Ahmed Alelg
201507470
ICS-431:Operating Systems
Due Date: 12^{th} April, 2020

Homework - 4
Term: 192

Theoretical part

Chapter 6: Synchronization

Problem 6.5
Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

Solution: Since one processor can not block others, Processes can still access or modify the data simultaneously through other processors. Hence using interrupts can not achieve mutual exclusion.

Problem 6.7
Describe two kernel data structures in which race conditions are possible. Be sure to include a description of how a race condition can occur.

Solution: A race condition occurs when two or more processes are trying to modify the same data simultaneously. Consider two processes trying to create a file at the same time, the file control block needs to create a record for each one, there will be a race on which one will get the next free sector. Or in an open file table for multi-user, when two more users are manipulating it.

Problem 6.9
Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:
`typedef struct {int available;} lock;`
`(available == 0)` indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this `struct`, illustrate how the following functions can be implemented using the `test_and_set()` and `compare_and_swap()` instructions: `void acquire(lock *mutex)`, `void release(lock *mutex))`

Solution:

```
1
2  int test_and_set(int *available) {
3    int temp = *available;
4    *available = 1;
5    return temp;
6  }
7
8  int compare_and_swap(int *value, int expected, int new){
9    int temp = *value;
10   if (temp == expected)
11     *value = new;
12   return temp;
13 }
14
15 void aquire(lock *mutex){
16   while( test_and_set(&mutex->available) ); // blocks on available=1
17 }
18
19 /* another possible implementaiton */
20
21 void aquire(lock *mutex) {
```

```
22    while( compare_and_swap(&mutex->available, 0, 1) );
23  }
24
25  void release(lock *mutex){
26    mutex->available = 0; // release the lock
27  }
28
```

Problem 6.12
Assume that a context switch takes T time. Suggest an upper bound(in terms of T) for holding a spinlock. If the spinlock is held for any longer, a mutex lock (where waiting threads are put to sleep) is a better alternative.

Solution: if spinlock takes more than $2T$ it is better to put it to sleep; since it will be more costly than the overhead time of putting it to sleep.

Chapter 7: Deadlocks

Problem 7.1
Consider the traffic deadlock depicted in Figure 7.10.

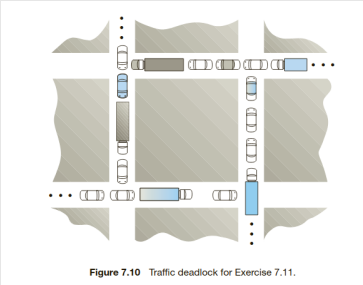

Figure 7.10 Traffic deadlock for Exercise 7.11.

a. Show that the four necessary conditions for deadlock hold in this example.

b. State a simple rule for avoiding deadlocks in this system

Solution:

a. (a) Mutual exclusion: No two vehicles can exists on the same place.
   (b) Hold and Wait: each vehicle is holding a place and waiting for other places to be cleared out.
   (c) Non-preemption condition: the cars can not be taken out of the road.
   (d) Circular wait: each car is blocked and is waiting for the next car in the lane in a circular manner.

   Given that all conditions hold, we conclude a deadlock has occurred.

b. at least one of the following solutions is required to prevent a deadlock:

   (a) Allow two side-lanes to cancel mutual exclusion.
   (b) Only let a vehicle pass an intersection if the traffic is moving to cancel hold and wait.
   (c) preempt the cars (possibly using a helicopter) at all intersections.

Problem 7.5
Compare the circular-wait scheme with the various deadlock-avoidance schemes (like the bankers algorithm) with respect to the following issues:

a. Runtime overheads

b. System throughput

Solution:

a. **the deadlock-avoidance schemes are more intensive** since they computes the process usage of resources during its whole life cycle. Compared to **Circulr-wait schemes which are less intensive** since they only enumerates over the processes checking for ordering inconsistencies.

b. More resources can be allocated to processes when using deadlock-avoidance schemes hence the throughput will be more compared to circular waiting schemes, which tries to prevent circular formations.

Problem 7.7
Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources.Show that the system is deadlock free

Solution: If a process was allocated with 2 resources it must release them before moving on to the next process, hence we would always have 4 available resources and no deadlock can occur.

Problem 7.12
Consider the following snapshot of a system:

	Allocation	*Max*
	A B C D	*A B C D*
P_0	3 0 1 4	5 1 1 7
P_1	2 2 1 0	3 2 1 1
P_2	3 1 2 1	3 3 2 1
P_3	0 5 1 0	4 6 1 2
P_4	4 2 1 2	6 3 2 5

Using the bankers algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the processes may complete. Otherwise, illustrate why the state is unsafe

a. Available=(0, 3, 0, 1)

b. Available=(1, 0, 0, 2)

Solution:

	Needed
P_0	2103
P_1	1001
P_2	0200
P_3	4102
P_4	2113

a. unsafe; 0.3.0.1 $\xrightarrow{P_2}$ 3.4.2.2 $\xrightarrow{P_1}$ 5.6.3.2 $\xrightarrow{P_3}$ 5.11.4.2 □
No possible assignment, all the remaining processes require at least 3 instances of D.

b. safe with the sequence $\{P_1, P_2, P_3, P_4, P_0\}$; since 1.0.0.2 $\xrightarrow{P_1}$ 3.2.1.2 $\xrightarrow{P_2}$ 6.3.3.3 $\xrightarrow{P_3}$ 6.8.4.3 $\xrightarrow{P_4}$ 10.10.5.5 $\xrightarrow{P_0}$ 13.10.6.9 ✓

Chapter 8: Memory managment

Problem 8.2
Consider the following process for generating binaries. A compiler issued to generate the object code for individual modules, and a linkage editor is used to combine multiple object modules into a single program binary. How does the linkage editor change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linkage editor to facilitate

the memory-binding tasks of the linkage editor?

Solution: The linker replaces the static addresses into relative ones. The information about the addresses locations and their names are needed in order for the process to take place.

Problem 8.4
Most systems allow a program to allocate more memory to its address space during execution. Allocation of data in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?

a. Contiguous memory allocation

b. Pure segmentation

c. Pure paging

Solution:

a. We need to know all the total size to be allocated beforehand, or limit the processes by a certain size; otherwise the whole process must be relocated.

b. Allocate it within the heap segment, if the space is not enough search for a block in the physical memory that is big enough to accommodate it.

c. Find a free slot in the page table and associate it with a frame, if no free frame is found then lookup for a victim process for swapping.

Problem 8.10
Explain why address space identifiers (ASIDs) are used.

Solution: It is used to uniquely identify a process's address space, this will secure it from other unintended access by other processes.

Problem 8.14
What is the maximum amount of physical memory?

Solution: It is specified by the hardware RAM. And also by the architecture of the OS; a 32bit OS for example can address up to 4GB of memory.

Practical Part

Problem 1-7.17
Implement your solution to Exercise 7.15 using `POSIX` synchronization. In particular, represent northbound and southbound farmers as separate threads. Once a farmer is on the bridge, the associated thread will sleep for a random period of time, representing traveling across the bridge.Design your program so that you can create several threads representing the northbound and southbound farmers

Solution:

```
Farmer from the north crossed the bridge: 1
Farmer from the south crossed the bridge: 2
Farmer from the north crossed the bridge: 3
Farmer from the south crossed the bridge: 4
Farmer from the north crossed the bridge: 5
Farmer from the south crossed the bridge: 6
Farmer from the north crossed the bridge: 7
Farmer from the south crossed the bridge: 8
Farmer from the north crossed the bridge: 9
Farmer from the south crossed the bridge: 10
Total farmers: 10
Total farmers passed: 10
```

Please find the source code in the attached files.

```c

#include<stdio.h>
#include <pthread.h>
#include<stdint.h>
#include <unistd.h>
#include <stdlib.h>

static int bridge = 0;
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

static void* farmerFunc(void *arg) {
int s;
int turn = *((int *) arg);

s = pthread_mutex_lock(&lock);
sleep( rand() % 3);
bridge ++;
s = pthread_mutex_unlock(&lock);
if(turn % 2)
printf("Farmer from the south crossed the bridge: %d\n", bridge);
else
printf("Farmer from the north crossed the bridge: %d\n", bridge);



return NULL;
}

int main() {
int total_farmers = 10;
pthread_t farmers[10];

for (int i=0; i<10; i+=1) {
int *p1 = malloc(sizeof(*p1));
*p1 = i;
pthread_create(&farmers[i], NULL, farmerFunc, p1 );

}

for (int i = 0; i<10; i+=1)
pthread_join(farmers[i], NULL);

printf("Total farmers: %d\nTotal farmers passed: %d\n", total_farmers, bridge);
return 0;
}


```

Problem 1-8.25
Assume that a system has a 32-bit virtual address with a 4-KB page size. Write a C program that is passed a virtual address (in decimal) on the command line and have it output the page number and offset for the given address. As an example, your program would run as follows: `./a.out 19986` Your program would output:
`The address 19986 contains:`
`page number = 4`
`offset = 3602`
Writing this program will require using the appropriate data type to store 32 bits. We encourage you to use unsigned data types as well.

Solution:

```
js@js-delta:~/Desktop/workbench/C/OS/hmwk4$ make
gcc -o 8_25 8_25.c && ./8_25 19986 && rm 8_25

The address 19986 contains:
page numer = 4
offset = 3602
```

Please find the source code in the attached files.

```c

#include<stdio.h>
#include <stdlib.h>
#define VIRTUAL_ADDR_SIZE 32
#define PAGE_SIZE 12 //4096B=4kB

int main(int argc, char *argv[]) {
  unsigned int virt_addr = atoi(argv[1]);
  unsigned int pageNumber  = virt_addr >> PAGE_SIZE;
  unsigned int distance = VIRTUAL_ADDR_SIZE - PAGE_SIZE;
  unsigned int offset = (virt_addr << (distance)) >> distance ;

  printf("\nThe address %u contains:\npage numer = %u\noffset = %u\n", virt_addr,
 pageNumber, offset);
}

```

Problem 2

The goal of this programming assignment is to give you a basic introduction to the students IPC mechanisms and synchronization using semaphores. You need to write and run C programs (as processes on your Linux machine), and monitor their behavior. Consider the following problem: A program is to be written to print all numbers between 1 and 1000 (inclusive) that are not (evenly) divisible by either 2 or 3. This problem is to be solved using three processes (P_0, P_1, P_2) and two one-integer buffers (B_0 and B_1)as follows:

1. P_0 is to generate the integers from 1 to 1000, and place them in B_0 one at a time. After placing 1000 in the buffer, P_0 places the sentinel 0 in the buffer, and terminates.

2. P_1 is to read successive integers from B_0. If a value is not divisible by 2, the value is placed in B_1. If the value is positive and divisible by 2, it is ignored. If the value is 0, 0 is placed in B_1, and P_1 terminates.

3. P_2 is to read successive integers from B_1. If a value is not divisible by 3, it is printed. If the value is positive and divisible by 3, it is ignored. If the value is 0, P_2 terminates.

Write a program to implement P_0, P_1, and P_2 as separate processes and B_0 and B_1 as separate pieces of shared memory {each the size of just one integer. Use semaphores to coordinate processing. Access to B_0 should be independent of access to B_1; for example, P_0 could be writing into B_0 while either P_1 was writing into B_1 or P_2 was reading.

Solution: *Please find the source code in the attached files.*

```
(base) js@js-delta:~/Desktop/workbenc
gcc -pthread -o 2 2.c && ./2 && rm 2
1
5
7
11
13
17
```

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <stdlib.h>
#define MAX_COUNT_VAL 1000

typedef struct {
  unsigned int data;
  sem_t mutex;
  sem_t read; // producer checks if it has been read before writing
  sem_t wrote; // consumer checks if it has been written before reading
} Buffer;

Buffer buf[2];


Buffer* init_buffer() {
  Buffer *buf = (Buffer *) malloc(sizeof(Buffer));
  sem_init(&buf->mutex, 0, 1);
  sem_init(&buf->read,  0, 1);
  sem_init(&buf->wrote, 0, 0);
  return buf;
}

void buf_write(Buffer* buf, unsigned int val, int pr){
  sem_wait(&buf->read);
  sem_wait(&buf->mutex);
    buf->data = val;
  sem_post(&buf->mutex);
  sem_post(&buf->wrote);


}

unsigned int buf_read(Buffer* buf) {
  unsigned int val;
  sem_wait(&buf->wrote);
  sem_wait(&buf->mutex);
    val = buf->data;
  sem_post(&buf->mutex);
  sem_post(&buf->read);
  return val;
}

/* p1 */
void* begin_count(void * args){
  for(unsigned int i = 1; i<MAX_COUNT_VAL+1; i++){
    buf_write(&buf[0], i, 1);
  }
  buf_write(&buf[0], 0, 1);
  return NULL;
}

/* p2 */
void* div_two(void * args){
  for(;;) {

    unsigned int tmp = buf_read(&buf[0]);
    if( (tmp % 2) != 0 ){
      buf_write(&buf[1], tmp, 2);
    }

    if(tmp == 0){
      buf_write(&buf[1], tmp, 2);
```

```
66        return NULL;
67        break;
68      }
69    }
70  }
71
72
73  /* p3 */
74  void* div_three(void * args){
75    for(;;){
76      unsigned int tmp = buf_read(&buf[1]);
77      if(tmp == 0){
78        return NULL;
79        break;
80      }
81
82      if( (tmp % 3) != 0 ) {
83      printf("%u\n", tmp);
84      }
85
86    }
87  }
88
89  int main () {
90    buf[0] = *init_buffer();
91    buf[1] = *init_buffer();
92    pthread_t p1, p2, p3;
93
94    pthread_create(&p1, NULL, begin_count, NULL);
95    pthread_create(&p2, NULL, div_two, NULL);
96    pthread_create(&p3, NULL, div_three, NULL);
97
98    pthread_join(p1, NULL);
99    pthread_join(p2, NULL);
100   pthread_join(p3, NULL);
101
102   return 0;
103
104 }
```

Problem 3

The aim of this problem is to help you learn some of the CPU scheduling algorithm discussed in the class by implementing and simulating their performance. You need to program the following scheduling algorithms (Preferably in C): FCFS, SJF (with preemption) and Round Robin Scheduling. Each program will read from a file containing a list of processes with per-defined data for the process. The program will simulate the execution of the processes. It will print out the time taken by each process to complete (turnaround time) and the wait time and compute the average turnaround time for all processes to execute as well as the standard deviation of that average.

Inputs: A filename from the keyboard read the file for the predefined data, and a possible time slice size (depending on which algorithm is used). The file containing the information on the processes will have each process on a separate line. The processes will be in the file in the order in which they arrive at the OS. Each line will have a process name that will be a string. Following that will be the arrival time of the process. The arrival time will be in reference to the previous process. Following this will be the total execution time. Next will be the elapsed time between I/O interrupts (system calls), next will be the time spent waiting and processing the I/O and finally the priority of the process as an integer (smaller values will have higher priority). It will look like this

P_1	0	20.0	1.5	5.0	2
P_2	2	15.0	2.0	6.0	1
P_3	6	27.0	1.8	3.5	4
P_4	4	36.0	2.1	2.6	3
x	x	x	x	x	x

Where, the xx indicates the end of the data.

Outputs: A prompt for which file is to be read. A prompt for the amount of time for a time-slice. A list of each process and the time it took for it to complete. Then an average time for processes to complete. Finally the standard deviation for the average time for processes to complete.

Example:

The name of the file to be read: filename

process name turnaround time total wait time

The standard deviation for the average process completion time was? For RR Scheduling: The time slice if required for your algorithm will be 3. You may assume that a swap (context switching time) is small enough that it can be ignored. Vary the time slice/quantum of RR scheduling from 1 to 10sec(in steps of 1 sec) and plot a graph showing how the average turnaround time for processes vary with time slice/quantum. Also, plot a graph showing how the average waiting time for processes varies with time slice/quantum.

Solution: Please find the source code in the attached files.

```
FCFS

#       Waiting t(CPU)  t(all)
P1      0.0     20.0    26.5
P2      18.0    33.0    41.0
P3      65.0    92.0    97.3
P4      31.0    67.0    71.69999999999999

Avg. cpu turnaround: 53.0
Avg. total turnaround: 59.125
Stdv. cpu turnaround: 32.69046751985457
Stdv. total turnaround: 31.66684649071749
Avg. wait: 28.5
Stdv. wait: 27.452990122510638

Scheduling Order:
P1 P2 P4 P3
```

(a) FCFS scheduling

```
Preemptive SJF

#       Waiting t(CPU)  t(all)
P1      15.0    35.0    41.5
P2      0       15.0    23.0
P3      29.0    56.0    61.3
P4      58.0    94.0    98.69999999999999

Avg. cpu turnaround: 50.0
Avg. total turnaround: 56.125
Stdv. cpu turnaround: 33.77375707064091
Stdv. total turnaround: 32.406622265621365
Avg. wait: 25.5
Stdv. wait: 24.691429012243635

Scheduling Order:
P1 P2 P1 P3 P4
```

(b) Preemptive SJF scheduling

```
RR

#       Waiting t(CPU)  t(all)
P1      45      65.0    71.5
P2      37.0    52.0    60.0
P3      56.0    83.0    88.3
P4      58.0    94.0    98.69999999999999

Avg. cpu turnaround: 73.5
Avg. total turnaround: 79.625
Stdv. cpu turnaround: 18.663690238892556
Stdv. total turnaround: 17.226602489560534
Avg. wait: 49.0
Stdv. wait: 9.83192080250175

Scheduling Order:
P1 P2 P1 P4 P3 P2 P1 P4 P3 P2 P1 P4 P3 P2 P1 P4 P3 P2 P1 P4 P3 P1 P4 P3 P4 P3 P4 P3 P4 P3 P4
```

(c) RR scheduling

Figure 1: The results running the implemented algorithms

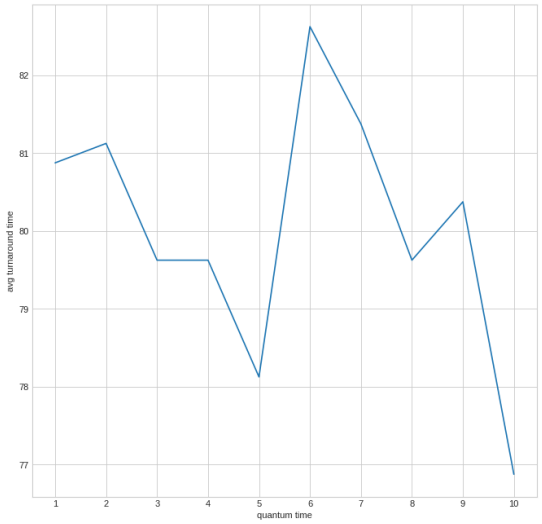

Figure 2: Average RR total turnaround time as quantum time varies between 1 to 10

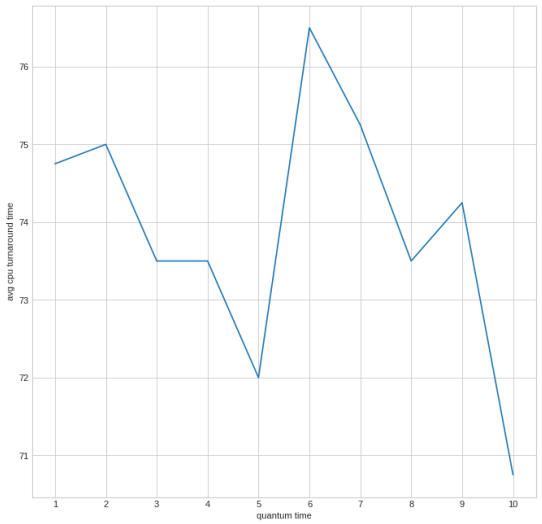

Figure 3: Average RR CPU turnaround time

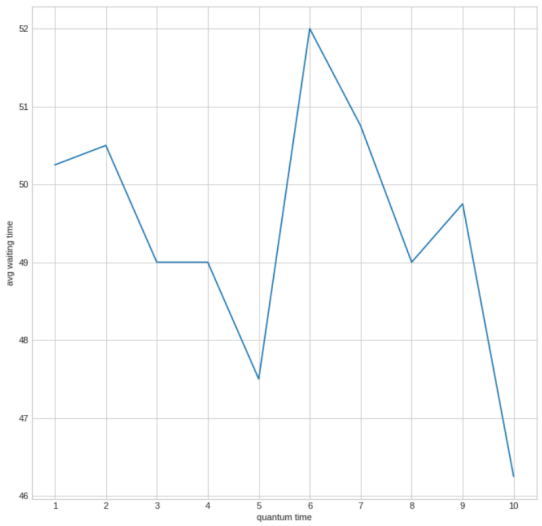

Figure 4: Average RR waiting time has the same behavior as turnaround time