6.004 Worksheet
L20 – Synchronization

Semaphores (Dijkstra)

Programming construct for synchronization:

- NEW DATA TYPE: semaphore, an integer ≥ 0
  \[ \text{semaphore } s = k; \] // initialize to K
- NEW OPERATIONS (defined on semaphores):
  \[ \text{wait(semaphore } s) \]
  - wait until \( s > 0 \), then \( s = s - 1 \)
  \[ \text{signal(semaphore } s) \]
  - \( s = s + 1 \) (one waiting process may now be able to proceed)
- SEMANTIC GUARANTEE: A semaphore \( s \) initialized to \( K \)
  enforces the constraint:

\[ \text{signal}(s) \leq \text{wait}(s), K \]

Often you will see
\( P \) used for \( \text{wait}(s) \)
and
\( V \) used for \( \text{signal}(s) \)
\( P \) = “pokker” meaning
\( V \) = “verhogen” (increase)

Semaphores for Precedence

Goal: want statement A2 in process A to complete
before statement B4 in Process B begins.

\[ \text{Process } A \]
\[ A1; \]
\[ A2; \]
\[ \text{signal(s);} \]
\[ A3; \]
\[ A4; \]
\[ A5; \]

\[ \text{Process } B \]
\[ B1; \]
\[ B2; \]
\[ \text{wait(s);} \]
\[ B3; \]
\[ B4; \]
\[ B5; \]

Recipe:
- Declare semaphore = 0
- \text{signal(s)} at start of arrow
- \text{wait(s)} at end of arrow

Semaphores for Resource Allocation

Abstract problem:
- POOL of K resources
- Many processes, each needs resource for occasional
  uninterrupted period
- MUST guarantee that at most K resources are in use at any time.

Semaphore Solution:

In shared memory:
\[ \text{semaphore } s = k; \] // K resources

Using resources:
\[ \text{wait}(s); \] // Allocate a resource
\[ \text{signal}(s); \] // return it to pool

Invariant: Semaphore value = number of resources left in pool

Dealing With Deadlocks

Cooperating processes:
- Establish a fixed ordering to shared resources and require
  all requests to be made in the prescribed order

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[lock[lock[lock[a]]]]);
  signal(lock[lock[lock[b]]]);
}

Unconstrained processes:
- O/S discovers circular wait & kills waiting process
- Transaction model
- Hard problem

Summary

Communication among parallel threads or asynchronous processes requires synchronization....
- Precedence constraints: a partial ordering among operations
- Semaphores as a mechanism for enforcing precedence constraints
- Mutual exclusion (critical sections, atomic transactions) as a
  common compound precedence constraint
- Solving Mutual Exclusion via binary semaphores
- Synchronization serializes operations, limits parallel execution

Many alternative synchronization mechanisms exist!

Deadlock:
- Consequence of undisciplined use of synchronization mechanism
- Can be avoided in special cases, detected and corrected in others
Problem 1.

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

\[
\text{particle buffer}[100]; \quad // \text{holds 100 particles}
\]

Semaphores and initial values: \( P=0, S=50, M=1 \)

<table>
<thead>
<tr>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLoop:</td>
<td>CLoop:</td>
</tr>
<tr>
<td>Produce pair P1, P2</td>
<td><strong>WAIT(P)</strong></td>
</tr>
<tr>
<td><strong>WAIT(S); WAIT(M)</strong></td>
<td>Fetch P1 from buffer</td>
</tr>
<tr>
<td>Place P1 in buffer</td>
<td>Fetch P2 from buffer</td>
</tr>
<tr>
<td>Place P2 in buffer</td>
<td><strong>SIGNAL(S)</strong></td>
</tr>
<tr>
<td><strong>SIGNAL(M); SIGNAL(P)</strong></td>
<td>Package and ship</td>
</tr>
<tr>
<td>Go to PLoop</td>
<td>Go to CLoop</td>
</tr>
</tbody>
</table>

\( P = \# \text{ of particle pairs in buffer, enforces rule 4} \)
\( S = \# \text{ of pair spaces in buffer, enforces rules 1 and 3} \)
\( M = \text{mutual exclusion lock, enforces rule 2 (when there are multiple producers)} \)
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.

(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

**Loop 1:**
- `wait(a)`
- `print("H")`
- `print("E")`
- `signal(b)`
- `goto Loop1`

### Process 2

**Loop 2:**
- `wait(b)`
- `print("L")`
- `signal(c)`
- `goto Loop2`

### Process 3

**Loop 3:**
- `wait(c)`
- `print("O")`
- `signal(c)`
- `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:

Process A | Process B
---|---
... | ...
A1: LD(counter, R0) | B1: LD(counter, R0)
ADDC(R0, 1, R0) | ADDC(R0, 2, R0)
A2: ST(R0, counter) | B2: ST(R0, counter)
... | ...

(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.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
</tr>
</tbody>
</table>

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:** X=0, Y=0

Process A | Process B
---|---
... | ...
A1: LD(counter, R0) | B1: LD(counter, R0)
ADDC(R0, 1, R0) | ADDC(R0, 2, R0)
wait(x) | signal(x); wait(y)
A2: ST(R0, counter) | B2: ST(R0, counter)
signal(y) | ...

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

**Semaphores:** X=0, Y=0

Process A | Process B
---|---
... | ...
A1: LD(counter, R0) | B1: LD(counter, R0)
signal(x) | signal(y)
ADDC(R0, 1, R0) | ADDC(R0, 2, R0)
wait(y) | wait(x)
A2: ST(R0, counter) | B2: ST(R0, counter)
... | ...
Problem 4.

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 ≤ B, i.e., A precedes B. Similarly, we know that C ≤ 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 ≤ C is satisfied, i.e., execution of P1 finishes before execution of P2 begins.

Add WAIT and SIGNAL statements so that B ≤ C

Semaphore initial values: S=0

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...Section A code…</td>
<td>wait(S) ... Section C code…</td>
</tr>
<tr>
<td>...Section B code…</td>
<td>...Section D code…</td>
</tr>
<tr>
<td>signal(S)</td>
<td></td>
</tr>
</tbody>
</table>

(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: M=1

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait(M) ...Section A code…</td>
<td>wait(M) ... Section C code…</td>
</tr>
<tr>
<td>...Section B code…</td>
<td>...Section D code…</td>
</tr>
<tr>
<td>signal(M)</td>
<td>signal(M)</td>
</tr>
</tbody>
</table>
Please add \texttt{WAIT(...)} and \texttt{SIGNAL(...)} statements as needed in the spaces below so that $A \leq D$ and $C \leq B$, i.e., the first section (A and C) of \textbf{both} processes completes execution before the second section (B or D) of \textbf{either} process begins execution.

\textbf{Add WAIT and SIGNAL statements so that $A \leq D$ and $C \leq B$}

Semaphore initial values: $S=0$, $T=0$

\begin{tabular}{|l|l|}
\hline
\textbf{Process P1} & \textbf{Process P2} \\
\hline
...Section A code... & ...Section C code...
\texttt{signal}(S) & \texttt{signal}(T)
\texttt{wait}(T) & \texttt{wait}(S)
...Section B code... & ...Section D code...
\hline
\end{tabular}
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!).

```c
// Semaphores shared by all students, provide initial values
semaphore SU = __________, SL = __________, L = __________;

// code for going downstairs
Down() {
    wait(L);
    wait(SU);
    Enter SU;

    Exit SU/enter landing;
    signal(SU);
    wait(SL);
    Exit landing/enter SL;
    signal(L);
    Exit SL;
    signal(SL)
}

// code for going upstairs
Up() {
    wait(L);
    wait(SL);
    Enter SL;

    Exit SL/enter landing;
    signal(SL);
    wait(SU);
    Exit landing/enter SU;
    signal(L);
    Exit SU;
    signal(SU)
}
```

1 2 3
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?

A value of $1$ for $S$ allows at most one WAIT to
succeed – others will stall until first one SIGNALs. 

Initial value for $S$: 1

This implements mutual exclusion.
(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*.

| Initial semaphore values: $s_1 = 1$, $s_2 = 1$, $s_3 = 1$ |
|---|---|---|
| P1: | P2: | P3: |
| wait($s_1$); | wait($s_2$); | wait($s_1$); |
| wait($s_2$); | wait($s_3$); | wait($s_2$); |
| print(“1”); | print(“2”); | wait($s_3$); |
| signal($s_2$); | signal($s_3$); | signal($s_3$); |
| signal($s_1$); | signal($s_2$); | signal($s_2$);

Uses a global ordering for semaphores ($S_1 > S_2 > S_3$). No deadlock possible.

Can it deadlock?  
YES  NO  Can’t tell

Deadlock scenario: 
P1 holds S1, waiting on S2  
P3 holds S2, waiting on S1

Can it deadlock?  
YES  NO  Can’t tell

| Initial semaphore values: $s_1 = 2$, $s_2 = 1$, $s_3 = 1$ |
|---|---|---|
| P1: | P2: | P3: |
| wait($s_1$); | wait($s_2$); | wait($s_2$); |
| wait($s_2$); | wait($s_3$); | wait($s_3$); |
| print(“1”); | print(“2”); | wait($s_1$); |
| signal($s_2$); | signal($s_3$); | signal($s_1$); |
| signal($s_1$); | signal($s_2$); | signal($s_3$); |

Now both WAIT(S1) statements will succeed.

Can it deadlock?  
YES  NO  Can’t tell