CS 3013 - Operating Systems

Project 2 (90 points)
Assigned: Tuesday, September 6, 2022
Checkpoint: Tuesday, September 13, 2022 at 11:59:59pm
Due: Friday, September 16, 2022 at 11:59:59pm

Project 2: Synchronization and Concurrency

The following two problems will give students a chance to practice synchronization in C programming. The students must create threads for each of the actors and debug in a synchronous way. Each thread must use a degree of randomness in delays to test the scheduling. The random numbers should be seeded using `srand()` with the number in a `seed.txt` file (like in Project 1). This will allow students to change the seed value to test different scenarios while allowing one to create reproducible scenarios for debugging purposes.

The students must solve one of these synchronization problems with both mutexes and condition variables (e.g., `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_cond_wait`, and `pthread_cond_signal`) and the other with semaphores (e.g., `sem_wait`, `sem_post`, and `sem_init`). The students can decide which to use for each problem, but **students cannot use the same primitive type for both**. Both can be solved with either, but one way may be more straightforward than the other. All scheduling must be uncontrolled; students **may not** use a centralized coordinator that chooses which thread runs at any given time.

Problem 1: An OmniSportsPark of Dreams

After drawing a minor league baseball team to the city, Worcester realized that the field and stadium could be better used if it was used for multiple sports. With a level of innovation possible only within the WPI Innovation Studio, a team of intrepid WPI students designed a solution: the Worcester OmniSportsPark.

What is an OmniSportsPark, one might ask? It is a sports facility that can house multiple types of sports within a single facility. Baseball? Check. Football? Check. The Other Football? Them too. With a flip of 308 switches (and 5 levers), the park can transform to different types of competition fields. This would truly bring Worcester sports efficiency to a new level.

Well, except for one thing. It turns out that an OmniSportsPark is rather expensive to build. It was so expensive that Worcester had to forgo an event planner for the facility. Seeing this as an opportunity for a completely contrived Operating Systems project, the WPI Computer Science department set out to prove that sports participants would be able to autonomously manage the use of the facility in a distributed way.

Statement of Work

The WPI CS operating systems professors agreed to assign the problem of distributed sports team coordination to their CS 3013 students. Each project team must construct a simulator for the different sports and players to show that the facility can accomplish the goals. The following constraints were set for the simulator assignment:

1. **Sports Variety and Players:** The OmniSportsPark isn't quite "omni" ... it's more "mostly." The park needs to support three different types of sports: baseball, football, and soccer (which is sometimes called "footie" or "real football," depending on who one wishes to upset). The simulator shall have 36 baseball players (enough for four teams of 9 players), 44 football players (enough for four teams of 11 players), and 44 soccer players[1] (enough for four full teams of 11 players).
2. **Field Maxima and Minima:** To play baseball, the field must have exactly 18 baseball players (9 for each team). If there are not enough baseball players, none should take the field. Any baseball players

[1] Sockets? Soccerers? Sourcerors? There has to be a word for this.

in excess of 18 must wait until there is another opportunity to play. Likewise, football (the American variety) requires 22 players to be available (11 for each team). While soccer normally requires 22 players (again, 11 for each team), Worcester has agreed to allow team sizes to range from 1 to 11 on the field, so long as teams are evenly matched. Accordingly, soccer can be played with any even number of soccer players up to 22.

3. **Team Safety:** When the field is configured for a particular type of sport, it is considered unsafe for players of a different type of sport to take the field simultaneously. In other words, if the field is set for soccer, no football players or baseball players should be allowed on the field, even if they promise not to use their hands. The same is true for all the sports: only one type of player should use the field at a time.

4. **Maximum Parallelism:** If a pair of soccer players are using the field, any other pairs of soccer players may join the field until the 22 player limit is reached. Teams should try to maximize the number of soccer players unless meeting other constraints (such as fairness, described next). Football and baseball require an exact number of players (i.e., the minimum and maximum are the same), so there is no separate requirement to maximize the player count for those sports.

5. **Fair Performance Opportunities:** Every sports type should have an opportunity to perform without having to wait forever. The soccer players may not deprive the baseball players from playing simply by continuously having a new pair of soccer players join the field every time a pair leaves, thus monopolizing the field. Likewise, baseball players cannot prevent football players from taking the field by colluding to always favor the next batch of baseball players (and the same is true for all other combinations of teams). (In private, the OS programmer types call this the "starvation" constraint, but they were afraid to use that term around the players.)

6. **Game Time Variability and Bounds:** For baseball and football, the players must reach a consensus on the length of the game, but that game length should be selected random. Worcester scoffs at the notion of regulation football game quarter lengths and timeout limits. Meanwhile, baseball players in general seem to scoff at well-defined inning lengths. The Worcester soccer players are more free spirited: each pair of soccer players (again, one player on each team) will decide on a time length to play and can leave the field when they are done. When a pair departs, the game proceeds unless the field becomes empty. When a pair of players depart, another pair of players may take their place to maximize parallelism.

7. **Pre-game Napping:** If a player or (or set of players) is ready to go onto the field, they must either immediately enter the field (if permitted under the field limit and constraints #2 (Team Safety) and #3 (Fair Performance Opportunities)) or they must take a nap. Importantly, a player thread may not busy-wait, since that is directly correlated with player fretting and anxiety.

Students must implement code that creates the requisite number of baseball, football, and soccer players thread, with each thread representing a single player. Students must use either semaphores or the combination of mutexes and condition variables to properly synchronize the field. Each baseball, football, or soccer player thread should identify itself (by name or number and player type) and which field slot (e.g., from 1 to the field limit) it is using when it enters or leaves the field. Students should simulate sports activity on the field using the `sleep()` call with a random wait time for the team (or pair, in the case of soccer players). Students should use a random sleep time at the beginning of each thread to emulate players arriving at the OmniSportsPlex at different times. Students may choose to have player compete only twice (a double header) or to return to play continuously until the simulation is manually aborted (e.g., with a Control-C signal). Between each of these competitions, a player should only return to the ready state after waiting a random period of time after their last competition (simulating a rest period).

Students may wonder if they need to keep track of which players are on which teams. To reduce complexity, we do not require students to keep track of teams at all. If they wish to model teams anyway, students can simply the first half of the players on the field are associated with Team 1 and the second half are associated with Team 2 and that teams do not persist after the players leave the field.

Students should explain how their solution avoids depriving the different sports player types of the field in a text file, `problem1_explanation.txt`, that will be submitted along with the source code.

Problem 2: Taking Flight

The aviation industry is having a rough time these days and is in search of creative solutions. In the spirit of "less is more," one proposal calls for eliminating air traffic control for airport runways and instead leaving it to the individual pilots to share relevant information through a series of broadcasts about needed runway space (technically referred to as "calling dibs"). The reaction to this proposal has varied. One expert referred to it as "a dubious plan that can only lead to calamity." Another called it "a complicated exercise that only an operating systems professor could love." In this project, we aim to prove that latter expert correct.

To determine whether the proposal could be feasible, our class will be exploring a simulation of an airport airfield and determining how independent threads of execution, each representing airplanes, could collectively manage the runways. The simulation will include varying planes with different runway usage requirements. The airfield we will examine is shown in Figure 1.

To reduce the complexity of the simulation, we make a set of simplifying assumptions. First, planes can taxi to the runways without worrying about collisions (i.e., they can move about the blue lines in Figure 1 freely) and thus we can exclude those paths from our analysis. Second, we can break the airfield into six non-overlapping regions (and we do not have to worry about the little segments of runways between the regions). Third, planes will be of only two types: large or small. Fourth, no two planes may be in the same region at the same time (otherwise, we call that a "bad day"). And finally, planes travel in straight lines once entering the runway in immediate succession, without stopping or teleporting[2].

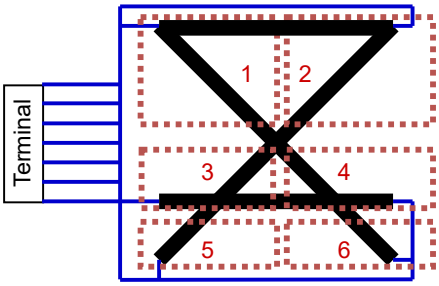

Figure 1: The Airport Runways and Regions.

A large plane must use a three-segment continuous straight runway (i.e., regions 1, 4, and 6 or alternatively regions 2, 3, and 5). A small plane can use any two-segment continuous straight runway (e.g., regions 1 and 2, regions 3 and 4, regions 1 and 4, regions 2 and 3, regions 3 and 5, and regions 4 and 6). Planes may enter runways from either end and travel in each direction (i.e., a large plane may proceed through regions 1, 4, and then 6 or in the opposite direction, from 6, 4, and then 1).

In this simulation, students should generate 15 threads for large planes and 30 threads for small planes. These threads will run continuously, cycling through a set of states until the simulation ends. Each thread will be in one (and only one) of the following states, which proceed in order: idle at the terminal, awaiting takeoff, taking off, flying, awaiting landing, and landing. Upon completing the landing, the plane will go to the terminal, thus entering the "idle at the terminal" state, where it can begin the next cycle. The states are described as follows:

1. **Idle at the Terminal:** This state involves a random sleep. The simulation begins with each plane waiting a random time at the terminal. This wait can be simulated with a random time parameter with the `usleep` function. Before sleeping, the thread should announce the intended sleep duration. After landing, a plane returns to the terminal to idle, again for a random time.

2. **Awaiting Takeoff:** The plane randomly chooses a set of runways for takeoff that is appropriate for the plane type (as described above). It then chooses an ordering in which it will use those runways. For example, for a large plane, it randomly choose to either use regions {1, 4, 6} or regions {2, 3, 5} as the set it will use. It will then randomly choose whether to use those regions in the order where the numbers increase (e.g., 1 then 4 then 6) or decrease (e.g., 6 then 4 then 1). The selection of the set of runways and their order must be done **without** awareness of other planes' goals or the current runway

[2]The Federal Aviation Administration was pretty insistent on the "no teleporting" simulator requirement. Apparently, teleportation is a competing industry that involves too many entanglements.

occupancy. Once the set of runways and their order is selected, the plane must print its intended runway region order to the console. The plane will taxi to the appropriate starting point, waiting on a blue line just outside the initial region. If the plane can proceed immediately, it should advance to the next state and print the console that it is doing so. Otherwise, it should print a note to the console indicating it must wait. It then should sleep on a semaphore or condition variable until it is able to proceed.

3. **Taking Off:** The plane will move to the first region in its list and announce its identity and position with a printout. It will sleep for a random amount of time in that region. Upon completing that sleep, it proceeds directly to the next region on the list without delaying or coordination. It again announces its identity and position with a printout and randomly sleeps and advances until it has left the final region on its list, at which point it advances to the next state. Note that these sleeps are representing the plane rolling down the runway at some velocity. The requirement to move to the next region without delay or coordination represents the fact that once a plane starts its takeoff, it cannot instantly stop and wait for a space to become clear. If a plane's movement into the next region would result in two planes being in the same region at the same time, it is considered a collision and a failure. This project must ensure this never happens, but students cannot pause a plane during takeoff to avoid it.

4. **Flying:** In this state, the plane is flying in the air and it is assumed to not be able to collide with any other planes. After all, our job is just to coordinate the runways, not the airspace. This state is simulated using a random sleep. Before sleeping, the thread should announce the intended sleep duration. When that sleep ends, the plane will proceed to the next state. Students can think of the plane flying in a circle since the plane will takeoff and land at the same airport. This lets us stress test the runways of the simulated airport without needing to simulate other airports.

5. **Awaiting Landing:** This state is similar to Awaiting Takeoff: the plane randomly select a set of compatible runways regions and the order it will traverse them. It prints that information to the console. If the plane is able to land, it will print a console note and proceed to the next state immediately. Otherwise, it should print a console note about the delay and sleep on a semaphore or condition variable until it is able to proceed.

6. **Landing:** This state is similar to Taking Off. The regions selected during the Awaiting Landing state are traversed in the selected order with a random sleep on each region before immediately traversing the next. As with takeoffs, the plane must announce its identity and runway position with each region traversed. Upon leaving the last region, the plane transitions to the Idle at the Terminal state.

When a plane enters each state, it must print to the console a little message about its situation. This status message must include its identity (e.g., "Plane Thread 4"), its type ("(Large)" or "(Small)"), its state, and any relevant details for that state as described earlier. Students can imagine that the planes are able to communicate with each other to call dibs on runways spots and indicate when they have cleared each spot. One can expect that the pilots of other planes will accurately track this information and respect it (e.g., the threads can share a globally array or diagram representing the runways and their use). One can envision this shared state as each plane having a co-pilot who records broadcast information about runway use on a whiteboard.

The simulator should advance planes as efficiently as possible by maximizing parallelism while avoiding deadlock or starvation. All planes should be guaranteed to be able to make forward progress (e.g., fairness should be ensured so planes cannot starve out other planes). It is not fair for a plane to be stuck in an Awaiting Takeoff or an Awaiting Landing state for a long time. If two planes can safely use different runway spots at the same time, they should do so. Planes cannot assume the speed at which other planes will take off or land (e.g., since runway use time is random, it is possible for one plane to rear end a plane in front of it even if they are using the same runways in the same order). Planes cannot pass each other by swapping positions (e.g., a plane going from region 2 to region 3 would collide with a plane going from region 3 to region 2). For simplicity, we can assume there is a bridge and tunnel setup at the junction of the X with the runways, so a plane moving from region 1 to region 4 can proceed at the same time as a plane moving from region 2 to region 3 without fear of collision.

When students have completed their solution, they should explain how the solution meets the requirements described above in a text file, `problem2_explanation.txt`, accompanying their code. They must specifically describe any scenarios that are guaranteed to deadlock and any rules that can ensure a deadlock is avoided. They must discuss any decreased parallelism that may result from any deadlock avoidance mechanisms they introduce and argue how they minimized that decrease.

Checkpoint Contributions

Students must submit work that demonstrates substantial progress towards completing the project on the checkpoint date. Substantial progress is judged at the discretion of the grader to allow students flexibility in prioritizing their efforts. However, as an example, any assignment in which one of the two synchronization problems is addressed will receive full credit towards the checkpoint. **Projects that fail to submit a checkpoint demonstrating significant progress will incur a 10% penalty during final project grading.**

Deliverables and Grading

When submitting the project, please include the following:

- All of the files containing the code for all parts of the assignment.
- The `problem1_explanation.txt` explanation for the OmniSportsPark problem.
- The `problem2_explanation.txt` explanation for the Taking Flight problem.
- The test files or input (e.g., specific random seed values) that the team used to convince themselves (and others) that the programs actually work.
- Output from the tests showing the system's functionality.
- A document called README.txt explaining the project, any defects, and anything that the team feels the grader should know when grading the project. Only plaintext write-ups are accepted.

Please compress all the files together as a single .zip archive for submission. As with all projects, please only submit standard zip files for compression; **.rar, .7z, and other custom file formats will not be accepted**.

The project programming is only a portion of the project. Students should use the following checklist in turning in their projects to avoid forgetting any deliverables:

1. Sign up for a project partner or have one assigned (URL: `https://cerebro.cs.wpi.edu/cs3013-shue/request_teammate.php`),
2. Submit the project code and documentation via InstructAssist (URL: `https://cerebro.cs.wpi.edu/cs3013-shue/files.php`), and
3. Complete a Partner Evaluation (URL: `https://cerebro.cs.wpi.edu/cs3013-shue/evals.php`).

A grading rubric has been provided at the end of this specification to give a guide for how the project will be graded. No points can be earned for a task that has a prerequisite unless that prerequisite is working well enough to support the dependent task. Students will receive a scanned markup of this rubric as part of their project grading feedback. Students are expected to contribute equally on their teams; unequal contributions may result in different scores for the students on a team.

Project 2 – Synchronization – Grading Sheet/Rubric

Evaluation?

Grader: _____ Student Name: _____ _____

Date/Time: _____ Student Name: _____ _____

Team ID: _____ Student Name: _____ _____

Late?: _____

Checkpoint?: _____ Project Score: / 90

Earned	Weight	Task ID	Description
_____	0	0	Primitives – Students must use a different primitive for Part 1 and Part 2. If the same primitive is used for both, students may choose which part will be graded. The other part is to be assigned a score of 0.
_____	15	1	Part 1 – Threads/processes implemented correctly with a high degree of parallelism. Prerequisite: Task 0.
_____	15	2	Part 1 – Correct mutual exclusion and prevention of deadlocks. Prerequisite: Task 1.
_____	15	3	Part 1 – Solution ensures fairness/starvation prevention and good explanation in problem1_explanation.txt. Prerequisite: Task 1.
_____	15	4	Part 2 – Threads/processes implemented correctly with a high degree of parallelism. Prerequisite: Task 0.
_____	15	5	Part 2 – Correct mutual exclusion and prevention of deadlocks. Prerequisite: Task 4.
_____	15	6	Part 2 – Solution ensures fairness/starvation prevention and good explanation in problem2_explanation.txt. Prerequisite: Task 5.

Grader Notes: