, esses dois processos permanecem bloqueados e continuarão assim até o programa finalizar.

7. Descrever o que funciona no núcleo desenvolvido e o que NÃO está funcionando. Em caso de não funcionamento, dizer qual é a sua visão do porquê deste não funcionamento.

Dos testes que realizamos nenhum indicou erro ou não funcionamento de algum requisito do projeto, logo falaremos sobre o funcionamento do núcleo.

Inicialmente os processos são criados a partir da primitiva mproc_create, cada um com seus argumentos de prioridade, função associada e argumento. Feito isso, o scheduler é acionado, e fica rodando até que todos os processos que estão na lista de readys tenham concluído sua execução, ou seja, até o momento em que a lista seja vazia.

Quando o scheduler começa a executar, é pego da lista de readys o primeiro elemento, pois esta lista, na verdade é uma fifo. Este processo executa, podendo bloquear, finalizar ou ceder voluntariamente a cpu (yield). Ao finalizar, é ativada a lógica dos processos bloqueados, como descrita na primitiva mproc_join e em seguida um novo processo é selecionado, o primeiro da lista de readys. E assim a execução continua até que todos os processos tenham finalizado.

Para os processos que realizam yield e os que realizam bloqueio, a descrição do que ocorre foi explicada nos itens 5 e 6 respectivamente.

8. Qual a metodologia de teste utilizada? Isto é, quais foram os passos (e programas) efetuados para testar o núcleo desenvolvido? Foi utilizado um debugger? Qual?

Para debugar o programa utilizamos muita análise de conteúdo de variáveis com printf e com breakpoints (getchar). Mas também utilizamos o GDB, que é um debugger que nos possibilita ver o que ocorre com o conteúdo das variáveis em tempo de execução ou mesmo o que aconteceu quando ocorre uma falha de segmentação (se alocamos um ponteiro e não desalocamos, acessos a trechos de memória que não pertencem ao nosso código, etc).

Também utilizamos o valgrind com electric fence, porém muito pouco, pois não quisemos perder muito tempo estudando a documentação do debugger, preferindo dar mais atenção ao trabalho e usar técnicas de debug que estamos mais familiarizados.

(http://www.parl.clemson.edu/~wjones/summerStudents/memoryDebugging/pres.pdf)

9. Quais as principais dificuldades encontradas e quais as soluções empregadas para contorná-las.

Inicialmente tivemos problemas com a composição das filas. Muitas vezes esquecemos de desalocar pcbs ao removê-los das listas, ou mesmo realizamos a atualização dos ponteiros da fila encadeada de forma errada, o que gerou falha de segmentação. Porém revisamos o código, realizamos novos testes e consertamos esses problemas.

O maior problema enfrentado foi com relação à troca de contexto. Ao realizar a troca de contexto do scheduler com um processo sempre tínhamos como retorno uma falha de segmentação. Por fim acabamos descobrindo, juntamente ao mestrando Eduardo, que

estávamos inicializando uma variável indevidamente da estrutura que havíamos criado e portanto ocorria o erro.