CS2106 Operating Systems

Semester 1 2021/2022

Tutorial 5

Synchronization

Note: Synchronization is important to both multi-threaded and multi-process programs. Hence, we will use the term **task** in this tutorial, i.e. we will not distinguish between process and thread.

1. **(Race conditions)** Consider the following two concurrent tasks:

```
int i = 0; // shared variable, initialized to 0
```

Task A	**Task B**
`plus = 0;` `while (i < 10) {` `  i++;` `  plus++;` `}` `print "A wins";`	`minus = 0;` `while (i > -10) {` `  i--;` `  minus++;` `}` `print "B wins";`

   a. Which task will win (i.e. print out the "**`X wins`**" message first)?
   b. If one of the tasks reaches the end of execution, will the other also finish eventually?
   c. If we allow **Task A** to start first, will that help **Task A** to win?
   d. Express the final value of "**`i`**" using "**`plus`**" and "**`minus`**".

2. **(Readers-writers problem)** As discussed in the lecture, the solution presented was not fair to the writer. The main issue is that reader tasks that arrive later than a writer task can jump queue and join the readers currently reading the data structure.

   Modify the simple solution such that once a writer "indicates interest" in modifying the data structure D, any later reader or writer will have to wait. The changes required are quite minor: only a couple new lines of code.

   (A hint can be found on the bottom of the last page; don't peek if you want to challenge yourself! ☺)

3. **(General semaphores)** We mentioned that a general semaphore (S > 1) can be implemented by using one or more **binary semaphores** (S == 0 or 1). Consider the following attempt:

```
int count = <initially: any non-negative integer>;
Semaphore mutex = 1; // binary semaphore
Semaphore queue = 0; // binary semaphore, for blocking tasks
```

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue)
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

Note: for ease of discussion, we allow the count to go negative in order to keep track of the number of task blocked on queue.

a. The solution is **very close**, but unfortunately can still have **undefined behavior** in some execution scenarios. Give one such execution scenario to illustrate the issue. (Hint: a binary semaphore's value can only be S = 0 or S = 1.)
b. Correct the attempt. Note that you only need very small changes to the two functions.

4. **(AY 19/20 Midterm)** Implement an intra-process mutual exclusion mechanism (a lock) **using Unix pipes**. Your implementation **must not use any synchronization construct** such as a mutex (`pthread_mutex_t`) or semaphore (`sem_t`).

   Information to refresh your memory:
   - In multithreaded processes, file descriptors are shared between all threads in a process. If multiple threads simultaneously call `read()` on a file descriptor, only one thread will be successful in reading any available data up to the buffer size provided, the others will remain blocked until more data is available to be read.
   - The read end of a pipe is at index 0, the write end at index 1.
   - System calls signatures for `read, write, open, close, pipe` (you may not need all of them):
     ```
     int pipe(int pipefd[2]);
     int open(const char *pathname, int flags, mode_t mode);
     int close(int fd);
     ssize_t read(int fd, void *buf, size_t count);
     ssize_t write(int fd, const void *buf, size_t count);
     ```
   - Marks will not be deducted for minor syntax errors.

Write your lock implementation below in the spaces provided. The definition of `struct pipelock` is already sufficient as-is, but feel free to add any other members you might need. You need to implement `lock_init`, `lock_acquire`, and `lock_release`.

Line#	Code
1 2 3	**/* Define a pipe-based lock */** struct pipelock { int fd[2]; };
11 12	**/* Initialize lock */** void lock_init(struct pipelock *lock) { }
21 22	**/* Function used to acquire lock */** void lock_acquire(struct pipelock *lock) { }
31 32	**/* Release lock */** void lock_release(struct pipelock * lock) { }

5. **(Dining Philosophers)** Our philosophers in the lecture are all left-handed (they pick up the left chopstick first). If we force **one of them** to be a right-hander, i.e. pick up the right chopstick before the left, then it is claimed that the philosophers can eat without explicit synchronization. Do you think this is a **deadlock-free solution** to the dining philosopher problem? You can support your claim informally (i.e. no need for a formal proof), but it must be convincing ☺.

Questions for your own exploration

6. **(Semaphore)** In cooperating concurrent tasks, sometimes we need to ensure that **all N tasks** have reached a certain point in the code before proceeding. This specific synchronization mechanism is commonly known as a **barrier**. Example usage:

```
//some code

Barrier(N);  // The first N-1 tasks reaching this point
             // will be blocked.
             // The arrival of the Nth task will release
             // all N tasks.

// Code here only gets executed after all N tasks
// reach the barrier above.
```

   Use **semaphores** to implement a **one-time use `Barrier()`** function **without using any loops.** Remember to indicate your variables declarations clearly.

7. **(Race condition)** Suppose the following two tasks share the variables X and Y, and they each run the line of code given below. What are the possible final results for X and Y? X and Y are both initialized to 1 at the beginning.

Task A	Task B
`X = Y + 1`	`Y = X + 1`

   If we use critical section to enclose the code for task A and B:

Task A	Task B
`EnterCS()` `X = Y + 1` `ExitCS()`	`EnterCS()` `Y = X + 1` `ExitCS()`

   What are the possible final results for X and Y?

8. Can disabling interrupts avoid race conditions? If yes, would disabling interrupts be a good way of avoiding race conditions? Explain.

Hint for Q2: Let's prevent new readers from entering the room if there is a waiting writer.