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 I

```
int x = 0;
```

```
//Thread T0
for (int i=0; i<5; i++) {
   x = x + 1;
}
```

```
//Thread T0
for (int j=0; j<5; j++) {
   x = x + 2;
}
```

- Consider two concurrent threads T0, T1, which access a shared variable x that has been initialized to 0. There is no mutex protection.
- Q1: What are the minimum, maximum, and all possible values of x after the two threads have completed execution?
- Q2: Suppose we protect statement ‘x = x+2’ in Thread B within a critical section using a mutex lock. What are all the minimum, maximum, and all possible final values of x?

Concurrency I Answer

```
int x = 0;
```

```
//Thread T0
for (int i=0; i<5; i++) {
   x = x + 1;
}
```

```
//Thread T0
for (int j=0; j<5; j++) {
   x = x + 2;
}
```

- ANS1: Possible values of x after the two threads have completed execution: 5,…15. Min: 5. Max: 15.
  - The x=x+2 statements can be “erased” by being between the load and store of an x=x+1 statement, and vice versa. Each x=x+1 statement can either do nothing (if erased by Thread T1) or increase x by 1. Each x=x+2 statement can either do nothing (if erased by Thread T0) or increase x by 2. Since there are at least four stores from Thread T0 prior to the load for the last statement, the last load reads x value of at least 4. Since there are 5 of each type, and since x starts at 0, x has min 5 and max (5*1)+(5*2)=15. Possible values are 5, 6, 7,…15, e.g., If three increments from Thread T0 and two increments from Thread T1 are applied, then x=(3×1)+(2×2)=7.
- ANS2: Possible final values of x are 5, 7, 9, 11, 13, or 15. Min: 5. Max: 15.
  - Since the x=x+2 statements are atomic, the x=x+1 statements can never be “erased” because the load and store phases of x=x+2 cannot be separated. Thus, our final value is at least 5 (from Thread T0) with from 0 to 5 successful updates of x=x+1. When one x=x+2 is not erased, x has value 5+2=7. When two x=x+2 is not erased, x has value 5+4=9, and so on.

Concurrency II

- 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 II Answer

- Min 50, max 130
- Since each thread may read the value of int D, then write them in arbitrary order, overwriting each other's updates. Other possible results include 110, 120, 70…

Recall: Locks: Loads/Stores

- This implementation does not ensure mutual exclusion, since both threads may grab the lock:
- After Thread 1 reads flag==0 and exits the while loop, it is preempted/interrupted by Thread 2, which also reads flag==0 and exits the while loop. Then both threads set flag=1 and enter the critical section.
- Root cause: Lock is not an atomic operation!

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

flag = 0

Thread 1	Thread 2
call lock()	
while (flag == 1)	
interrupt: switch to Thread 2	
	call lock()
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Mutual Exclusion I

```
Boolean S0, S1;
S0=false, S1=false;
```

```
//Thread T0
while (true) {
    //Spin-waits if S0 == S1
    while (S0 == S1);
    //Critical section
    S0 = S1;
}
```

```
//Thread T1
while (true) {
    //Spin-waits if S0 != S1
    while (S0 != S1);
    //Critical section
    S1 = !S0;
}
```

- Does it achieve one of more of the correctness properties of a concurrent program:
  - Mutual exclusion: Only one thread in critical section at a time
  - Progress (deadlock-free): If several simultaneous requests, must allow one to proceed
  - Bounded waiting (starvation-free): Must eventually allow each waiting thread to enter
- Does it need the TestAndSet() instruction for atomic execution like the previous slide “Locks: Loads/Stores”?
- What is its major flaw?
- ANS:

Mutual Exclusion I: Sample Execution

```
Boolean S0, S1;
S0=false, S1=false;
```

```
//Thread T0
while (true) {
    while (S0 == S1);
    //Critical section
    S0 = S1;
}
```

```
//Thread T1
while (true) {
    while (S0 != S1);
    //Critical section
    S1 = !S0;
}
```

- T0 and T1 take turns to enter the critical section in strict alternation order .

	S0	S1
Init	F	F
T1 in CS		
	F	T
T0 in CS		
	T	T
T1 in CS		
	T	F
T0 in CS		
	F	F

Mutual Exclusion I Answer

- Mutual Exclusion: Achieved. Only one thread can enter its critical section at a time because the conditions S0 == S1 and S0 != S1 ensure that only one thread can proceed.

- Progress (Deadlock-Free): Achieved. It is not possible for each thread to be blocked forever waiting for each other.

- Bounded Waiting (Starvation-Free): Achieved. Both threads enter each one's critical section in strict alternation order, i.e., T0, T1, T0, T1…

- TestAndSet Instruction: Not required. The solution uses simple Boolean variables and logical operations. In the previous slide "Locks: Loads/Stores", all threads read and update a single global shared flag variable, so CPU atomic instructions like TestAndSet is needed to ensure atomicity of (read+modify+write) of the shared flag variable. But in this solution, each thread reads both S0 and S1, but T0 only updates S0 and T1 only updates S1, so no mutual exclusion is needed.

- Major Flaw: The algorithm relies on both threads actively participating in strict alternation order, i.e., T0, T1, T0, T1… If one thread stops due to some program bug or crashing, or is delayed indefinitely, the other thread might be blocked forever, leading to a potential deadlock. This may not happen for the example of two simple while loops, but it is just for illustration, whereas in reality each thread may run a large program with complex control flow, and use these instructions as lock/unlock instructions.

Mutual Exclusion II

```
Boolean flag[2];
flag[0]=false, flag[1]=false;
```

```
//Thread T0
while (true) {
     flag[0] = true;
     while (flag[1]==true);
     /* Critical Section */
     flag[0] = false;
}
```

```
//Thread T1
while (true) {
     flag[1] = true;
     while (flag[0]==true);
     /* Critical Section */
     flag[1] = false;
}
```

- Does it achieve one of more of the correctness properties of a concurrent program:
  - Mutual exclusion: Only one thread in critical section at a time
  - Progress (deadlock-free): If several simultaneous requests, must allow one to proceed
  - Bounded waiting (starvation-free): Must eventually allow each waiting thread to enter
- ANS:

Mutual Exclusion II: Sample Execution & Answer

```
Boolean flag[2];
flag[0]=false, flag[1]=false;
```

```
//Thread T0
while (true) {
    flag[0] = true;
    while (flag[1]==true);
    /* Critical Section */
    flag[0] = false;
}
```

```
//Thread T1
while (true) {
    flag[1] = true;
    while (flag[0]==true);
    /* Critical Section */
    flag[1] = false;
}
```

	Flag[0]	Flag[1]
Init	F	F
T0 tries	T	F
T0 in CS		
	F	F
T1 tries	F	T
T1 in CS		
	F	F
T0 tries	T	F
T1 tries	T	T
Deadlock		

- Mutual Exclusion: Achieved. The use of flags ensures that only one thread can enter its critical section at a time.
- Progress (Deadlock-Free): Not satisfied. If both threads set their flags simultaneously, they will block each other indefinitely, resulting in deadlock.
- Bounded Waiting (Starvation-Free): Achieved. One thread cannot repeatedly enter the CS and starve the other thread, if the other thread is waiting.

Mutual Exclusion III (Peterson's Solution)

```
Boolean flag[2];
flag[0]=false, flag[1]=false;
int turn = 0;
```

```
//Thread T0
while (true) {
    flag[0] = true;
    turn = 1;
    while (flag[1]==true && turn==1);
    /* Critical Section */
    flag[0] = false;
}
```

```
//Thread T1
while (true) {
    flag[1] = true;
    turn = 0;
    while (flag[0]==true && turn==0);
    /* Critical Section */
    flag[1] = false;
}
```

- Does it achieve one of more of the correctness properties of a concurrent program:
  - Mutual exclusion: Only one thread in critical section at a time
  - Progress (deadlock-free): If several simultaneous requests, must allow one to proceed
  - Bounded waiting (starvation-free): Must eventually allow each waiting thread to enter
- ANS:

Mutual Exclusion III (Peterson's Solution): Sample Execution & Answer

- Mutual Exclusion: Achieved. The combination of flag and turn ensures that only one thread can enter its critical section at a time.
- Progress (Deadlock-Free): Achieved. The turn variable ensures that if both threads want to enter their critical sections, one will eventually proceed. It is not possible for each thread to be blocked forever waiting for each other.
- Bounded Waiting (Starvation-Free): Achieved. Each thread gets a fair chance to enter its critical section due to the alternation enforced by the turn variable.

	Flag[0]	Flag[1]	turn
Init	F	F	0
T0 tries	T	F	1
T0 in CS			
	F	F	1
T1 tries	F	T	0
T1 in CS			
	F	F	0
T0 tries	T	F	1
T1 tries	T	T	0
T0 in CS	(T1 cannot enter CS)		
	F	T	0
T0 tries	T	T	1
T1 in CS	(T0 cannot enter CS)		
	T	F	1

Mutual Exclusion III (Peterson's Solution Variation)

```
Boolean flag[2];
flag[0]=false, flag[1]=false;
int turn = 0;
```

```
//Thread T0
while (true) {
    flag[0] = true;
    turn = 0;
    while (flag[1]==true && turn==1);
    /* Critical Section */
    flag[0] = false;
}
```

```
//Thread T1
while (true) {
    flag[1] = true;
    turn = 1;
    while (flag[0]==true && turn==0);
    /* Critical Section */
    flag[1] = false;
}
```

- Does it achieve one of more of the correctness properties of a concurrent program:
  - Mutual exclusion: Only one thread in critical section at a time
  - Progress (deadlock-free): If several simultaneous requests, must allow one to proceed
  - Bounded waiting (starvation-free): Must eventually allow each waiting thread to enter
- ANS:

Mutual Exclusion III (Peterson's Solution Variation) Sample Execution & Answer

- This variation is similar to Peterson's Solution but with an incorrect implementation of the turn variable:
- Mutual Exclusion: Achieved. Only one thread can enter its critical section at a time due to the conditions on flag and turn.
- Progress (Deadlock-Free): Achieved. It is not possible for each thread to be blocked forever waiting for each other.
- Bounded Waiting (Starvation-Free): Not satisfied. A thread may be indefinitely delayed if the other repeatedly sets its flag and does not allow alternation via the turn variable, i.e., one thread can repeatedly enter the CS and starve the other thread.
- TestAndSet Instruction: Not required.
- Major Flaw: Incorrect handling of the turn variable leads to potential livelock or starvation.

	Flag[0]	**Flag[1]**	**turn**
Init	F	F	0
T0 tries	T	F	0
T0 in CS			
	F	F	0
T1 tries	F	T	1
T1 in CS			
	F	F	1
T0 tries	T	F	0
T1 tries	T	T	1
T1 in CS			
	T	F	1
T1 tries	T	T	1
T1 in CS			
	T	F	1
T0 experiences starvation			

Readers/Writers Solution using Monitors, Prefers Writers

```
int AR=0: Number of active readers;
int WR=0: Number of waiting readers;
int AW=0: Number of active writers;
int WW=0: Number of waiting writers;
Condition okToRead, okToWrite;
mutex_t mutex = 1;
```

Q: Rewrite it to prefer readers

```
Reader() {
    mutex_lock(&mutex);
    while ((AW + WW) > 0) {//Is it safe to read?
       WR++; //No. Writers exist
       cond_wait(&okToRead,&mutex);
       WR--; //No longer waiting
    }
    AR++;    //Reader active!
    mutex_unlock(&mutex);
    AccessDatabase(ReadOnly);
    mutex_lock(&mutex);
    AR--;    //No longer active
    if (AR == 0 && WW > 0)//No other active
readers
       cond_signal(&okToWrite);//Wake up one
writer
    mutex_unlock(&mutex);
}
```

```
Writer() {
    mutex_lock(&mutex);
    while ((AW + AR) > 0) {//Is it safe to
write?
       WW++; //No. Active users exist
       cond_wait(&okToWrite,&mutex);
       WW--; //No longer waiting
    }
    AW++; //Writer active!
    mutex_unlock(&mutex);
    AccessDatabase(ReadWrite);
    mutex_lock(&mutex);
    AW--;    //No longer active
    if (WW > 0){//Give priority to writers
       cond_signal(&okToWrite);//Wake up one
writer
    } else if (WR > 0) {//Otherwise, wake reader
       cond_broadcast(&okToRead);//Wake all
readers
    }
    mutex_unlock(&mutex);
}
```

Readers/Writers Solution using Monitors, Prefers Writers

ANS: Rewrite it to prefer readers

```
int AR=0: Number of active readers;
int WR=0: Number of waiting readers;
int AW=0: Number of active writers;
int WW=0: Number of waiting writers;
Condition okToRead, okToWrite;
mutex_t mutex = 1;
```

```
Reader() {
    mutex_lock(&mutex);
    while (AW > 0) {//Is it safe to read?
       WR++; //No. Writers exist
       cond_wait(&okToRead,&mutex);
       WR--; //No longer waiting
    }
    AR++;    //Reader active!
    mutex_unlock(&mutex);
    AccessDatabase(ReadOnly);
    mutex_lock(&mutex);
    AR--;    //No longer active
    if (AR == 0 && WW > 0)//No other active
readers
       cond_signal(&okToWrite);//Wake up one
writer
    mutex_unlock(&mutex);
}
```

```
Writer() {
    mutex_lock(&mutex);
    while ((AW + AR) > 0) {//Is it safe to
write?
       WW++; //No. Active users exist
       cond_wait(&okToWrite,&mutex);
       WW--; //No longer waiting
    }
    AW++; //Writer active!
    mutex_unlock(&mutex);
    AccessDatabase(ReadWrite);
    mutex_lock(&mutex);
    AW--;    //No longer active
    if (WR > 0) {//Wake all readers
       cond_broadcast(&okToRead);//Wake all
readers
    } else if (WW > 0){Otherwise, wake one
writer
       cond_signal(&okToWrite);//Wake up one
writer
    }
    mutex_unlock(&mutex);
}
```

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 I

```
//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 }
```

Deadlock scenario 1:

- 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
- This results in a circular waiting condition, where each thread grabs one lock and requests the other.

```
//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 }
```

Deadlock scenario 2:

- 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
- (Other interleavings are possible, e.g., t1 grabs lock1, t2 grabs lock2 requests lock 1, t1 requests lock 2)
- 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

Deadlocks II

```
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.