CS314 - Operating Systems Homework 2 (Fall '16)

1 Administrivia

Homeworks should be done individually and turned in via Moodle.

LaTeX version of this file is here: `/home/cs314/hw2.tex` To generate a pdf with your solutions copy the above file to someplace in your home directory and compile with `pdflatex hw2.tex`

2 Get On With It

1. (20 pts) Consider the following C code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int a = 0;
int main(){
  fork();
  printf("%d\n", ++a);
  fork();
  printf("%d\n", ++a);
  fork();
  printf("%d\n", ++a);
  exit(0);
}
```

   - (10 pts) Including the original process, how many processes end up calling exit()? Show your work or modified code and output showing the correct count.

     The correct answer is 8. I arrived at the solution by realizing that the final `printf("%d\n", ++a);` is being called as often as the `exit(0);` statement, so by compiling the code and running the BASh command `./a.out | grep 3 | wc -l` would return the number of times "3" was printed, which would be the equal to the number of processes.

     Additionally, since `fork()` creates two processes for every one, by raising two to the number of instances of `fork()`, we can get the total number of processes, in this case $2^3 = 8$.

   - (10 pts) What is one possible output of this program? Explain.

     One possible outcome would be

     ```
     1
     2
     3
     1
     ```

```
2
3
1
2
3
1
2
3
1
2
3
1
2
3
1
2
3
1
2
3
```

This is because the processes are shuffled randomly together, and do not execute in any particular order except when in reference to itself.

2. (20 pts) Included here is a trivial threaded program. The intended purpose is to construct a list of numbers from 1 to MAX-1, where each number occurs only once. Run the program on os.cs.s.siue.edu and report what you observe in the output (hint: use wc to count the number of lines in the output), answering the questions: Is the output correct? If it is incorrect, why is it incorrect?

   Now, if you've found that the output is incorrect, fix the program to produce the intended output. Describe what you've done, explaining why it works. Include your source as part of the answer.

   Also, In your answer, include the execution times you've observed for both the original program and for one you've fixed to produce correct output. (see: `man time`) Try to answer at least the following: How is the execution time different? Why is it different? How does the result change your expectations of improving performance by dividing work into threads?

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

#define MAX 100000
FILE* out;

int main()  {
  pthread_t f3_thread, f2_thread, f1_thread;
  void *f1();
  int i = 0;
  out = fopen("numbers", "w+");
  pthread_create(&f1_thread,NULL,f1,&i);
  pthread_create(&f2_thread,NULL,f1,&i);
  pthread_create(&f3_thread,NULL,f1,&i);
  pthread_join(f1_thread, NULL);
  pthread_join(f2_thread, NULL);
```

```
  pthread_join(f3_thread, NULL);
  fclose(out);
  return 0;
}

void *f1(int *x){
  while(*x < MAX){
    fprintf(out,"%d\n", *x);
    (*x)++;
  }
  pthread_exit(0);
}
```

This code is problematic as it has a race condition intrinsic to its execution. It relies on the shared integer i to know which number to print out next. The problem therefore arises when a number is being processed when the thread is interrupted, and the next thread begins to process the same value.

A possible solution to the problem is to re-write the code like this

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define MAX 100000
FILE* out;
sem_t sem;

int main()  {
  pthread_t f3_thread, f2_thread, f1_thread;
  void *f1();
  int i = 0;
  sem_init(&sem, 0, 1);
  out = fopen("numbers", "w+");
  pthread_create(&f1_thread,NULL,f1,&i);
  pthread_create(&f2_thread,NULL,f1,&i);
  pthread_create(&f3_thread,NULL,f1,&i);
  pthread_join(f1_thread, NULL);
  pthread_join(f2_thread, NULL);
  pthread_join(f3_thread, NULL);
  fclose(out);
  return 0;
}

void *f1(int *x){
  while(*x < MAX){
    sem_wait(&sem);
    //We need to check that condition again
    if(*x < MAX)
    {
        fprintf(out,"%d\n", *x);
        (*x)++;
    }
    sem_post(&sem);
  }
  pthread_exit(0);
}
```

This code uses a semaphore to restrict access to the check-print-modify section of code. The semaphore will only allow one thread to enter the critical section at once, and the first thing it does is check that its value for `i` is still a valid one. The tradeoff here is for time. The original program executes in around 30 ms, whereas my corrected version takes around 90 ms. This is because the corrected program has times where a thread might be idle, whereas the uncorrected version has all its threads running throughout execution.

3. (20 pts) Here is another trivial threaded program. Each thread simply prints the value of the passed argument (ignore compiler warnings; we are abusing the argument pointer to pass an integer). Enter this program and run it a bunch of times. You'll notice that the order of thread execution is random. Without changing main(), use semaphores to enforce that threads always get executed in order, so that what is printed are the values 1, 2, and 3, in that order.

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

int main()  {
  pthread_t thread1, thread2, thread3;
  void *f1();

  pthread_create(&thread1,NULL,f1,1);
  pthread_create(&thread2,NULL,f1,2);
  pthread_create(&thread3,NULL,f1,3);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  pthread_join(thread3, NULL);

  return 0;
}

void *f1(int x){

  printf("%d\n", x);

  pthread_exit(0);
}
```

A possible solution to this is shown here.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define NUM_THREADS 3

sem_t turnstile1, turnstile2, mutex;
int count = 0, cycles = 0;

int main()  {
  pthread_t thread1, thread2, thread3;
  void *f1();
  sem_init(&turnstile1, 0, 0);
  sem_init(&turnstile2, 0, 1);
  sem_init(&mutex, 0, 1);

  pthread_create(&thread1,NULL,f1,1);
  pthread_create(&thread2,NULL,f1,2);
  pthread_create(&thread3,NULL,f1,3);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  pthread_join(thread3, NULL);

  return 0;
}

void *f1(int x)
{
  while(cycles < NUM_THREADS)
  {
    //Rendezvous
    sem_wait(&mutex);
```

```
    count++;
    if(count == NUM_THREADS)
    {
      sem_wait(&turnstile2);
      sem_post(&turnstile1);
    }
    sem_post(&mutex);
    sem_wait(&turnstile1);


    //Critical point
    if(cycles + 1 == x)
    {
      printf("%d\n", x);
      cycles++;
    }

    sem_post(&turnstile1);
    //Rendezvous part 2
    sem_wait(&mutex);
    count--;
    if(count == 0)
    {
      sem_wait(&turnstile1);
      sem_post(&turnstile2);
    }
    sem_post(&mutex);
    sem_wait(&turnstile2);
    sem_post(&turnstile2);
  }
  pthread_exit(0);
}
```

This is a general solution for any known number of threads. It is a reusable barrier modifed to allow single access to a critical section, where the number is printed if it is the correct number. When all the numbers have been printed, the loop terminates, and all the threads are released.

4. (20 pts) Rewrite the example from the "Dining Philosophers" slide (Chapter 2 slides; also found in Figure 2-47 in Tanenbaum) to working C code. Write a main() that creates five threads (one for each philosopher). Demonstrate that your solution works by printing something when each philosopher changes state. Let the program exit when each philosopher has eaten at least twice.

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

#define N        5
#define LEFT     ( i + N - 1) % N
#define RIGHT    ( i + 1) % 1
#define THINKING 0
#define HUNGRY   1
#define EATING   2
#define MEALS    2

int state[N];

pthread_t threads[N];

sem_t mutex;
sem_t s[N];

void *philosopher(int*);
```

```
void take_forks(int);
void put_forks(int);
void test(int);
void think(int);
void eat(int);

int main()
{
  sem_init(&mutex, 0, 1);
  for(int i = 0; i < N; i++)
  {
    sem_init(s + i, 0, 1);
  }
  for(int i = 0; i < N; i++)
  {
    pthread_create(threads + i, NULL, philosopher, &i);
  }
  for(int i = 0; i < N; i++)
  {
    pthread_join(threads[i], NULL);
  }
  return 0;
}

void *philosopher(int *i)
{
  for(int j = 0; j < MEALS; j++)
  {
    think(*i);
    take_forks(*i);
    eat(*i);
    put_forks(*i);
  }

  pthread_exit(0);
}

void take_forks(int i)
{
  sem_wait(&mutex);
  state[i] = HUNGRY;
  test(i);
  sem_post(&mutex);
  sem_wait(s + i);
}

void put_forks(int i)
{
  sem_wait(&mutex);
  state[i] = THINKING;
  test(RIGHT);
  test(LEFT);
  sem_post(&mutex);
}

void test(int i)
{
  if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
  {
    state[i] = EATING;
    sem_post(s + i);
  }
}

void think(int i)
{
  printf("Philosopher %d: Hmm...\n", i);
}

void eat(int i)
{
  printf("Philosopher %d: Nom\n", i);
```

```
}
```

5. (20 pts) Write a shell script that produces a file of sequential numbers by reading the last number in the file, adding 1 to it, and then appending it to the file. Run one instance of the script in the background and one in the foreground, each accessing the same file. Show an example from the output of the race condition manifesting itself. What is the critical region? Modify the script to prevent the race. See the notes that follow.

   Here's a basic bash script to get you started:

```
#!/bin/bash
if [ ! -f numbers ]; then echo 0 > numbers; fi
count=0
while [[ $count != 100 ]]; do
#maybe some code goes here
count=‘expr $count + 1‘
#perhaps some more code goes here
done
```

   Output from the race condition looks something like this.

```
0
1
2
2
3
3
4
5
5
6
7
8
9
10
11
7
8
8
9
9
10
10
11
12
13
14
14
15
15
16
16
17
17
18
18
19
19
20
20
21
22
```

22
23
23
24
25
25
26
26
27
27
28
28
29
29
30
30
31
31
32
32
33
33
34
35
35
36
36
37
37
38
39
40
41
42
43
44
45
46
47
48
48
49
50
50
51
51
52
52
53
53
54
54
55
56
56
57
57
58
58
59
60
61
61
62
62
63
64
64
65
65
66
66

67
67
68
68
69
69
70
70
71
71
72
72
73
73
74
74
75
76
77
78
79
79
80
80
81
82
83
83
84
84
85
86
86
87
87
88
88
89
89
90
90
91
91
92
92
93
93
94
94
95
95
96
96
97
97
98
98
99
100
100
101
101
102
102
103
103
104
104
105
106
106
107

107
108
108
109
109
110
110
111
111
112
112
113
114
115
116
117

The critical section is the read in and the append to file was the critical section. I used a busy waiting solution where the script will check constantly against the existance of the lock file. I release the mutex by rm-ing the lock file.

```
#!/bin/bash
if [ ! -f numbers ]; then echo 0 > numbers; fi
count=0
while [[ $count != 100 ]]; do
    while true; do
        if ln numbers numbers.lock 2> /dev/null; then break; fi;
    done
    next=‘tail -n 1 numbers‘
    next=‘expr $next + 1‘
    echo $next >> numbers
    count=‘expr $count + 1‘
    rm numbers.lock
done
```

3 What to Turn In

Turn in one file, preferrably a tgz or zip, containing the answers to the questions above, along with all working code, where applicable.