Problem 1. Simple questions

1. The runtime system could be blocking/unblocking a thread, or is scheduling queues. Therefore, some problem might occur when the clock interrupt handler check when to do thread switching, since they might be in an inconsistent state. One way to solve it is to use flag when the runtime system is entered. The clock handler would see this and set its own flag, then return. When the runtime system finished, it would check the clock flag, see if a clock interrupt occurred, then run clock handler.

2. Yes, but with limited efficiency. Before any thread made a system call, an alarm should be set, and then execute. If the thread is blocked, the alarm should return the control back to the thread package. Since most of the system call won't have blocking, and the alarm should be cleared out, every single system call should be viewed as 3 system calls, which is not very ideal. Since we need to maintain the alarms to avoid them expired, extra work need to be done.

Problem 2. Monitors

The drawback of this solution is that it slows down the overall performance. When a process is blocked by waituntil, the system has to wait for the boolean expression finish computing, which wastes a lot of time and computing resources.

Problem 3. Race condition in Bash

1. The following code shows a typical example of a program with racing possibility. The detailed output is in README.md.

```bash
#!/bin/bash
file=text.txt

if ! [ -f $file ]
then
        echo 0 > $file
fi

for i in `seq 1 100`
do
        cnt=$(tail -1 $file)
        echo $((++cnt)) >> $file
done
```

2. The following piece of code shows the version using file lock, which solves the issue.

```sh
#!/bin/sh
file=text.txt

if ! [ -f $file ]
```

```bash
then
        echo 0 > $file
fi

for i in `seq 1 100`
do
    exec 3<$file
    flock -x 3
        cnt=$(tail -1 $file)
        echo $((++cnt)) >> $file
    flock -u 3
    exec 3>&-
done
```

Problem 4. Programming with semaphores

1. The semaphore header file is located at `/usr/include/linux/semaphore.h`
2. There are four main functions defined in `semaphore.h`

```c
int sem_init(sem_t *sem, int pshared, unsigned int value);
// sem: the semaphore to be initialized
// pshared: the signal to be decided to share among process or
↪  thread, where 0 denote shared by thread
// value: the initial value of signal
// Return 0 when normal, return -1 when fails

int sem_destroy(sem_t *sem);
// sem: semaphore to be destoryed
// Return 0 when normal, return -1 when fails

int sem_wait(sem_t *sem);
// Waiting signal, if the signal > 0, then --signal and return. If
↪  the signal is 0, then block the thread.
// Return 0 when normal, return -1 when fails

int sem_post(sem_t *sem);
// Release signal, ++signal
// Return 0 when normal, return -1 when fails
```

3. The following piece of code use semaphore to protect the variable from being accessed by two process at the same time.

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

```c
#include <semaphore.h>
#define N 1000000
int count = 0;

void *thread_count(void *a) {
    int i, tmp;
    sem_t *Sem = a;
    for (i = 0; i < N; i++) {
        sem_wait(Sem);
        tmp = count;
        tmp = tmp + 1;
        count = tmp;
        sem_post(Sem);
    }
}

int main(int argc, char *argv[]) {
    int i;
    sem_t Sem;
    pthread_t *t = malloc(2 * sizeof(pthread_t));
    if (sem_init(&Sem, 0, 1)) {
        printf("ERROR init semaphore\n");
        exit(0);
    }
    for (i = 0; i < 2; i++) {
        if (pthread_create(t + i, NULL, thread_count, &Sem)) {
            fprintf(stderr, "ERROR creating thread %d\n", i);
            exit(1);
        }
    }
    for (i = 0; i < 2; i++) {
        if (pthread_join(*(t + i), NULL)) {
            fprintf(stderr, "ERROR joining thread\n");
            exit(1);
        }
    }
    if (count < 2 * N)
        printf("Count is %d, but should be %d\n", count, 2 * N);
    else
        printf("Count is[%d]\n", count);
    free(t);
    pthread_exit(NULL);
}
```