VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Operating system

Lab 3 - Exercises

Scheduling

Advisor(s): Kha Sang
Student(s): Lê Bùi Trung Dũng 2210573

HO CHI MINH CITY, MAY 2024

Contents

List of Figures

List of Tables

Listings

1 Exercise 1: Implement the FIFO scheduler policy

Vì `wrkid_busy[]` minh họa cho việc worker ứng với id có đang bận hay không, ta có thể kiểm tra lần lượt, nếu `wrkid_busy[index] == 0` thì cập nhật `wrkid_busy[index] = 1` và thực hiện `return index` với `index` là id của worker tương ứng. Ngoài ra ta có thể cập nhập `last_worker` để theo dõi `wrkid` của worker sau cuối vừa mới bị thay đổi `wrkid_busy` thành 0, điều này sẽ giúp cho những lần tìm kiếm kế tiếp, các `worker` có thể được lần lượt sử dụng.

```
int bkwrk_get_worker()
{
  /* TODO Implement the scheduler to select the resource
     entity
   * The return value is the ID of the worker which is not
     currently
   * busy or wrkid_busy[1] == 0
   */
  /**used to track the index of last worker*/
  static int last_worker = 0;
  for (int i = 0 ; i < MAX_WORKER; i++)
  {
    /*check for MAX_WORKER times*/
    int index = (i + last_worker)%MAX_WORKER;
      if (wrkid_busy[index] == 0)
      {
        /*bkwk_assign_worker() also do that*/
        wrkid_busy[index] = 1;
        /*the next find will start at index + 1*/
        last_worker = index + 1;
        return index;
      }
  }
    /**
     * if all workers are busy
     * return -1 to let bktask_assign_woker fail
    */
    return -1;
```

```
}
```

Ta có kết quả thực thi như sau:

Output 1

```
bkwrk_create_worker got worker 22102
bkwrk_create_worker got worker 22103
bkwrk_create_worker got worker 22104
bkwrk_create_worker got worker 22105
bkwrk_create_worker got worker 22106
bkwrk_create_worker got worker 22107
bkwrk_create_worker got worker 22108
bkwrk_create_worker got worker 22109
bkwrk_create_worker got worker 22110
bkwrk_create_worker got worker 22111
Assign tsk 0 wrk 0
Assign tsk 1 wrk 1
Assign tsk 2 wrk 2
worker wake 0 up
Task func - Hello from 1
worker wake 1 up
Task func - Hello from 2
worker wake 2 up
Task func - Hello from 5
```

Output 2

```
bkwrk_create_worker got worker 23131
bkwrk_create_worker got worker 23132
bkwrk_create_worker got worker 23133
bkwrk_create_worker got worker 23134
bkwrk_create_worker got worker 23135
bkwrk_create_worker got worker 23136
bkwrk_create_worker got worker 23137
bkwrk_create_worker got worker 23138
bkwrk_create_worker got worker 23139
bkwrk_create_worker got worker 23140
Assign tsk 0 wrk 0
worker wake 0 up
Task func - Hello from 1
Assign tsk 1 wrk 1
Assign tsk 2 wrk 2
worker wake 1 up
Task func - Hello from 2
worker wake 2 up
Task func - Hello from 5
```

2 Exercise 2: Implement the worker using fork()

Bởi vì `fork()` khác `clone()` ở chỗ `clone()` có thể tạo ra các threads và chúng chia sẽ memory, vì vậy để có đáp án như `clone()`, ta sẽ phải sử dụng IPC để chia sẽ `worker[]` và `wrkid_busy[]`. Vì vậy ta sẽ áp dụng `shmget()` và `shmat()` để tạo và liên kết các vùng shared memory. Vì vật ở mỗi `function()`, ta sẽ sử dụng shared memory, và trong các processes con ta cũng sẽ sử dụng đoạn source code bên dưới để có thể cho phép các process con share memory như `clone()`.

```
key_t my_key   = ftok("./bkwrk.c",1000);
key_t my_key_1 = ftok("./bktask.c",1000);
key_t my_key_2 = ftok("./main.c",1000);
```

```
int shmid1 = shmget(my_key,sizeof(struct bkworker_t)*MAX_WORKER,0666|IPC_CREAT);
int shmid2 = shmget(my_key_1,sizeof(int)*MAX_WORKER,0666|IPC_CREAT);
int shmid3 = shmget(my_key_2,sizeof(int)*MAX_WORKER,0666|IPC_CREAT);

int* wrkid_busy = (int*)shmat(shmid2,NULL,0);
struct bkworker_t* worker = (struct bkworker_t*)shmat(shmid1,NULL,0);
int* argument = (int*)shmat(shmid3,NULL,0);
```

Các processes con sẽ chia sẽ 3 vùng shared memory bao gồm `wrkid_busy`, `worker` và `argument` (dùng để lưu các giá trị tham số cho các process con). Ta có hiện thực ở các process con của threadpool như sau:

```
/* TODO: Implement fork version of create worker */
/* fork() will create a new process which is a copy
of  the parent process*/
sigset_t set;
int s;
sigemptyset( & set);
sigaddset( & set, SIGQUIT);
sigaddset( & set, SIGUSR1);
sigprocmask(SIG_BLOCK, & set, NULL);
int pid = fork();
if (pid == 0)
{
    /**
    * CHILDREN
    */
    key_t my_key   = ftok("./bkwrk.c",1000);
    key_t my_key_1 = ftok("./bktask.c",1000);
    key_t my_key_2 = ftok("./main.c",1000);
    int shmid1 = shmget(my_key,sizeof(struct bkworker_t)*MAX_WORKER,0666|IPC_CREAT);
```

```
20      int shmid2 =
           shmget(my_key_1,sizeof(int)*MAX_WORKER,0666|IPC_CREAT);
21      int shmid3 = shmget(my_key_2,sizeof(int)*
           MAX_WORKER,0666|IPC_CREAT);
22
23      int* wrkid_busy = (int*)shmat(shmid2,NULL,0);
24      struct bkworker_t* worker = (struct
           bkworker_t*)shmat(shmid1,NULL,0);
25      int* argument = (int*)shmat(shmid3,NULL,0);
26
27      sigset_t set;
28      int sig;
29      int s;
30      wrkid_busy[i] = 0;
31      /* Taking the mask for waking up */
32      sigemptyset( & set);
33      sigaddset( & set, SIGUSR1);
34      sigaddset( & set, SIGQUIT);
35      #ifdef DEBUG
36          fprintf(stderr, "worker %i start living tid %d \n",
               i, getpid());
37          fflush(stderr);
38      #endif
39      struct bkworker_t* wrk = &worker[i];
40      while (1)
41      {
42          /* wait for signal */
43          s = sigwait( & set, & sig);
44          if (s != 0)
45          continue;
46
47      #ifdef INFO
48          fprintf(stderr, "worker wake %d up\n", i);
49      #endif
50      /* Busy running */
```

```
51      if (wrk ->func == NULL)
52      {
53          printf("Null function\n");
54      }
55      else
56      {
57          wrk ->func((void*)&argument[i]);
58      }
59      /* Advertise I DONE WORKING */
60
61      wrkid_busy[i] = 0;
62      worker[i].func = NULL;
63      worker[i].arg = NULL;
64      worker[i].bktaskid = -1;
65      argument[i] = -999;
66      }
67
68  return 0;
```

Vấn đề của cách hiện thực này:

- Phải bắt buộc thêm các shared memory và sửa đổi code bên ngoài `/*TODO*/` (bad practice but...).
- Phải bắt buộc implement `end()` để quá trình cha có thể kết thúc các quá trình con một các trực tiếp tránh để các quá trình còn thành `orphan` và phải thực hiện `shmdt` và `shclt` để đảm bảo shared memory được giải quyết sau khi sử dụng xong bởi các processes cha và con.

```
1  void end()
2  {
3      for (int i = 0; i < MAX_WORKER; i++)
4          kill(wrkid_tid[i], SIGTERM);
5      key_t my_key   = ftok("./bkwrk.c",1000);
6      key_t my_key_1 = ftok("./bktask.c",1000);
7      key_t my_key_2 = ftok("./main.c",1000);
```

```
8       int shmid1 = shmget(my_key,sizeof(struct
            bkworker_t)*MAX_WORKER,0666|IPC_CREAT);
9       int shmid2 =
            shmget(my_key_1,sizeof(int)*MAX_WORKER,0666|IPC_CREAT);
10      int shmid3 = shmget(my_key_2,sizeof(int)*
            MAX_WORKER,0666|IPC_CREAT);
11      shmctl(shmid1,IPC_RMID,0);
12      shmctl(shmid2,IPC_RMID,0);
13      shmctl(shmid3,IPC_RMID,0);
14  }
```

Một kết quả khi chạy chương trình với hiện thực `fork()` thay vì `clone()`:

Output 1

```
bkwrk_create_worker got worker 26170
bkwrk_create_worker got worker 26171
bkwrk_create_worker got worker 26172
bkwrk_create_worker got worker 26173
bkwrk_create_worker got worker 26174
bkwrk_create_worker got worker 26175
bkwrk_create_worker got worker 26176
bkwrk_create_worker got worker 26177
bkwrk_create_worker got worker 26178
bkwrk_create_worker got worker 26179
Assign tsk 0 wrk 0
Assign tsk 1 wrk 1
worker wake 0 up
Assign tsk 2 wrk 2
Task func - Hello from 1
worker wake 1 up
worker wake 2 up
Task func - Hello from 2
Task func - Hello from 5
```

Output 2

```
bkwrk_create_worker got worker 26258
bkwrk_create_worker got worker 26259
bkwrk_create_worker got worker 26260
bkwrk_create_worker got worker 26261
bkwrk_create_worker got worker 26262
bkwrk_create_worker got worker 26263
bkwrk_create_worker got worker 26264
bkwrk_create_worker got worker 26265
bkwrk_create_worker got worker 26266
bkwrk_create_worker got worker 26267
Assign tsk 0 wrk 0
Assign tsk 1 wrk 1
worker wake 0 up
Assign tsk 2 wrk 2
Task func - Hello from 1
worker wake 1 up
Task func - Hello from 2
worker wake 2 up
Task func - Hello from 5
```

Ngoài ra, ta có kết quả từ việc chạy `#define STRESS_TEST` như kết quả sau, ta có thể với

15 tasks, và 10 processes trong thread pools, các tasks được tạo ra và chuyển ra bất cứ processes nào đang trong trạng thái `not busy` và việc tìm kiếm sẽ tiến hành từ `worker` đã `busy` trước đó.

```
bkwrk_create_worker got worker 26622
bkwrk_create_worker got worker 26623
bkwrk_create_worker got worker 26624
bkwrk_create_worker got worker 26625
bkwrk_create_worker got worker 26626
bkwrk_create_worker got worker 26627
bkwrk_create_worker got worker 26628
bkwrk_create_worker got worker 26629
bkwrk_create_worker got worker 26630
bkwrk_create_worker got worker 26631
Assign tsk 0 wrk 0
Assign tsk 1 wrk 1
worker wake 0 up
Assign tsk 2 wrk 2
Task func - Hello from 1
worker wake 1 up
worker wake 2 up
Task func - Hello from 2
Task func - Hello from 5
Assign tsk 3 wrk 3
Assign tsk 4 wrk 4
worker wake 3 up
Assign tsk 5 wrk 5
Task func - Hello from 0
Assign tsk 6 wrk 6
worker wake 4 up
Task func - Hello from 1
Assign tsk 7 wrk 7
worker wake 5 up
Task func - Hello from 2
Assign tsk 8 wrk 8
worker wake 6 up
```

```
Assign tsk 9 wrk 9
Task func - Hello from 3
worker wake 8 up
Assign tsk 10 wrk 0
Task func - Hello from 5
worker wake 7 up
Task func - Hello from 4
Assign tsk 11 wrk 1
worker wake 9 up
worker wake 0 up
Task func - Hello from 7
Assign tsk 12 wrk 2
Task func - Hello from 6
Assign tsk 13 wrk 3
worker wake 2 up
worker wake 1 up
Task func - Hello from 9
Task func - Hello from 8
worker wake 3 up
Assign tsk 14 wrk 4
Task func - Hello from 10
Assign tsk 15 wrk 5
worker wake 4 up
Task func - Hello from 11
Assign tsk 16 wrk 6
worker wake 5 up
Task func - Hello from 12
worker wake 6 up
Task func - Hello from 13
Assign tsk 17 wrk 7
worker wake 7 up
Task func - Hello from 14
```

3 Exercise 3: Framework for fork() and join() in theory

Dựa vào multi-tasking programming, ta có thể xây framework cho fork() và join(). Các hiện thực như sau:

1. Khởi tạo: Tạo một pool của các worker threads ở thời điểm đầu khi bắt đầu process, những threads này sẽ đợi các tasks để thực hiện.
2. fork(): Khi một task có thể được thực hiện song song, main thread có thể chia task thành những sub tasks và gắn các sub tasks vào những free thread pool.
3. Mỗi worker thread sẽ thực hiện sub tasks độc lập.
4. join(): Khi một worker thread thực hiện xong, nó sẽ signal main thread, khi tất cả sub tasks được hoàn thành thì join cũng được hoàn thành.
5. Main thread tiếp tục thực hiện.

Ta có thể viết psuedocode như sau:

```
/**
    Pseudocode for fork() and join()
    using thread pool
*/
// Initialization
create_worker_threads();

main()
{
    // Fork
    tasks = split_task_into_subtasks(big_task);
    for each task in tasks
        assign_task_to_worker_thread(task);
    // Join
    results = [];
    for each task in tasks
        results.append(wait_for_worker_thread(task));
    // Continue with sequential code...
```

```
19      use(results);
20  }
```

4 Exercise 4: What is OpenMP in C

OpenMP bao gồm một tập các chỉ dẫn biên dịch `#pragma` nhằm chỉ dẫn cho chương trình hoạt động. Các pragma được thiết kế nhằm mục đích nếu các trình biên dịch không hỗ trợ thì chương trình vẫn có thể hoạt động bình thường, nhưng sẽ không có bất kỳ tác vụ song song nào được thực hiện như khi sử dụng OpenMP. OpenMP là một công cụ mạnh mẽ cho lập trình song song trong C và C++. Nó cho phép người dùng viết mã chạy trên nhiều lõi hoặc luồng mà không cần phải trực tiếp tạo ra các process hay threads bởi người dùng một cách trực tiếp. Ta có thể xét một ví dụ đơn giản sau đây:

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(int argc, char** argv)
5  {
6      #pragma omp parallel
7      {
8          printf("Hello from process: %d\n",
               omp_get_thread_num());
9      }
10     return 0;
11 }
```

Tất cả các chỉ dẫn OpenMP trong C/C++ đều được dùng thông qua `#pragma omp` theo sau là các thông số và kết thúc bằng một ký hiệu xuống dòng. `#pragma` chỉ được áp dụng vào đoạn chương trình ngay sau nó, ngoại trừ lệnh barrier và flush. Một vài các syntax phổ biến trong OpenMP:

- Chỉ dẫn `parrallel`: Chỉ dẫn parrallel bắt đầu một đoạn code thực hiện song song. Nó tạo ra một team bao gồm N threads (N được chỉ định tại thời điểm chạy chương trình, thông thường bằng số nhân CPU, nhưng có thể bị ảnh hưởng bởi một số lý

do khác), các lệnh xử lý tiếp theo ngay sau `#pragma` hoặc block tiếp theo (trong giới hạn) sẽ được thực hiện song song.

- Chỉ dẫn lặp `for`: Chỉ dẫn for sẽ chia vòng lặp for mà mỗi thread trong nhóm thread hiện tại thực hiện một phần của vòng lặp. Trong ví dụ dưới đây vì vòng lặp đã được chia ra cho các thread trong nhóm thực hiện, nên kết quả của vòng lặp có thể sẽ không theo thứ tự từ `0` đến `9`.

```
#pragma omp for
for(int i = 0; i < 10; ++i)
{
    printf(" %d", i);
}
```

Một ví dụ sử dụng OpenMP:

```
#include <stdio.h>
#include <omp.h>
#define SIZE 100
int main()
{
    int i, sum = 0;
    int a[SIZE];
    // Initialise array
    for(i = 0; i < SIZE; i++)
        a[i] = 1;
    //Calculate the sum
    #pragma omp parallel for reduction(+:sum)
    for(i = 0; i < SIZE; i++)
        sum += a[i];
    printf("Sum: %d\n", sum);
    return 0;
}
```

Trong đoạn mã trên, chúng ta sử dụng chỉ thị `#pragma omp parallel for reduction(+:sum)` để chia công việc tính tổng các phần tử của mảng ra cho nhiều luồng. `reduction(+:sum)`

là một cấu trúc giảm trong OpenMP, nó cho phép tổng hợp kết quả từ tất cả các luồng vào một biến duy nhất (sum trong trường hợp này).