Software Básico 2008.2

Trabalho Prático 3: Escalonamento de Threads

Data de entrega: 01/12/2008
Trabalho individual ou em dupla

1 Objetivo

Utilizando primitivas de manipulação de contexto de processos no Linux ou rotinas em assembly (feitas por você ou obtidas da rede), desenvolver um pacote gerenciador de threads preemptivas em nível de usuário que suporte diferentes políticas de escalonamento.

2 Problema

Neste trabalho, você deverá implementar um pacote gerenciador de threads preemptivas. A implementação será feita no contexto de um único processo de usuário no Unix. Ou seja, você tem um único processo Unix que poderá criar várias threads para executar diferentes atividades independentes. Este processo deverá usar o pacote gerenciador para criar, terminar e gerenciar as threads criadas. O Unix continuará a ver seu programa como um único processo. Logo, a fim de proporcionar um melhor desempenho, o seu pacote deve fornecer primitivas para **escalonamento** da execução das threads, assumindo que as mesmas podem ser preemptadas em função da política de escalonamento adotada. O seu pacote deve fornecer a possibilidade de diferentes políticas de escalonamento. A aplicação do usuário, ao inicializar o gerenciador, deverá determinar qual política será adotada.

O gerenciador deve ser capaz de chavear de uma thread para outra, quando a primeira executa explicitamente uma das chamadas que resultam em interrupção de sua execução (vide seção a seguir), tal como a *mythread_yield()*, ou quando ela é preemptada.

3 Implementação

3.1 Threads Preemptivas

O pacote gerenciador de threads deve implementar a seguintes funções para uso pelas aplicações:

- *void mythread_init(char sched_type)* : inicializa o pacote the threads. Neste caso, o programa principal que executa esta função se torna a primeira thread, de número zero. O parâmetro define a política de escalonamento a ser adotada pelo gerenciador para esta aplicação conforme apresentado na seção seguinte.
- *void mythread_t mythread_create(mythread_func f, void *param, int sched_info)* : cria uma nova thread a partir da função f dada, que será iniciada com o parâmetro

indicado, retornando o identificador da nova thread. O tipo *mythread_func* é definido como:

*typedef void (*mythread_func)(void*);*

que pertence à thread correspondente à execução corrente. As funções reais usadas podem receber um inteiro ou um apontador para um tipo específico. Deve ser usado *type casting* para ajustar o apontador da função e do parâmetro ao tipo genérico. O último parâmetro *sched_info* corresponde à prioridade da thread. Ele deverá ser um valor inteiro de 1 a 5, sendo 5 a maior prioridade. Este parâmetro somente é utilizado caso o gerenciador escolhido em *mythread_init* seja SCHED_PRIO. Neste caso, a thread criada por *mythread_init* automaticamente recebe prioridade 5.

- *mythread_t mythread_self(void)* : retorna o identificador da thread em execução.
- *void mythread_destroy(mythread_t t)* : destrói a thread indicada, mesmo que ela esteja pronta para executar ou suspensa.
- *void mythread_yield(void)* : suspende a execução da thread corrente e retorna o controle para o gerenciador de threads, que deverá escolher a thread para executar em seguida, conforme política de escalonamento escolhida.
- *void mythread_suspend(void)* : semelhante à anterior, mas torna a thread corrente indisponível para execução, isto é, ela não voltará a executar até que alguma outra thread libere-a através de uma chamada à próxima função.
- int mythread_reactivate(mythread_t athread) : retorna a thread indicada à condição de executável. Retorna um valor diferente de zero se a thread indicada estava realmente suspensa e foi reativada com sucesso e zero caso contrário.
- int mythread_exit(void) : termina a thread atual, retornando o controle para o gerenciador.
- int mythread_join(mythread_t athread) : bloqueia a thread que executa a chamada até que a thread identificada como parâmetro termine, retornando zero em caso de sucesso.
- int mythread_join_all(void) : bloqueia a thread que executa a chamada até que não haja nenhuma outra thread em condição de executar. Retorna zero se não há mais threads, um valor maior que zero igual ao número de threads suspensas (se tais existirem) ou um valor menor que zero em caso de erro (se fizer sentido na sua implementação).

Além disso, como uma thread pode ser interrompida a qualquer instante (preempção), é preciso também definir uma forma de garantir "exclusão mútua", isto é, garantir que sequências de instruções que devam ser atômicas em relação ao resto do programa não sejam interrompidas. Para tanto, você deverá implementar também as seguintes funções:

- *mymutex_t* myt_mutex_create(void)* : cria uma variável do tipo *mymutex_t* , para servir de identificador de uma região de exclusão mútua.

- *void myt_mutex_lock(mymutex_t* mtx)* : se *mtx* não estiver travado, a thread em execução pode prosseguir, travando *mtx*; caso contrário, ela deve ser suspensa até que *mtx* seja destravado.

- *void myt_mutex_unlock(mymutex_t* mtx)* : destrava *mtx*; caso haja outras threads suspensas tentando executar *myt_mutex_lock* na mesma variável, uma delas deve ser liberada para continuar, o que resultará em novo travamento de *mtx*.

As operações de travamento e destravamento podem ser implementadas utilizando-se o mesmo princípio usado nas funções *mythread_suspend* e *mythread_reactivate*.

Por fim, você deverá implementar primitivas para comunicação e trocas de dados entre **duas** threads via trocas de mensagens. Para trocar mensagens, duas threads devem criar um canal de comunicação através de uma fila de mensagem. Filas de mensagens têm tamanho limitado a até QUEUE_SIZE bytes. Cada mensagem é na verdade uma estrutura de dados contendo um número variável de bytes (*char*∗) e um indicador da direção do fluxo (por exemplo: qual das duas threads participantes do canal é a origem da mensagem). O seu gerenciador de threads deverá ser responsável pela manutenção e gerenciamento das filas de mensagens, cada uma identificada unicamente através de um identificador criado pelo gerenciador e utilizado para envio e recebimento de mensagens pelas threads participantes. Para tanto, você deverá incluir as seguintes primitivas no seu pacote de gerenciamento: de threads:

- *int myqueue_create(int key):* esta função cria uma fila de mensagens. O parâmetro *key* é utilizado pelas duas threads que desejam se comunicar para que possam estabelecer um canal único de comunicação. Em outras palavras, duas threads que desejam se comunicar via uma fila de mensagens devem chamar esta função passando a mesma chave *key* como parâmetro. A primeira chamada resulta na criação e inicialização das estruturas de dados associada à fila bem como na vinculação da thread chamadora como uma das participantes do canal de comunicação. A segunda chamada, realizada por uma thread dif erente (você deverá testar isto), vincula esta última como segundo participante do canal. A partir daí, o canal está pronto para ser utilizado. Novas chamadas de *myqueue_create* com mesmo parâmetro devem indicar erro. Esta função retorna -1 em caso de erro e um identificador da fila criada em caso de sucesso.

- *int mythread_sendmsg(int queue_id, mymsg_t msg, int bytes)* : esta função insere a mensagem indicada na fila de mensagens cujo identificador é *queue_id*. O terceiro parâmetro indica o tamanho da mensagem (do vetor com mensagem, não incluindo o indicador de direção) em bytes. Caso não haja espaço disponível na fila de mensagens da thread destino, a mensagem deverá ser truncada. Esta função retorna o número de bytes da mensagem efetivamente inseridos na fila do destino.

- *int mythread_recvmsg(int queue_id, mymsg_t *msg, int *byte s, char flag)* : esta função retira uma mensagem da fila com identificador *queue_id*, inserindo-a na estrutura *msg*, definida pela thread chamadora. O parâmetro bytes define o número

máximo de bytes que podem ser lidos da fila para a estrutura *msg*. Ele é ajustado para o número de bytes da mensagem efetivamente recebidos. A função retorna o número de bytes recebidos, caso uma mensagem seja recebida com sucesso. O parâmetro $flag$ determina a ação a ser tomada caso não haja uma mensagem na fila para ser recebida. Se $flag$ é igual a NON_BLOCK, a função deve retornar imediatamente, retornando um valor -1. Caso $flag$ seja igual a $BLOCK$, então a thread chamadora deverá ser temporariamente suspensa até que uma mensagem seja recebida.

Para simplificar a sua implementação, você pode assumir que não mais que $NTHREADS = 100$ threads existirão no mesmo tempo no sistema. Você deve discutir no seu relatório as implicações desta premissa para o projeto do seu gerenciador.

3.2 Escalonamento

O seu gerenciador de threads deve permitir que a aplicação escolha uma d entre as seguintes políticas:

- SCHED_RR: esta política é a *round robin*. Nela, as threads em estado executável estão em um fila. O gerenciador define intervalos de tempo (*quanta*) para execução de cada thread. Terminado este tempo, a thread em execução é suspensa de forma transparente, inserida no fim da fila de executáveis, e a próxima thread na fila é escolhida para executar. A duração de cada intervalo de tempo é um parâmetro do seu gerenciador.

- SCHED_PRIO: esta política de escalonamento leva a prioridade (1 a 5, sendo 5 a mais alta) de cada thread em consideração. Threads com prioridade mais alta assumem a execução primeiro. A iniciação de uma nova thread causa a preempção da thread atualmente em execução, caso esta última tenha prioridade menor. Você deve elaborar algum mecanismo para evitar inanição de threads com prioridades mais baixas e discutí-lo com detalhes no relatório.

Para implementar a política *round robin*, você deverá usar as funções *getitimer()* e *setitimer()* e tratadores de sinais associado ao sinal de relógio (SIGALRM), de forma semelhante ao que foi feito no TP1.

4 Recursos Disponíveis

O mais difícil na implementação de threads é o chaveamento entre elas. Como fazer isto no nível de usuário? Basta lembrar que o que define o fluxo de execução de um processo, além da CPU, é o conteúdo da pilha do sistema, identificada pelo *stack pointer*. A pilha original é definida pelo sistema operacional e utilizada de acordo com as convenções definidas pelo compilador. Para se conseguir diversos fluxos de execução, precisamos então de diversas pilhas, para que cada uma guarde o histórico de chamadas de função de cada fluxo. Além disso, precisamos da habilidade de salvar e restaurar o estado da CPU (o conteúdo dos registradores e as flags) para cada thread. Como não é preciso

acesso aos registradores especiais de mapeamento de memória (que são comuns para o processo todo e não particulares por thread), não é preciso executar em modo kernel para chavear o contexto. Isto normalmente é feito utilizando código assembly apropriado, não em linguagem de alto nível. Alternativamente, hoje isso já está previsto na biblioteca padrão C do Linux, que define funções que podem ser usadas, com algum cuidado, para se fazer isso.

No C/Unix é possível pular de uma parte do programa para outra utilizando as funções *setjmp* e *longjmp* (confira o manual on-line do Linux, *mansetjmp* e *manlongjmp*). O processo de se salvar o estado em um *setjmp* é exatamente o mesmo necessário para se salvar o contexto da thread e o *longjmp* faz a mudança de contexto. (Notar que há problemas de compatibilidade com certas versões do Kernel linux/GCC/Libc.) Outra opç ao (recomendada neste trabalho) é usar as funes *makecontext*, *setcontext*, *getcontext*, *swapcontext* (confira o manual) que são chamadas de sistema especialmente definidas para lidar com trocas de contexto. Essas ferramentas facilitam principalmente a questão de armazenamento, troca e carga de contexto, sendo bem documentadas na Web e no manual on-line do Linux. Um exemplo de utilização de contextos do linux pode se encontrado no arquivo $exemplo - contexto.c$ (disponível no site da disciplina).

As definições necessárias para implementação do pacote de gerenciamento de threads preemptivas estão no arquivo *mythread.h* (também disponível no site da disciplina). Além disto, exemplos de programas de teste serão disponibilizados no site também.

5 O que deve ser apresentado

Cada aluno/dupla deve entregar, além do código da implementação, um relatório curto que deve conter uma descrição da arquitetura adotada e a justificativa para sua escolha da primitiva de manipulação de contexto, as estruturas de dados utilizadas e decisões de implementação não documentadas nesta especificação.

6 Submissão Eletrônica

Os trabalhos deverão ser entregues (juntamente com o relatório gerado) em um arquivo do tipo zip ou tar.gz através do LearnLoop (turma SoftBas). **O código fonte deve ser bem comentado.**

7 Observações Gerais

1. Comece a fazer este trabalho logo, enquanto o problema está fresco na memória e o prazo para terminá-lo está tão longe quanto jamais poderá estar.

2. **Vão valer pontos a clareza, identação e comentários no programa, bem como a qualidade da documentação que o acompanhe.**

3. O trabalho deve ser desenvolvido de forma independente por cada aluno/dupla. Não é permitido discutir os aspectos do programa e soluções adotadas com outro s alunos,

e é terminantemente proibido compartilhar programas ou trecho de programas. **Tal comportamento será punido severamente.**

Última alteração: 10 de novembro de 2008