Coming Up

- Two Quizzes (will open Thursday 12:00 AM and due by Sunday 11:00 PM).
  - One on multithread programming
  - And another on synchronization and deadlocks
  - Guides will be posted.

- **Midterm Written Exam II**
  - Per the syllabus original dates:
    - Thursday, Nov 14th, 09:40AM-11:05 AM, 50 points,
    - A guide similar to the first midterm exam will be posted on Friday.
    - Most of the questions will be from Chapters 3, 4, 5, and 6.
      - (Process and Thread Management, Concurrency and Synchronization)
      - Will involve programming questions on threading (C/C++ & Java) and race-conditions

1

1

University of Wisconsin-Stout

CS 442, Operating Systems,

Saleh M. Alnaeli, Ph.D.

Fall, 2019

Concurrency and Synchronization

2

2

Multiple Processes

- Operating System design is concerned with the management of processes and threads:
  - Multiprogramming
    - The management of multiple processes within a uniprocessor system
  - Multiprocessing
    - The management of multiple processes within a multiprocessor
  - Distributed Processing
    - The management of multiple processes executing on multiple, distributed computer systems
    - The recent proliferation of clusters is a prime example of this type of system

3

"Too Much Milk" Example

time	you	your roommate
3:00	arrive home	
3:05	look in fridge, no milk	
3:10	leave for grocery store	
3:15		arrive home
3:20	arrive at store	look in fridge, no milk
3:25	buy milk, leave	leave for grocery store
3:30		
3:35	arrive home	arrive at store
3:36	put milk in fridge	
3:40		buy milk, leave
3:45		
3:50		arrive home
3:51		put milk in fridge
3:51	oh, no! too much milk!!	

- What is the problem with above code?

4

4

"Too Much Milk" Example

time	you	your roommate
3:00	arrive home	
3:05	look in fridge, no milk	
3:10	leave for grocery store	
3:15		arrive home
3:20	arrive at store	look in fridge, no milk
3:25	buy milk, leave	leave for grocery store
3:30		
3:35	arrive home	arrive at store
3:36	put milk in fridge	
3:40		buy milk, leave
3:45		
3:50		arrive home
3:51		put milk in fridge
3:51	oh, no! too much milk!!	

- *atomic operation* – cannot be interleaved with
- problem with above code is that lines
  "Look in fridge, no milk" through *"Put milk in fridge"*
  are not atomic.

5

5

"Running Contest" Example

Thread A Thread B

```
global variable i = 0

while (i < 10)              while (i > -10)
  i = i + 1                       i = i - 1
print "A wins"              print "B wins"
```

Race Condition
- Occurs when multiple processes or threads read and write data items
- The final result depends on the order of execution

- questions:
  - who wins? is it guaranteed that someone wins?
  - what if both threads have their own CPU, running concurrently at exactly the same speed? is it guaranteed that it goes on forever?
  - what if they are sharing a CPU?
  - what is the atomicity of the operations?

CPP implementation → Next Slide

Updated by Dr. Alnaeli
From Dr. Nesterenko's OS Presentation

6

CPP Implementation of Running Contest

```
#include <iostream>       // std::cout
#include <thread>         // std::thread
using namespace std;

int countValue = 0; //global variable
void task1();
void task2();

int main()
{
    //creating threads
    thread th1(task1);
    thread th2(task2);
    th1.join();
    th2.join();

    return 0;
}
```

```
void task1() {
    while (countValue < 5) {
        countValue = countValue + 1;
        cout << "\nA: countValue =" <<
        countValue << endl;
    }
    cout << "\n A wins" << endl;
    exit(0);
}
void task2() {
    while (countValue > -5) {
        countValue = countValue - 1;
        cout << "\nB: countValue =" <<
        countValue << endl;
    }
    cout<< "\n B wins" << endl;
    exit(0);
}
```

7

Demo raceMillion.cpp

```
#include <iostream>
#include <thread>
#include <vector>
using namespace std;
int counter = 0;
void race(){
int count = 0;
        std::thread::id tid = this_thread::get_id();
        this_thread::yield();

        for (int i = 0; i < 1000000; i++) {
                counter = counter + 1;
                count++;
        }

        cout << "\nThread  " << tid << "  calculated " << count << endl;
```

```
int main(void) {
vector <thread> raceThreads;

        for (int i = 0; i < 2; i++)
                raceThreads.push_back(thread(race));
        for (thread& t : raceThreads)
                t.join();
        cout << "The final counter is " << counter << endl;
}
```

- What is the expected value of printed by main thread **counter** after the other threads are done?
- What is the actual value?
- What are the expected values of the local **count** printed by each thread?
- What are the actual value?

8

Race Condition
C/C++ Example. Sum of Matrix elements.

- For full version, please see Piazza.

```
void parallel_Matrix_Sum(int from, int to) {
    for (int draw = from; draw <= to; ++draw) {
        for (int dcolumn = 0; dcolumn < matrixC.size(); ++dcolumn) {
            sum = sum + matrixC[draw][dcolumn]; //Race condition, supposed to be a Critical Section,}
    }
}
```

Solution: We have to make it mutually exclusive so that only a single thread can access the critical section. (see next slide ☺)

9

9

Synchronization Solution in Java for the exercise from last lecture

```
// Provides synchronized access
public class SyncCounter
{
  private int count;
  public SyncCounter()  {    count = 0;  }

  public synchronized void increment() // Make it Mutually
   Exclusive
  {
    count++;  // one thread at a time
  }
  public String toString()
  {
    return "Count is:\t" + count;
  }
} // end of class SyncCounter
```

```
public class Demo05
{
  public static void main(String[] args) throws
   InterruptedException
  {
    SyncCounter sc = new SyncCounter();
    Runnable r1 = new Increase2(sc, 5000);
    Runnable r2 = new Increase2(sc, 5000);
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);
    t1.start(); t2.start();
    t1.join(); t2.join();
    System.out.println("Count is: " + sc);
  }
} // end of class Demo05
```

Output is 10000

10

Race Condition
C/C++ Example. Sum of Matrix elements.

- For full version, please see Piazza.

```
void parallel_Matrix_Sum(int from, int to) {
    for (int draw = from; draw <= to; ++draw) {
        for (int dcolumn = 0; dcolumn < matrixC.size(); ++dcolumn) {
            sum = sum + matrixC[draw][dcolumn]; //Race condition, supposed to be a Critical Section,}
    }
}
```

Solution: We have to make it mutually exclusive so that only a single thread can access the critical section. (see next slide ☺)

11

11

C/C++ Solutions (example using Mutex Object)

- **Mutex** is a program object that provides Mutual Exclusion. That is, it is created so that multiple program thread can take turns sharing the same resource, such as access to a shared variable or a file.
- Make sure to include the <mutex> library. #include <mutex> // std::mutex
- For full version, please see Piazza.

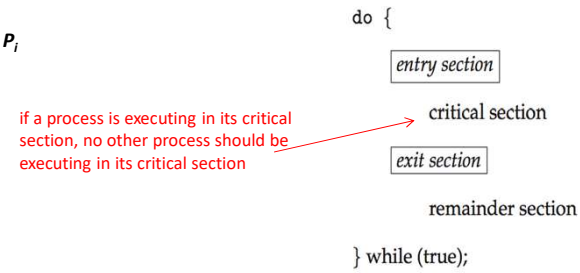
```
void parallel_Matrix_Sum(int from, int to) {
    for (int draw = from; draw <= to; ++draw) {
        for (int dcolumn = 0; dcolumn < matrixC.size(); ++dcolumn) {
            mtx.lock();   // one thread at a time. Thus, critical section
            sum = sum + matrixC[draw][dcolumn]; // No Race condition, critical section
            mtx.unlock();
        }
    }
}
//Advantages (Safety and correctness)
// Disadvantages: Overhead of having the threads to wait.
//Do an experiment. (compare with and without Mutex object
```

12

12

Critical Section/MX

General structure of process P_i



- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section/MX problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

13

Critical Regions

Requirements to avoid race conditions:

1. **No** two processes may be simultaneously inside their critical regions.
2. **No** assumptions may be made about speeds or the number of CPUs.
3. **No** process running outside its critical region may block other processes.
4. **No** process should have to wait forever to enter its critical region.

14

Mutual Exclusion

- *mutual exclusion* (MX) – elementary synchronization problem
- there is a set of threads that indefinitely alternate between executing the *critical section* (CS) of code and non-critical section
  - each thread may execute non-critical section indefinitely
  - each thread expected to execute the CS finitely long
- before entering the CS, a thread *requests* it
- a solution to MX should ensure
  - ***safety*** – at most one thread at a time execute the CS
  - ***liveness*** – a thread requesting the CS is eventually given a chance to enter it
    - liveness violation: ***starvation*** – a requesting thread is not allowed to enter the CS
      - **deadlock** – all threads are blocked and cannot proceed
      - **livelock** – threads continue to operate but none could enter the CS
- could be solved
  - algorithmically (in user space)
  - with OS support

15

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

16

Synchronization Terminology

atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

17

Mutual Exclusion: Hardware Support

- Disabling Interrupt
- Special Machine Instructions
  - Next Slides

18

18

Interrupt Disabling

- In a uniprocessor system, concurrent processes cannot have overlapped execution; they can only be interleaved.
- Furthermore, a process will continue to run until it invokes an OS service or until it is interrupted.
- Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted.
- This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts.
- A process can then enforce mutual exclusion in the following way

```
while (true) {
   /* disable interrupts */;
      /* critical section */;
   /* enable interrupts */;
   /* remainder */;
}
```

➢ **Problems:**

- The efficiency of execution could be degraded
  - processor is limited in its ability to interleave processes.
- will not work in a multiprocessor architecture.

19

19

Special Machine Instructions

- In a multiprocessor configuration, several processors share access to a common main memory. No interrupt mechanism between them could support MX.
- But, at the hardware level, access to a memory location excludes any other access to the same location.
- With this fact, processor designers proposed machine instructions that carry out two actions atomically. So, access to a CS can be blocked when busy.
- The important characteristic of this instruction is that it is executed atomically. Thus, if two test_and_set() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

```
int compare_and_swap(int *word, int testval, int newvalue) {
    int oldval = *word;
    if (*oldval == testval)
            *word = newvalue;
    return oldval;
}
    //The definition of the compare and swap() instruction
```

20

20

Machine Instructions for Mutual Exclusion II

- The memory location has been updated if the returned value is the same as the test value.
- This atomic instruction therefore has two parts: A compare is made between a memory value and a test value; if the values are the same, a swap occurs.
- The entire compare & swap function is carried out atomically—that is, it is not subject to interruption.

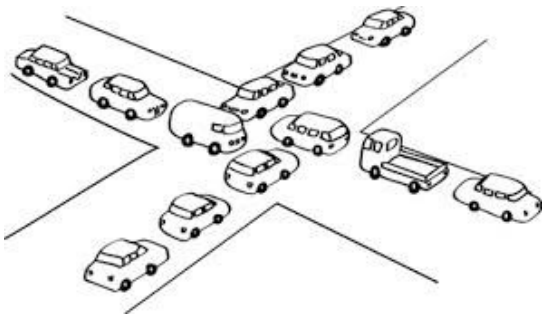
```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
         /* do nothing */;
     /* critical section */;
     bolt = 0;
     /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));

}
```

- **PROPERTIES OF THE MACHINE-INSTRUCTION APPROACH**
  - advantages:
    - Easy to verify
    - Safety is guarantee
    - Works with single/multi processors and many CSs
  - problems:
    - Busy waiting.
    - violates liveness, why?
    - Deadlock is possible
      - Example: P1 and P2 (Higher priority), P1 in CS, P2 scheduled and Busy waiting for P1, P1 never scheduled due to lower priority. Thus, **deadlock**!

21

21

What is a Deadlock?



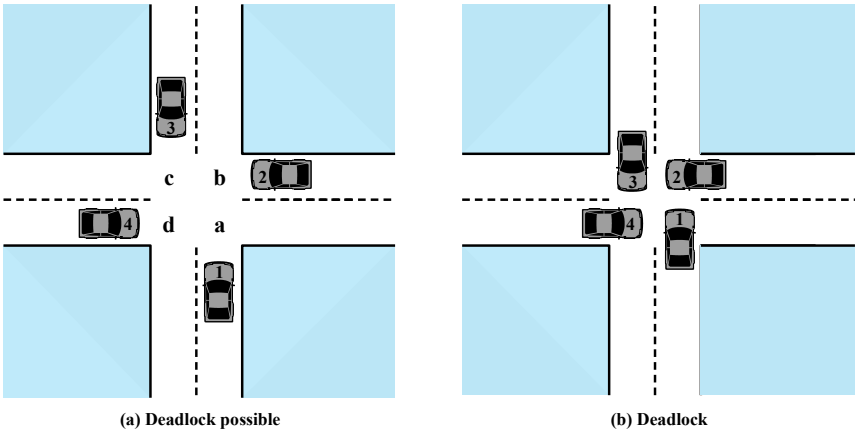
A situation wherein each of a collection of processes is waiting for something from other processes in the collection. Since all are waiting, none can provide any of the things being waited for.

22

22

Deadlock Illustration

All deadlocks involve conflicting needs for resources by two or more processes.



(a) Deadlock possible (b) Deadlock

four-way stop intersection at approximately the same time.
- **Car 1**, traveling north, needs quadrants a and b.
- **Car 2** needs quadrants b and c.
- **Car 3** needs quadrants c and d.
- **Car 4** needs quadrants d and a.

if all four cars ignore the rules and proceed (cautiously) into the intersection at the same time, then each car seizes one resource (one quadrant) but cannot proceed because the required second resource has already been seized by another car. This is an actual deadlock.

23

Network Environment

Scanner

How about if the scanner and the blue-ray driver are shared by many users and applications.
If an app needs to scan a document and send it directly to the Blue-ray, it will need to acquire both devices.
How about of two apps need to do the same thig but programmed in a different ways. One acquires the BR driver first and the other acquires the scanner first?

24

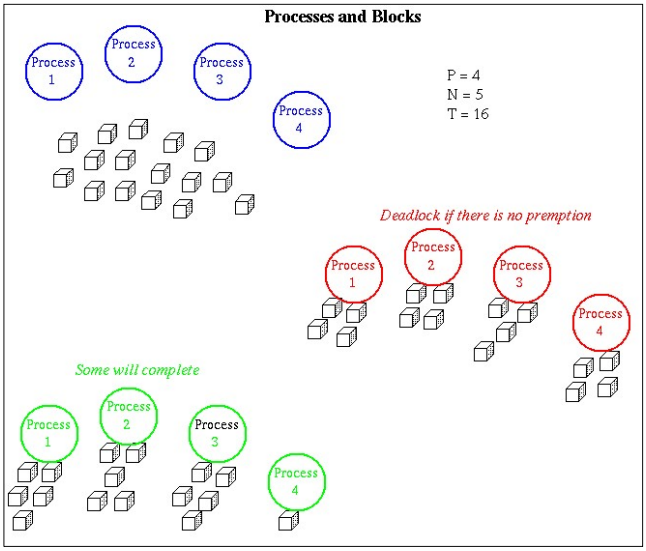
24

Deadlock Cont. Block building contest example

- ✓ Deadlock can occur over separate resources, as in previous example, or even over separate copies of a single resource.

- **Block building contest**: suppose each of a collection of processes is trying to produce a result by acquiring a number of resources one after the other.
  - P processes.
  - N blocks needed by each process.
  - T total blocks.

Will the processes all be able to complete their jobs?



25

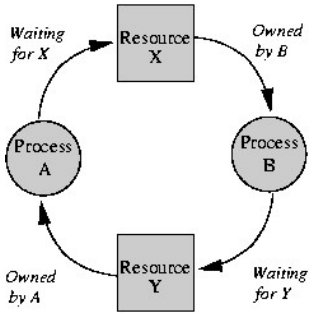
Deadlock Conditions

In general, there are four **conditions for deadlock**:

- ✓ **No sharing of resources.** They can be used by only one process at a time.
- ✓ **No preemption.** Once given, a resource cannot be taken away.
- ✓ **Multiple independent requests.** Processes do not ask for resources all at once.
- ✓ **Circularity in the resource graph.** The resource graph describes who has what and who wants what.

These are relatively simple-minded cases:

- Do not know in advance how many resources a process will need.
- Processes may release and re-request the same resource.
- Deadlock can occur over anything involving waiting,
  - for example messages in a pipe system. Hard for OS to control.



26

Solutions to the deadlock problem

Solutions to the deadlock problem fall into two general categories:

- **Prevention**: organize the system so that it is impossible for deadlock ever to occur.
  - May lead to less efficient resource utilization in order to guarantee no deadlocks.
  - constrain resource requests to prevent at least one of the four conditions of deadlock.
- **Avoidance**: determine when the system is deadlocked and then take drastic action.
  - requires knowledge of future process resource requests.
  - Approach 1: Do not start a process if its demands might lead to deadlock.
  - Approach 2: Do not grant an incremental resource request to a process if this allocation
- **Detection**: determine when the system is deadlocked and then take drastic action.
  - Requires termination of one or more processes in order to release their resources. Usually this is not practical.

27

27

Deadlock Prevention

organize the system so that it is impossible for deadlock ever to occur.

- May lead to less efficient resource utilization in order to guarantee no deadlocks.
- constrain resource requests to prevent at least one of the four conditions of deadlock.

Deadlock prevention: must find a way to eliminate one of the four necessary conditions for deadlock:

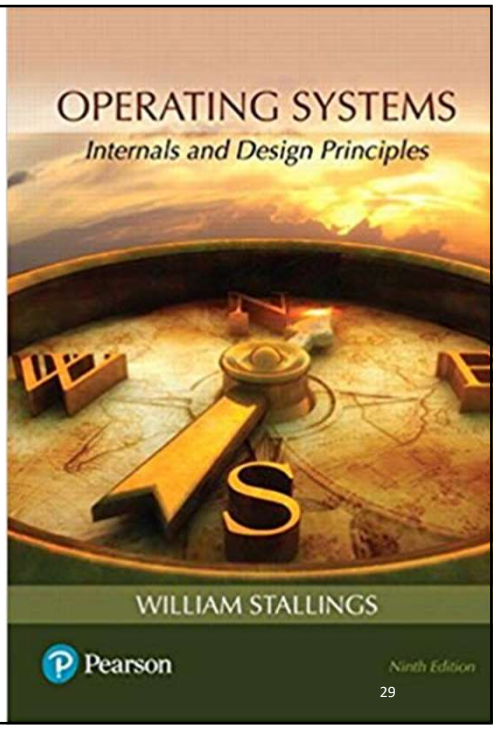
- Create enough resources so that there is always plenty for all.
- Do not allow waiting.
- Do not allow exclusive access. (not practical)
- Allow preemption.
- Make process ask for everything at once. Either get them all or wait for them all. (Tricky to implement).
- Make *ordered* or *hierarchical* requests. (e.g. ask for all S's, then all T's, etc.)
  - All processes must follow the same ordering scheme.

28

28

Reading Task (more on deadlock)

- Please read the sections 6.1 to 6.4.
- Literature review recommended
- A quiz will follow this task (deadline Sunday, Nov 10th, 11:00 PM.)
  - Quiz will be online via Canvas and will open on Friday 12:00 PM.
- Will have in-class discussion on the topic on Monday, the 11th .
  - Please be prepared to share your findings.



29

29

Mutual Exclusion with **Busy Waiting** Software Solutions

- **Lock Variable**
  - Using a single shared lock. Initially 0.
  - If a process wants to enter the critical section, it tests the lock. If 0 then it sets it to 1 and proceeds to the critical region. If it is already 1 (critical region is occupied), it waits until it is 0 (available).
  - (impractical. Same problem; Race condition)
    - **P1** reads the lock and sees 0, and before updating it to 1 it **P2** is scheduled and reads the lock and gets 0 and updates it to 1 and enters the critical region. **P1** now is back again and scheduled and sets the lock to 1. Thus now we have a problem, **P1** and **P2** in the critical section at the same time.
- **Strict Alternation**
  - Next Slides
- **Semaphores**
- **Mutex (Done!)**
- **Monitors**
- **Message Passing**

30

30

Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

- threads take turns entering CS.
  - (a) checks if it's her turn. If it is not (`turn!=0`) then it waits doing nothing
  - if it is `(a)`'s turn it proceeds with it's CS and setting turn to 1 giving `(b)` an opportunity to enter her CS
- In both cases, be sure to note the semicolons terminating the while statements.
- **advantages**:
  - enforces safety
- **problems**:
  - Do you see any problem in this algorithm?
  - Work with a partner!

31

Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

- threads take turns entering CS.
  - (a) checks if it's her turn. If it is not (`turn!=0`) then it waits doing nothing
  - if it is `(a)`'s turn it proceeds with it's CS and setting turn to 1 giving `(b)` an opportunity to enter her CS
- In both cases, be sure to note the semicolons terminating the while statements.
- **advantages**:
  - enforces safety
- **problems**:
  - violates liveness, why?

  If p1 sets turn to p2 and p1 nonCS is really long and p2 nonCS is really short, p2 may want to enter its CS before p1 gets to its CS. So p2 is the only one wanting to enter the CS but must wait for p1 to finish its nonCS and make its way back to its CS, enter its CS and set turn to p2.

32

MX algorithm 2a

```
t1 ( ) {
      while (true) {
        while (t2_in_CS == true)
            ;  /* do nothing */
        t1_in_CS = true;
        CS
        t1_in_CS = false;
        non-CS
      }
}

t2 ( ) {
      while (true) {
        while (t1_in_CS == true)
            ;  /* do nothing */
        t2_in_CS = true;
        CS
        t2_in_CS = false;
        non-CS
      }
}
```

- before entering CS thread checks if the other is in CS
  - `t1_in_CS,t2_in_CS` indicate if corresponding thread is in CS
- **advantage**:
  - enforces liveness
- **problems**:
  - Do you see any problem in this algorithm?
  - Work with a partner!

33

33

MX algorithm 2a

```
t1 ( ) {
      while (true) {
        while (t2_in_CS == true)
            ;  /* do nothing */
        t1_in_CS = true;
        CS
        t1_in_CS = false;
        non-CS
      }
}

t2 ( ) {
      while (true) {
        while (t1_in_CS == true)
            ;  /* do nothing */
        t2_in_CS = true;
        CS
        t2_in_CS = false;
        non-CS
      }
}
```

- before entering CS thread checks if the other is in CS
  - `t1_in_CS,t2_in_CS` indicate if corresponding thread is in CS
- **advantage**:
  - enforces liveness
- **problem**:
  - **no safety** – after `t1` decides that `t2` is not in CS it is already in CS. Yet it takes time for `t1` to set its own flag `t1_in_CS` to `true`

34

34

MX algorithm 2b

```
t1 ( ) {
    while (true) {
        t1_in_CS = true;
        while (t2_in_CS == true)
        ;    /* do nothing */
        CS
        t1_in_CS = false;
        non-CS
    }
}
t2 ( ) {
    while (true) {
        t2_in_CS = true;
        while (t1_in_CS == true)
        ;  /* do nothing */
        CS
        t2_in_CS = false;
        non-CS
    }
}
```

- let's move the setting of the **`t1_in_CS`** flag outside of the neighbor's flag checking
- advantage: safety enforced
- **problems**:
  - Do you see any problem in this algorithm?
  - Work with a partner!

35

35

MX algorithm 2b

```
t1 ( ) {
    while (true) {
        t1_in_CS = true;
        while (t2_in_CS == true)
        ;    /* do nothing */
        CS
        t1_in_CS = false;
        non-CS
    }
}
t2 ( ) {
    while (true) {
        t2_in_CS = true;
        while (t1_in_CS == true)
        ;  /* do nothing */
        CS
        t2_in_CS = false;
        non-CS
    }
}
```

- let's move the setting of the **`t1_in_CS`** flag outside of the neighbor's flag checking
- advantage: safety enforced
- problem – liveness violated
  - threads deadlock - both threads cannot get to CS if they set their flags at the same time!

36

36

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share variables:
  - **`int turn;`**
  - **`Boolean t1_wants_in_CS and t2_wants_in_CS`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The variables **`tx_wants_in_CS`** array is used to indicate if a process x is ready to enter the critical section.

37

MX algorithm 3 (Peterson's alg.)

```
1   t1 ( ) {
        while (true) {
3          t1_wants_in_CS = true;
           turn = 2;
5          while (t2_wants_in_CS == true
             && turn == 2)
7           ;  /* do nothing */
           CS
9          t1_wants_in_CS = false;
           non-CS
        }
11  }

13  t2 ( ) {
        while (true) {
15         t2_wants_in_CS = true;
           turn = 1;
17         while (t1_wants_in_CS == true
             && turn == 1)
19          ;  /* do nothing */
           CS
21         t2_wants_in_CS = false;
           non-CS
        }
23  }
```

- join algs 1 and 2
- condition of **`while`** of **`t1`** does NOT hold (**`t1`** is allowed to proceed to CS) if:
  - **`t2`** is out of CS (**`t2_wants_in_CS != true`**)
  - **`t1`** is blocked on its **`while (turn == 1)`**: **`t2`** set **`turn`** to 1 after **`t2`** executed: **`turn = 2`**
- enforces safety
- enforces liveness (no deadlock)
  - **`t1`** cannot be blocked when:
  - **`t2`** is in ***`non-CS`***: **`t2_wants_in_CS == false`**
  - **`t2`** is in **`while`**: **`turn == 1`**
- Disadvantages
  - It involves Busy waiting
  - It is limited to 2 processes.

38

Multiple Process MX (bakery alg.)

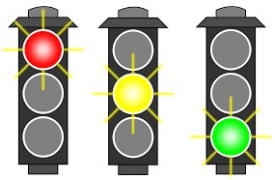
```
int i; /* unique process id */
while (true)  {
   choosing[i]=true;
   number[i]=max(number[0],…,
             number[n-1])+1;
   choosing[i]=false;
   for (j=0; j<n; j++) {
      while (choosing[j])
       ;   /* do nothing */
      while (number[j]>0 &&
         (number[j]<number[i] ||
          number[j]==number[i] &&
          j<i))
       ;   /do nothing */
   }
   CS
   number[i] = 0;
   non-CS
}
```

- bakery alg. is a generalization of Peterson's alg. for **n** processes.
- each process has unique identifier
- the process takes a number when CS is needed; in case of a tie process ids are used
- the process with the lowest number is allowed to proceed
- why does it check that **choosing** is not set when trying to access CS?

39

39

Semaphores



40

Semaphore Idea

- **Dijkstra 1965. The fundamental principle** :
- Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place (**wait**) until it has received a specific **signal**.
- Any complex coordination requirement can be satisfied by the appropriate structure of signals.
- For signaling, special variables called **semaphores** are used.
- To transmit a signal via semaphore **s** , a process executes the primitive **semSignal**(s).
- To receive a signal via semaphore s , a process executes the primitive **semWait**(s) ;
- if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.
- Original operation names are: (V for Signal, and P for Wait). Read the story in literature.

41

41

Semaphore Operations

To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which only three operations are defined:

1. A semaphore may be initialized to a nonnegative integer value.

2. The semWait operation decrements the semaphore value. If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution.

3. The semSignal operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

- **Notes**:

✓ The semWait and semSignal primitives are assumed to be atomic.

✓ Other than these three operations, there is no way to inspect or manipulate Semaphores.

✓ In general, there is no way to know before a process decrements a semaphore whether it will block or not.

✓ After a process increments a semaphore and another process gets woken up, both processes continue running concurrently. There is no way to know which process, if either, will continue immediately on a uniprocessor system.

✓ When you signal a semaphore, you don't necessarily know whether another process is waiting, so the number of unblocked processes may be zero or one.

42

42

A Definition of Semaphore Primitives
(general semaphore)

```
struct semaphore {
        int count;
        queueType queue;
};
void semWait(semaphore s)
{
        s.count--;
        if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
        }
}
void semSignal(semaphore s)
{
        s.count++;
        if (s.count<= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
        }
}
```

43

43

Binary Semaphore

- A binary semaphore may only take on the values 0 and 1 and can be defined by the following three operations:

1. A binary semaphore may be initialized to 0 or 1.
2. The semWaitB operation checks the semaphore value. If the value is **zero**, then the process executing the semWaitB is blocked. If the value is **one**, then the value is changed to **zero** and the process continues execution.
3. The semSignalB operation checks to see if any processes are blocked on this semaphore (semaphore value equals 0). If so, then a process blocked by a semWaitB operation is unblocked. If no processes are blocked, then the value of the semaphore is set to **one**.

- Note: the nonbinary semaphore is often referred to as either a **counting semaphore** or a **general semaphore**

44

44

Strong/Weak Semaphores

😐A queue is used to hold processes waiting on the semaphore

Strong Semaphores

- the process that has been blocked the longest is released from the queue first (FIFO)

Weak Semaphores

- the order in which processes are removed from the queue is not specified

45

Definition of Binary Semaphore Primitives

```
struct binary_semaphore {
       enum {zero, one} value;
       queueType queue;
};
void semWaitB(binary_semaphore s)
{
       if (s.value == one)
            s.value = zero;
       else {
                    /* place this process in s.queue */;
                    /* block this process */;
       }
}
void semSignalB(semaphore s)
{
       if (s.queue is empty())
            s.value = one;
       else {
                    /* remove a process P from s.queue */;
                    /* place process P on ready list */;
       }
}
```

46

46

Mutual Exclusion Using Semaphores

- The semaphore is initialized to 1. Thus, the first process that executes a semWait will be able to enter the critical section immediately, setting the value of s to 0.
- Any other process attempting to enter the critical section will find it busy and will be blocked, setting the value of s to –1.
- Any number of processes may attempt entry; each such unsuccessful attempt results in a further decrement of the value of s .
- When the process that initially entered its critical section departs, s is incremented and one of the blocked processes (if any) is removed from the queue of blocked processes associated with the semaphore and put in a Ready state.
- When it is next scheduled by the OS, it may enter the critical section.

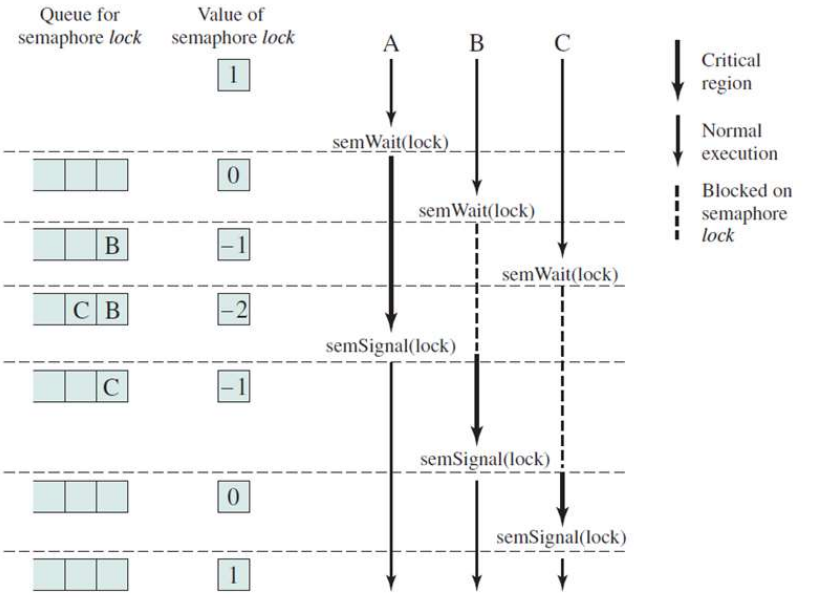
```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section */;
          semSignal(s);
          /* remainder */;
     }
}
void main()
{
     parbegin (P(1), P(2),…, P(n));
}
```

47

47

Processes Accessing Shared Data Protected by a Semaphore

Note that normal execution can proceed in parallel but that critical regions are serialized



48

48

"Too much milk" with semaphores

Too much milk ///(V for semSignalB, and P for semWaitB)

Thread A	Thread B
`P(fridge);`	`P(fridge);`
`if (noMilk){`	`if (noMilk){`
`buy milk;`	`buy milk;`
`noMilk=false;`	`noMilk=false;`
`}`	`}`
`V(fridge);`	`V(fridge);`

- "fridge" is a semaphore initialized to 1, noMilk is a shared variable

Execution:

After:	s	queue	A	B
	1			
A: `P(fridge);`	0		in CS	
B: `P(fridge);`	**-1**	**B**	**in CS**	**waiting**
A: `V(fridge);`	0		finish	ready, in CS
B: `V(fridge);`	**1**			**finish**

49

Producer/Consumer Problem

General Situation:

- one or more producers are generating data and placing these in a buffer
- a single consumer is taking items out of the buffer one at time
- only one producer or consumer may access the buffer at any one time

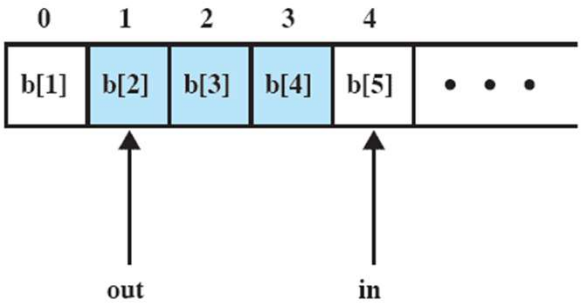
The Problem:

- ensure that the producer can't add data into full buffer and consumer can't remove data from an empty buffer

50

Buffer Structure

Infinite Buffer for the Producer/Consumer Problem



51

Producer/consumer problem (bounded Buffer)

```
int p, c, buff[B],
front=0, rear=0;
semaphore empty(B),
      full(0),

producer() {
   while (true) {
   /* produce p */
   wait(empty);
   buff[rear]=p;
   rear=(rear+1) % B;
   signal(full);
   }
}

consumer () {
   while (true) {
   wait(full);
   c=buff[front];
   front=(front+1) % B;
   signal(empty);
   /* consume c */
   }
}
```

- bounded **buff** holds items added by **producer** and removed by **consumer**. **producer** blocks when buffer is full; **consumer** – when empty
- this variant – single producer, single consumer
- **p** - item generated by producer
- **c** - item utilized by consumer
- **empty** - if open - **producer** may proceed
- **full** - if open - **consumer** may proceed

52

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

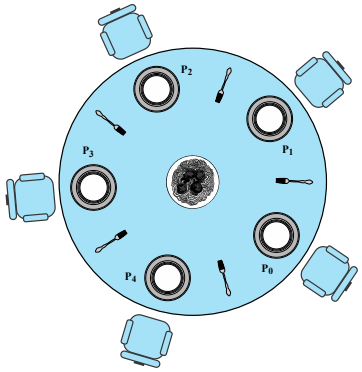
Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

53

53

Dining Philosophers Problem

- Five philosophers sit at a circular table doing one of two things: ***eating*** or ***thinking***.
- While eating, they are not thinking, and while thinking, they are not eating.
- This spaghetti is difficult to serve and eat with a single fork, so each philosopher **must acquire two forks to eat**.
- Each philosopher **can only use the forks on their immediate left and right**.
- Each philosopher can acquire forks only **one at a time**.



- ✓ No two philosophers can use the same fork at the same time (**mutual exclusion**)
- ✓ No philosopher must starve to death
  - (avoid deadlock and starvation)

54

A First Solution to the Dining Philosophers Problem using semaphores

```
/* program     diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```

This solution, alas, leads to deadlock: If all of the philosophers are hungry at the same time, they all sit down, they all pick up the fork on their left, and they all reach out for the other fork, which is not there. In this undignified position, all philosophers starve.

55

A Second Solution to the Dining Philosophers Problem using semaphores

```
/*program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
     think();
     wait (room);
     wait (fork[i]);
     wait (fork [(i+1) mod 5]);
     eat();
     signal (fork [(i+1) mod 5]);
     signal (fork[i]);
     signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

This solution is free of deadlock and starvation.

56

Solution for Philosophers' dining table

```
static void Eat(Semaphore numEating, Semaphore leftFork, Semaphore rightFork)
{
    SemaphoreWait(numEating);   // wait until can try to get forks
    SemaphoreWait(leftFork);    // get left
    SemaphoreWait(rightFork);   // get right
    printf("%s eating!\n", ThreadName());
    RandomDelay(5000,25000);    // "eat" for random time
    SemaphoreSignal(leftFork); // let go
    SemaphoreSignal(rightFork);
    SemaphoreSignal(numEating);
}
```

57

57

Mutex

C/C++ Solutions (example using Mutex Object)

- **Mutex** is a program object that provides Mutual Exclusion. That is, it is created so that multiple program thread can take turns sharing the same resource, such as access to a shared variable or a file.
- Make sure to include the <mutex> library. #include <mutex> // std::mutex
- For full version, please see **Piazza.**

```
void parallel_Matrix_Sum(int from, int to) {
    for (int draw = from; draw <= to; ++draw) {
        for (int dcolumn = 0; dcolumn < matrixC.size(); ++dcolumn) {
            mtx.lock();    // one thread at a time. Thus, critical section
            sum = sum + matrixC[draw][dcolumn]; // No Race condition, critical section
            mtx.unlock();
        }
    }
}

//Advantages (Safety and correctness). Practical from correctness perspective.
//Problems you need to pay attention to:
//Overhead of having the threads to wait. But, no other choice.
//Do an experiment. (compare sequential version to parallel with Mutex object)
```

58

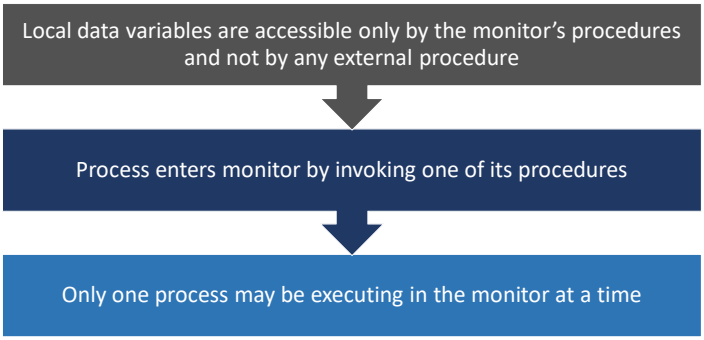
58

Monitors

- *Monitors* are a high-level data abstraction tool combining three features:
  - Shared data.
  - Operations on the data.
  - Synchronization, scheduling.
- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
  - Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java
- Has also been implemented as a program library
- Software module consisting of one or more procedures, an initialization sequence, and local data

59

Monitor Characteristics



- ✓ a shared data structure can be protected by placing it in a monitor.
- ✓ To be useful for concurrent processing, the monitor must include synchronization tools.
  - For example, suppose a process invokes the monitor and, while in the monitor, must be blocked until some condition is satisfied. The monitor must be released so that some other process may enter it.

60

Java only allows one condition variable (implicit) per object.
Here is the same solution in Java:

```
class QueueHandler {
   final static int BUFFSIZE = 200;
   private int first;
   private int last;
   private int buff[BUFFSIZE];

   private int ModIncr(int v) {
      return (v+1)%BUFFSIZE;
   }

   public QueueHandler (int val)
   {
      first = last = 0;
   }

   public synchronized void AddToQueue (int val) {
   {
      while (ModIncr(last) == first) {
         try { wait(); }
         catch (InterruptedException e) {}
      }
      buff[last] = val;
      last = ModIncr(last);
      notify();
   } //end of AddToQueue
   public synchronized int RemoveFromQueue ();
   {
      while (first == last) {
         try { wait(); }
         catch (InterruptedException e) {}
      }
      int ret = buff[first];
      first = ModIncr(first);
      notify();
      return ret;
   } // end of RemoveFromQueue
} //end of class QueueHandler
```

61

61

A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                  /* buffer pointers */
int count;                                  /* number of items in buffer */
cond notfull, notempty;          /* condition variables for synchronization */

void append (char x)
{
     if (count == N) cwait(notfull);      /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);                       /* resume any waiting consumer */
}
void take (char x)
{
     if (count == 0) cwait(notempty);   /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                   /* one fewer item in buffer */
     csignal(notfull);                        /* resume any waiting producer */
}
{                                                          /* monitor body */
     nextin = 0; nextout = 0; count = 0;          /* buffer initially empty */
}
```

```
void producer()
{
     char x;
     while (true) {
     produce(x);
     append(x);
     }
}
void consumer()
{
     char x;
     while (true) {
       take(x);
       consume(x);
     }
}
void main()
{
     parbegin (producer, consumer);
}
```


62

A Solution to the Dining Philosophers Problem Using a Monitor

```
monitor dining_controller;
cond ForkReady[5];         /* condition variable for synchronization */
boolean fork[5] = {true};       /* availability status of each fork */

void get forks(int pid)          /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork[left])
      cwait(ForkReady[left]);         /* queue on condition variable */
   fork[left] = false;
   /*grant the right fork*/
   if (!fork[right])
      cwait(ForkReady[right]);        /* queue on condition variable */
   fork[right] = false:
}
void release forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork[left] = true;
   else                    /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork[right] = true;
   else                    /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

```
void philosopher[k=0 to 4]           /* the five philosopher clients */
{
   while (true) {
      <think>;
      get_forks(k);         /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);     /* client releases forks via the monitor */
   }
}
```

63

Condition variables

Condition variables: things to wait on. Two types:
(1) classic Hoare condition variables and (2) Java condition variables.

Hoare condition variables:
•**condition.wait():** release monitor lock, put process to sleep. When process wakes up again, re-acquire monitor lock immediately.
•**condition.notify():** wake up one process waiting on the condition variable (FIFO). If no process is waiting then do nothing.
•**condition.broadcast():** wake up all processes waiting on the condition variable. If no process is waiting then do nothing.

Java condition variables:
•**wait():** release monitor lock on current object; put thread to sleep.
•**notify():** wake up one process waiting on the condition; this process will try to reacquire the monitor lock. If no process is waiting then do nothing.
•**notifyall():** wake up all processes waiting on the condition; each process will try to reacquire the monitor lock. (Of course, only one at a time will acquire the lock.) If no process is waiting then do nothing.

64

64

Monitors Summary

- ✓ Was not present in very many languages, but extremely useful. Java made monitors *much* more popular and well known.
- ✓ Semaphores use a single structure for both exclusion and scheduling, monitors use different structures for each.
- ✓ A mechanism similar to wait/notify is used internally to Unix for scheduling OS processes.
- ✓ Monitors are **more** than just a synchronization mechanism. Basing an operating system on them is an important decision about the structure of the entire system.

65

65

Thank you

66

66

References

- Some of the materials and slides are from:
  - Modern Operating Systems (4th Edition)
    - Book, by Andrew S. Tanenbaum (Author), Herbert Bos
  - Operating System Concepts – 9th Edition (Book)
    - Book by Abraham Silberschatz
  - Operating Systems: Principles and Practice– 2nd Edition (Book)
    - Book by Thomas Anderson
  - Operating Systems: Internals and Design Principles 7th Edition
  - Book, by William Stallings
- Some slides in this presentation are taken from Dr. Mikhail Nesterenko's
  - Operating System 2012 class presentations
  - My old school
  - Permission was guaranteed ☺

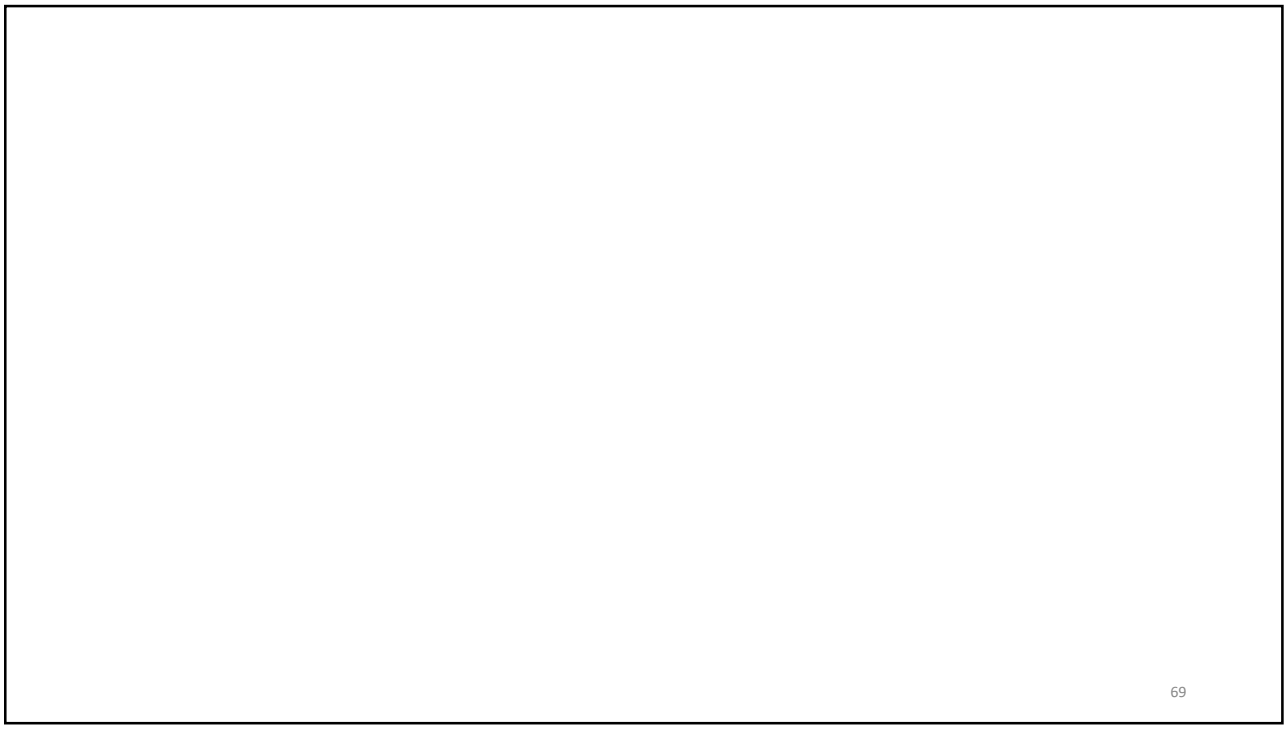
67

67

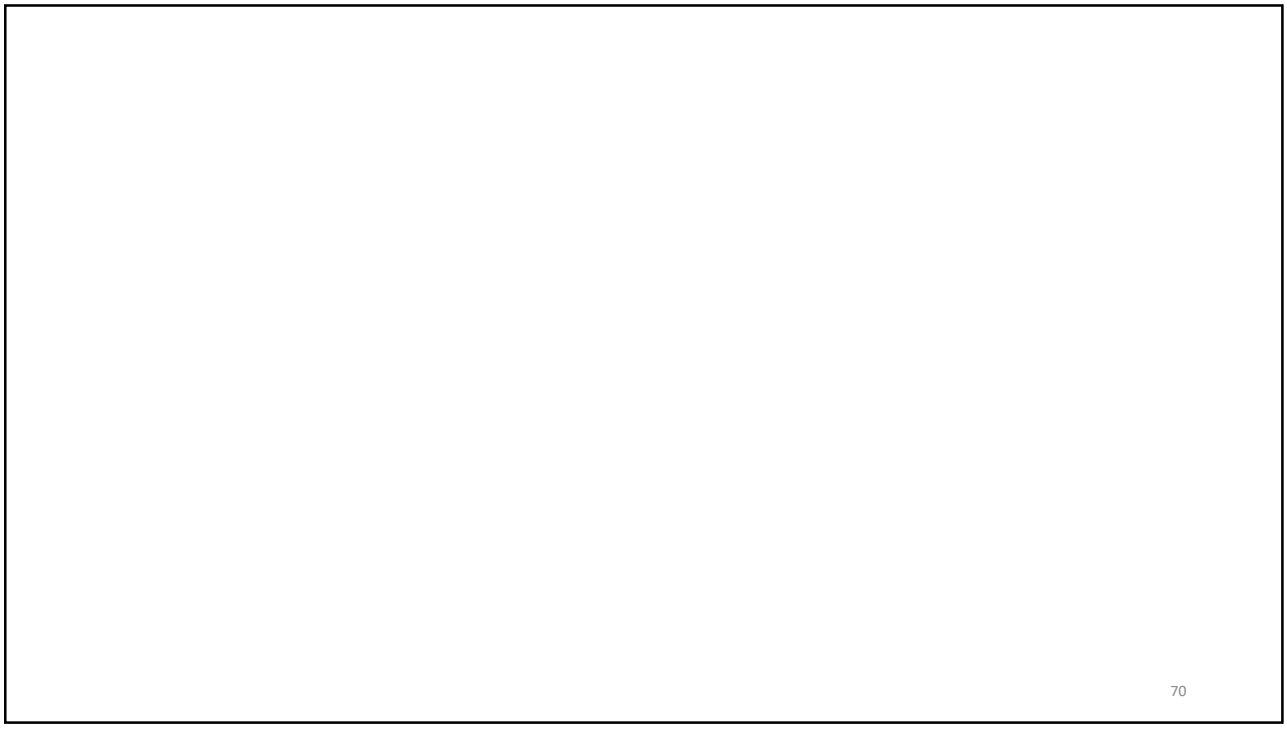
References 2

- Dr. Saleh Alnaeli, The easiest and fastest path to CS.
  - IBM.com
- Object-Oriented Data Structures Using Java (3rd edition),
  - by Dale, Joyce, and Weems (code and slides)
- Algorithm Design: Foundations, Analysis, and Internet Examples
  - by Michael T. Goodrich, Roberto Tamassia
- Data Structures and Algorithms in Java 6/E
  - by Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser
  - {slides, text, and code}
- Oracle, Java Documentation
  - https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html
- Intel (pictures, slides 6 and 7)
  - Intel.com
- OS Course, Barton P. Miller, UW-Madison

68



69



70