Relatório do Trabalho Prático – Sistemas Operacionais

Ediana da Silva de Souza[1], Êrica Peters do Carmo[1]

[1]Universidade do Estado de Santa Catarina (UDESC) - Joinville/SC

edianadasilvadesouza@gmail.com, ericapetersc@gmail.com

1. Introdução

O presente relatório foi desenvolvido como parte do Trabalho Prático da disciplina de Sistemas Operacionais do curso de Bacharelado em Ciência da Computação da Universidade do Estado de Santa Catarina.

A próxima seção descreve a forma como foi implementado o controle de concorrência e a API utilizada no programa `principal.c`. Nesse programa, dado um argumento `x`, a thread principal (`main`) cria outras `x+1` threads. Dessas, uma thread é responsável por consumir os votos em uma lista e inseri-los no placar, enquanto as outras são responsáveis por produzir os votos a partir de um arquivo de entrada (o nome do arquivo é também passado como argumento) e inseri-los na lista para depois serem consumidos.

Ao final da execução, o programa imprime na tela e salva um arquivo de saída com o placar da votação, composto pelo nome dos candidatos, quantidade de votos de cada um, quantidade de votos total (válidos e inválidos) e tamanho máximo da fila de votos.

2. Controle de Concorrência

O programa `principal.c` utiliza a API de Pthreads, especificamente mutexes e variáveis de condição.

Nesse programa, a lista onde os votos são armazenados após serem produzidos é a região crítica. Para garantir exclusão mútua nessa região, foi utilizado um mutex (`the_mutex`) responsável por liberar e bloquear as threads durante a sua execução.

A thread consumidora executa o comando `pthread_mutex_lock(&the_mutex)` no início da sua execução e a cada vez que for consumir um voto (enquanto a lista de votos não estiver vazia e as threads produtoras não tiverem sido encerradas). Após consumir um voto, a thread consumidora executa o comando `pthread_mutex_unlock(&the_mutex)` e libera a região crítica.

As threads produtoras executam o comando `pthread_mutex_lock(&the_mutex)` antes de inserir um voto na lista, para garantir acesso exclusivo à região crítica. Após a inserção, elas executam o comando `pthread_mutex_unlock(&the_mutex)`.

Para garantir a sincronização entre as threads foi utilizada a variável de condição `condc`. Através do comando `pthread_cond_signal(&conc)` é enviado um sinal ao consumidor para que ele saia do seu estado de bloqueio e continue sua execução. Esse comando é acionado sempre que as threads de produção inserem elementos na lista e

quando todas as threads se encerram.

3. Estruturas de dados utilizadas

No programa foram utilizadas duas estruturas de dados semelhantes responsáveis por fazer o armazenamento de diferentes informações.

A lista_candidatos é uma lista encadeada com os campos referentes ao nome do candidato, número do candidato e quantidade de votos. As informações referentes ao nome e ao número são lidas e definidas na função `inicializa_lista_candidatos`. A quantidade de votos de cada nó é alterada de acordo com o consumo de votos que a thread consumidora está executando na lista_contagem. A cada número consumido, a thread anda pela lista até encontrar o nó com o mesmo número do candidato e incrementa em um a quantidade de votos.

A lista_contagem é uma lista encadeada com um campo referente a um numero inteiro que guarda o número do candidato votado. Esse número é produzido (lido em um arquivo e inserido na lista) pelas threads consumidoras.

4. Conclusões

O programa principal.c realiza suas funções conforme o esperado, contabilizando os votos de uma eleição. A partir da utilização das threads, mutexes e variáveis de condição, foi possível impedir que várias threads alterassem a região crítica simultaneamente e garantir o bom funcionamento da aplicação.