Chapter 12, Bryant and O’Hallaron

CONCURRENT PROGRAMMING

Thread synchronization

• What will the following code output?

```
#define LOOPS 10000000
static int count = 0;

void *counter(void *args) {
    int i;
    for (i = 0; i < LOOPS; i++)
        count++;
    printf("Executed %d times\n", i);
    return NULL;
}

int main() {
    pthread_t tids[2];
    pthread_create(&tids[0], NULL, counter, NULL);
    pthread_create(&tids[1], NULL, counter, NULL);
    pthread_join(tids[0], NULL);
    pthread_join(tids[1], NULL);
    printf("Count: %d\n", count);
    return 0;
}
```
Thread synchronization, cont.

- The first two lines will be:
  Executed 10000000 times
- And the last will be:
  Count: 11398345
- Or maybe:
  Count: 12398354
- Or even...
  Count: 15892348
- But almost definitely not "Count: 20000000" Why not?

Thread synchronization errors

- We might expect the increment to be an atomic operation, but it isn't
  – i++ requires loading i's value into a register, updating that value by adding 1 to it, then storing the result back into i's location in memory
- Consider this schedule of events:
  – Thread A loads i's value
  – Thread B loads i's value
  – A updates register
  – B updates register
  – A stores register value in i
  – B stores register value in i
- What is the value stored in i now?
  – Only one more than it was before!

Thread synchronization errors, cont.

- In this example, we need to ensure that when one thread is in the middle of its load-update-store set of instructions, the other thread isn't in its own set
- For a given thread, these instructions constitute a critical section that should not have other threads' critical sections interleaved within them

Semaphores

- To properly implement a critical section, we can use semaphores
- These are special global variables, with nonnegative integer values, that can only be changed by two operations
  – in most literature, called P and V
  – we'll restrict ourselves to discussion of binary semaphores here, but semaphores can be more than just 0/1 values
- Used to ensure that only one thread is in a critical section at a time
Semaphores, cont.

- $P(s)$, or wait operation
  - If $s$ is 0, waits until $s$ is 1
  - If $s$ is 1, sets $s$ to 0 (decrements $s$)
- $V(s)$, or post operation
  - If $s$ is 0, sets $s$ to 1 (increments $s$), and restarts exactly one of the processes waiting for $s$ to be 1
  - Should not be called except after a $P(s)$ call
- All elements of these operations are indivisible (i.e., these operations are atomic)

Using semaphores

- Three functions, all in `<semaphore.h>`
- Each return 0 on success, -1 on error

  ```c
  int sem_init(sem_t *s, 0, unsigned int value);
  ```
  - initializes the semaphore pointed at by $s$ to value
  - You must initialize semaphores before use
  - The second argument will always be 0 for our purposes

  ```c
  int sem_wait(sem_t *s);
  ```
  - Performs the $P(s)$ operation

  ```c
  int sem_post(sem_t *s);
  ```
  - Performs the $V(s)$ operation

Example using semaphores

```c
#define LOOPS 10000000
static int count = 0;
static sem_t mutex;
void *counter(void *args) {
  int i;
  for (i = 0; i < LOOPS; i++) {
    sem_wait(&mutex);
    count++;
    sem_post(&mutex);
  }
  printf("Executed %d times\n", i);
  return NULL;
}

int main() {
  pthread_t tids[2];
  sem_init(&mutex, 0, 1);
  pthread_create(&tids[0], NULL, counter, NULL);
  pthread_create(&tids[1], NULL, counter, NULL);
  pthread_join(tids[0], NULL);
  pthread_join(tids[1], NULL);
  printf("Count: %d\n", count);
  return 0;
}
```

Thread safety

- Thread-safe functions are those that always produce correct results when called by concurrent threads
- One class of thread-unsafe functions is those functions that don't protect shared variables
- Use of semaphores to protect access to shared variables is one way we can make thread-unsafe functions thread-safe
- The fixed example will now report a final count of 20000000, as we expect, but it runs significantly more slowly
Thread safety, cont.

- Functions that keep state between invocations are also thread-unsafe
  - state can be held by global and/or static vars
- Consider this implementation of a pseudo-random number generator:
  
  ```c
  unsigned int last;
  
  int rand(void) {
    last = last*1103515245 + 12345;
    return (unsigned int) (last / 65536) % 32768;
  }
  
  void srand(unsigned int seed) {
    last = seed;
  }
  ```

- We can only fix this by requiring callers to supply the seed value on each call, eliminating use of global/static data:
  
  ```c
  int rand(unsigned int *last) {
    *last = *last * 1103515245 + 12345;
    return (unsigned int) (*last / 65536) % 32768;
  }
  ```

- Although now all calls to this function require changing, which could lead to bugs, especially in larger programs
- If your code is threaded and relies on a dependable sequence of results, you cannot have global (or static) data accessed in your functions called by threads
  - protecting the shared data with semaphores helps, but still doesn't ensure ordering across threads

Some functions also return pointers to static data:

```c
char *itoa(int n) {
  static char buffer[50];
  sprintf(buffer, "%d", n);
  return buffer;
}
```

- This approach breaks down if multiple threads use the function, as one thread could end up using another thread's results
- Solution: protect calls to these functions with semaphores, and make deep copies before allowing other threads access to the function

Reentrancy

- A reentrant function relies on no shared data at all
  - "thread safe" ≠ "reentrant"
    - "x is reentrant" implies "x is thread safe"
- We made a reentrant form of the rand() function earlier
- Unix systems provide reentrant versions of most thread-unsafe functions, but their interfaces sometimes vary across platforms
Race condition

• A race condition occurs when a program depends on one thread reaching a point $x$ before another thread reaches a point $y$.

• The following program is supposed to print four points on the line $y = 3x + 2$, but instead, prints the same point $(3, 11)$ 4 times, due to a race condition.

```c
/* include statements omitted */
#define NUM_THREADS 4
typedef struct {
  int x, y;
} Point;

void *print_point(void *arg) {
  Point p = *(Point *) arg;
  printf("(%d, %d)\n", p.x, p.y);
  return NULL;
}

int main() {
  pthread_t tids[NUM_THREADS];
  int i;
  Point pt;
  for (i = 0; i < NUM_THREADS; i++) {
    pt.x = i;
    pt.y = 3 * i + 2;
    pthread_create(&tids[i], NULL, print_point, &pt);
  }
  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tids[i], NULL);
  return 0;
}
```

Eliminating the race condition

• The struct being passed to the peer threads is changed by the main thread before the peer threads access it!

• We can use dynamically allocated memory to make sure each thread has a struct dedicated to it.

• The main thread will create the struct, and each thread will destroy its struct once it has obtained all the needed values from the struct.

```c
/* include statements and definitions omitted */
void *print_point(void *arg) {
  Point p = *(Point *) arg;
  free(arg);
  printf("(%d, %d)\n", p.x, p.y);
  return NULL;
}

int main() {
  pthread_t tids[NUM_THREADS];
  int i;
  Point *pt_ptr;
  for (i = 0; i < NUM_THREADS; i++) {
    pt_ptr = malloc(sizeof(*pt_ptr));
    pt_ptr->x = i;
ptr->y = 3 * i + 2;
    pthread_create(&tids[i], NULL, print_point, pt_ptr);
  }
  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tids[i], NULL);
  return 0;
}
```