Computer Operating Systems

Practice Session 8: Example Synchronization Problems

T. Tolga Sarı (sarita@itu.edu.tr)
Sultan Çoğay (cogay@itu.edu.tr)
Doğukan Arslan (arslan.dogukan@itu.edu.tr)

April 20, 2022

Today

Problem

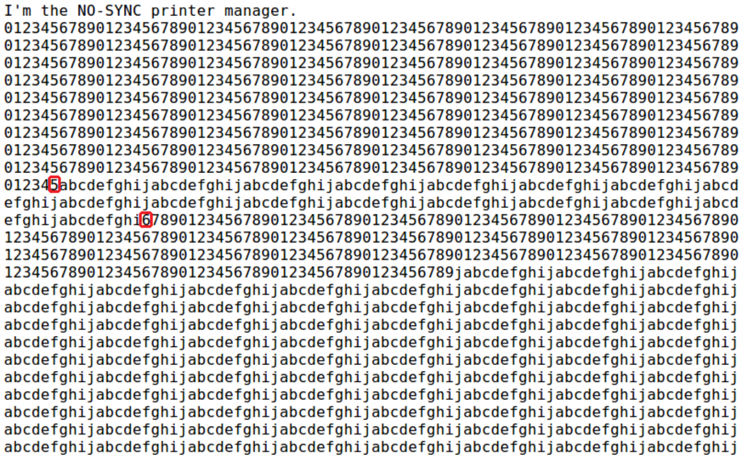
Purpose: Control a Printer System

- System has 1 printer
- Printer prints out (to the screen) a single character for each page it actually prints out
- Within a short period, jobs are given to this printer as to print out 100 copies (for each doc.) of two different documents
- These documents should be printed by protecting the integrity of each document

Without Synchronization

```c
#include <pthread.h>
#include <stdio.h>

void* printThis(void* typ){
  int i,j;
  char* str = (char)typ=='a'?"abcdefghij":"0123456789"; // 2 types of contents
  for(i=0; i<100; i++) // 100 separate print jobs
      for(j=0; j<10; j++) // of 10 pages each
        printf("%c", str[j]); // each character represents a page
  pthread_exit(NULL);
}

int main(void){
  printf("I'm the NO-SYNC printer manager.\n");
  setvbuf(stdout, (char*)NULL, _IONBF, 0); // no-buffer printf
  pthread_t a,n;   // create two threads (a thread for each set of documents)
  pthread_create(&a, NULL, printThis, (void *)'a');
  pthread_create(&n, NULL, printThis, (void *)'n');
  // wait for the threads to finish
  pthread_join(a, NULL);
  pthread_join(n, NULL);
  pthread_exit(NULL);
  return 0;
}
```

Example Output (Without Synchronization)



```
I'm the NO-SYNC printer manager.
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
012345abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcd
efghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcd
efghijabcdefghi67890123456789012345678901234567890123456789012345678901234567890
12345678901234567890123456789012345678901234567890123456789012345678901234567890
12345678901234567890123456789012345678901234567890123456789012345678901234567890
1234567890123456789012345678901234567890123456789jabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
abcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghijabcdefghij
```

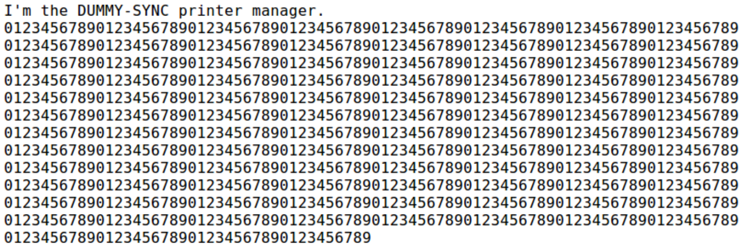
Synchronization Attempt

```
#include <pthread.h>
#include <stdio.h>

int s;

void* printThis(void* typ){
    int i,j;
    // create two types of contents for print jobs
    char* str = (char)typ=='a'?"abcdefghij":"0123456789";
    for(i=0; i<100; i++) // 100 separate print jobs
        if(s>0){
            s--; // lock other threads
            for(j=0;j<10;j++)
                printf("%c", str[j]);
            s++; // unlock other threads
        }
    pthread_exit(NULL);
}
```

Synchronization Attempt

```c
int main(void){
  printf("I'm the DUMMY-SYNC printer manager.\n");
  setvbuf(stdout, (char*)NULL, _IONBF, 0); // no-buffer printf
  s = 1; // printer is initially available
  pthread_t a,n;   // create two threads (a thread for each set of documents)
  pthread_create(&a, NULL, printThis, (void *)'a');
  pthread_create(&n, NULL, printThis, (void *)'n');
  // wait for the threads to finish
  pthread_join(a, NULL);
  pthread_join(n, NULL);
  pthread_exit(NULL);
  return 0;
}
```

Example Output (Synchronization Attempt)



```
I'm the DUMMY-SYNC printer manager.
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789
```

Problem: in `printThis()` function, no behavior (e.g., waiting) is defined for the case $s = 0$

Successful Synchronization

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>

int initKeys(char* argv[]) {
    char cwd[256];
    char* keyString;

    // get current working directory
    getcwd(cwd, 256);

    // form keystring
    keyString = malloc(strlen(cwd) + strlen(argv[0]) + 1);
    strcpy(keyString, cwd);
```

Successful Synchronization

```c

    int someKey = ftok(keyString, 1);

    //  deallocate keystring
    free(keyString);

    return someKey;

    // if you use get_current_dir_name, result of that call must be deallocated
     by caller
    // because it creates its own buffer to store path.
    // but in this example we use getcwd and our own buffer.
}

// to create a semaphore for mutual exclusion(value=1)
int s;

void create(char *argv[]){
    int someKey = initKeys(argv);
    s = semget(someKey, 1, 0700|IPC_CREAT);
    semctl(s, 0, SETVAL, 1); // semaphore value = 1
}

void sem_signal(int semid, int val){ // semaphore increment operation
```

Successful Synchronization

```
    semaphore.sem_num=0;
    semaphore.sem_op=val;
    semaphore.sem_flg=1;
    semop(semid, &semaphore, 1);
}

void sem_wait(int semid, int val){ // semaphore decrement operation
    struct sembuf semaphore;
    semaphore.sem_num=0;
    semaphore.sem_op=(-1*val);
    semaphore.sem_flg=1;
    semop(semid, &semaphore, 1);
}

void increase(int sid){ // to increase semaphore value by 1
  sem_signal(s,1);
}

void decrease(int sid){ // to decrease semaphore value by 1
  sem_wait(s,1);
}
```

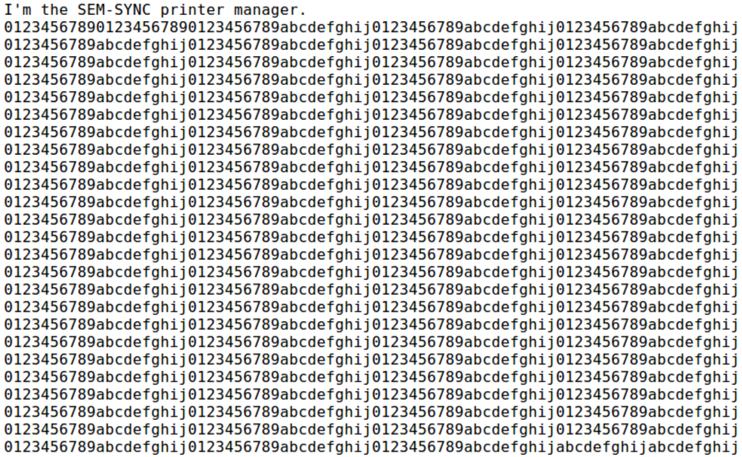
Successful Synchronization

```

void* printThis(void* typ){
    int i,j;
    char* str = (char)typ=='a'?"abcdefghij":"0123456789"; // 2 types of contents
    for(i=0; i<100; i++) {// 100 separate print jobs
        decrease(s); // lock other threads
        for(j=0;j<10;j++)
            printf("%c", str[j]);
        increase(s);// unlock other threads
    }
    pthread_exit(NULL);
}
int main(int argc, char *argv[]){
  printf("I'm the SEM-SYNC printer manager.\n");

  setvbuf(stdout, (char*)NULL, _IONBF, 0); // no-buffer printf

  create(argv);   // create a semaphore for mutual exclusion

  pthread_t a,n;   // create two threads (a thread for each set of documents)
  pthread_create(&a, NULL, printThis, (void *)'a');
  pthread_create(&n, NULL, printThis, (void *)'n');
```

Successful Synchronization

```
    pthread_create(&n, NULL, printThis, (void *)'n');

    pthread_join(a, NULL);
    pthread_join(n, NULL);

    semctl(s, 0, IPC_RMID, 0); // removing the created semaphore

    pthread_exit(NULL);
    return 0;
}
```

Example Output (Successful Synchronization)



```
I'm the SEM-SYNC printer manager.
01234567890123456789012345678 9abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij0123456789abcdefghij
0123456789abcdefghij0123456789abcdefghij0123456789abcdefghijabcdefghijabcdefghij
```

Water (H_2O) Production Problem

- We want to model the production of a water molecule using two threads for modelling oxygen (O) and hydrogen (H).
- A water molecule is assembled by using an O thread and 2 H threads (by using the `bond()` function)
- A thread can call `bond()` function for (i+1)th molecule if and only if 3 threads producing the ith molecule have already called `bond()` function and have completed their jobs.
  - An O thread waits for 2 H threads if they are not ready.
  - Similarly, an H thread waits for another H thread and an O thread.
- A `barrier` is used to make each thread wait until a complete water molecule is ready.

Prepare the synchronization steps for oxygen and hydrogen threads required for the described behavior above.

Water (H_2O) Production Problem

Required Variables and Initial Values:

- `mutex = Semaphore(1)` Mutex: MUTual EXclusion
- `oxygen = 0` Counter for oxygen atoms, protected by mutex
- `hydrogen = 0` Counter for hydrogen atoms, protected by mutex
- `barrier = Barrier(3)` allows for the next group to proceed after the group of 3 required threads are combined into a water molecule (by calling `bond()` function)
- `oxyQueue = Semaphore(0)` Semaphore on which oxygen atoms (threads) wait
- `hydroQueue = Semaphore(0)` Semaphore on which hydrogen atoms (threads) wait
- Initially `hydroQueue` & `oxyQueue` semaphores are locked (both are equal to zero)
- Basic functions related to queuing structures are named as follows:
  - `oxyQueue.wait()` : Enqueue an oxygen thread to oxygen queue
  - `hydroQueue.wait()` : Enqueue an hydrogen thread to hydrogen queue
  - `oxyQueue.signal()` : Dequeue an oxygen thread from oxygen queue
  - `hydroQueue.signal()` : Dequeue an hydrogen thread from hydrogen queue

Oxygen Thread Code Summary

An O thread that has arrived should allow 2 H threads to be processed by increasing `hydroQueue` semaphore by 2

- If there are at least two H threads waiting, it signals them and itself
- If not, it releases the mutex and waits

After bonding, threads wait at the `barrier` until all three threads have bonded, and then the oxygen thread releases the mutex.

```
mutex.wait()
oxygen += 1
if (hydrogen >= 2)
  hydroQueue.signal(2)
  hydrogen -= 2
  oxyQueue.signal()
  oxygen -=1
else
  mutex.signal()

oxyQueue.wait()
bond()
barrier.wait()
mutex.signal()
```

Hydrogen Thread Code Summary

H thread has a similar code:

```
mutex.wait()
hydrogen += 1
if (hydrogen >=2 and oxygen >=1)
  hydroQueue.signal(2)
  hydrogen -= 2
  oxyQueue.signal()
  oxygen -= 1
else
  mutex.signal()

hydroQueue.wait()
bond()
barrier.wait()
```

Note: At the end of each bonding, only the oxygen thread releases the mutex to guarantee that the mutex is signalled once (there is only one oxygen in each water molecule).

Sushi Bar Problem

- Imagine a sushi bar with 5 seats. If you arrive while there is an empty seat, you can take a seat immediately. But if you arrive when all 5 seats are full, that means that all of them are dining together, and you will have to wait for the entire party to leave before you sit down.
- Design a program for simulating customers entering and leaving this sushi bar.

Variables Used in the Solution

```
eating = waiting = 0 // keep track of the number of threads
mutex = Semaphore(1) // mutex protects both counters
block = Semaphore(0) // incoming customers' queue(regular meaning)
must_wait = False // indicates that the bar is full
```

Solution Attempt

```
mutex.wait()
if must_wait:
  waiting += 1
  mutex.signal()
  block.wait()
  mutex.wait()     // reacquire mutex
  waiting -= 1

eating += 1
must_wait = (eating == 5)
mutex.signal()

// eat sushi

mutex.wait()
eating -= 1
if eating == 0:
  n = min(5, waiting)
  block.signal(n)
  must_wait = False
mutex.signal()
```

Solution Attempt

```
 1 mutex.wait()
 2 if must_wait:
 3   waiting += 1
 4   mutex.signal()
 5   block.wait()
 6   mutex.wait()     // reacquire mutex
 7   waiting -= 1
 8
 9 eating += 1
10 must_wait = (eating == 5)
11 mutex.signal()
12
13 // eat sushi
14
15 mutex.wait()
16 eating -= 1
17 if eating == 0:
18   n = min(5, waiting)
19   block.signal(n)
20   must_wait = False
21 mutex.signal()
```

The problem is at Line 6. It is possible for newly arrived threads to take all the seats before the waiting threads.

A Solution

The reason a waiting customer reacquires mutex is to update eating/waiting state.

- Make the departing customer, who already has the mutex, do the updating.

```
mutex.wait()
if must_wait:
  waiting += 1
  mutex.signal()
  block.wait()
else:
  eating += 1
  must_wait = (eating == 5)
  mutex.signal()

// eat sushi

mutex.wait()
eating -= 1
if eating == 0:
  n = min(5, waiting)
  waiting -= n
  eating += n
  must_wait = (eating == 5)
  block.signal(n)
mutex.signal()
```

Another Solution

- If there are fewer than 5 customers at the bar and no one waiting, an entering customer just increments `eating` and releases the mutex. The fifth customer sets `must_wait`.
- If there are 5 customers at the bar, entering customers `block` until the last customer at the bar clears `must_wait` and signals `block`.
  - The signaling thread gives up the mutex and the waiting thread receives it.
  - This process continues, with each thread passing the mutex to the next until there are no more chairs or no more waiting threads.

Another Solution

```
mutex.wait()
if must_wait:
  waiting += 1
  mutex.signal()
  block.wait()    // when we resume, we have the mutex
  waiting -= 1

eating += 1
must_wait = (eating == 5)
if waiting and not must_wait:
  block.signal()     // and pass the mutex
else:
  mutex.signal()

// eat sushi

mutex.wait()
eating -= 1
if eating == 0: must_wait = False

if waiting and not must_wait:
  block.signal()     // and pass the mutex
else:
  mutex.signal()
```

Party Problem

For a party to be held in the dormitory, below constraints are defined:

- Any number of students can be in a room at the same time.
- Dorm manager can step into the room on below two conditions:
  - To search for the room if there is no student inside
  - To terminate the party if there are more than 50 students in the room
- When the manager is inside the room, no other student can come in but insider students can come out
- Manager leaves the room only when it becomes empty
- There exists only one manager

Implement a simulation holding for above constraints defined

Party Problem

Required Variables and Initial Values:

- `student = 0` Number of students in the room
- `manager = "notInRoom"` Holds the status of the manager (enum)
- `mutex = Semaphore(1)` For protecting student and manager status
- `gateLock = Semaphore(1)` Used for holding incoming students when the manager is inside
- `roomReady = Semaphore(0)` Manager is outside and the room is empty or there is a party with 50 or more students.
- `studentsOut = Semaphore(0)` Manager is inside and all students has left the room

Code Summary for the Manager

```
mutex.wait()
if student > 0 and student < 50:
  manager = 'waiting'
  mutex.signal()
  roomReady.wait()    // and get mutex from the student.

if student >= 50:  // student count should be >= 50 for terminating the party
  manager = 'inRoom'
  WarnStudentsToLeave()
  gateLock.wait()       // lock entrance
  mutex.signal()
  studentsOut.wait()
  gateLock.signal()     // remove the lock on entrance

else: // student count should be 0 for searching
  searchRoom()

manager = 'notInRoom'
mutex.signal()
```

- The manager waits until the room is empty or there are 50 or more students.
- The manager terminates the party if there are 50 or more students.
- The manager searches the room if the room is empty.

Code Summary for the Student

```
mutex.wait()
if manager == 'inRoom':
  mutex.signal()
  gateLock.wait()
  gateLock.signal()
  mutex.wait()
student += 1
if student == 50 and manager == 'waiting':
  roomReady.signal()     // pass roomReady to manager
else:
  mutex.signal()
party()
mutex.wait()
student -= 1
if student == 0 and manager == 'waiting':
  roomReady.signal()     // pass roomReady to manager
else if student == 0 and manager == 'inRoom':
  studentsOut.signal()     // pass studentsOut to manager
else:
  mutex.signal()
```

- If the manager is waiting, then the 50th student in or the last one out has to signal `roomReady`.
- If the manager is in the room, the last student out signals `studentsOut`.

Dining Savages Problem

A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary.

When a savage wants to eat, he helps himself from the pot, unless it is empty.

If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot.

Write code for the passengers and car that enforces these constraints.

Dining Savages Problem

The synchronization constraints are:

- Savages cannot invoke getServingFromPot if the pot is empty.
- The cook can invoke putServingsInPot only if the pot is empty.

Required Variables and Initial Values:

```
// Counter
servings = 0

// Mutex semaphore for counter
mutex = Semaphore (1)

// indicates that the pot is empty
emptyPot = Semaphore (0)

// indicates that the pot is full
fullPot = Semaphore (0)
```

Savage Thread

- As each savage passes through the mutex, he checks the pot.
- If it is empty, he signals the cook and waits.
- Otherwise, he decrements servings and gets a serving from the pot

```
mutex.wait()
if servings == 0:
    emptyPot.signal()
    fullPot.wait()
servings -= 1
getServingFromPot()
mutex.signal()

eat ()
```

Cook Thread

- When the cook runs putServingsInPot, we know that the savage that holds the mutex is waiting on fullPot. So the cook could access servings safely.

```
emptyPot.wait()
putServingsInPot(M)
servings = M
fullPot.signal()
```

Roller Coaster Problem

Suppose there are `n` passenger threads and a car thread. The passengers repeatedly wait to take rides in the car, which can hold `C` passengers, where `C < n`. The car can go around the tracks only when it is full.

- Passengers should invoke `board` and `unboard`.
- The car should invoke `load`, `run` and `unload`.
- Passengers cannot `board` until the car has invoked `load`
- The car cannot depart until `C` passengers have boarded.
- Passengers cannot `unboard` until the car has invoked `unload`.

Write code for the passengers and car that enforces these constraints.

Required Variables and Initial Values

```
// Counters
boarders = 0
unboarders = 0
// Mutex semaphores for counters
mutex = Semaphore(1)
mutex2 = Semaphore(1)
// Passenger queues
boardQueue = Semaphore(0)
unboardQueue = Semaphore(0)
// The status of the car
allAboard = Semaphore(0)    // full
allAshore = Semaphore(0)    // empty
```

Car Thread

Before each ride:

- The car signals `C` passengers to board, then waits for the last one to signal `allAboard`

After each ride:

- The car signals all `C` passengers to unboard, then waits the last one to signal `allAshore`

```
load()
boardQueue.signal(C)
allAboard.wait()

run()

unload()
unboardQueue.signal(C)
allAshore.wait()
```

Passenger Thread

Passengers wait for the car before boarding, naturally, and wait for the car to stop before leaving. The last passenger to board/unboard signals the car and resets the relevant passenger counter.

```
boardQueue.wait()
board()
mutex.wait()
  boarders += 1
  if boarders == C:
    allAboard.signal()
    boarders = 0
mutex.signal()

unboardQueue.wait()
unboard()
mutex2.wait()
  unboarders += 1
  if unboarders == C:
    allAshore.signal()
    unboarders = 0
mutex2.signal()
```

Multi-car Roller Coaster Problem

Some additional constraints need to be satisfied for the case of multiple cars:

- Only one car can be boarding at a time.
- Multiple cars can be on the track concurrently.
- Since cars can't pass each other, they have to unload in the same order they boarded.
- All the threads from one carload must disembark before any of the threads from subsequent carloads.

Modify the previous solution to handle these additional constraints. You can assume that there are `m` cars, and that each car has a local variable named `i` that contains an identifier between 0 and `m - 1`.

Required Variables and Initial Values

- Two lists of semaphores are defined to keep the cars in order:
  - `loadingArea` represents the loading area where passengers board a car
  - `unloadingArea` represents the unloading area where passengers unboard a car
- Each list contains one semaphore for each car.
- To enforce the ordering constraints among the cars, only one semaphore in each list is unlocked at any time.
  - Initially, only the semaphores for Car 0 are unlocked.
  - Before loading/unloading, each car waits on its own semaphore, and it signals the next car as it leaves.

```
loadingArea = [Semaphore(0) for i in range(m)]
loadingArea[1].signal()
unloadingArea = [Semaphore(0) for i in range(m)]
unloadingArea[1].signal()
```

The function `next` computes the identifier of the next car in the sequence (wrapping around from `m - 1` to 0):

```
def next(i):
  return (i + 1) % m
```

Modified Car Thread

No need to modify the passenger thread as using 2 mutex semaphores allows a passenger to board a car while another passenger is unboarding the next car.

```
loadingArea[i].wait()
load()
boardQueue.signal(C)
allAboard.wait()
loadingArea[next(i)].signal()

run()

unloadingArea[i].wait()
unload()
unboardQueue.signal(C)
allAshore.wait()
unloadingArea[next(i)].signal()
```

Santa Claus Problem

Santa Claus sleeps in his shop at the North Pole and can only be awakened by either

1. all nine reindeer being back from their vacation in the South Pacific,
2. or some of the elves having difficulty making toys.

To allow Santa to get some sleep, the elves can only wake him when three of them have problems. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return.

If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready. (It is assumed that the reindeer do not want to leave the tropics, and therefore they stay there until the last possible moment.)

Santa Claus Problem

- After the ninth reindeer arrives, Santa must invoke `prepareSleigh`, and then all nine reindeer must invoke `getHitched`.
- After the third elf arrives, Santa must invoke `helpElves`. Concurrently, all three elves should invoke `getHelp`.
- All three elves must invoke `getHelp` before any additional elves enter.

Santa should run in a loop so he can help many sets of elves. We can assume that there are exactly 9 reindeer, but there may be any number of elves. Write code for Santa, reindeer and elf threads that satisfies the given constraints.

Required Variables and Initial Values

```
// Counters
elves = 0
reindeer = 0
// Mutex semaphore for counters
mutex = Semaphore(1)
// To wake Santa up
santaSem = Semaphore(0)
// For reindeer to get hitched
reindeerSem = Semaphore(0)
// To prevent additional elves from entering
// while three elves are being helped
elfTex = Semaphore(1)
```

Santa Thread

- When Santa is woken up by a signal (santaSem), he checks which of the two conditions holds and either deals with the reindeer or the waiting elves.
- If there are nine reindeer waiting, Santa invokes prepareSleigh, then signals reindeerSem nine times, allowing the reindeer to invoke getHitched.
- If there are elves waiting, Santa just invokes helpElves.

```
santaSem.wait()
mutex.wait()
  if reindeer == 9:
    prepareSleigh()
    reindeerSem.signal(9)
  else if elves == 3:
    helpElves()
mutex.signal()
```

Reindeer Thread

- The ninth reindeer being back from vacation signals Santa and then joins the other reindeer waiting on `reindeerSem`.
- When Santa signals, the reindeer all execute `getHitched`.

```
mutex.wait()
  reindeer += 1
  if reindeer == 9:
    santaSem.signal()
mutex.signal()

reindeerSem.wait()
getHitched()
```

Elf Thread

- The last elf to enter holds `elfTex`, stopping other elves from entering until all three elves have invoked `getHelp`.
- The last elf to leave releases `elfTex`, allowing the next batch of elves to enter.

```
elfTex.wait()
mutex.wait()
  elves += 1
  if elves == 3:
    santaSem.signal()
  else
    elfTex.signal()
mutex.signal()

getHelp()

mutex.wait()
  elves -= 1
  if elves == 0:
    elfTex.signal()
mutex.signal()
```

References

- Downey, A. B. (2008). The little book of semaphores. Version 2.1.5. Green Tea Press.