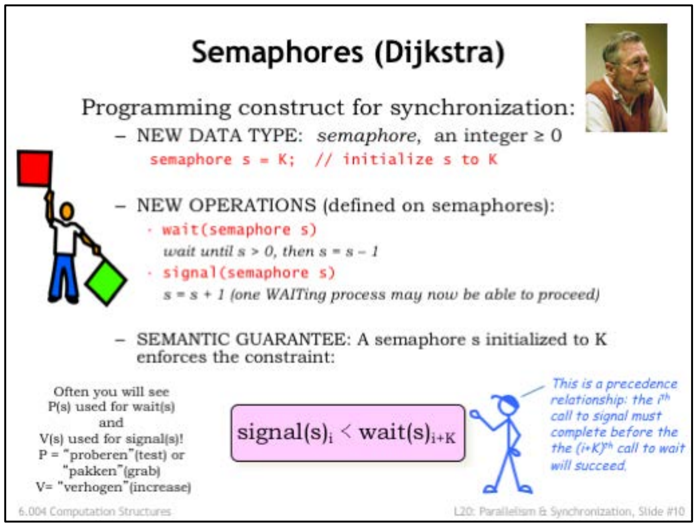
Computation Structures

Synchronization Worksheet



Semaphores (Dijkstra)

Programming construct for synchronization:

- NEW DATA TYPE: *semaphore*, an integer ≥ 0

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

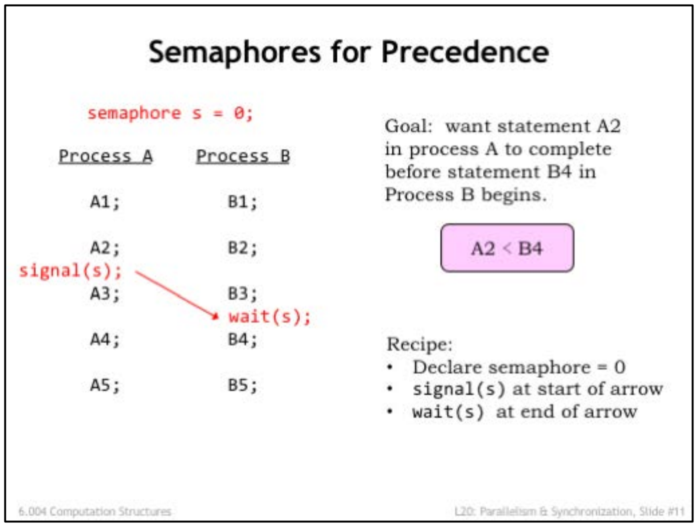
- NEW OPERATIONS (defined on semaphores):
  - `wait(semaphore s)`
    wait until s > 0, then s = s – 1
  - `signal(semaphore s)`
    s = s + 1 (one WAITing process may now be able to proceed)
- SEMANTIC GUARANTEE: A semaphore s initialized to K enforces the constraint:

$$\boxed{\mathrm{signal(s)}_i < \mathrm{wait(s)}_{i+K}}$$

Often you will see P(s) used for wait(s) and V(s) used for signal(s)! P = "proberen" (test) or "pakken" (grab) V= "verhogen" (increase)

This is a precedence relationship: the i^{th} call to signal must complete before the the $(i+K)^{th}$ call to wait will succeed.

6.004 Computation Structures L20: Parallelism & Synchronization, Slide #10



Semaphores for Precedence

```
semaphore s = 0;

Process A      Process B

  A1;            B1;

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

  A5;            B5;
```

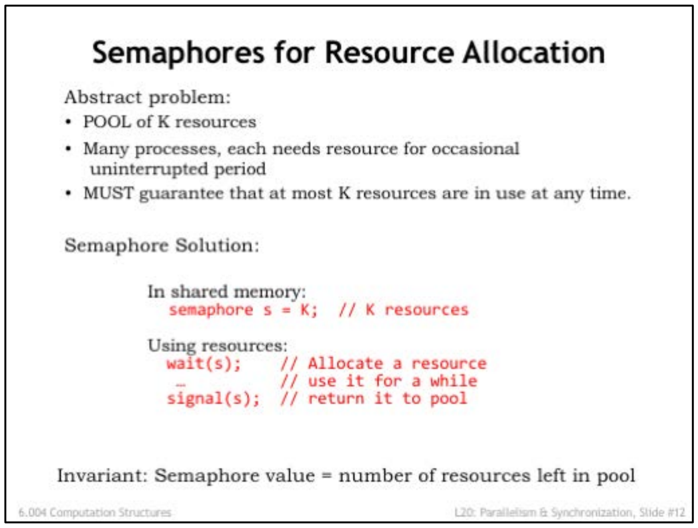
Goal: want statement A2 in process A to complete before statement B4 in Process B begins.

$$\boxed{A2 < B4}$$

Recipe:

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

6.004 Computation Structures L20: Parallelism & Synchronization, Slide #11



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:

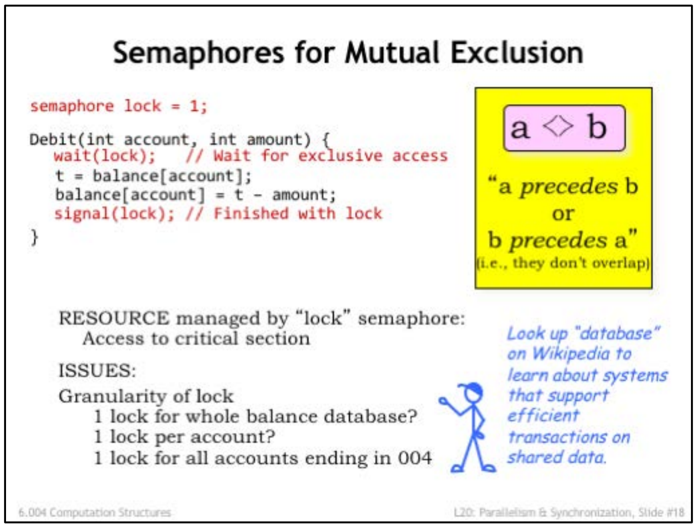
```
semaphore s = K;  // K resources
```

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

6.004 Computation Structures L20: Parallelism & Synchronization, Slide #12



Semaphores for Mutual Exclusion

```
semaphore lock = 1;

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

$$\boxed{a <> b}$$

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

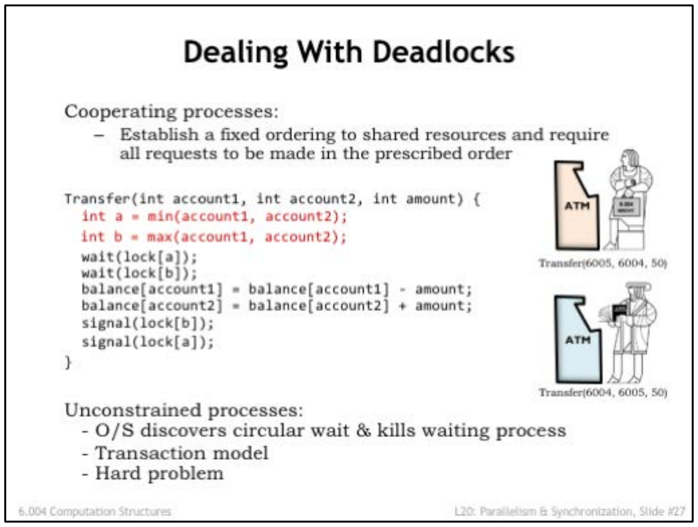
RESOURCE managed by "lock" semaphore: Access to critical section

ISSUES:

Granularity of lock
 1 lock for whole balance database?
 1 lock per account?
 1 lock for all accounts ending in 004

Look up "database" on Wikipedia to learn about systems that support efficient transactions on shared data.

6.004 Computation Structures L20: Parallelism & Synchronization, Slide #18



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

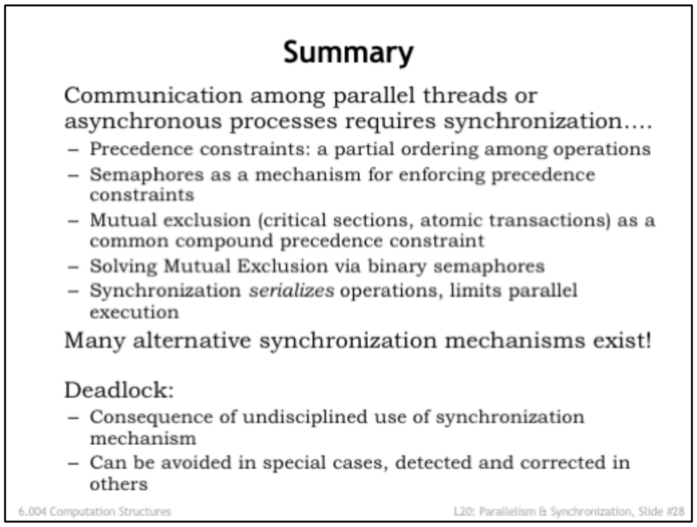
Transfer(6005, 6004, 50)

Transfer(6004, 6005, 50)

Unconstrained processes:

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

6.004 Computation Structures L20: Parallelism & Synchronization, Slide #27



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

6.004 Computation Structures L20: Parallelism & Synchronization, Slide #28

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
particle buffer[100];  // holds 100 particles

Semaphores and initial values: Semaphore pair = 0, capicty = 50. lock=1.
```

```
Producer
PLoop:

   Produce pair P1, P2
   wait(capicity) . wait(lock).
   Place P1 in buffer

   Place P2 in buffer
   signal (pair) ; signal(lock)
   Go to PLoop
```

```
Consumer
CLoop:

   wait(pair)
   Fetch P1 from buffer

   Fetch P2 from buffer
   signal(capicty).
   Package and ship

   Go to CLoop
```

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): ~~YES~~ NO. **HLOE (YES/NO):** YES **LOL (YES/NO):** YES.

? If the system start at print("E"), this is OK!

(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 = 1; b = 0; c = 0;

Process 1                 Process 2                 Process 3

Loop1:                    Loop2:                    Loop3:

    wait(a)                   wait(b)                   wait(c)
                                                        wait(c)
    print("H")                print("L")                print("O")

    print("E")                signal(c)

    signal(b)
    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:

```
Process A        counter=10        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.

A1 A2 B1 B2: counter = 13	**B1 A1 B2 A2: counter =** 11
A1 B1 A2 B2: counter = 12	**B1 A1 A2 B2: counter =** 12
A1 B1 B2 A2: counter = 11	**B1 B2 A1 A2: counter =** 13.

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: read =0 , write = 0

Process A                          Process B
…                                  …
A1: LD(counter,R0)                 B1: LD(counter,R0)
                                       signal (write)
    ADDC(R0,1,R0)                      ADDC(R0,2,R0)
    wait (write)                       wait ( read).
A2: ST(R0,counter)                 B2: ST(R0,counter)
…   signal ( read)                 …
```

(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 (Y)                         signal(X).
    ADDC(R0,1,R0)                      ADDC(R0,2,R0)
    wait (X)                           wait (Y).
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: X = 0.

Process P1	Process P2
	wait (X)
…Section A code…	… Section C code…
…Section B code…	…Section D code…
signal(X)	

(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: X = 1

Process P1	Process P2
wait (X)	Wait (X)
…Section A code…	… Section C code…
…Section B code…	…Section D code…
signal (X)	signal (X)

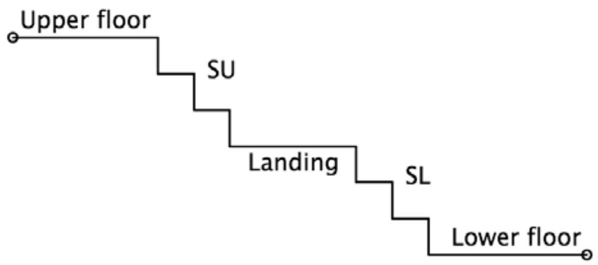
(C) Please add WAIT(…) and SIGNAL(…) statements as needed in the spaces below so that A ≤ D *and* C ≤ B, i.e., the first section (A and C) of **both** processes completes execution before the second section (B or D) of **either** process begins execution.

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

Semaphore initial values: X=0, Y=0.

Process P1	Process P2
…Section A code…	… Section C code…
signal (Y)	signal (X) :
wait (X)	wait (Y)
…Section B code…	…Section D code…

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 = ___1___, SL = ___1___, L = ___3___;
```

```
// code for going downstairs
Down() {
    wait (SU) ; wait (L)
    Enter SU;
    signal (SU)
    Exit SU/enter landing;
    wait (SL)
    Exit landing/enter SL;

    Exit SL;
    signal (SL)
    signal (L)
}
```

```
// code for going upstairs
Up() {
    wait (SL) ; wait (L)
    Enter SL;
    signal (SL)
    Exit SL/enter landing;
    wait (SU)
    Exit landing/enter SU;

    Exit SU;
    signal (L)
    signal (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?

used as a lock.

Initial value for S: 1.

(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);
```

$S_1 \rightarrow S_2$.

$S_2 \rightarrow S_3$.

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);
```

$S_2 \rightarrow S_3 \rightarrow S_1$

$S_1 \rightarrow S_2$.

Can it deadlock?

[✓ **YES**] **NO** **Can't tell**

circyle.

```
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);
```

$S_1 \rightarrow S_2$.

$S_2 \rightarrow S_3$.

$S_2 \rightarrow S_3 \rightarrow S_1$

Can it deadlock?

YES [✓ **NO**] **Can't tell**