6.004 Worksheet Questions
L21 – Synchronization

Semaphores (Dijkstra, 1962)

Programming construct for synchronization:

- New data type: *semaphore*, an integer ≥ 0

```
semaphore s = K;  // initialize s to K
```

- New operations (defined on semaphores):
  - `wait(semaphore s)`
    wait until s > 0, then s = s – 1
  - `signal(semaphore s)`
    s = s + 1 (one waiting thread may now be able to proceed)
- Semantic guarantee: A semaphore s initialized to K enforces the precedence constraint:

$$\text{signal}(s)_i \prec \text{wait}(s)_{i+K}$$

The i^{th} call to signal(s) must complete
before the $(i+K)^{th}$ call to wait(s) completes

Semaphores for Precedence

```
semaphore s = 0;

Thread A        Thread B

A1;             B1;

A2;             B2;
signal(s);
A3;             B3;
                wait(s);
A4;             B4;

A5;             B5;
```

Goal: Want statement A2 in thread A to complete before statement B4 in thread B begins.

$$A2 \prec B4$$

Recipe:

- Declare semaphore = 0
- `signal(s)` at start of arrow
- `wait(s)` at end of arrow

Semaphores for Resource Allocation

Abstract problem:

- Pool of K resources
- Many threads, each needs resource for occasional uninterrupted period
- Must guarantee that at most K resources are in use at any time

Solution using semaphores:

In shared memory:

```
semaphore s = K;  // K resources
```

Using resources:

```
wait(s);     // Allocate a resource
…            // use it for a while
signal(s);   // return it to pool
```

Invariant: Semaphore value = number of resources left in pool

Semaphores for Mutual Exclusion

```
semaphore lock = 1;

void debit(int account, int amount) {
   wait(lock);   // Wait for exclusive access
   t = balance[account];
   balance[account] = t – amount;
   signal(lock); // Finished with lock
}
```

a <> b

"a *precedes* b
or
b *precedes* a"
(i.e., they don't overlap)

Lock controls access to critical section

Issue: Lock granularity

One lock for all accounts?
One lock per account?
One lock for all accounts ending in 004?

Synchronization: The Dark Side

The naïve use of synchronization constraints can introduce its own set of problems, particularly when a thread requires access to more than one protected resource.

```
void transfer(int account1, int account2, int amount) {
   wait(lock[account1]);
   wait(lock[account2]);
   balance[account1] = balance[account1] - amount;
   balance[account2] = balance[account2] + amount;
   signal(lock[account2]);
   signal(lock[account1]);
}
```

transfer(6031, 6004, 50)

transfer(6004, 6031, 50)

What can go wrong here?

```
Thread 1: wait(lock[6031]);
Thread 2: wait(lock[6004]);
Thread 1: wait(lock[6004]); // cannot complete
                            // until thread 2 signals
Thread 2: wait(lock[6031]); // cannot complete
                            // until thread 1 signals
```

No thread can make progress → Deadlock

Example: Dealing With Deadlocks

Can you fix the transfer method to avoid deadlock?

```
void transfer(int account1, int account2, int amount) {
  int a = min(account1, account2);
  int b = max(account1, account2);
  wait(lock[a]);
  wait(lock[b]);
  balance[account1] = balance[account1] - amount;
  balance[account2] = balance[account2] + amount;
  signal(lock[b]);
  signal(lock[a]);
}
```

Transfer(6031, 6004, 50)

Transfer(6004, 6031, 50)

Problem 1. ★

P1 and P2 are processes that run concurrently. P1 has two sections of code where section A is followed by section B. Similarly, P2 has two sections: C followed by D. Within each process execution proceeds sequentially, so we are guaranteed that A $\preceq$ B, i.e., A precedes B. Similarly, we know that C $\preceq$ D. There is no looping; each process runs exactly once. You will be asked to add semaphores to the programs – you may need to use more than one semaphore. Please give the initial values of any semaphores you use. For full credit use a minimum number of semaphores and don't introduce any unnecessary precedence constraints.

(A) Please add WAIT(…) and SIGNAL(…) statements as needed in the spaces below so that the precedence constraint B $\preceq$ C is satisfied, i.e., execution of P1 finishes before execution of P2 begins.

Add WAIT and SIGNAL statements so that B $\preceq$ C

Semaphore initial values: _____

Process P1	Process P2
…Section A code…	… Section C code…
…Section B code…	…Section D code…

(B) Please add WAIT(…) and SIGNAL(…) statements as needed in the spaces below so that D ≤ A *or* B ≤ C, i.e., executions of P1 and P2 cannot overlap, but are allowed to occur in either order.

Add WAIT and SIGNAL statements so that D ≤ A *or* B ≤ C

Semaphore initial values: _____

Process P1	Process P2
…Section A code…	… Section C code…
…Section B code…	…Section D code…

(C) Please add WAIT(…) and SIGNAL(…) statements as needed in the spaces below so that A ≤ D *and* C ≤ B, i.e., the first section (A and C) of **both** processes completes execution before the second section (B or D) of **either** process begins execution.

Add WAIT and SIGNAL statements so that A ≤ D *and* C ≤ B

Semaphore initial values: _____

Process P1	Process P2
…Section A code…	… Section C code…
…Section B code…	…Section D code…

Problem 2 ★

The following three processes are run on a shared processor. They can coordinate their execution via shared semaphores that respond to the standard signal(S) and wait(S) procedures. Their intent is to print the word HELLO. Assume that execution may switch between any of the three processes at any point in time.

```
Process 1                    Process 2                    Process 3
Loop1: print("H")            Loop2: print("L")            Loop3: print("O")
       print("E")            goto Loop2                   goto Loop3
       goto Loop1
```

(A) Assuming that no semaphores are being used, for each of the following sequences of characters, specify whether or not this system could produce that output.

LEHO (YES/NO): _____ **HLOE (YES/NO): _____** **LOL (YES/NO): _____**

(B) You would like to ensure that only the sequence HELLO can be printed and that it will be printed exactly once. Add any missing wait(S) and signal(S) calls to the code below (where S is one of a, b or c) to ensure that the three processes can only print HELLO exactly once. Remember to specify the **initial value** for each of your semaphores. *Recall that semaphores cannot be initialized to negative numbers.*

```
Semaphores: a = ___; b = ___; c = ___;

Process 1          Process 2          Process 3

Loop1:             Loop2:             Loop3:

    wait(a)            wait(b)            wait(c)

    print("H")         print("L")         print("O")

    print("E")

    signal(b)

    goto Loop1         goto Loop2         goto Loop3
```

Problem 3 ★

The following pair of processes share the variable `counter`, which has been given an initial value of 10 before execution of either process begins and is stored at address 0x100:

```
Process A             Process B
…       …
A1: lw x1, 0x100(x0)  B1: lw x1, 0x100(x0)
    addi x1, 1, x1        addi x1, 2, x1
A2: sw x1, 0x100(x0)  B2: sw x1, 0x100(x0)
…       …
```

(A) If Processes A and B are run on a timesharing system, there are six possible orders in which the LD and ST instructions might be executed. For each of the orderings, please give the final value of the counter variable.

A1 A2 B1 B2: counter = _____ **B1 A1 B2 A2: counter = _____**

A1 B1 A2 B2: counter = _____ **B1 A1 A2 B2: counter = _____**

A1 B1 B2 A2: counter = _____ **B1 B2 A1 A2: counter = _____**

In the following two questions you are asked to modify the original programs for processes A and B by adding the minimum number of semaphores and signal and wait operations to guarantee that the final result of executing the two processes will be a specific value for counter. Give the initial values for every semaphore you introduce. For full credit, your solution should allow *all* execution orders that result in the required value.

(B) Add semaphores (with initial values) so that the final value of counter is 12.

```
Semaphores: _____

Process A             Process B
…       …
A1: lw x1, 0x100(x0)  B1: lw x1, 0x100(x0)

    addi x1, 1, x1        addi x1, 2, x1

A2: sw x1, 0x100(x0)  B2: sw x1, 0x100(x0)
…       …
```

(C) Add semaphores (with initial values), so that the final value of counter is **not** 13.

```
Semaphores: _____________________________

Process A             Process B
…       …
A1: lw x1, 0x100(x0)  B1: lw x1, 0x100(x0)

    addi x1, 1, x1        addi x1, 2, x1

A2: sw x1, 0x100(x0)  B2: sw x1, 0x100(x0)
…       …
```

Problem 4 ★

Schro Dinger has a company that produces pairs of entangled particles, which are then packaged and sent to manufacturers of quantum computers. Since it's a complicated process, there are multiple machines that produce particle pairs; each machine runs the Producer code shown below.

The completed particle pairs are placed in the particle buffer, where they take up 2 of the buffer locations. There's a single packaging machine that takes a particle pair from the particle buffer and prepares it for shipment; the packing machine runs the Consumer code shown below.

To prevent any violations of the boundary conditions the following rules must be followed:

1. A production machine can only place a particle pair in the buffer if there are two spaces available.
2. The particle pair must be stored in consecutive buffer locations, i.e., a particle from some other production machine can't appear between the particles that make up the pair.
3. The capacity of the buffer (100 particles, or 50 particle pairs) can't be exceeded.
4. The packaging machine breaks if it accesses the buffer and finds it empty – it should only proceed when there are at least two particles in the buffer.

Schro has heard of semaphores but is unsure how to use them to ensure the rules are followed.

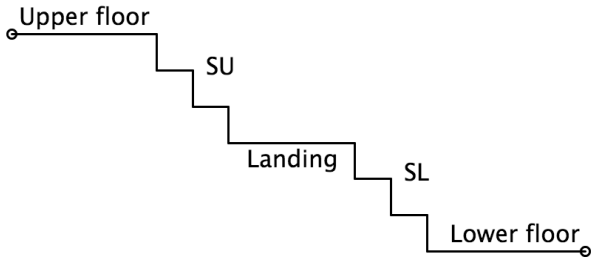
- Please insert the appropriate semaphores, WAITs, and SIGNALs into the Producer and Consumer code to ensure correct operation and to prevent deadlock.
- Be sure to indicate initial values for any semaphores you use.
- Remember: **there are multiple producers and a single consumer**!
- For full credit, use a minimum number of semaphores and don't introduce unnecessary precedence constraints.

```
                    Shared Memory
          particle buffer[100];  // holds 100 particles

      Semaphores and initial values: ____________________________
```

```
          Producer
PLoop:

   Produce pair P1, P2

   Place P1 in buffer

   Place P2 in buffer

   Go to PLoop
```

```
          Consumer
CLoop:

   Fetch P1 from buffer

   Fetch P2 from buffer

   Package and ship

   Go to CLoop
```

Problem 5



The MIT Safety Office is worried about congestion on stairs and has decided to implement a semaphore-based traffic-control system. Most connections between floors have two flights of stairs with an intermediate landing (see figure). The constraints the Safety Office wishes to enforce are

- Only 1 person at a time on each flight of stairs
- A maximum of 3 persons on a landing
- As a few traffic constraints as possible
- No deadlock (a particular concern if there's bidirectional travel)

Assume stair traffic is unidirectional: once on a flight of stairs, people continue up or down until they've reached their destination floor (no backing up!), although they may pause at the landing.

There are three semaphores: they control the upper flight of stairs (SU), the landing (L), and the lower flight of stairs (SL). Please provide appropriate initial values for these semaphores and add the necessary wait() and signal() calls to the Down() and Up() procedures below. Note that the Down() and Up() routines will be executed by many students simultaneously and the semaphores are the only way their code has of interacting with other instances of the Down() and Up() routines. To get full credit your code must avoid deadlock and enforce the stair and landing occupancy constraints. **Hint**: for half credit, implement a solution where only 1 person at time is in-between floors (but be careful of deadlock here too!).

```
// Semaphores shared by all students, provide initial values

semaphore SU = ________, SL = __________, L = __________;
```

```
// code for going downstairs
Down() {

    Enter SU;

    Exit SU/enter landing;

    Exit landing/enter SL;

    Exit SL;

}
```

```
// code for going upstairs
Up() {

    Enter SL;

    Exit SL/enter landing;

    Exit landing/enter SU;

    Exit SU;

}
```

Problem 6

(A) Semaphore S is used to implement mutual exclusion on accesses to a shared buffer. No other semaphores are used. What should its initial value be?

Initial value for S: _____

(B) Indicate whether each of the following sets of semaphore-synchronized processes can deadlock. The last two cases are variants of the first one; differences are *underlined*.

Circle answers below

```
Initial semaphore values: s1 = 1, s2 = 1, s3 = 1
P1:               P2:               P3:
wait(s1);         wait(s2);         wait(s1);
wait(s2);         wait(s3);         wait(s2);
print("1");       print("2");       wait(s3);
signal(s2);       signal(s3);       print("3");
signal(s1);       signal(s2);       signal(s3);
                                    signal(s2);
                                    signal(s1);
```

Can it deadlock?

YES NO Can't tell

```
Initial semaphore values: s1 = 1, s2 = 1, s3 = 1
P1:               P2:               P3:
wait(s1);         wait(s2);         wait(s2);
wait(s2);         wait(s3);         wait(s3);
print("1");       print("2");       wait(s1);
signal(s2);       signal(s3);       print("3");
signal(s1);       signal(s2);       signal(s1);
                                    signal(s3);
                                    signal(s2);
```

Can it deadlock?

YES NO Can't tell

```
Initial semaphore values: s1 = 2, s2 = 1, s3 = 1
P1:               P2:               P3:
wait(s1);         wait(s2);         wait(s2);
wait(s2);         wait(s3);         wait(s3);
print("1");       print("2");       wait(s1);
signal(s2);       signal(s3);       print("3");
signal(s1);       signal(s2);       signal(s1);
                                    signal(s3);
                                    signal(s2);
```

Can it deadlock?

YES NO Can't tell

Problem 7 (From Past Quiz)

MIT has been developing its own online grading system, LearnScope, which it hopes will replace how people grade homework and exams both in MIT and beyond.

LearnScope prides itself on parallelized grading by starting multiple threads running the `gradeExams` function described below in pseudocode.

Shared Memory:

```
// exams is an array containing the exams
exams = [exam0, exam1, exam2, ... , exam99];
next_exam = 0;
```

```
gradeExams:

    // Get ungraded exam
    exam = exams[next_exam]
    next_exam = next_exam + 1

    // Grade exam
    grade(exam)

    goto gradeExams
```

(A) Suppose two threads, A and B, are running the `gradeExams` code above without any synchronization. For each of the following failure scenarios, circle whether it is possible to happen or not:

1. A and B start grading the same exam:

   Possible / **Not Possible**

2. A gets exam k+1 before B gets exam k

   Possible / Not Possible

3. An exam is skipped and is never graded by either A or B:

   Possible / Not Possible

Despite LearnScope's advanced parallel grading technology, it cannot interface with 6.004's gradebook module. Instead, Silvina manually enters the grades into the gradebook as LearnScope completes its grading. When the gradeExams function finishes grading an exam, it emails a score report for the student to Silvina.

She will read a score report email, enter the grade into the gradebook, and repeat, as represented by the enterGrades function below. Silvina doesn't like having more than 10 unread score report emails at a time and also does not like checking her email when she doesn't have any new messages. Assume that there is more than 1 and fewer than 10 threads running the gradeExams function.

(B) **Add semaphores below to enforce these constraints:**

1. No two LearnScope threads should ever grade the same exam
2. Silvina should never have more than 10 unread LearnScope score report emails
3. readScoreReportEmail() should not be called until there is an unread score report email from LearnScope
4. After all 100 exams are claimed by LearnScope threads, LearnScope should stop attempting to grade exams
5. As long as there are still unclaimed exams, avoid deadlock
6. Use as few semaphores as possible, and do not add any additional precedence constraints

Shared Memory:

```
exams = [exam0, exam1, exam2, ... , exam99];
next_exam = 0;

// Specify your semaphores and initial values here
```

gradeExams:

```
    // Get ungraded exam
    exam = exams[next_exam]

    next_exam = next_exam + 1

    // Grade exam
    grade(exam)

    emailScoreReport(exam)

    goto gradeExams
```

enterGrades:

```
    readScoreReportEmail()

    enterGrade()

    goto enterGrades
```