# CSC 112: Computer Operating Systems
# Lecture 3

# Synchronization

Department of Computer Science,

Hofstra University

Acknowledgement: Lecture slides based on UC Berkeley CS 162: Operating Systems and System Programming
And NTNU course on Operating Systems

## Concurrency

- Consider three concurrent threads T1, T2, T3, which access a shared variable D that has been initialized to 100. There is no mutex protection. What are the minimum and maximum possible values of D after the three threads have completed execution?
- ANS:

```
//Initialization
int D=100;
//Thread T1
void main(){
D=D+20;
}
//Thread T2
void main(){
D=D-50;
}
//Thread T3
void main(){
D=D+10;
}
```

## Concurrency

- Consider the methods used by threads T1 and T2 for accessing their critical sections whenever needed, as given below. (These are the only methods for accessing the critical section, and T1 and T2 each runs a separate higher-level program that uses these methods which is not shown.) The initial values of S1 and S2 are arbitrary.
- Which of the following is true?
- (a) Mutual exclusion but not progress
- (b) Progress but not mutual exclusion
- (c) Neither mutual exclusion nor progress
- ANS:

```
//Initialization to arbitrary values
Boolean S1={true or false};
Boolean S2={true or false};
```

```
//Method use by Thread T1
while (S1 == S2);
//Critical section
S1 = S2;
```

```
//Method use by Thread T2
while (S2 != S1);
//Critical section
S2 = !S1;
```

## Mutual Exclusion

```
//Initialization
Boolean wants1=false,
Boolean wants2=false;
```

```
//Thread T1
while (true) {
    wants1 = true;
    while (wants2 == true);
    /* Critical Section */
    wants1 = false;
}
/* Remainder section */
```

```
//Thread T2
while (true) {
    wants2 = true;
    while (wants1 == true);
    /* Critical Section */
    wants2 = false;
}
/* Remainder section */
```

- Which one of the following statements is TRUE about the above construct?
- (a) It does not ensure mutual exclusion.
- (b) It does not ensure bounded waiting.
- (c) It requires that processes enter the critical section in strict alternation.
- (d) It does not prevent deadlocks, but ensures mutual exclusion.
- ANS:

# Race Conditions

Consider the two threads each executing t1 and t2. Values of shared variables y and z are initialized to 0

```
int y=0, z=0;
```

```
1 t1(){
2    int x;
3    x = y + z;
4 }
```

```
1 t2(){
2    y = 1;
3    z = 2;
4 }
```

Q. Give all possible final values for x and the corresponding order of execution of instructions in t1 and t2.

1) t1 runs to the end first; then t2 runs to the end: x = 0+0 = 0

2) t2 to line 2; then t1 to the end; then t2 to the end: x = 1+0 = 1

3) t2 to the end; then t1 to the end: x = 1+2 = 3

*Are there other possibilities giving additional values?*

## Race Conditions

- Addition operation x=y+z consist of multiple machine instructions in assembly language:
    A. fetch operand y into register r1
    B. fetch operand z into register r2
    C. add r1 + r2, store result in r3
    D. store r3 in memory location of x
- If a task switch to t2 occurs between machine instructions A and B; then t2 runs to completion before switching back to t1, then:
    - y is read as 0 (t2 didn't set y yet)
    - z is read as 2 (t2 sets z before execution instruction B of add. in t1)
    - the sum is then x = 0 + 2 = 2

```
int y=0, z=0;
```

```
t1(){
    int x;
    x = y + z;
}
```

```
t2(){
    y = 1;
    z = 2;
}
```

## Race Conditions

Q. Give a solution using semaphores.
Solution: we protect the addition x = y + z within a critical section, *using* a binary semaphore (mutex). This code guarantees that x can never have the value 1 or 2, possible values are x = 0, 3
(Line “int x” can be outside or inside the critical section with no difference. We use a slightly different notation of s.wait()/s.signal() to denote sem_wait(&s) and sem_post(&s).

```
int y=0, z=0;
semaphore s = 1;
```

```
t1(){
    int x;
    s.wait();
    x = y + z;
   s.signal();
}
```

```
t2(){
   s.wait();
   y = 1;
   z = 2;
   s.signal();
}
```

# Semaphores I

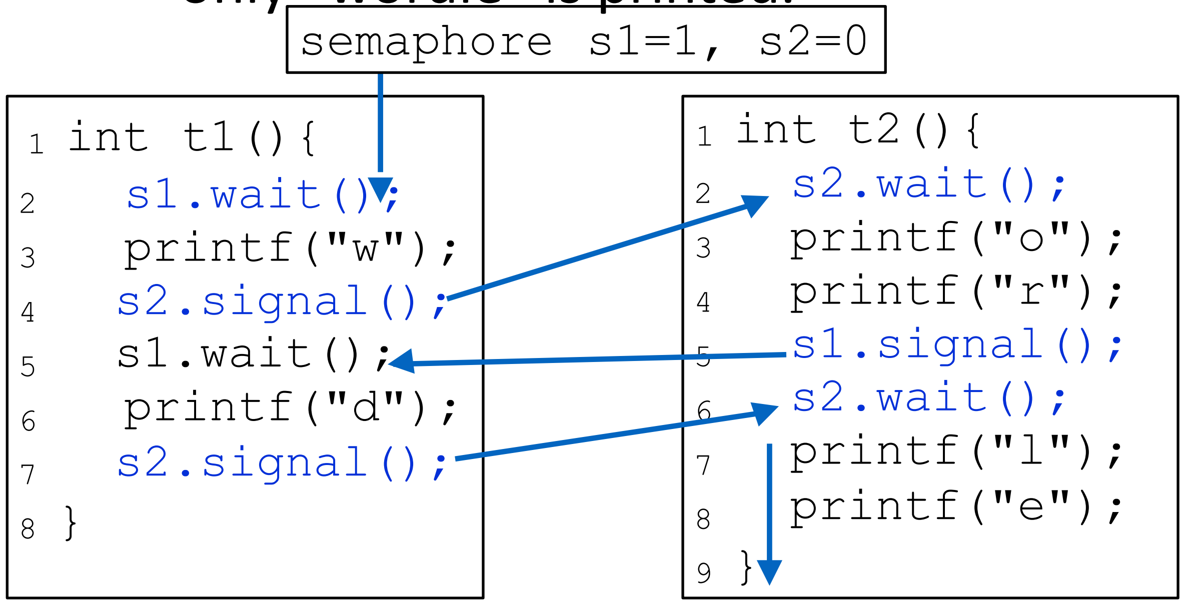
t1:

```
int t1()  {
   printf("w");
   printf("d");
}
```

t2:

```
int t2()  {
   printf("o");
   printf("r");
   printf("l");
   printf("e");
}
```

Q. Use semaphores and insert wait/signal calls into the two threads so that only “wordle” is printed.



```
semaphore s1=1, s2=0
```

```
int t1(){
   s1.wait();
   printf("w");
   s2.signal();
   s1.wait();
   printf("d");
   s2.signal();
}
```

```
int t2(){
   s2.wait();
   printf("o");
   printf("r");
   s1.signal();
   s2.wait();
   printf("l");
   printf("e");
}
```

- t1 has to run first to print "w", so s1 should be initialized to 1.
- t2 has to wait until the "w" has been printed by t1, then it is woken up by t1 calling s2.signal(), so s2 should be initialized to 0.

## Semaphores II

- The following three functions of a program f1(), f2(), f3() run in separate threads each and print some prime numbers. All three threads are ready to run at the same time. Use synchronization using the semaphores S1, S2 and S3 and wait/signal operations on the semaphores to ensure that the program outputs the prime numbers in increasing order (2, 3, 5, 7, 11, 13).

```
Semaphore S1=0;
Semaphore S2=0;
Semaphore S3=0;
f1() {
    printf("3");
    printf("5");
}
f2() {
    printf("2");
    printf("13");
}
f3() {
    printf(“7");
    printf("11");
}
```

## Semaphores II Solution

- Solution 1 (left): With initial values of all semaphores = 0, only f2 can run, prints 2, signals S1 and then waits for S2. S1.signal() starts f1, which was waiting for S1 and can now print 3 and 5 and then signal S3. S3.signal() now starts f3, which prints 7 and 11 and signals S2. This returns execution to f2, which can then finally print 13.
- Solution 2(right): s2 has initial value 1, so f2 calls S2.wait() and runs first. The rest of the same as Solution 1. You can see that initializing s2=0 has the same effect as initializing s2=1 and let f2 call S2.wait() first. So Solution 1 is better with one less call to wait().

```
semaphore S1=0;
semaphore S2=0;
semaphore S3=0;
f1() {
    S1.wait();
    printf("3");
    printf("5");
    S3.signal();
}
f2() {
    printf("2");
    S1.signal();
    S2.wait();
    printf("13");
}
f3() {
    S3.wait();
    printf("7");
    printf("11");
    S2.signal();
    }
```

```
semaphore S1=0;
semaphore S2=1;
semaphore S3=0;
f1() {
    S1.wait();
    printf("3");
    printf("5");
    S3.signal();
}
f2() {
    S2.wait();
    printf("2");
    S1.signal();
    S2.wait();
    printf("13");
}
f3() {
    S3.wait();
    printf("7");
    printf("11");
    S2.signal();
    }
```

## Semaphores III

```
semaphore s_a=0, s_b=0, s_c=0;
```

```
int t1() {
  while(1) {
    printf("A");
    s_c.signal();
    s_a.wait();
  }
}
```

```
int t2() {
  while(1) {
    printf("B");
    s_c.signal();
    s_b.wait();
  }
}
```

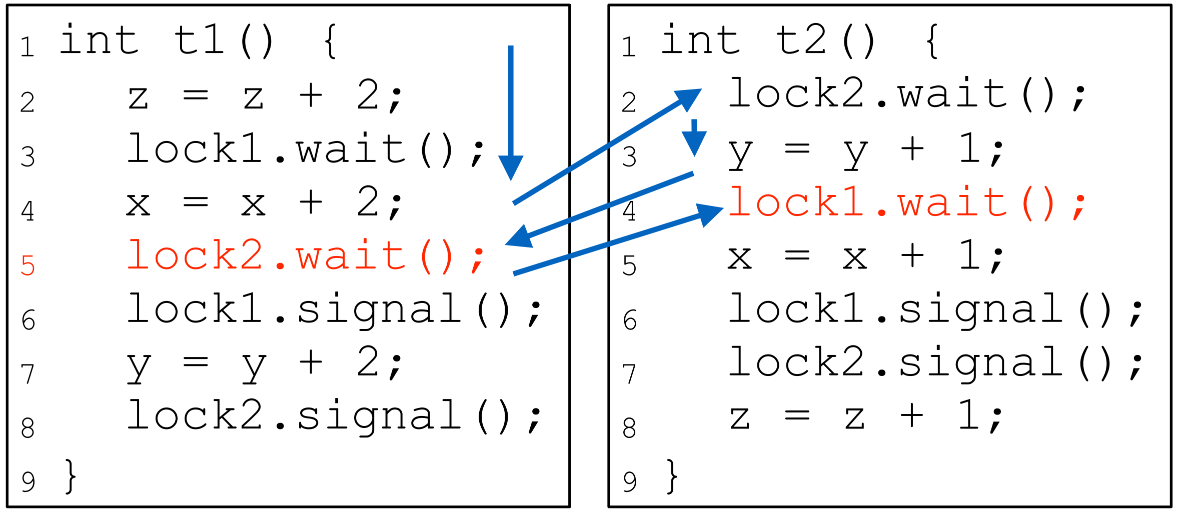
```
int t3() {
  while(1) {
    s_c.wait();
    s_c.wait();
    printf("C");
    s_a.signal();
    s_b.signal();
  }
}
```

Q. Which strings can be output when running the 3 threads in parallel?

- Either t1 or t2 could start first, so the first letter can be A or B
- Then both t1 and t2 signal s_c, only after both have signalled s_c, t3 can start and print C
- t3 signals s_a and s_b, which start in arbitrary order again
- Accordingly, the output is a regular expression ((AB|BA)C)+
    - Print A or B in arbitrary order, then print C, then the process repeats

## Deadlocks

```
//Initialization
int x=0, y=0, z=0;
semaphore lock1=1, lock2=1;
```



```
1 int t1() {
2     z = z + 2;
3     lock1.wait();
4     x = x + 2;
5     lock2.wait();
6     lock1.signal();
7     y = y + 2;
8     lock2.signal();
9 }
```

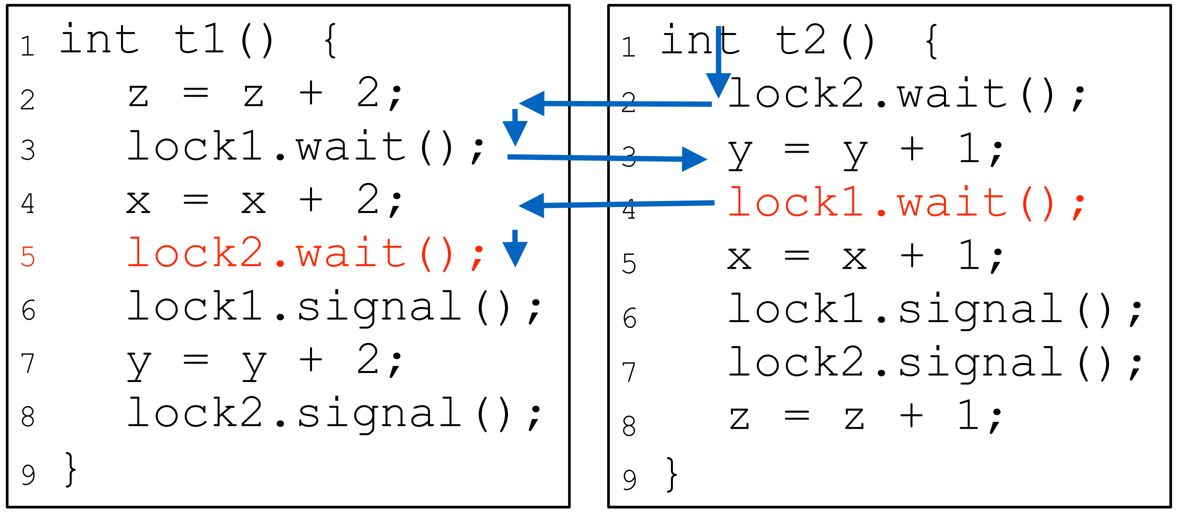
```
1 int t2() {
2     lock2.wait();
3     y = y + 1;
4     lock1.wait();
5     x = x + 1;
6     lock1.signal();
7     lock2.signal();
8     z = z + 1;
9 }
```

a. Executing the threads in parallel could result in a deadlock. Why?

- t1 runs first until line 4 (so lock1=0, lock2=1); switch to t2
- t2 starts and runs until line 3 (so lock1=0, lock2=0); back to t1
- t1 waits for lock2 in line 5 ↯ switch to t2, waits for lock1 in line 4
- This results in a *circular waiting condition* which is not resolved

## Quiz: Deadlocks

```
//Initialization
int x=0, y=0, z=0;
semaphore lock1=1, lock2=1;
```



```
1 int t1() {
2     z = z + 2;
3     lock1.wait();
4     x = x + 2;
5     lock2.wait();
6     lock1.signal();
7     y = y + 2;
8     lock2.signal();
9 }
```

```
1 int t2() {
2     lock2.wait();
3     y = y + 1;
4     lock1.wait();
5     x = x + 1;
6     lock1.signal();
7     lock2.signal();
8     z = z + 1;
9 }
```

Q. Executing the threads in parallel could result in a deadlock. Why?

- t2 runs first until line 2 (so lock2=0, lock1=1); switch to t1
- t1 starts and runs until line 3 (so lock1=0, lock2=0); back to t2
- t2 waits for lock2 in line 4; switch to t1, waits for lock1 in line 5

Note: There are other possible interleavings, as long as each thread grabs one lock and requests the other. You can remove all other statements and only leave the lock wait() instructions and get into this deadlock.)

# Deadlocks

```
int x=0, y=0, z=0;
semaphore lock1=1, lock2=1;
```

```
1 int t1() {
2    z = z + 2;
3    lock1.wait();
4    x = x + 2;
5    lock2.wait();
6    lock1.signal();
7    y = y + 2;
8    lock2.signal();
9 }
```

```
1 int t2() {
2    lock2.wait();
3    y = y + 1;
4    lock1.wait();
5    x = x + 1;
6    lock1.signal();
7    lock2.signal();
8    z = z + 1;
9 }
```

- Q. What are the possible values of x, y and z in the deadlock state?
- t1 runs until Line 5 lock2.wait() and t2 runs until Line 4 lock1.wait(), so x = 2, y = 1, z = 2
- Q. What are the possible values of x, y and z if the program finishes successfully without a deadlock?
- t1 runs first to the end, then t2 (or vice versa): x=3, y=3, z=3
- In t1, lock1.signal() sets lock1=1, lock2.signal() sets lock2=1, this exiting the critical sections protected by lock1 and lock2.
- Since Line 2 of t1 “z=z+2”, and Line 8 of t2 “z=z+1” are not protected within a critical section, a thread switch may occur in the middle of each line, e.g.,
  - t2 Line 8 reads z=0; before z is written back; switch to t1 Line 2, run t1 to the end; switch to t2 Line 8, write back z=0+1=1.
  - Or, t1 Line 2 reads z=0; before z is written back; switch to t2 Line 2, run t2 to the end; switch to t1 Line 2, write back z=0+2=2.
- Note: to prevent deadlocks, every thread should acquire locks in the same order, e.g. both acquire lock1 before lock2, or both acquire lock2 before lock1