CS2106 Operating Systems

Semester 1 2021/2022

Tutorial 5 Solutions

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`**".

ANS :

   a. Non-deterministic. Either task can win, depending on the scheduling.
   b. Yes.
   c. Probably no guarantee. However, if the time quantum is large enough for task A to reach 10 before B joins the ready queue, then A will win.
   d. Not possible. The intuitive answer of "`plus – minus == i`" is **false.**

      The "`i++`" introduces a race condition between the tasks. Updates to i can be lost due to unfortunate interleaving, e.g.

Task A	**Task B**
`load i -> R1  // R1 = 0`	
`R1 + 1 -> R1  // R1 = 1`	
	`load i -> R2  // R2 = 0`
	`R2 - 1 -> R2  // R2 = -1`
	`store R2 -> i // i = -1`
`store R1 -> i  // i = 1`	

The above loses one subtraction. So, "plus − minus ≠ i" afterwards.

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! ☺)

ANS:

Basic idea: add a "revolving door" to the "room". This door normally allows everyone to enter one by one. However, as soon as someone bars the door, it will block all late comers.

Additional declarations:

```
Semaphore revDoor = 1;
```

Original code in black color, new code in red.

Writer

```
wait(revDoor); // blocks the "door"

wait(emptyRoom);
modifies data
signal(emptyRoom);

signal(revDoor);
```

Reader

```
wait(revDoor);
signal(revDoor); // immediately let another task pass through
                 // UNLESS the writer "blocks" the door.

wait(mutex);
nReader++;
if (nReader == 1)
    wait(roomEmpty);
signal(mutex);

Reads data

wait(mutex);
nReader--;
if (nReader == 0)
     signal(roomEmpty);
signal(mutex);
```

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.

ANS:

a. The issue is task interleaving between "`signal(mutex)`" and "`wait(queue)`" in `GeneralWait()` function. Consider the scenario where count is 0, two tasks A and B execute `GeneralWait()`, as task A clears the "`signal(mutex)`", task B gets to executes until the same line. At this point, count is -2. Suppose two other tasks C and D now executes `GeneralSignal()` in turns, both of them will perform `signal(queue)` due to the count -2. Since `queue` is a binary semaphore, the 2nd `signal()` will have undefined behavior (remember that we cannot have S = 2 for binary semaphore).

b. Corrected version:

```
int count = <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 removed
```

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

`    signal(mutex);` `}`	`    }` `}`

Using the same execution scenario in (a), task D will not reach the 2nd `signal(queue)` as it will block on `mutex`. Once either task A or B pass the `wait(queue)`, they will `signal(mutex)`, allowing task D to proceed. At this point in time, the value of `queue` is now 0, so the subsequent `signal(queue)` by task D will not result in undefined behaviour.

Reference: D. Hemmendinger, "A correct implementation of general semaphores", Operating Systems Review, vol. 22, no. 3 (July, 1988), pp. 42-44.

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) {`
	
	`    pipe(lock->fd);`
	`    write(lock->fd[1], "a", 1);`

```
    // The first write is meant to initialize the lock such that exactly one thread can acquire the lock.


}
21  /* Function used to acquire lock */
22  void lock_acquire(struct pipelock *lock) {

        char c;
        read(lock->fd[0], &c, 1);

    // read will block if there is no byte in the pipe.

    // Closing the reading or writing end of the pipe in a thread will cause closing that end for all threads of the process (shared variable). Also, it will prevent all other threads to acquire or release the lock.
    }
31  /* Release lock */
32  void lock_release(struct pipelock * lock) {

        write(lock->fd[1], "a", 1);

    // Need to write/read exactly one byte to simulate increment/decrement by 1 of a semaphore. It might work with multiple bytes, but you need to take care how many bytes are read/written.



    }
```

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 ☺.

ANS:

The claim is TRUE. Informal argument below.

For ease of discussion, let's refer to the right-hander as **R**.

If **R** grabbed the right chopstick then managed to grab the left chopstick THEN
R can eat ➔ not a deadlock.

If **R** grabbed the right chopstick but the left chopstick is taken THEN
The left neighbor of R has already gotten both chopstick s ➔ eating ➔ eventually release chopsticks.

If **R** cannot grab the right chopstick THEN
The right neighbor of R has taken its left chopstick. Worst case scenario: all remaining left-handers hold on to their left chopsticks. However, the left neighbor of R will be able to take its right chopstick because R is still trying to get its right fork.

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.

ANS:

```
int arrived = 0; // shared variable
Semaphore mutex = 1; // binary semaphore to provide mutual exclusion
Semaphore waitQ = 0; // for N - 1 processes to block on


Barrier(N) {
    wait(mutex);
    arrived++;
    signal(mutex);
    if (arrived == N)
        signal(waitQ);

    wait(waitQ);
    signal(waitQ);
}
```

This is not a **reusable barrier**, as **`waitQ`** may have undefined value afterwards, e.g. if tasks interleave just before `if (arrived == N)`, multiple tasks can execute the first `signal(waitQ)` resulting in `waitQ > 0` at the end. Note that this does not affect correctness.

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?

ANS:
The code can be roughly translated to the following pseudo machine instructions:

Task A	Task B

`$r1 <- Y`
`$r1 <- $r1 + 1`
`X <- $r1`
	`$r2 <- X`
	`$r2 <- $r2 + 1`
	`Y <- $r2`

The pair of load and store instructions (1st and 3rd instruction) for each task is the main cause of the race condition issue. Omitting the middle instructions, we have the following six interleaving patterns:

`$r1 <- Y`
`X <- $r1`
`$r2 <- X`
`Y <- $r2`
	`$r1 <- Y`
	`$r2 <- X`
	`X <- $r1`
	`Y <- $r2`
		`$r1 <- Y`
		`$r2 <- X`
		`Y <- $r2`
		`X <- $r1`

X = 2, Y = 3	X = 2, Y = 2	X = 2, Y = 2

`$r2 <- X`
`$r1 <- Y`
`X <- $r1`
`Y <- $r2`
	`$r2 <- X`
	`$r1 <- Y`
	`Y <- $r2`
	`X <- $r1`
		`$r2 <- X`
		`Y <- $r2`
		`$r1 <- Y`
		`X <- $r1`

X = 2, Y = 2	X = 2, Y = 2	X = 3, Y = 2

With the critical section, there are only two possible interleaving patterns. Since all the steps of the high level language statement is executed together, we only need to consider the interleaving at the statement level:

`X = Y + 1` `Y = X + 1`	`Y = X + 1` `X = Y + 1`
X = 2, Y = 3	X = 3, Y = 2

Comparing the execution sequences, it is clear that race conditions allows "impossible" execution results. Out of the six sequences from the first part, only two gives "correct" results.

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

Answer:

Yes, disabling interrupts and enabling them back is equivalent to acquiring a universal lock and releasing it, respectively. Without interrupts, there will be no quantum-based process switching (because even timer interrupts cannot happen); hence only one process is running. However:

- This will not work in a multi-core/multi-processor environment since another process may enter the critical section while running on a different core.
- User code may not have the privileges needed to disable timer interrupts
- Disabling timer interrupts means that many scheduling algorithms will not work properly
- Disabling non-timer interrupts means that high-priority interrupts that may not even share any data with the critical section may be missed.
- Many important wakeup signals are provided by interrupt service routines and these would be missed by the running process. A process can easily block on a semaphore and stay blocked indefinitely, because there is nobody to send a wakeup signal.
- if a program disables interrupts and hangs, the entire system will no longer work since it cannot switch tasks and perform anything else.

One should therefore not disable interrupts for the purpose of locking.

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