Sistemas Operacionais I
Chaveamento de Contexto

Vítor De Araújo – 173285 – Turma B

1. Introdução

O objetivo do trabalho é estudar formas de implementação de chaveamento de processos, bem como implementar um escalonador de threads não-preemptivo em C. Foram analisadas duas formas de implementação: uma utilizando as funções `setjmp` e `longjmp`, e uma utilizando as funções da família `getcontext`.

2. Análise das formas de implementação

As funções `setjmp` e `longjmp` fazem parte da biblioteca padrão do C, e permitem realizar desvios não-locais de execução. A função `setjmp` armazena um valor descrevendo o contexto atual em uma variável; esse valor pode ser utilizado mais tarde pela função `longjmp` para retornar ao ponto onde o contexto foi salvo. Embora permitam chaveamento de contexto, essas funções não permitem salvar o estado da pilha de um processo, mas apenas a posição do *stack pointer* de um dado contexto. Assim, elas podem ser usadas para saltar de um contexto mais profundo para um mais baixo na *call stack*, mas o contrário não pode ser feito de maneira segura. Threads criadas com essas funções teriam que iniciar do mesmo ponto sempre que fossem re-escalonadas.

Já as funções da família `getcontext` permitem que cada thread possua sua própria pilha. Assim, uma thread pode salvar seu contexto e liberar o processador para outra função, e sua execução poderá ser retomada exatamente do ponto onde o contexto foi salvo, sem que haja risco de a outra thread ter sobrescrito a pilha. São quatro as funções dessa família: `getcontext`, que obtém o contexto atual; `setcontext`, que ativa um contexto previamente salvo; `makecontext`, que altera um contexto previamente salvo, fazendo com que ele chame uma função específica quando esse contexto for chamado, ao invés de saltar para o ponto onde o contexto foi originalmente salvo; e `swapcontext`, que simultaneamente salva o contexto atual e ativa um novo contexto. Essas funções armazenam o contexto em uma estrutura do tipo `ucontext_t`, que possui, entre outros, um campo para especificação de uma pilha individual para a thread.

Foi escolhida a família `getcontext` de funções para a implementação do escalonador de threads devido a essa flexibilidade de permitir retomar a execução de uma thread do ponto em que parou.

3. Implementação

O escalonador implementa uma API para criação e manipulação de threads previamente especificada. A implementação está contida no arquivo `sched.c`, e o header especificando a API em `sched.h`. Todas as funções da API foram implementadas.

A fila de threads foi implementada como uma lista encadeada circular, através de uma estrutura `thread_t`. Ela contém os seguintes campos:

- `tid` - Identificador da thread
- `context` - Informação de contexto
- `numwaiters` - Número de processos que estão aguardando o término da thread
- `status` - Contém o exit status da thread após seu término
- `next` - Ponteiro para a próxima thread na fila.

O escalonador mantém duas variáveis globais, `so_current` e `so_previous`, que apontam para o primeiro e o último processo da fila, respectivamente. Também há uma variável `so_exit_context`, que contém um contexto utilizado para encerrar threads que não chamam explicitamente a função `so_exit`. A função `so_init` inicializa essas variáveis, cria uma estrutura `thread_t` correspondendo à thread principal, e coloca-a como única thread na fila.

A função `so_create` é usada para adicionar novas threads à fila. Ela cria um novo contexto com as funções `getcontext` e `makecontext`, aloca uma pilha para o contexto, ajusta o ponteiro `uc_next` do contexto de modo que o contexto `so_exit_context` seja ativado se a thread for encerrada não-explicitamente, e coloca a thread no final da fila do escalonador.

Tanto `so_main` quanto `so_create` utilizam a função auxiliar `so_make_thread` para alocar a estrutura da thread e atribuir valores padrão a seus campos.

A função `so_yield` transfere o controle da CPU para a próxima thread na fila, ajustando os valores das variáveis `so_current` e `so_previous` e invocando a função `swapcontext`, salvando assim o contexto corrente na thread que está deixando de executar, e ativando o contexto da thread seguinte.

A função `so_exit` encerra a execução de uma thread, removendo-a da fila, ajustando o campo status da estrutura `thread_t` com o valor apropriado, e o campo `next` para `NULL`. Se não há threads a esperando, a memória usada pela pilha e pela estrutura são liberadas; esse procedimento é realizado pela função `so_forgo_thread`.

A função `so_join` espera que a thread especificada como argumento termine. Enquanto a thread esperada não terminar, ela chama `so_yield`, transferindo o controle para a próxima thread. Para testar pelo término da thread, a função mantém um ponteiro para a thread, obtido com a função `so_find_thread`, e verifica se o campo `next` da estrutura é `NULL`. Quando ocorre o término da thread, a função recupera seu exit status, decrementa `numwaiters`, e chama `so_forgo_thread` para liberar a thread se ninguém mais a estiver esperando.

Outra maneira de implementar a `so_yield` seria ter uma fila separada para threads que estão bloqueadas esperando por outras. Para isso seria necessário, além de manter a lista de bloqueados (para que uma thread bloqueada pudesse ser encontrada pela `so_find_thread`), manter em cada thread uma lista de threads que a esperam (para que se possa reativá-las quando a thread termina). Isso requeriria desassociar a estrutura da thread da estrutura da lista (pois uma thread poderia estar em duas listas). Por simplicidade, optei pela primeira abordagem.

Finalmente, a função `so_getid` retorna o identificador da thread em execução.