UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
INF01151 – SISTEMAS OPERACIONAIS II N

ATIVIDADE DE PROGRAMAÇÃO: EXCLUSÃO MÚTUA IMPLEMENTADA EM SOFTWARE

Adilson Enio Pierog – 00158803
Andres Grendene Pacheco - 00264397
Mateus Luiz Salvi – 00229787
Luís Filipe Martini Gastmann - 00276150

Nesta atividade, o objetivo é encapsular o pseudocódigo do Algoritmo da Padaria nas primitivas:

void lamport_mutex_lock (int thread_id);
void lamport_mutex_unlock (int thread_id);

O trabalho consiste nos seguintes arquivos:

- program.c: Contém o programa principal, o qual deve ser compilado;

- lamport.c: Implementação do algoritmo baseado no Algoritmo da Padaria de Lamport para a exclusão mútua;

- lamport.h: Header com importação de bibliotecas, definição de constante e funções definidas;

- liblamport.so: Biblioteca com as primitivas implementadas, vinculada dinamicamente ao programa, também deve ser compilada.

- makefile: para facilitar a compilação e gerar o binário executável: *program* e a livraria liblamport.

O programa principal, arquivo “program.c”, cria múltiplas threads para executar uma tarefa simples de incrementar uma variável compartilhada. Ele utiliza a biblioteca POSIX thread para gerenciamento de threads se iniciado com o parâmetro “ptrhead”. Caso iniciado com o parâmetro “lamport”, o programa fará o gerenciamento das threads através do algoritmo de lamport. O programa valor final da variável *accumulator* após todas as threads terem terminado seus trabalhos.

Exemplo de execução: ./program ptrhead

“libbakery.c”: Esse arquivo é uma implementação do algoritmo baseado no Algoritmo da Padaria de Lamport para a exclusão mútua em sistemas concorrentes com múltiplas threads. O objetivo do programa é controlar o acesso de várias threads a uma seção crítica, garantindo que apenas uma thread entre nessa região por vez. Por sua vez, “libbakery.h” importa as bibliotecas pthread.h (para ser usada quando necessário) além de outras livrarias nativas necessárias para execução. Ele define que serão criadas dez threads, e após, inicializa variáveis, implementa mecanismo de bloqueio, libera o “mutex“, e contém a função que representa o processo da thread (thread_process(void *arg)).

Esse algoritmo é chamado de "padaria" porque funciona de maneira semelhante a um sistema de senhas de uma padaria: cada thread pega um número ao tentar entrar na seção crítica, e a com o número mais baixo (ou a que chegou antes) é

a próxima a entrar. O algoritmo garante que não haja deadlocks e que as threads entrem na seção crítica de forma justa.

Este programa, então, deve garantir que até N threads possam competir por uma seção crítica, mas apenas uma de cada vez consiga acesso a ela, respeitando a ordem de chegada, conforme o algoritmo de Lamport.

Execução:
A seguir, mostramos a execução do programa em seus dois modos (Lamport e pThread) e comparamos resultados. Usamos a função built-in time do bash para medir o tempo de execução do programa:

```
mateus@Compiuter:/mnt/d/Facul/Sistemas Operacionais II/ThreadedNetworkAdder/Atividade 1 - Algorítmo da padaria$ time LD_LIBRARY_PATH=. ./program lamport

lamport
Joined with thread id 0
Joined with thread id 1
Joined with thread id 2

Final accumulator value: 9000000

real    0m0.172s
user    0m0.425s
sys     0m0.000s
mateus@Compiuter:/mnt/d/Facul/Sistemas Operacionais II/ThreadedNetworkAdder/Atividade 1 - Algorítmo da padaria$ time LD_LIBRARY_PATH=. ./program pthread

pthread
Joined with thread id 0
Joined with thread id 1
Joined with thread id 2

Final accumulator value: 9000000

real    0m0.011s
user    0m0.002s
sys     0m0.000s
mateus@Compiuter:/mnt/d/Facul/Sistemas Operacionais II/ThreadedNetworkAdder/Atividade 1 - Algorítmo da padaria$
```

Implementação:

Funções de *lock* e *unlock* para o algoritmo da padaria:

```c
void lamport_mutex_lock(int i)
{
  choosing[i] = true;
  ticket[i] = max_ticket() + 1;
  choosing[i] = false;
  int j;
  for (j = 0; j < N_THREADS; j++)
  {
    while (choosing[j])
    {
    }
    while (ticket[j] != 0 && ((ticket[j] < ticket[i]) || (ticket[j] == ticket[i] && j < i)))
    {
    }
  }
}

void lamport_mutex_unlock(int thread_id)
{
  ticket[thread_id] = 0;
}
```

Funções de threading respectivas ao modo de execução escolhido através de parâmetro “lamport” ou “pthread”:

```c
void *lamport_thread_process(void *arg)
{
  int i = *((int *)arg);
  int interactionsCounter = 0;

  lamport_mutex_lock(i);

  do
  {
    accumulator++;
    interactionsCounter++;
  } while (interactionsCounter != N_ITERACTIONS);

  lamport_mutex_unlock(i);

  return NULL;
}

void *hw_thread_process(void *arg)
{
  int i = *((int *)arg);
  int interactionsCounter = 0;

  pthread_mutex_lock(&lock);

  do
  {
    accumulator++;
    interactionsCounter++;
  } while (interactionsCounter != N_ITERACTIONS);

  pthread_mutex_unlock(&lock);

  return NULL;
}
```

Conclusão:

O tempo de execução é consideravelmente menor usando-se primitivas em hardware, mesmo que aqui mostramos apenas um tempo de execução para cada, na média o padrão se manteve, com pequenas variações irrelevantes e esperadas.

Tivemos problemas com o algoritmo da padaria quando usando-se as funções de lock e unlock dentro do ciclo de soma, resultando em uma soma final menor do que a esperada, o que não acontece se o ciclo inteiro é considerado como zona crítica, estando entre as chamadas de lock e unlock, onde cada thread executa seus 3 milhões de incrementos de uma só vez antes de liberar quantum à outra thread. Este problema não ocorreu usando-se a livraria pthreads. Também observou-se que a execução é consideravelmente mais rápida quando o ciclo de soma inteiro é tratado

como zona crítica em ambos os casos. Portanto, optamos por utilizar o ciclo de soma inteiro como zona crítica para as duas rotinas, para fins de comparação.