VE482 Homework 4

Weili Shi | 519370910011 | Oct 31, 2021

Ex.1

- **Consider a system in which threads are implemented entirely in user space, with the run-time system getting a clock interrupt once a second. Suppose that a clock interrupt occurs while some thread is executing in the run-time system. What problem might occur? Can you suggest a way to solve it?**

  A clock interrupt may result in unexpected behavior when a certain thread is about to enter or leave the critical region, or about to get blocked or get unblocked.

  Solution: the runtime system temporarily disables interrupts during critical stages of execution. However, the interrupt cannot be lost. When the system receives an interrupt, it takes a note. When the system considers appropriate, it checks the note and if there was an interrupt, the system runs the interrupt handler.

- **Suppose that an operating system does not have anything like the select system call (man select for more details on the command) to see in advance if it is safe to read from a file, pipe, or device, but it does allow alarm clocks to be set that interrupt blocked system calls. Is it possible to implement a threads package in user space under these conditions? Discuss.**

  Yes it is possible. The thread that wants to read first sets an alarm clock, then it reads. If it gets blocked, the alarm clock will interrupt the thread. If it does not get blocked, the alarm clock is reset. However the timing of the alarm clock can be tricky.

Ex. 2 Monitors

During the lecture monitors were introduced (3.30). They use condition variables as well as two instructions, wait and signal. A different approach would be to have only one operation called `waituntil`, which would check the value of a boolean expression and only allow a process to run when it evaluates as True. What would be the drawback of such a solution?

`waituntil` can work fine, however it consumes lots of resource. When a process / thread gets blocked by the `waituntil`, it has to wait in a tight loop for the boolean value, checking whether it is true. If we use `wait` and `signal`, the process / thread can safely release the CPU because it will be waken up later by `signal` from another process / thread.

Ex. 3 Race condition in bash

1. The race condition quickly happens, at line 22 and 23, both are 22.

   `ex3.sh`

   ```
   #!/bin/bash

   FILE=./file

   if ! test -f $FILE ; then
   ```

```
  echo 1 > $FILE
fi

for i in {1..20}
do
  last=$(tail -n 1 $FILE)
  ((last+=1))
  echo $last >> $FILE
done
```

`ex3_driver.sh`

```
#!/bin/bash

for i in {1..10}
do
  ./ex3.sh; ./ex3.sh&
done
```

2. modified script:

`ex3_flock.sh`

```
#!/bin/bash

FILE=./file

if ! [ -s $FILE ]; then
  echo 1 > $FILE
fi

for i in {1..20}
do
  (
    flock -n 33
    last=$(tail -n 1 $FILE)
    ((last+=1))
    echo $last >> $FILE
  )
  33>>$FILE
done
```

`ex3_driver.sh`

```
#!/bin/bash

rm file

for i in {1..10}
do
  # ./ex3.sh; ./ex3.sh&
  ./ex3_flock.sh; ./ex3_flock.sh &
done
```

Ex 4. Semaphore

`semaphore.h` is located in `/usr/include/semaphore.h`, more details can be found in manpages.

```
man sem_init
man sem_wait
man sem_post
man sem_destroy
```

Adjusted program is shown below:

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

#define N 1000000
int count = 0;
sem_t sem; // define sem

void * thread_count(void *a) {
    int i, tmp;
    for(i = 0; i < N; i++) {
        sem_wait(&sem); // down
        tmp = count;
        tmp = tmp+1;
        count = tmp;
        sem_post(&sem); // up
    }
    return NULL;
}

int main(int argc, char * argv[]) {
    int i;
    pthread_t *t=malloc(2*sizeof(pthread_t));
    // initialize semaphore
    if (sem_init(&sem, 0, 1)) {
      fprintf(stderr, "ERROR initializing semaphore\n");
      exit(1);
    }
    for (i = 0; i < 2; i++) {
        if (pthread_create(t+i, NULL, thread_count, NULL)) {
            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);
    pthread_exit(NULL);
    free(t);
```

```
    sem_destroy(&sem);
    return 0;
}

// more details in ./ex4
```