Discussion of a Possible Solution for HW2

Deniz Sayın

Contents

Note: *Agent* refers to both proper privates and sneaky smokers.

1 Part I

The point of this part is being very easy since there is only one thing to consider: proper privates not intersecting.

1.1 Per-cell Mutexes

Now, a common approach here was having per-cell mutexes and having privates acquiring areas lock down all of the mutexes in the area one by one. Avoiding deadlocks is easy with this approach since you can have a total-order between the mutexes. For example, always going from top-left to bottom right when acquiring them will avoid deadlocks which you can do pretty much inadvertantly.

The issue with the per-cell mutex approach is that it's not all-or-nothing: privates hold some locks even while waiting, which can block new privates from acquiring areas. See the first case in complex_scenarios.pdf as an example.

1.2 Semaphore Sets

Another way to deal with this would be to use System V semaphore sets which can up/down multiple semaphores atomically. However, these are somewhat clunky semaphores designed with inter-process synchronization in mind and have a limit of 32,000 semaphores per set in Linux, which makes them not viable for our case, since grids can have more than 32,000 cells. They would still be fine for most small cases though.

1.3 Mutex-protected Data Structure

A more common approach is having a bunch of data structures you use for synchronization and protecting them with a mutex, like the following pseudo-code:

```
// Locking the area, sleep on cond if can't lock yet
lock(&mutex);
while (area_not_available())
    wait(&cond, &mutex);
lockdown_area();
unlock(&mutex);

// .. do stuff here, like cleaning up the area
// ..
// ..

// Unlock the area
lock(&mutex);
unlock_area();
wake_waiting_intersecting_privates(); // use signal here
unlock(&mutex);
```

One wasteful part here is waking all intersecting privates, even though only one will be able to acquire the lock. But we may be in a bad spot if we wake just one: one intersecting private may be blocked by another intersection as well. Since checking that would also be inefficient, and hoping there won't be too many intersecting privates with just one area, I opted to wake up all intersecting privates.

In a real-life scenario, when there are multiple viable solutions, it is important to analyse the use cases of your application and choose an approach that makes the most sense based on that.

Now, many of you wanted to avoid this kind of approach at first since it seems unnatural: independent privates can't lock areas in parallel, they each have to do their locking sequentially, but are parallel once they finish locking. This is pretty normal though when you have something to synchronize! Consider the read-lock implementation for the reader-writer problem in the lecture notes:

```
// Acquiring a read-lock
wait(mutex);
readcount++;
if (readcount == 1)
    wait(write);
signal(mutex);

// .. read here
// ..
// ..

// Releasing the read-lock
wait(mutex);
readcount--;
if (readcount == 0)
    signal(write);
signal(mutex);
```

The mutex here is necessary to protect the readcount variable. This means that multiple readers can't acquire the lock in parallel, but they can read in parallel once they have all acquired the lock. Think of it like a conference hall with one door: only one person can enter at a time, but there can be many people inside at the same time. It just works!™

1.4 Implementing with an Area List

My own example implementation tries to keep it simple by simply using two lists, one for the areas of active privates with `actives` and another for the areas of sleeping privates in `waiters`. Each private has their own condition variable they can sleep on. The pseudo-code is something like this:

```
/* Begin: locking the area */
lock(&mutex);
if (intersects_with(actives)) {
    // Insert into waiting list
    append(waiters, private);

    // Wait for no more intersections
    do {
        wait(&private->cv, &mutex);
    } while(intersects_with(actives));

    // No more intersections, remove from waiter list
    remove(waiters, private);
}
// Add to the active list and unlock the mutex
append(actives, private);
/* End: locking the area */

// Do stuff here
// ..
// ..

/* Begin: unlocking the area */
lock(&mutex);

// Remove from active list, wake up all intersecting privates
remove(actives, private);
for (waiting_private : get_intersecting_privates(waiters, private))
    signal(&waiting_private->cv);

unlock(&mutex);
/* End: unlocking the area */
```

Of course, this approach is inefficient since finding intersections is $\mathcal{O}(n)$, but it's enough for our purposes. In a real-life scenario, it could make sense to use something like an R-tree or quadtree for possibly $\mathcal{O}(\log n)$ intersection checks.

2 Part 2

2.1 No Other Notifications After an Order is Given

So far so good, now it's time for the order mechanism! This is where the homework gets much more complicated. A commander is supposed to give orders at certain time instances, which is fine. But synchronization is difficult due to the following constraint:

Constraint: No notifications other than order acknowledgments should be sent after an order

notification.

2.1.1 Pseudocode

Since it is possible for proper privates to be anywhere in their code when an order is supposed to be given, this constraint is not satisfied naturally. We need to prevent an order notification from being sent while proper privates are doing things. We can first approach this by noting critical sections in private code where no order should be given. The private pseudo-code is something like this:

```
notify(CREATED);

for (region : regions) {

    area_lock(region);
    notify(LOCKED);

    for (cell : region) {
        while (cell > 0) {
            sleep(rest_time);
            cell--;
            notify(GATHERED);
        }
    }

    area_unlock(region);
    notify(UNLOCKED);
}

notify(EXITED);
```

We need to avoid giving orders while privates are in certain small regions of code:

```
notify(CREATED);

for (region : regions) {

    area_lock(region);
    /* Begin: region 0 */
    notify(LOCKED);
    /* End: region 0 */

    for (cell : region) {
        while (cell > 0) {
            sleep(rest_time);
            /* Begin: region 1 */
            cell--;
            notify(GATHERED);
            /* End: region 1 */
        }
    }

    /* Begin: region 2 */
    notify(UNLOCKED);
    area_unlock(region);
    /* End: region 2 */
}

notify(EXITED);
/* End: region 2 when exiting loop */
```

Now, these regions are not exact, but there is some reasoning behind the choices:

- Region 0: Prevents locking notifications, the private can easily unlock the region again silently when responding to break/stop. Why not start it before `area_lock(region)`? Because that is a recipe for deadlocks in case the private has to wait for no intersections.
- Region 1: Simple, just prevent orders while the private is in the small region for decrementing cell values.
- Region 2: Prevent orders while notifying of unlocks. This is more open, it's also possible to remove the area unlocking from the region depending on how you handle orders, and it's also possible to combine it with the end of region 1 by moving the end of region 1 to the start of the inner loop, just before `sleep(rest_time)`. It's also important to extend this region to the include the exit notification when exiting the loop, to prevent an 'exit' notification right after an order is given.

So, this is how the code could become to satisfy all the requirements:

```
notify(CREATED);

lock(order_lock); // To be consistent with the unlock at the start
for (region : regions) {

    unlock(order_lock); // To unlock when restarting loop

    area_lock(region);
    lock(order_lock); // Prevent orders after here
    check_for_orders();
    notify(LOCKED);

    for (cell : region) {
        while (cell > 0) {
            unlock(order_lock); // Allow again before sleeping
            sleep(rest_time);
            lock(order_lock); // Prevent again after wakeup
            check_for_orders();
            cell--;
            notify(GATHERED);
        }
    }

    // Do note that the order_lock is still locked here
    notify(UNLOCKED);
    area_unlock(region);
}

notify(EXITED);
unlock(order_lock);
```

Of course, the private has to check for the existence of an order right after locking the `order_lock` every time and react accordingly. e.g. release the area and send a "took a break" or "stopped" notification and whatnot. Not too difficult to refactor into a function to prevent a mess.

2.1.2 Lock Type

Now the issue is the kind of lock to use here. It would be possible to make the `order_lock` a simple mutex. Totally possible. The only disadvantages are the fact that independent privates would not be able to stay in the same critical region which is bad but kind of insignificant since they already have to hold a mutex for notify and the rest of the code is really short. The second disadvantage is that starving is possible, the commander that wants to give an order is not guaranteed to get the mutex before other incoming privates.

To handle both of these issues, I opted to use a read-write lock with writer priority, which is a common twist on the reader-writer problem (it's even in the little book of semaphores). With

this implementation, privates can be in critical regions in parallel since they acquire a read-lock on the order. More importantly, whenever the commander (writer) wants to lock for giving an order, all privates coming after the commander will wait for the commander to acquire the lock first. This solves the starving issue: the commander will be able to give the order as soon as all privates *currently* in a critical region unlock their read-locks. See `sm_soldier_lock()` and `sm_soldier_unlock()` to see how reader lock and unlock and `sm_commander_order()` to see how the commander locks and unlocks in my own implementation.

Another fancy thing with this is that a private can *relock* the order lock (i.e. unlock and then lock immediately after) to *allow* a commander waiting to give orders to give them. Not necessary, but fun to think about.

2.1.3 What About Those Stuck in the Area Locking?

In case privates are sleeping in `area_lock()`, they will wait for the privates currently holding the areas to unlock them after getting a break/stop order. Then, they will be able to lock the area, notice that they have gotten an order, and will unlock the area again. Kind of wasteful since they are locking the area just to unlock it again immediately. Not a huge issue and it allows us to keep the order and area synchronization independent. This is a problem we'll have to solve in Part III though.

2.2 Waking up From Sleep to get Orders

So far we've handled the synchronization of orders, but privates cannot yet wake up from their rest in-between cigbutts when they get an order. This is because we can't interrupt sleep-style functions except by signaling the thread. Handling this without signals is easy though: we just need a condition variable for issuing orders to sleep on with a timed wait. Here's a simplified version.

```
lock(order_lock);
while (order != stop && order != break) {
    int r = timed_wait(cond_order_issued, order_lock, rest_time);
    if (r == TIMEOUT)
        break;
}
```

This corresponds to `pthread_cond_timedwait` in the pthread library. The commander will broadcast to `cond_order_issued` after giving an order. The main thread acts as the commander thread sending orders in my implementation. See the `cigrest` function for a more real version of this that also interacts with the read-write lock mechanism.

Using signals here makes life difficult!

2.3 Orders in Quick Succession

This is a problematic one, which is why it just 2 points in the final grading.

For a commander to be able to wait for everyone to receive the order, we can use some counting, i.e. we need to know how many soldiers there are and how many have received the order. The

code for the commander could be like:

The hard part here is that the number of soldiers is changing, not all of them will leave at the same time. One soldier could continue from a break and exit, while another has not even acknowledged the continue order yet. It is important to be very careful with the synchronization of these numbers (active soldiers, soldiers who have acknowledged the order) to not mess up the notification scheme. It can end in a deadlock very easily.

Explaining the solution in detail here would be a bit unnecessary since it's just a small part of the homework, so you can check the code in `sm_commander_await_orders_received` in `sync_mechanism.c` and track from there if you want to see one possible approach.

2.4 Other Details

Other implementation details like making sure to unlock areas during a break/stop and handling consecutive same orders are easy to implement, so I won't be covering them here.

3 Part 3

Part 3 just wanted to see you extend the existing scheme with another agent. The core ideas are the same, you just have to add a bit more sauce on top.

3.1 Locking Interactions between Proper Privates and Sneaky Smokers

Sneaky smokers slightly complicate the area locking mechanism since they can intersect with each other, but not by very much. An overview of the area interactions:

- **Proper Private - Proper Private:** Areas should not intersect.
- **Proper Private - Sneaky Smoker:** Areas should not intersect.
- **Sneaky Smoker - Sneaky Smoker:** Cells should not be the same. In other words, areas should not be exactly the same.

In my implementation, I added a second pair of lists for active and waiting sneaky smokers. The locking and unlocking is only slightly different:

- **Proper Private**
  - **Locking:** Check for intersections with active proper privates and sneaky smokers. Wait for there to be none and then acquire the area, adding yourself to the active list.
  - **Unlocking:** Remove yourself from the active list. Wake all intersecting proper privates and sneaky smokers.
- **Sneaky Smoker**
  - **Locking:** Check for intersections with active proper privates and exact same area with sneaky smokers. Wait for there to be none and then acquire the area, adding yourself to the active list.

- **Unlocking:** Remove yourself from the active list. Wake all intersecting proper privates and one sneaky smoker that is waiting for the same area (no need to wake multiple smokers).

If you used a boolean grid for synchronization, adding a second grid or an extra enumeration value for areas locked by sneaky smokers would help.

3.2 Reacting to BREAK! Properly

This is the main problem (at least in the implementation I've described so far) when moving from part 2 to part 3. Consider the following scenarios and see how they differ.

3.2.1 Scenario 1 (Part 2)

- P1, P2 and P3 exist as proper privates.
- P1 and P2 are currently gathering, while P3 is waiting for P1 to unlock its area, sleeping on a condvar.
- A BREAK! order is issued.
- P1 and P2 notice the order, and take a break. They unlock their areas before entering the break.
- P3 now wakes up to acquire the area it was waiting for since P1 unlocked it. It then notices the break order and also enters the break. Doesn't really matter if it locks/unlocks the area unnecessarily.

As you can see, in this scenario, there is no need for specific interaction between area locking and orders. Since privates entering a break will wake up other privates and they will then notice that an order has been issued.

3.2.2 Scenario 2 (Part 3)

- P1 is a proper private, S1 is a sneaky smoker.
- S1 is littering an area, and P1 is waiting for S1 to unlock the area.
- A BREAK! order is issued.
- S1 obviously does not react to the order since sneaky smokers do not take breaks. P1 remains in wait for the area to be unlocked and can not notice the order. The break order fails.

In Scenario 2 you can notice that decoupling the area locking and orders is no longer possible since proper privates waiting for areas from sneaky smokers will not wake up due to the area not being unlocked.

To solve this, both order lock variables and area checking data structures need to be tied to the same mutex. Privates waiting for an area will also check for the existence of orders, and the commander will wake all proper privates in the waiting list when giving an order, in addition to those waiting between cigarettes.

And that's it for part 3. Not much after part 2, but I wanted you to think about how you can combine the order and area locking mechanisms to deal with the problem I just described. Hopefully you did it, or you may have just gotten from Part 3 cases not containing orders. Sneaky!

4 FAQ

Some questions came up a lot during the homework, and I want to leave their answers here as possibly useful future reference:

4.1 How to use Condition Variables?

Not a question many asked, but by far the most important since I've seen them being misused a ton in student codes. So it's time for a small demo.

This is important, because as a student I also used to not understand condition variables, avoided them and hated them. But they are in fact pretty great and much more generic than semaphores! But, some care is necessary in their use to avoid missing events and signals.

A very simple scenario. Let's say in our thread we want to wait for a shared variable named `event` to be set to 1. Our condition variable associated with this will be `event_cv`.

4.1.1 Wrong Approach

A lot of code tries to handle it like this:

```
if (!event)
    pthread_cond_wait(&event_cv, &some_random_mutex);
```

While this seems nice and straightforward, it is riddled with synchronization issues. First of all, `event` is a shared variable and needs to be protected from race conditions by a mutex.

The mutex protecting `event` must be locked before checking to avoid races, and then `pthread_cond_wait` should be called with that specific mutex, since what `pthread_cond_wait` does is **release** that mutex.

Changes in `event` will be missed often with this code due to changes occurring between `if (!event)` and the wait call.

Even worse is waiting without checking:

```
pthread_cond_wait(&event_cv, &some_random_mutex);
```

The condition variable is just a waiting queue and is not magically tied to the condition you are waiting for by default. You handle that with checks and the extra mutex. This will cause problems *very often* since signals are lost on condition variables, unlike semaphores.

So: We lock the mutex to prevent races on the condition we are checking, then pass that mutex to the wait function so that it can unlock it, allowing other threads to change the condition. When

waking up, the given mutex will have been re-acquired. Which makes sense, because we need to make sure no one can change `event` now that we've been woken up believing it to be `1`.

Let's call the mutex protecting the event variable `event_mutex`. This leads us to the following code:

```
pthread_mutex_lock(&event_mutex);
if (!event)
    pthread_cond_wait(&event_cv, &event_mutex);
// event_mutex will be acquired on wake
```

There is still an issue here though! With Mesa semantics, the case for pthreads, there is no guarantee that our thread will wake up immediately after signaled. Such a scenario is possible:

- Thread 1 is sleeping on `event_cv`.
- Thread 2 locks, sets `event = 1` and signals `event_cv`, unlocks.
- Thread 3 is scheduled, locks, sets `event = 0` for some reason and unlocks.
- Finally, Thread 1 is scheduled again. It wakes up with the lock, and continues the code believing `event == 1`, even though it has been changed by another thread in between!

Because of the possibility of such scenarios, the check needs to be done in a `while` loop. Since the thread may not wake up as soon as it gets signaled, something else may happen in between changing the condition that caused it to wake up. This means the thread would need to sleep again. Such events are called *spurious wakeups*, because to the thread, it seems like it got woken up for no reason, due to `event` still being equal to 0. Even though it became 1 and only became 0 again after that.

4.1.2 Correct Approach

This finally leads us to the correct code:

```
pthread_mutex_lock(&event_mutex);
while (!event)
    pthread_cond_wait(&event_cv, &event_mutex);

// event_mutex will be acquired on wake
// do stuff

pthread_mutex_unlock(&event_mutex);
```

For the waker, this is simpler:

```
pthread_mutex_lock(&event_mutex);
event = 1;
pthread_mutex_unlock(&event_mutex);
pthread_cond_signal(&event_cv);
```

Signaling before or after unlocking is a bit of a debate but not very important. You can read this SO post for more details: `https://stackoverflow.com/questions/6419117/signal-and-unlock-order`

4.1.3 Another Wrong Approach

You may wonder why the mutex has to be passed to the wait function, instead of something like:

```
pthread_mutex_lock(&event_mutex);
while (!event) {
    pthread_mutex_unlock(&event_mutex);
    pthread_cond_wait(&event_cv);
    pthread_mutex_lock(&event_mutex);
}
// ...
```

This is because there is tight coupling between the mutex and the condition variable. If you release the mutex before waiting, `event` could change in between and you would miss it.

4.2 How to check for EOF?

The homework required you to be able to check for EOF in part 1 and part 2 inputs if you've implemented the next parts, since explicit numbers were not given for the number of orders and sneaky smokers.

This caused a lot more grief than I expected! Many students used overly complicated schemes, or just did not check it causing garbage counts for order and sneaky smoker numbers, even though the need to check for EOF was also the present in the given example test cases.

Here are some simple ways I know of, shown for a variable `num_orders` keeping the count of orders. You would use the same for the number of sneaky smokers.

4.2.1 C

Most `<stdio.h>` functions return EOF when they fail due to reaching the end of file. Also on other failures, but we can ignore that.

```
// Standard code
scanf(" %d", &num_orders);
// Code with check
if (scanf(" %d", &num_orders) == EOF)
    num_orders = 0;
```

4.2.2 C++

The simplest way is using the overloaded `!` operator for streams.

```
// Standard code
cin >> num_orders;
// Code with check
if (!(cin >> num_orders)) // Failed to extract
    num_orders = 0;
```

An alternative:

```
cin >> num_orders;
if (cin.fail())
    num_orders = 0;
```

A pitfall, which won't work since EOF hasn't been reached *before* you try to extract the next value:

```
if (!cin.eof()) // True, since the stream is not consumed yet
    cin >> num_orders; // Will consume the last newline and fail
// Now cin.eof() will be true. But not before cin >> num_orders!
```

4.3 How to wake up from the sleep between gatherings when an order is given?

See 2.2, the answer should not be signals!