Multiprocessing: Pthreads API

COSC 208, Introduction to Computer Systems, 2022-05-03

Announcements

- Project 4 due Thursday, May 5

Outline

- pthreads API
- Creating multiple threads

Warm-up: Pthread API

Q1: *What are all possible outputs produced by this program?*

```c
#include <pthread.h>
void *printer(void *arg) {
    char *ch = (char*)arg;
    printf("I am %c\n", *ch);
    return NULL;
}
int main() {
    pthread_t thread1, thread2;
    char *ch1 = malloc(sizeof(char));
    *ch1 = 'X';
    char *ch2 = malloc(sizeof(char));
    *ch2 = 'Y';
    pthread_create(&thread1, NULL, &printer, ch1);
    pthread_create(&thread2, NULL, &printer, ch2);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
}
```

~~~~
```
I am X
I am Y
```
OR
```
I am Y
I am X
```
~~~~

Pthreads API

- Can create and wait for threads to finish, just like processes, but API is different
- Use the pthreads library—`#include <pthread.h>`
- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine)(void*), void * arg)`
  - `thread`—a struct that stores metadata for the thread
  - `attr`—configuration settings for the thread
  - `start_routine`—the function to start executing when the thread starts
    - Pass a pointer to a function
  - `arg`—an argument passed to the aforementioned function
  - *How do we create a new process?*—`fork`
- `int pthread_join(pthread_t thread, void **value_ptr)`
  - `thread`—the same struct passed at thread creation; used to identify the thread we want to wait for
  - `value_ptr`—the location where the function return value should be stored
    - Notice it's a pointer to a void pointer and the `start_routine` function specified in create returns a void pointer
  - *How do we wait for a process to finish?*—`wait` or `waitpid`
- Q2: *What are all possible outputs produced by this program?*

```
#include <pthread.h>
void *printer(void *arg) {
    char *ch = (char*)arg;
    printf("I am %c\n", *ch);
    return NULL;
}
int main() {
    pthread_t thread1, thread2;
    char *ch = malloc(sizeof(char));
    *ch = 'P';
    pthread_create(&thread1, NULL, &printer, ch);
    pthread_join(thread1, NULL);
    *ch = 'Q';
    pthread_create(&thread2, NULL, &printer, ch);
    pthread_join(thread2, NULL);
}
```

```
I am P
I am Q
```

Creating multiple threads

- Create an array of `pthread_t` and an array of arguments
- Call `pthread_create` within a loop
- Call `pthread_join` within a separate loop
- Example

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5
void *simple(void *arg) {
    int *id = (int *)arg;
    printf("I am thread %d\n", *id);
    return NULL;
}
int main() {
    pthread_t threads[NUM_THREADS];
    int ids[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++) {
        ids[i] = i+1;
        pthread_create(&(threads[i]), NULL, &simple, &(ids[i]));
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("All threads finished\n");
}
```

Returning values from threads

- *When does a thread end?* — when the function passed to `pthread_create` finishes (i.e., returns)
- *What happens to a function's parameters and local variables when the function returns?* — they no longer exist (i.e., the stack frame is destroyed)
- *Where should we store a value that should exist even after a function returns?* — on the heap
- Need to store a thread's return value on the heap
- Thread returns a pointer to the value on the heap

- Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
void *length(void *arg) {
    char *str = (char *)arg;
    int *len = malloc(sizeof(int));
    *len = strlen(str);
    return len;
}
int main() {
    pthread_t thread;
    char *phrase = "Hello, threads!";
    pthread_create(&thread, NULL, &length, phrase);
    int *result = NULL;
    pthread_join(thread, (void *)&result);
    printf("Length: %d\n", *result);
    free(result);
}
```

- `pthread_join` returns `0` if successful, or an error number
- To get the pointer returned by the thread, we need to pass a location where the pointer can be stored — i.e., we need to pass a double pointer

Practice writing multi-threaded programs

- Q3: *Write a function called `sum_array` which takes an array of `ARRAY_LEN` integers and returns the sum of the integers. Your function should have the appropriate prototype/implementation to serve as the entry point for a thread. Assume `ARRAY_LEN` is a constant which has been `#define`d.*

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_LEN 10
#define NUM_ARRAYS 5

void *sum_array(void *args) {
    int *nums = (int *)args;
    int *sum = malloc(sizeof(int));
    *sum = 0;
    for (int i = 0; i < ARRAY_LEN; i++) {
        *sum += nums[i];
    }
    return sum;
}
```

- Q4: *Write a function called `sum_matrix` which takes an array of `NUM_ARRAYS` arrays of integers (i.e., an `int **`) and returns the sum of all the integers. The function should create `NUM_ARRAYS` threads, each running the `sum_array` function for a single array of integers. Assume `NUM_ARRAYS` is a constant which has been `#define`d.*

```
int sum_matrix(int *matrix[]) {
    pthread_t threads[NUM_ARRAYS];
    for (int i = 0; i < NUM_ARRAYS; i++) {
        pthread_create(&(threads[i]), NULL, &sum_array, matrix[i]);
    }

    int total = 0;
    for (int i = 0; i < NUM_ARRAYS; i++) {
        int *sum;
        pthread_join(threads[i], (void **)(&sum));
        total += *sum;
        free(sum);
    }

    return total;
}

int main() {
    int *matrix[NUM_ARRAYS];
    for (int i = 0; i < NUM_ARRAYS; i++) {
        matrix[i] = malloc(sizeof(int) * ARRAY_LEN);
        for (int j = 0; j < ARRAY_LEN; j++) {
            matrix[i][j] = i * 100 + j;
        }
    }

    int sum = sum_matrix(matrix);
    printf("%d\n", sum);
}
```

Extra practice

- QA: *What are all possible outputs produced by this program?*

```
#include <stdio.h>
#include <pthread.h>
void *printer2(void *arg) {
    char *ch = (char*)arg;
    printf("Start %c\n", *ch);
    printf("End %c\n", *ch);
    return NULL;
}
int main() {
    pthread_t thread1, thread2;
    char *ch1 = malloc(sizeof(char));
    *ch1 = 'X';
    char *ch2 = malloc(sizeof(char));
    *ch2 = 'Y';
    pthread_create(&thread1, NULL, &printer2, ch1);
    pthread_create(&thread2, NULL, &printer2, ch2);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
}
```

```
Start X
End X
Start Y
End Y
```

OR

```
Start Y
End Y
Start X
End X
```

OR

```
Start X
Start Y
End Y
End X
```

OR

```
Start X
Start Y
End X
End Y
```

OR

```
Start Y
Start X
End X
End Y
```

OR

```
Start Y
Start X
End Y
End X
```