

2015 Introduction to Operating Systems

Midterm Exam

Total Points: 100

Name:

Student ID:

Attentions:

1. The exam is closed book, closed slides, closed notes
2. No electronic devices are allowed
3. Remember to write your name and student ID on the answer sheet

1. **[10pts]** Which of the following are typical features of a microkernel?

- (a). Loadable kernel modules
- (b). System services running in user-mode
- (c). User-mode drivers
- (d). Preemptive multitasking
- (e). Heavily relying on efficient IPC mechanism

b, c, e

2. **[10pts]** Which of the following are true about system calls?

- (a). User-mode programs rely on system calls to invoke functions in OS kernel
- (b). System calls cannot be invoked directly by user-mode code. In other words, user-mode programs must use library functions (e.g. `fopen`, `printf`) to invoke system calls indirectly
- (c). System calls cannot run in parallel. At any time, there can be at most one system call invocation on a system
- (d). System call invocation does not change the memory address space of the calling process
- (e). Context switch cannot take place during the execution of a system call

a, d

3. **[10pts]** Assume there are 1,000 processes of ProgramX (Figure 1) running on a system equipped with an Intel Core i5 processor. If we randomly pick one of the processes, which of the following CPU states (a~e as shown in Figure 2) is the most likely one that the process will be at?

```
int main()
{
    while(1);
    return 0;
}
```

Figure 1. ProgramX

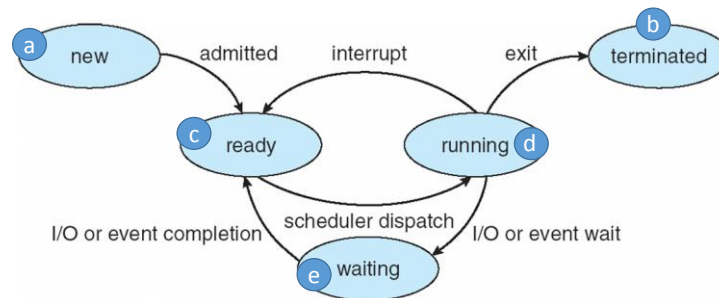


Figure 2. CPU scheduling state of a process

c

4. Refer to the example code on using Pipe IPC in Figure 3.
 - (a) **[3pts]** How many processes will be executing the code at Line 18? Note that the creation of child process may not succeed. You have to consider both cases.

Child process creation succeeded => 1

Child process creation failed => 0

- (b) **[3pts]** How many processes will be executing the code at Line 24? Note that the creation of child process may not succeed. You have to consider both cases.

Child process creation succeeded => 1

Child process creation failed => 0

- (c) **[5pts]** Does the program have any race conditions? If yes, please describe what are the race conditions?

No

- (d) **[5pts]** Assume the child process creation is successful, what will be shown on the screen after the execution of the program? If the program has race conditions that will affect the screen output results, you need to list all possible outputs.

NCTU NTHU

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(int argc, char **argv) {
7      pid_t childpid;
8      int fd[2];
9      char buffer[32];
10     int status;
11
12     pipe(fd);
13     childpid = fork();
14
15     if (childpid < 0) {
16         printf("fork failed\n");
17     } else if (childpid == 0) {
18         dup2(fd[1], 1);
19         close(fd[0]);
20         close(fd[1]);
21         printf("NCTU");
22         sprintf(buffer, "NTU");
23     } else {
24         dup2(fd[0], 0);
25         close(fd[0]);
26         close(fd[1]);
27         scanf("%s", buffer);
28         printf("%s NTHU", buffer);
29         waitpid(childpid, &status, 0);
30     }
31
32     return 0;
33 }

```

Figure 3. Pipes IPC

5. [9pts] Answer the following questions regarding process scheduling.

- (a). Under what conditions does round robin scheduling behave identically to FIFO?

When the burst durations are equal to or less than the scheduler time quantum.

- (b). Under what conditions does round robin perform poorly compared to FIFO?

If the scheduler time quantum is too small, RR (round-robin) will incur very high context switch overhead. That will increase average waiting time, turnaround time, and even response time beyond FIFO.

Round robin does not guarantee order of completion (early job may complete later)

- (c). What is the motivation behind shortest-job-first scheduling?

Let jobs with short CPU bursts to go first. This can lower the average waiting time. Likely, the short burst jobs will be doing I/O when long burst jobs are running. This improves the hardware resource utilization.

```
1  class CriticalSection
2  {
3      int locked;
4
5  public:
6
7      CriticalSection()
8      {
9          locked = false;
10     }
11
12     void lock()
13     {
14         while(locked == true);
15         locked = true;
16     }
17
18     void unlock()
19     {
20         locked = false;
21     }
22 };
```

Figure 4. Critical section lock implementation

6. [8pts] Figure 4 presents the implementation of a critical section lock in C++. Here we assume both memory reads and memory writes are atomic. Prove or disprove the lock satisfies mutual exclusion.

The lock does not satisfy mutual exclusion. A counter example is that when two threads trying to acquire the lock at the same time, both will pass the check at Line 14 and slip over the while loop. Both threads will acquire the lock. Mutual exclusion will be violated.

7. [8pts] Give an example of a synchronization lock that satisfies mutual-exclusion, freedom from deadlock, but not freedom from starvation.

The straightforward spinlock implementation (below) satisfies mutual exclusion, and freedom from deadlock. Mutual exclusion is guaranteed as a result of the atomic swap and memory read/writes. Deadlock freedom is guaranteed as the only blocking statement is the while loop. Per definition of deadlock, if no thread is running in the critical section, the thread who wishes to enter the critical section cannot be postponed indefinitely. However, that can only happen if the thread is blocked in the while loop. This is not possible, as the swap operation does not block. The while loop will break as soon as the atomic swap returns.

```
do {  
    key = TRUE;  
    while ( key == TRUE)  
        Swap (&lock, &key );  
  
    // critical section  
  
    lock = FALSE;  
  
    // remainder section  
  
} while (TRUE);
```

Supplement:

1. Other answers

- False
 - Peterson part 1, 2 (flag, victim): not free for deadlock. see slide 6.24, 6.25.
 - Peterson lock: free for starvation. see slide 6.29.
 - Semaphore with wait queue: free for starvation. see slide 6.47 ~ 6.49.
 - Deadlock prevention/detection algorithm in Ch7 isn't implemented in a single lock like spinlock. see supplement 2 for more detail.
- True: spinlock, mutex, semaphore ...

2. Deadlock meaning in Ch6 and Ch7 is a little different.

- Ch6 deadlock meaning is at slide 6.19.
- Ch7 deadlock meaning is at slide 7.4/7.7.

Problem 7 tell about single lock design in Ch6.

There is a good example in slide 7.17 to combine 2 concepts.

Algorithm for checking the resource order is implemented in `double_rq_lock()`, rather than implemented in `raw_spin_lock()`.

In Contrast, Ch6 discusses the characteristics of `raw_spin_lock()`. (3 requirements of critical section)

```
class CriticalSectionLockOne
{
    bool flag[2];

public:
    void lock()
    {
        int i = ThreadID.get();
        int j = 1-i;
        flag[i] = true;
        while(flag[j]) ; // wait
    }

    void unlock()
    {
        int i = ThreadID.get();
        flag[i] = false;
    }
};
```

Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.




Figure 5. CriticalSectionLockOne

8. **[8pts]** Prove that CriticalSectionLockOne in Figure 5 is free of deadlock. Otherwise, please give a counter-example.

Counter-example: Assume two threads attempt to acquire the lock. If the two threads set `flag[{id of the other thread}] = true` at the same time and enter the while-loop at the same time, both threads will get stuck in the while loop.

9. **[21pts]** One of the students came up with the parallel counter code in Figure 7 for homework #2. For experiment, the student uses a machine equipped with an Intel Core i7 (8 cores) processor and 16 GB RAM. The code is compiled with GNU C++ compiler without optimization. The only major workload running on the computer is the parallel counter. The experiment result is presented in Figure 6. Please answer the following questions:

- (a). Why does `Parallel` take a longer execution time (duration) than `Sequential`?

Access to the shared counter is serialized by lock. The critical section covers almost all of the workload (i.e. read, increment, and write of the shared counter). There is little workload to be parallelized by the use of threads. The call to lock/unlock incur additional overhead. The overhead can be even more significant if they result in system calls.

- (b). Why does `Race` take a shorter execution time than `Parallel`?

Overhead due to locking is avoided. Also, read access to the shared counter can be parallelized. Part of the counter incrementing (the part of the operation that takes place in each processor core) can be parallelized.

- (c). If we run `Parallel` again, will the output counter value remain 3000000000?

Yes

- (d). If we run `Race` again, will the output counter value remain 1129942545?

No

- (e). Why does `LockFree` take a shorter execution time than `Parallel`?

There is no locking overhead. Memory accesses are made to different locations, which can be parallelized.

- (f). Why does `LockFree` take a shorter execution time than `Race`?

Memory accesses in `Race` are made to the same memory location. As dictated by x86 microarchitecture, reads-after-writes and writes-after-writes on the counter will be serialized. Also, the caches in the processor cores will be constantly invalidated and reloaded from the main memory as each of the threads performs reads and writes to the counter. `LockFree` does not have the same issues.

- (g). If we change `SPACING` to 0, will that affect the experiment result in any way? If yes, please describe what will be affected.

The execution time of `LockFree` will become longer due to false sharing. Specifically, the three counter objects are likely to be placed on the same cache line. Write accesses to different counters can invalidate the cache of each other and degrade performance.

```

g++ a.cpp -lpthread
./a.out

SPACING = 128 bytes
Sequential: value=      3000000000 duration=7 secs
Parallel:   value=      3000000000 duration=336 secs
Race:       value=      1129942545 duration=17 secs
LockFree:   value=      3000000000 duration=2 secs

```

Figure 6. Parallel counter experiment result (SPACING = 128 bytes)

```

1  #include <pthread.h>
2  #include <time.h>
3  #include <iostream>
4  #include <stdint.h>
5
6  using namespace std;
7
8  #define SPACING 128
9
10 #define NUM_CORES 3
11 #define ITERATIONS 100000000L
12 #define COLUMN_WIDTH 12
13
14 class Counter
15 {
16 public:
17     uint64_t value;
18     bool bUseLock;
19     char spacings[SPACING];
20     void Increment() { value++; }
21 };
22
23 pthread_mutex_t mutex;
24 time_t start_time;
25
26 void* ThreadRunner(void* pCounter)
27 {
28     Counter* pC = (Counter*)pCounter;
29
30     unsigned int k;
31
32     if ( pC->bUseLock) {
33         for (k = 0; k < ITERATIONS; k++) {
34             pthread_mutex_lock(&mutex);
35             pC->Increment();
36             pthread_mutex_unlock(&mutex);
37         }
38     }
39     else {
40         for (k = 0; k < ITERATIONS; k++)
41             pC->Increment();
42     }
43 }
44
45 void LockFree()
46 {
47     time_t start_time;

```

```

48     int duration;
49     Counter x[NUM_CORES];
50     pthread_t tid[NUM_CORES];
51     int i;
52     uint64_t sum = 0;
53
54     for(i = 0; i < NUM_CORES; i++) {
55         x[i].value = 0;
56         x[i].bUseLock = false;
57     }
58
59     start_time = time(0);
60
61     for ( i = 0; i < NUM_CORES; i++)
62         pthread_create(&tid[i], NULL, ThreadRunner, &x[i]);
63
64     for ( i = 0; i < NUM_CORES; i++) {
65         pthread_join(tid[i], NULL);
66         sum += x[i].value;
67     }
68
69     cout.width(COLUMN_WIDTH);
70     cout<<std::left<<"LockFree: " ;
71     cout.width(COLUMN_WIDTH);
72     cout<<std::left<<"value="<<sum;
73     cout<<"\tduration="<< time(0) - start_time <<" secs"<<endl;
74
75 }
76
77 void Parallel()
78 {
79     time_t start_time;
80     int duration;
81     Counter x;
82     pthread_t tid[NUM_CORES];
83     int i;
84
85     x.value = 0;
86     x.bUseLock = true;
87
88     start_time = time(0);
89
90     for ( i = 0; i < NUM_CORES; i++)
91         pthread_create(&tid[i], NULL, ThreadRunner, &x);
92
93     for ( i = 0; i < NUM_CORES; i++)
94         pthread_join(tid[i], NULL);
95
96     cout.width(COLUMN_WIDTH);
97     cout<<std::left<<"Parallel: " ;
98     cout.width(COLUMN_WIDTH);
99     cout<<std::left<<"value="<<x.value;
100    cout<<"\tduration="<< time(0) - start_time <<" secs"<<endl;
101
102 }
103
104
105 void Race()
106 {
107     time_t start_time;
108     int duration;
109     Counter x;
110     pthread_t tid[NUM_CORES];

```

```

111     int i;
112
113     x.value = 0;
114     x.bUseLock = false;
115
116     start_time = time(0);
117
118     for ( i = 0; i < NUM_CORES; i++)
119         pthread_create(&tid[i], NULL, ThreadRunner, &x);
120
121     for ( i = 0; i < NUM_CORES; i++)
122         pthread_join(tid[i], NULL);
123
124     cout.width(COLUMN_WIDTH);
125     cout<<std::left<<"Race: " ;
126     cout.width(COLUMN_WIDTH);
127     cout<<std::left<<"value="<<x.value;
128     cout<<"\tduration="<< time(0) - start_time <<" secs"<<endl;
129
130 }
131
132
133 void Sequential()
134 {
135     int i;
136     Counter x;
137
138     x.value = 0;
139     x.bUseLock = false;
140
141     start_time = time(0);
142
143     for ( i = 0; i < NUM_CORES; i++)
144         ThreadRunner(&x);
145
146     cout.width(COLUMN_WIDTH);
147     cout<<std::left<<"Sequential: " ;
148     cout.width(COLUMN_WIDTH);
149     cout<<std::left<<"value="<<x.value;
150     cout<<"\tduration="<< time(0)-start_time <<" secs"<<endl;
151
152 }
153
154 int main()
155 {
156
157     cout<<"SPACING = " << SPACING <<" bytes"<<endl;
158     pthread_mutex_init(&mutex,0);
159     Sequential();
160     Parallel();
161     Race();
162     LockFree();
163     pthread_mutex_destroy(&mutex);
164
165     return 0;
166 }
167

```

Figure 7. Parallel Counter