COMP3231/9201/3891/9283 Operating Systems 2021/T1

UNSW

Tutorial Week 3

Questions and Answers

Synchronisation Problems

The following problems are designed to familiarise you with some of the problems that arise in concurrent programming and help you learn to identify and solve them.

Coordinating activities

1. What synchronisation mechanism or approach might one take to have one thread wait for another thread to update some state?

   Semaphore style answer (semaphore `updated`, with count initialised a 0)

   ```
   wait_for_update()
   {
      P(updated)
   }

   signal_update_occurred()
   {
      V(updated)
   }
   ```

   CV style answer with a variable `flag = 0`, a lock `l`, and a CV `cv` updated

   ```
   wait_for_update()
   {
     lock_acquire(l)
     while(flag == 0)
       cv_wait(cv,l)
     lock_release(l)
   }

   signal_update_occurred()
   {
   lock_acquire(l)
     flag = 1
     cv_signal(cv,l)
   lock_release(l)
   }
   ```

---

2. A particular abstraction only allows a maximum of 10 threads to enter the "room" at any point in time. Further threads attempting to enter the room have to wait at the door for another thread to exit the room. How could one implement a synchronisation approach to enforce the above restriction?

   Semaphore style answer (semaphore `room`, with count initialised to 10)

```
enter_room()
{
   P(room);
}

leave_room()
{
   V(room);
}
```

CV style answer with a variable `occupants = 0`, a lock `room_lock`, and a CV `room_cv` updated

```
enter_room()
{
  lock_acquire(room_lock);
  while(occupants == 10)
    cv_wait(room_cv,room_lock);
  occupants = occupants + 1;
  lock_release(room_lock);
}

leave_room()
{
  lock_acquire(room_lock);
    occupants = occupants - 1;
    if (occupants == 9)
      cv_signal(room_cv,room_lock);
  lock_release(room_lock);
}
```

---

3. Multiple threads are waiting for the same thing to happen (e.g. a disk block to arrive from disk). Write pseudo-code for a synchronising and waking the multiple threads waiting for the same event.

   A semaphore style answer with a variable `waiters_count = 0, disk_block_status = NOT_READY`, a semaphore `block_mutex` initialised to 1, and `block_sem` initialised to 0.

```
wait_block()
{
  P(block_mutex);
  if (disk_block_status == READY) {
    V(block_sem);
  } else {
    waiters_count = waiters_count + 1;
  }
  V(block_mutex);
  P(block_sem);
}

make_block_ready()
{
  P(block_mutex);
    disk_block_status = READY;
    while(waiters_count != 0) {
      waiters_count = waiters_count -1;
      V(block_sem);
    }
  V(block_mutex);
}
```

   CV style answer with a variable `disk_block_status = NOT_READY`, a lock `block_lock`, and a CV `block_cv` updated

```
wait_block()
{
```

```
  lock_acquire(block_lock);
  while(disk_block_status != READY)
    cv_wait(block_cv,block_lock);
  lock_release(room_lock);
}

make_block_ready()
{
  lock_acquire(block_lock);
  disk_block_status = READY;
  cv_broadcast(block_cv,block_lock);
  lock_release(block_lock);
}
```

Identify Deadlocks

4. Here are code samples for two threads that use semaphores (count initialised to 1). Give a sequence of execution and context switches in which these two threads can deadlock.

   Propose a change to one or both of them that makes deadlock impossible. What general principle do the original threads violate that causes them to deadlock?

```
semaphore *mutex, *data;

void me() {
        P(mutex);
        /* do something */

        P(data);
        /* do something else */

        V(mutex);

        /* clean up */
        V(data);
}

void you() {
        P(data)
        P(mutex);

        /* do something */

        V(data);
        V(mutex);
}
```

   The numbers inserted on the left indicate an execution order that results in deadlock.

```
semaphore *mutex, *data;

void me() {
1:      P(mutex);
        /* do something */

4:      P(data);
        /* do something else */

        V(mutex);

        /* clean up */
        V(data);
}
```

```
void you() {
2:      P(data)
3:      P(mutex);

        /* do something */

        V(data);
        V(mutex);
}
```

To prevent deadlock, ensure the semaphores are acquired in the same order.

```
semaphore *mutex, *data;

void me() {
        P(mutex);
        /* do something */

        P(data);
        /* do something else */

        V(mutex);

        /* clean up */
        V(data);
}

void you() {
        P(mutex);
        P(data)

        /* do something */

        V(data);
        V(mutex);
}
```

More Deadlock Identification

5. Here are two more threads. Can they deadlock? If so, give a concurrent execution in which they do and propose a change to one or both that makes them deadlock free.

```
lock *file1, *file2, *mutex;

void laurel() {
        lock_acquire(mutex);
        /* do something */

        lock_acquire(file1);
        /* write to file 1 */

        lock_acquire(file2);
        /* write to file 2 */

        lock_release(file1);
        lock_release(mutex);

        /* do something */

        lock_acquire(file1);

        /* read from file 1 */
        /* write to file 2 */
```

```
        lock_release(file2);
        lock_release(file1);
}

void hardy() {
        /* do stuff */

        lock_acquire(file1);
        /* read from file 1 */

        lock_acquire(file2);
        /* write to file 2 */

        lock_release(file1);
        lock_release(file2);

        lock_acquire(mutex);
        /* do something */
        lock_acquire(file1);
        /* write to file 1 */
        lock_release(file1);
        lock_release(mutex);
}
```

The way to look for deadlock potential in this example is to record the order of locking for both threads and look for inconsistencies.

- laurel: mutex -> file1 -> file2; file2->file1
- hardy: file1 -> file2; mutex -> file1

Note file1 and file2 can be acquired in two different orders, which can results in deadlock. See the example below for how it can occur.

```
lock *file1, *file2, *mutex;

void laurel() {
1:      lock_acquire(mutex);
        /* do something */

2:      lock_acquire(file1);
        /* write to file 1 */

3:      lock_acquire(file2);
        /* write to file 2 */

4:      lock_release(file1);
7:      lock_release(mutex);

        /* do something */

8:      lock_acquire(file1);

        /* read from file 1 */
        /* write to file 2 */

        lock_release(file2);
        lock_release(file1);
}

void hardy() {
        /* do stuff */

5:      lock_acquire(file1);
        /* read from file 1 */

6:      lock_acquire(file2);
        /* write to file 2 */
```

```
        lock_release(file1);
        lock_release(file2);

        lock_acquire(mutex);
        /* do something */
        lock_acquire(file1);
        /* write to file 1 */
        lock_release(file1);
        lock_release(mutex);
}
```

The way to prevent deadlock is to define a global locking order and ensure all threads adhere to the order. For this example, lets choose mutex -> file1 -> file2. This implies we should adjust laurel. Here are two versions of laurel that result in deadlock freedom (there are others).

```
lock *file1, *file2, *mutex;

void laurel() {
        lock_acquire(mutex);
        /* do something */

        lock_acquire(file1);
        /* write to file 1 */

        lock_acquire(file2);
        /* write to file 2 */

        lock_release(mutex);

        /* do something */

        /* read from file 1 */
        /* write to file 2 */

        lock_release(file2);
        lock_release(file1);
}
```

Or

```
void laurel() {
        lock_acquire(mutex);
        /* do something */

        lock_acquire(file1);
        /* write to file 1 */

        lock_acquire(file2);
        /* write to file 2 */

        lock_release(file1);
        lock_release(mutex);

        /* do something */

        lock_release(file2);
        lock_acquire(file1);
        lock_acquire(file2);

        /* read from file 1 */
        /* write to file 2 */

        lock_release(file2);
        lock_release(file1);
```

```
}
```

Synchronised Lists

6. Describe (and give pseudocode for) a synchronised linked list structure based on thread list code in the OS/161 codebase (`kern/thread/threadlist.c`). You may use semaphores, locks, and condition variables as you see fit. You must describe (a proof is not necessary) why your algorithm will not deadlock.

   In a general sense, the interface to the synchronised list is as follows.

   ```
   init(list_t *);
   add_head(list_t *list, node_t *node);
   add_tail(list_t *list, node_t *node);
   remove_head(list_t *list, node_t **node);
   remove_tail(list_t *list, node_t **node);
   insert_after(node_t *in_list, node_t *new_node);
   insert_before(node_t *in_list, node_t *new_node);
   remove(node_t *in_list);
   ```

   Make sure you clearly state your assumptions about the constraints on access to such a structure and how you ensure that these constraints are respected.

   In addition to a single lock solution, consider a solution involving a lock per node in the list. The instructive cases are `insert_after()` and `insert_before()`, and `remove()`

   The thread subsystem in OS/161 uses a linked list of threads to manage some of its state (`kern/thread/threadlist.c`). This structure is not synchronised. Why not?

   > In OS/161, the threadlist is a doubling linked list where all the code is written under the assumption that is executes mutually exclusively, which in practice is ensured by interrupt disabling on the uniprocessor. Hence, there is no synchronisation in the threadlist code itself.
   >
   > Writing a synchronised version of the general list management code could be done with a single lock that is acquired for all updates to the list and nodes itself. This would not allow independent updates to separate nodes in the list to proceed in parallel.
   >
   > One could use a lock per node in the list in an attempt to enable more parallelism, combined with locking in only one list direction when multiple locks are require for a single update. This is not as simple as it seems, see the sample code for how complex individual locking becomes for some operations. Given the number of locks required, and that operations all start at the head to avoid deadline, fine-grained locking in this case is worse than a single lock as it simply add overhead of more locks to acquire.

   ```
   #include
   #include
   #include
   #include "list.h"

   bool node_init(node_t *node)
   {
           struct lock *lock;
           lock = lock_create("node lock");
           if (lock == NULL)
                   return false;
           node->lock = lock;
           node->next = NULL;
           node->prev = NULL;
           return true;
   ```

```
}
void node_cleanup(node_t *node)
{
        lock_destroy(node->lock);
}

bool list_init(list_t *list)
{
        bool r;
        r = node_init(&list->head);
        if (! r)
                return false;
        r = node_init(&list->tail);
        if (! r) {
                node_cleanup(&list->head);
                return false;
        }
        list->head.next = &list->tail;
        list->tail.prev = & list->head;
        return true;
}


void list_cleanup(list_t *list)
{
        /* TBD */
        (void) list;
}

void add_head(list_t *list, node_t *node)
{
        insert_after(&list->head, node);
}

void add_tail(list_t *list, node_t *node)
{
        insert_before(list, &list->tail, node);
}

void insert_after(node_t *in_list, node_t *new_node)
{
        /* assumption, in_list is not removed before we successfully
           insert, this in only true for the head sentinal
           node */

        lock_acquire(new_node->lock); /* start new node in the locked state */

        /* acquire in order head -> tail */
        lock_acquire(in_list->lock);
        lock_acquire(in_list->next->lock);

        /* now can add */
        new_node->next = in_list->next;
        new_node->prev = in_list;

        in_list->next = new_node;
        new_node->next->prev = new_node;

        lock_release(new_node->next->lock);
        lock_release(new_node->lock);
        lock_release(in_list->lock);
}




static bool hand_over_locking(list_t *list, node_t *target)
{
        /* do hand over locking until we find the target */
```

```
        /* Either we find the target and return with target and
           predecessor locked or we return false if we don't find
           it */

        node_t *p, *t;

        p = &list->head;
        lock_acquire(p->lock);

        t = p->next;
        lock_acquire(t->lock);
        while (t != target) {
                if (t == &list->tail) {
                        /* can't find target */
                        lock_release(p->lock);
                        lock_release(t->lock);
                        return false;
                }
                lock_release(p->lock);
                p = t;
                t = t-> next;
                lock_acquire(t->lock);
        }
        return true;
}


bool insert_before(list_t *list, node_t *in_list, node_t *new_node)
{
        /* assume in_list is not removed before starting, this is only
         true for the tail sentinel node */

        bool r;

        lock_acquire(new_node->lock); /* again start in locked state */


        /* lock acquire in order head -> tail

           This fragment attempt to avoid hand-over locking by starting at the
           tail. It accesses prev without holding the lock, so it might change,
           and thus needs to check it's still prev after both locks are held.


           p = in_list->prev;
           lock_acquire(p->lock);
           lock_acquire(in_list->lock);

           while (p->next != in_list) {

                 a node must have got inserted
                 prior to getting the lock

                lock_release(p->lock);
                lock_release(in_list->lock);
                p = in_list->prev;
                lock_acquire(p->lock);
                lock_acquire(in_list->lock);
           }

           Yuck, this is livelock-able....  Basically one can't use the
           prev pointers to lock as we require the current node's lock
           prior to using prev, but then we can't lock the
           predecessor due to lock ordering needed to avoid deadlock.

        */


        /* we changed the prototype to include the list and use
```

```
           handover locking to lock in_list and predecessor */

        r = hand_over_locking(list, in_list);

        /* this should always succeed in a system that does not remove
           in_list */

        if (!r) {
                lock_release(new_node->lock);
                return r;
        }

        /* now have locks on prev and in_list */

        /* now can add */
        new_node->prev = in_list->prev;
        new_node->next = in_list;

        in_list->prev = new_node;
        new_node->prev->next = new_node;

        lock_release(new_node->prev->lock);
        lock_release(new_node->lock);
        lock_release(in_list->lock);
        return true;
}

static void remove_with_locks_held(node_t *in_list)
{
        in_list->prev->next = in_list->next;
        in_list->next->prev = in_list->prev;

}

void remove_head(list_t *list, node_t **node)
{
        node_t *n;
        lock_acquire(list->head.lock); /* sentinel head */
        lock_acquire(list->head.next->lock); /* removal candidate */

        if (list->head.next == &list->tail) {
                /* nothing to remove */
                lock_release(list->head.next->lock);
                lock_release(list->head.lock);
                *node = NULL; /* return null for empty list */
        }

        lock_acquire(list->head.next->next->lock); /* relies on lock
                                                      held above to use
                                                      the pointer */

        n = list->head.next;

        remove_with_locks_held(n);

        lock_release(list->head.lock); /* sentinel head */
        lock_release(n->lock); /* removed node */
        lock_release(list->head.next->lock); /* new head */

        *node = n;
}

void remove_tail(list_t *list, node_t **node)
{

/* leave this as an exercise, this is a handover involving three nodes
   where the last one is the tail */

        (void) list;
```

```
        (void) node;
}

bool remove(list_t *list, node_t *in_list)
{

        bool r;

        r = hand_over_locking(list, in_list);

        /* this should always succeed in a non-broken system,
           otherwise we have attempts to remove the same node more than
           once */

        if (!r)
                return r;

        /* in_list and predecessor is lock */

        lock_acquire(in_list->next->lock); /* grab the third lock */

        /* we can now remove the node */

        remove_with_locks_held(in_list);

        lock_release(in_list->prev->lock);
        lock_release(in_list->next->lock);
        lock_release(in_list->lock);
        return true;
}
```

Concurrency and Deadlock

7. For each of the following scenarios, one or more dining philosophers are going hungry. What is the condition the philosophers are suffering from?

    a. Each philosopher at the table has picked up his left fork, and is waiting for his right fork
    b. Only one philosopher is allowed to eat at a time. When more than one philosophy is hungry, the youngest one goes first. The oldest philosopher never gets to eat.
    c. Each philosopher, after picking up his left fork, puts it back down if he can't immediately pick up the right fork to give others a chance to eat. No philosopher is managing to eat despite lots of left fork activity.

        a. Deadlock
        b. Starvation
        c. Livelock

---

8. What is starvation, give an example?

    Starvation is where the system allocates resources according to some policy such that progress is being made, however one or more processes never receive the resources they require as a result of that policy.

    Example, a printer that is allocated based on "smallest print job first" in order to improve the response for small jobs. A large job on a busy system may never print and thus *starve*

---

9. Two processes are attempting to read independent blocks from a disk, which involves issuing a *seek* command and a *read* command. Each process is interrupted by the other in between its *seek* and *read*.

When a process discovers the other process has moved the disk head, it re-issues the original *seek* to re-position the head for itself, which is again interrupted prior to the *read*. This alternate seeking continues indefinitely, with neither process able to read their data from disk. Is this deadlock, starvation, or livelock? How would you change the system to prevent the problem?

> It is livelock. Allow each process to lock the disk and issue both commands together mutually exclusively and then release the lock.

---

10. Describe four ways to *prevent* deadlock by attacking the conditions required for deadlock.

    - Mutual exclusion condition
      - Make the resource sharable, i.e. allow concurrent access to read-only files. However, in general some resources are not shareable and require mutual exclusion.
    - Hold and wait condition
      - Dictate only a single resource can be held at any time. Not really practical.
      - Require that all required resource be obtained initially. If a resource is not available, all held resources must be releases before trying again - prone to livelock.
    - No preemption condition
      - Preempt the resource (take it away from the holder). Not always possible.
    - Circular wait condition
      - Order the resources numerically and request them in numerical order.

---

11. Answer the following questions about the tables.

    a. Compute what each process still might request and display in the columns labeled "still needs".
    b. Is the system in a safe or unsafe state? Why?

       Safe, feasible schedule p1,p4,p5,p2,p3

    c. Is the system deadlocked? Why or why not?

       No. There are not process remaining after the feasible schedule p1,p4,p5,p2,p3

    d. Which processes, if any, are or may become deadlocked?

       None

    e. Assume a request from p3 arrives for (0,1,0,0)
       1. Can the request be safely granted immediately?

          No

       2. In what state (deadlocked, safe, unsafe) would immediately granting the request leave the system?

          Unsafe

       3. Which processes, if any, are or may become deadlocked if the request is granted immediately?

          p2, p3

available			
r1	r2	r3	r4
2	1	0	0

	current allocation				maximum demand				still needs			
process	r1	r2	r3	r4	r1	r2	r3	r4	r1	r2	r3	r4
p1	0	0	1	2	0	0	1	2				
p2	2	0	0	0	2	7	5	0				
p3	0	0	3	4	6	6	5	6				
p4	2	3	5	4	4	3	5	6				
p5	0	3	3	2	0	6	5	2				

	current allocation				maximum demand				still needs			
process	r1	r2	r3	r4	r1	r2	r3	r4	r1	r2	r3	r4
p1	0	0	1	2	0	0	1	2	0	0	0	0
p2	2	0	0	0	2	7	5	0	0	7	5	0
p3	0	0	3	4	6	6	5	6	6	6	2	2
p4	2	3	5	4	4	3	5	6	2	0	0	2
p5	0	3	3	2	0	6	5	2	0	3	2	0

12. Solve the Dining Philosopher's problem below using locks and a different strategy to the one shown in lectures.

```
void take_both_forks(unsigned long phil_num)
{

/*
 * Take forks ensures mutually exclusive access to two forks
 * associated with the philosopher.
 *
 * The left fork number = phil_num
 * The right fork number = (phil_num + 1) % NUM_PHILOSPHERS
 */

}



void release_forks(unsigned long phil_num)
{
/*
 * Releases forks releases the mutually exclusive access to the
 * philosophers forks.
 */
```

The strategy is to use a lock per fork. To avoid deadlock we have to always acquire the lower numbered fork first, including handling the wrap-around case.

```
struct lock *fork_locks[NUM_PHILOSOPHERS];

void take_both_forks(unsigned long phil_num)
{
    int lower, higher;

    lower = phil_num; /* left fork */
    higher = (phil_num + 1) % NUM_PHILOSOPHERS; /* right fork */

    if (lower > higher) {
        /* swap lower/higher to avoid deadlock */
        lower = higher;
        higher = phil_num;
    }

    lock_acquire(fork_locks[lower]);
    lock_acquire(fork_locks[higher]);
```

```
}

void release_forks(unsigned long phil_num)
{
    lock_release(fork_locks[phil_num]);
    lock_release(fork_locks[(phil_num + 1) % NUM_PHILOSOPHERS]);
}
```

Page last modified: 11:27am on Monday, 1st of March, 2021

Screen Version

CRICOS Provider Number: 00098G