Міністерство Освіти І НАУКИ України
Національний університет "Львівська політехніка"

Інститут **ІКНІ**
Кафедра **ПЗ**

ЗВІТ

До лабораторної роботи № 6
На тему: *“Багатопоточність в операційній системі Linux. Створення, керування та синхронізація потоків”*
З дисципліни: *“Операційні системи”*

Лектор:
Старший викладач ПЗ
Грицай О.Д.

Виконав:
ст. гр. ПЗ-22
Солтисюк Д.А.

Прийняв:
Старший викладач ПЗ
Грицай О.Д.

« ____ » _____ 2022 р.

∑= ____

Львів – 2022

Тема роботи: Багатопоточність в операційній системі Linux. Створення, керування та синхронізація потоків.

Мета роботи: Ознайомитися з багатопоточністю в ОС Linux. Навчитись реалізовувати розпаралелювання алгоритмів за допомогою багатопоточності в ОС Linux. Навчитись використовувати різні механізми синхронізації потоків.

Індивідуальне завдання

1. Реалізувати заданий (згідно варіанту) алгоритм в окремому потоці.
2. Виконати розпаралелення заданого алгоритму на 2, 4, 8, 16 потоків.
3. Реалізувати можливість зміни/встановлення пріоритету потоку (для планування потоків) або встановлення відповідності виконання на ядрі.
4. Реалізувати можливість зробити потік від'єднаним.
5. Реалізувати можливість відміни потоку.
6. Реалізувати синхронізацію потоків за допомогою вказаних методів (згідно варіанту)
7. Порівняти час виконання задачі відповідно до кількості потоків і методу синхронізації (чи без синхронізації).
8. Результати виконання роботи оформити у звіт

Варіант завдання №2:

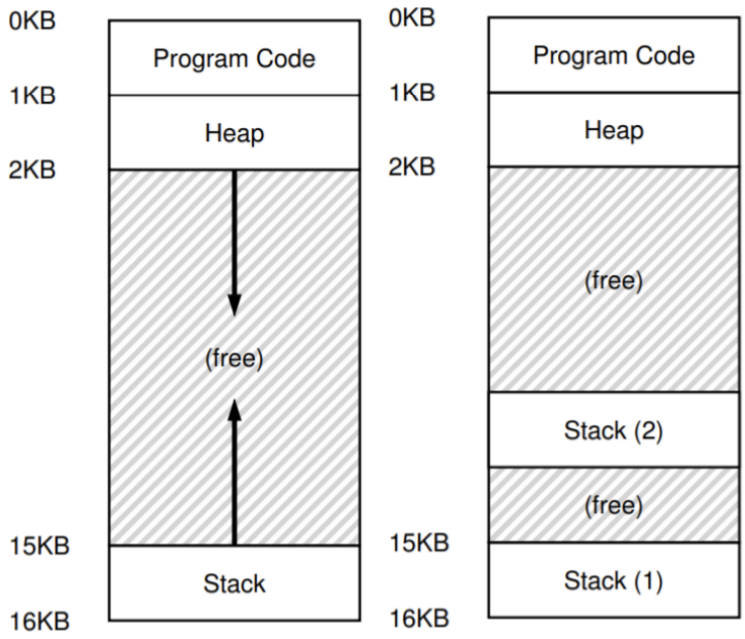
2. Обчислити суму елементів заданого масиву (кількість елементів >10000, елементи рандомні) (Синхронізація: семафор, спінблокуваввння)

Теоретичні відомості

Розглядаючи поняття процесу, визначають ще одну абстракцію для запущеного процесу: потік. У класичному уявленні існує єдина точка виконання в рамках програми (тобто єдиний потік контролю, на якому збираються та виконуються інструкції), багатопотокова програма має більш ніж одну точку виконання (тобто кілька потоків контролю, кожен з яких який отримується та виконується).

Кожен потік дуже схожий на окремий процес, за винятком однієї відмінності: вони мають спільний адресний простір і, отже, мають доступ до одних і тих же даних. Таким чином, стан одного потоку дуже подібний до стану процесу. Він має лічильник програм (PC), який відстежує, звідки програма отримує інструкції. Кожен потік має свій власний приватний набір реєстрів, який він використовує для обчислень; таким чином, якщо на одному процесорі працюють два потоки, при переході від запуску одного (T1) до запуску іншого (T2) має відбутися перемикання контексту. Контекстний перемикач між потоками дуже подібний до перемикання контекстів між процесами, оскільки перед запуском T2

необхідно зберегти регістр стану T1 і відновити стан реєстру T2. За допомогою процесів ми зберегли стан до блоку управління процесами (PCB); тепер нам знадобиться один або кілька блоків управління потоками (TCB) для збереження стану кожного потоку процесу. Однак є одна істотна відмінність у перемиканні контексту, який ми виконуємо між потоками порівняно з процесами: адресний простір залишається незмінним (тобто немає необхідності змінювати, яку таблицю сторінок ми використовуємо). Ще одна істотна відмінність між потоками та процесами стосується стека. У простій моделі адресного простору класичного процесу (однопотокового) є єдиний стек, який зазвичай знаходиться внизу адресного простору. Однак у багатопотоковому процесі кожен потік працює окремо і, звичайно, може залучати різні підпрограми для виконання будь -якої роботи. Замість одного стека в адресному просторі буде по одному на кожен потік.



На цьому малюнку можна побачити два стеки, розповсюджені по адресному простору процесу. Таким чином, будь-які змінні, параметри, повернені значення, що виділяються стеком, та інші речі, які розміщуємо у стеку, будуть розміщені у тому, що іноді називають локальним сховищем потоків, тобто стеком відповідного потоку. Раніше стек і купа могли зростати незалежно, і проблеми виникали лише тоді, коли в адресному просторі вичерпалося місце. Тут немає такої приємної ситуації, оскільки стеки, як правило, не повинні бути дуже великими (виняток становлять програми, які дуже часто використовують рекурсію).

Хід роботи

Спочатку напишу код для програми. Я виконуватиму її в консольному середовищі.

Також додам відлік часу, який рахуватиме, скільки потрібно чекати на результат Запрограмую рішення.

Код:

```
#include <pthread.h>
#include <semaphore.h>
#include <sys/resource.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

#include <algorithm>
#include <chrono>
#include <cmath>
#include <ctime>
#include <fstream>
#include <functional>
#include <iostream>
#include <random>
#include <thread>

int duration;
int sched_policy = SCHED_RR;
int sleep_time;

std::vector<double> numbers;
double resulting_sum = 0;

struct parameters {
  int from;
  int to;
  int method;
  bool with_sleep;
};

sem_t sum_semaphore;
pthread_spinlock_t sum_spinlock;
std::vector<pthread_t> thread_ids;

int thread_count = 0;

class Rand_double {
 public:
  Rand_double(double low, double high)
      : r(std::bind(std::uniform_real_distribution<>(low, high),
                    std::default_random_engine())) {}
```

```
  double operator()() { return r(); }

 private:
  std::function<double()> r;
};

void *count_array_sum(void *lparams) {
  parameters *params = (parameters *)lparams;
  // perform calculations
  int partial_sum = 0;
  for (int idx = params->from; idx < params->to; idx++) {
    partial_sum += numbers[idx];
  }

  // updated shared variable
  if (params->method == 1) {
    sem_wait(&sum_semaphore);
    resulting_sum += partial_sum;
    sem_post(&sum_semaphore);
  } else {
    pthread_spin_lock(&sum_spinlock);
    resulting_sum += partial_sum;
    pthread_spin_unlock(&sum_spinlock);
  }

  if (params->with_sleep) {
    sleep(sleep_time);
  }
}

void *_run_find_array_sum(void *) {
  const auto begin = std::chrono::high_resolution_clock::now();

  // wait for threads to complete
  for (int i = 0; i < thread_ids.size(); i++) {
    pthread_join(thread_ids[i], NULL);
  }

  const auto duration =
std::chrono::duration_cast<std::chrono::milliseconds>(
                             std::chrono::high_resolution_clock::now() -
begin)
                             .count();
  std::cout << "\n-------------\nComputed sum: " << resulting_sum << "\n";
  std::cout << "Time taken: " << duration << "ms\n-------------\n";
  // cleanup
  resulting_sum = 0;
  numbers.clear();
  thread_ids.clear();
  thread_count = 0;
}

void option_find_array_sum(bool with_sleep) {
```

```
parameters *params;
int method = 0;
int nums_size = 0;
Rand_double rd{1, 1000};

if (with_sleep) {
  std::cin.ignore();
  std::cout << "Enter sleep time: ";
  std::cin >> sleep_time;
}

std::cin.ignore();
std::cout << "Enter array size: ";
std::cin >> nums_size;

numbers.reserve(nums_size);
for (int i = 0; i < nums_size; i++) {
  numbers.push_back(rd());
}

std::cin.ignore();
std::cout << "Enter thread amount: ";
std::cin >> thread_count;

std::cout << "Choose method:\n0. Spinlock\n1. Semaphore\n";
std::cin >> method;
std::cin.ignore();

int batch_size = numbers.size() / thread_count;
int bonus_size = numbers.size() - batch_size * thread_count;
pthread_t thread_id;

for (int from = 0, to = batch_size; from < numbers.size();
     from = to, to = from + batch_size) {
  if (bonus_size) {
    to++;
    bonus_size--;
  }
  params = new parameters();
  params->from = from;
  params->to = to;
  params->method = method;
  params->with_sleep = with_sleep;

  pthread_attr_t attrs;
  pthread_attr_init(&attrs);
  if (pthread_attr_setschedpolicy(&attrs, sched_policy)) {
    std::cout << "Failed setting sched_policy";
  }
  pthread_create(&thread_id, &attrs, count_array_sum, params);
  pthread_attr_destroy(&attrs);

  thread_ids.push_back(thread_id);
}
```

```
  pthread_create(&thread_id, NULL, _run_find_array_sum, NULL);
  pthread_detach(thread_id);
}

void option_list_threads() {
  struct sched_param param;
  for (int i = 0; i < thread_ids.size(); i++) {
    pthread_getschedparam(thread_ids[i], &sched_policy, &param);
    std::cout << "idx: " << i << " | tid: " << thread_ids[i]
              << " | prio: " << param.sched_priority
              << " | max prio: " << sched_get_priority_max(sched_policy)
              << std::endl;
  }
}

pthread_t prompt_thread_id() {
  int index = 0;
  std::cout << "Enter idx: ";
  std::cin >> index;
  return thread_ids[std::min(index, thread_count - 1)];
}

void option_renice() {
  auto thread_id = prompt_thread_id();
  struct sched_param param;
  int priority = 0;

  std::cout << "Enter priority: \n";
  std::cin >> priority;

  param.sched_priority = priority;
  if (pthread_setschedparam(thread_id, sched_policy, &param)) {
    std::cout << "couldn't change prio";
  }
}

void option_detach() {
  if (pthread_detach(prompt_thread_id())) {
    std::cout << "couldn't detach thread";
  } else {
    std::cout << "thread has been detached\n";
  }
}

void option_cancel() {
  if (pthread_cancel(prompt_thread_id())) {
    std::cout << "couldn't cancel thread" << std::endl;
  } else {
    std::cout << "thread has been canceled" << std::endl;
  }
}

void option_exit() {
```

```
  pthread_spin_destroy(&sum_spinlock);
  sem_destroy(&sum_semaphore);
}

int main(int argc, char *argv[]) {
  pthread_spin_init(&sum_spinlock, PTHREAD_PROCESS_PRIVATE);
  sem_init(&sum_semaphore, 0, 1);

  char option;
  while (true) {
    std::cout << "Choose option:\n"
                 "f. Find array sum\n"
                 "F. Find array sum with sleep\n"
                 "p. Change thread prio\n"
                 "d. Detach thread\n"
                 "c. Cancel thread\n"
                 "l. List threads\n"
                 "q. Exit\n"
                 "Enter choice: ";
    std::cin >> option;
    std::cout << std::endl;

    switch (option) {
      case 'f':
        option_find_array_sum(false);
        break;
      case 'F':
        option_find_array_sum(true);
        break;
      case 'p':
        option_renice();
        break;
      case 'd':
        option_detach();
        break;
      case 'c':
        option_cancel();
        break;
      case 'l':
        option_list_threads();
        break;
      case 'q':
        option_exit();
        return 0;
    }

    std::cout << "\n\n";
  }
}
```

Протокол роботи

Запущу 1 нитках, щоб порівняти роботу без та із синхронізаціями.

Порахуємо сумму 1000000 випадкових double

Spinlock

```
Enter array size: 1000000
Enter thread amount: 1
Choose method:
0. Spinlock
1. Semaphore
0


Choose option:
f. Find array sum
F. Find array sum with sleep
p. Change thread prio
d. Detach thread
c. Cancel thread
l. List threads
q. Exit
Enter choice:
-------------
Computed sum: 5.00341e+08
Time taken: 75ms
-------------
```

Semaphore

```
Enter array size: 1000000
Enter thread amount: 1
Choose method:
0. Spinlock
1. Semaphore
1


Choose option:
f. Find array sum
F. Find array sum with sleep
p. Change thread prio
d. Detach thread
c. Cancel thread
l. List threads
q. Exit
Enter choice:
-------------
Computed sum: 5.00341e+08
Time taken: 83ms
-------------
```

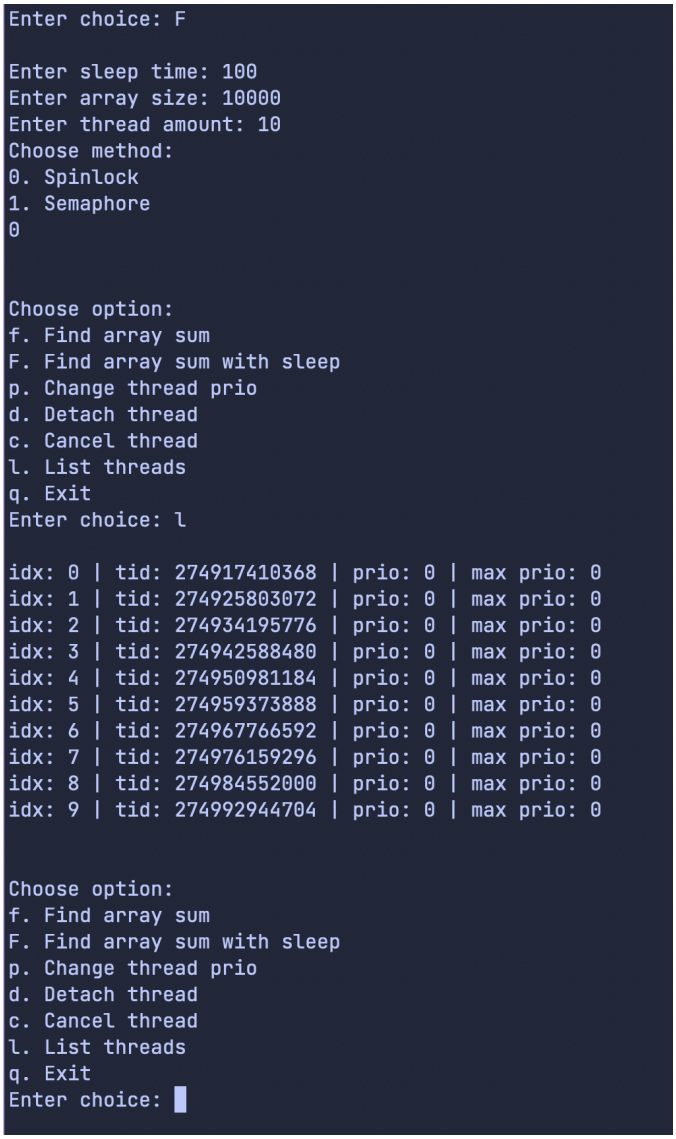
Час виконання приблизно однаковий

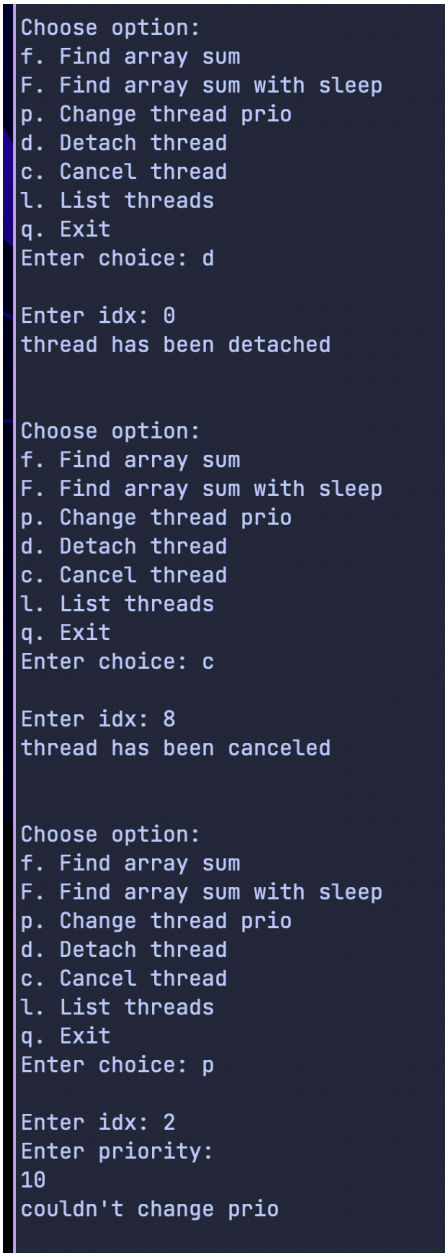
```
Enter array size: 1000000
Enter thread amount: 10
Choose method:
0. Spinlock
1. Semaphore
0


Choose option:
f. Find array sum
F. Find array sum with sleep
p. Change thread prio
d. Detach thread
c. Cancel thread
l. List threads
q. Exit
Enter choice:
-------------
Computed sum: 5.00341e+08
Time taken: 8ms
-------------
```

```
Enter choice:
Enter array size: 1000000
Enter thread amount: 10
Choose method:
0. Spinlock
1. Semaphore
1


Choose option:
f. Find array sum
F. Find array sum with sleep
p. Change thread prio
d. Detach thread
c. Cancel thread
l. List threads
q. Exit
Enter choice:
-------------
Computed sum: 5.00341e+08
Time taken: 10ms
-------------
```

Виконання в 10 нитках

Легко помітити прискорення виконання майже в 10 разів, що доводить ефективність паралельних обчислень. Семафор в порівнянні зі спінлоком витрачає більше процесерного часу, щоб не викликати race condition, в першу чергу це повязано з більшою к-стю системних викликів





Демонстарція операцій над потоками в Linux
(detach, cancel, sche_priority)

Висновок

У цій лабораторній роботі я навчився працювати із потоками та методами їхньої синхронізації, використовуючи Linux. Я ознайомився із спін-локом та семафором, зрозумів, який метод швидший та чому. Я розбив завдання на окремі частинки та використав синхронізацію, щоб отримати потрібний результат.