



Synchronization and Deadlocks

1



Projects

- If you are taking this course for 4 credits, we shall set up a bi-weekly one hour group meeting to discuss project progress.
 - Please send me your available times
 - Please use "CS 423 PROJECT MEETING" as subject line

2



Synchronization Questions

- How would you implement a monitor using mutexes?
 - A monitor exports a set of function calls and imposes the restriction that only one thread can execute inside it (i.e., be inside one of these calls) at any given time.

3



Synchronization Questions

- How would you implement rendezvous using mutexes?
 - The first thread that reaches the rendezvous point blocks waiting for the other thread. Once both reach the rendezvous point, they can proceed.

4



Synchronization Questions

- How would you implement barriers using mutexes?
 - Threads that reach a barrier block until all threads (in a group) have reached the barrier. Once all threads reach the barrier, they can proceed.

5



Synchronization Questions

- How would you implement read/write locks using mutexes?
 - A write operation cannot proceed concurrently with any other write or read operation
 - A read operation can proceed concurrently with other reads but not with write

6



The Mutex Implementation

- Lock (X)
 - Disable interrupts
 - If X is already locked
 - Put thread into mutex queue (do not put it in the scheduler ready queue)
 - Call scheduler to find next thread
 - Swap context
 - Else
 - Mark X as locked
 - Enable interrupts

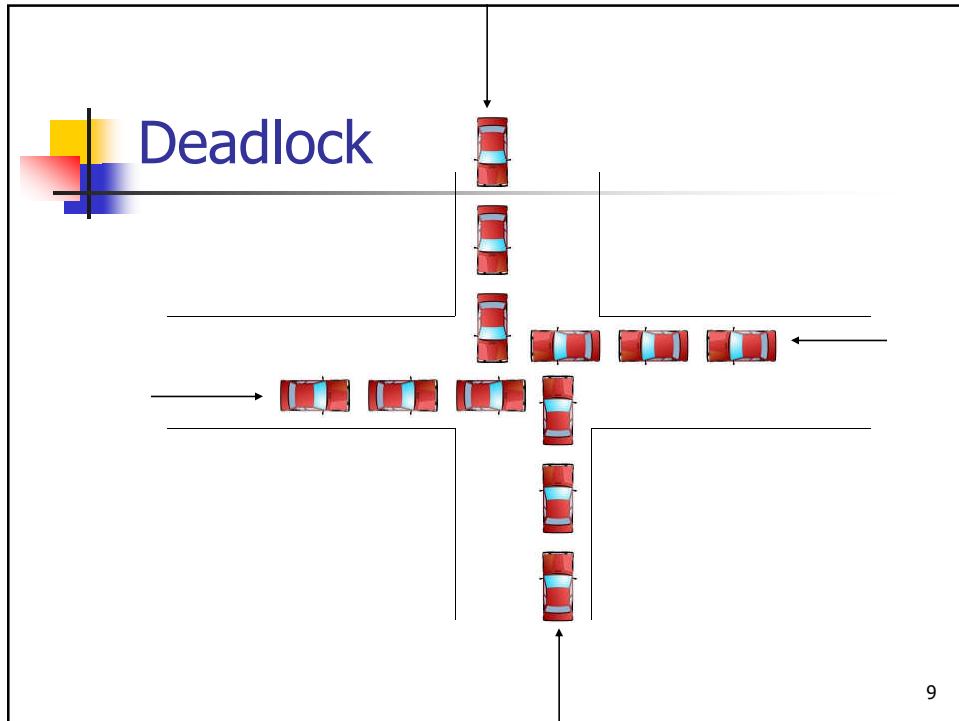
7



The Mutex Implementation

- Unlock (X)
 - Disable interrupts
 - If X is locked
 - If mutex queue is not empty
 - Dequeue next thread and put it in ready queue
 - Call scheduler to find next thread
 - If (next thread is not current thread) Swap context
 - Else mark X unlocked
 - Enable interrupts

8



Requirements for Deadlock

- What are they?

10



Requirements for Deadlock

- **Mutual exclusion**
 - Processes claim **exclusive** control of the resources they require
- **Hold-and-wait (a.k.a. wait-for) condition**
 - Processes hold resources already allocated to them while waiting for additional resources
- **No preemption condition**
 - Resources cannot be removed from the processes holding them until used to completion
- **Circular wait condition**
 - A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain

11



Deadlock Issues

- **Prevention**
 - design a system in such a way that deadlocks cannot occur, at least with respect to serially reusable resources.
- **Avoidance**
 - impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.
- **Detection**
 - in a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved.
- **Recovery**
 - after a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying the affected processes and starting them over.

12

Prevention: Mutual Exclusion

- Processes claim **exclusive** control of the resources they require
- How to break it?



13

Mutual Exclusion

- Processes claim **exclusive** control of the resources they require
- How to break it?
 - You are required to support mutual exclusion in MP2



14



Prevention: Wait-for Condition

- Processes hold resources already allocated to them while waiting for additional resources

- How to break it?

15



Wait-for Condition

- Processes hold resources already allocated to them while waiting for additional resources

- How to break it?
 - You are required to allow processes to lock individual resources at different times in MP2

16



Prevention: No Preemption Condition

- Resources cannot be removed from the processes holding them until used to completion
- How to break it?
 - You are required to enforce sequential access to critical sections in MP2

17



Prevention: Circular Wait Condition

- A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain
- How to break it?

18



Acquire locks in Ascending Order

- If I am holding lockID X, can acquire only lockID Y, $Y > X$.
 - Why does this prevent circular wait?

19



Deadlock Issues

- **Prevention**
 - design a system in such a way that deadlocks cannot occur, at least with respect to serially reusable resources.
- **Avoidance**
 - impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.
- **Detection**
 - in a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved.
- **Recovery**
 - after a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying the affected processes and starting them over.

20



Deadlock Issues

- **Prevention**
 - design a system in such a way that deadlocks cannot occur, at least with respect to serially reusable resources.
- **Avoidance**
 - impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.
- **Detection**
 - in a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved.
- **Recovery**
 - after a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying the affected processes and starting them over.

21



Avoidance

- **Avoidance:** impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches

22



Resource Allocation Graph

- **Nodes**
 - Processes
 - Resources
- **Arcs**
 - From lock to process = **lock assigned to (i.e., held by) process**
 - From process to resource = **process waits for (i.e., blocked on) lock**
- **Avoid circular wait**

23



The Mutex Implementation

- Lock (X)
 - Disable interrupts
 - If X is already locked
 - Put thread into mutex queue (do not put it in the scheduler ready queue)
 - Call scheduler to find next thread
 - Swap context
 - Else
 - Mark X as locked
 - Enable interrupts

24



The Mutex Implementation

- Lock (X)
 - Disable interrupts
 - If circular wait then set error code
 - Else
 - If X is already locked
 - Put thread into mutex queue (do not put it in the scheduler ready queue)
 - Call scheduler to find next thread
 - Swap context
 - Else
 - Mark X as locked
 - Update wait-for graph
 - Enable interrupts

25



The Mutex Implementation

- Unlock (X)
 - Disable interrupts
 - If X is locked
 - If mutex queue is not empty
 - Dequeue next thread and put it in ready queue
 - Call scheduler to find next thread
 - If (next thread is not current thread) Swap context
 - Else mark X unlocked
 - Update wait-for graph
 - Enable interrupts

26

Example Wait-for Graph

A diagram illustrating a wait-for graph. It features a central node labeled "Lock R" and a node labeled "Process A". A curved arrow points from "Lock R" to "Process A", indicating that Process A is the holder of Lock R.

- Process A executes lock (R)
 - A gets R

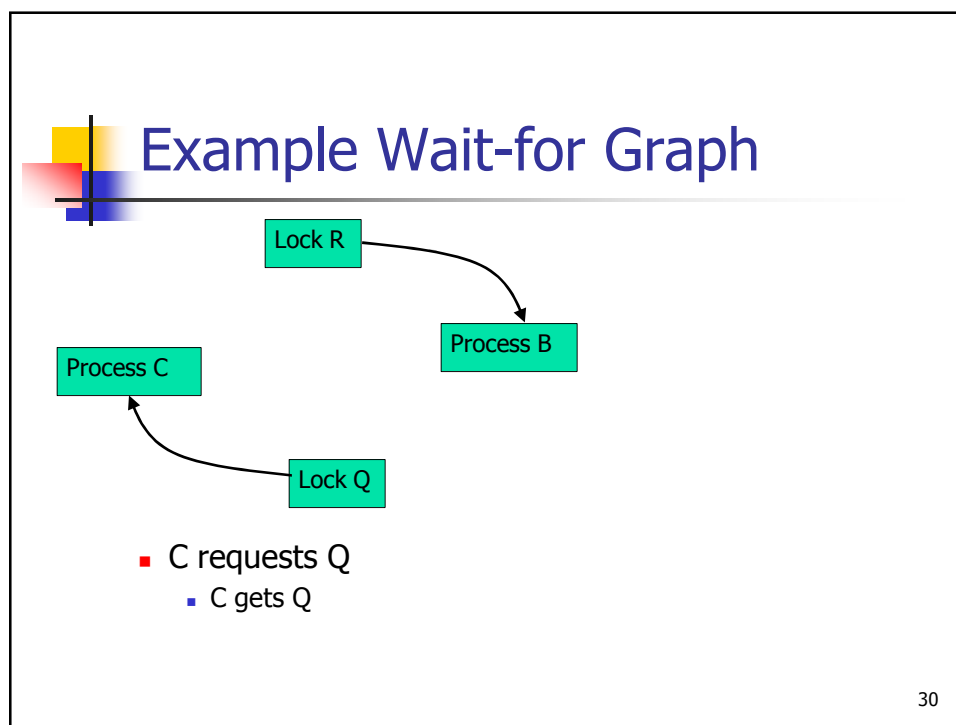
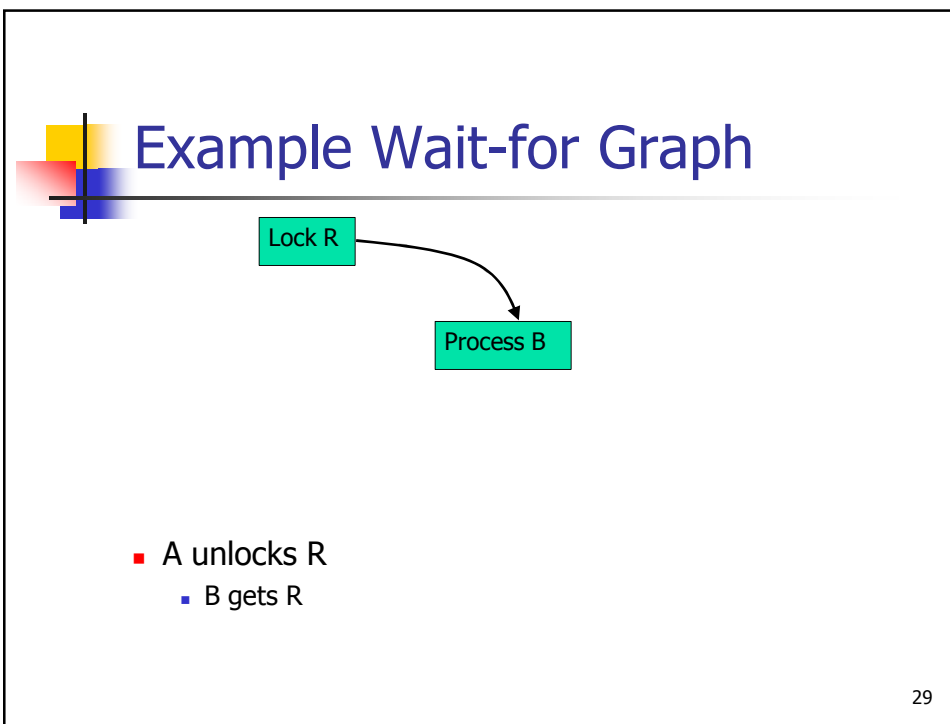
27

Example Wait-for Graph

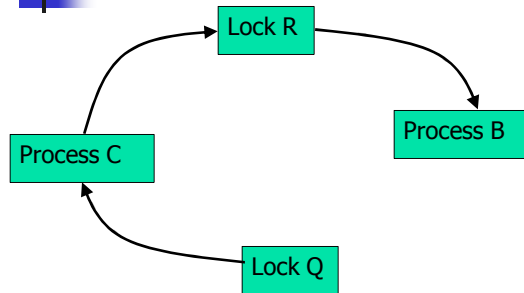
A diagram illustrating a wait-for graph. It features a central node labeled "Lock R", a node labeled "Process A", and a node labeled "Process B". A curved arrow points from "Lock R" to "Process A", indicating that Process A is the holder of Lock R. Another curved arrow points from "Process B" to "Lock R", indicating that Process B is requesting Lock R.

- Process B is requests lock R
 - B blocks waiting for R

28



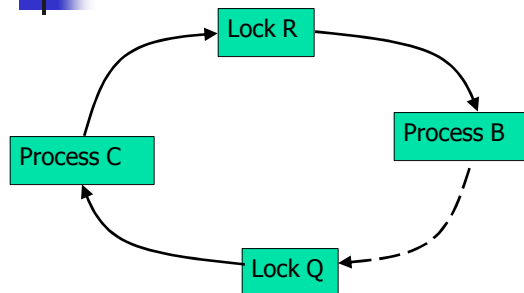
Example Wait-for Graph



- C requests R
- C blocks on R

31

Example Wait-for Graph



- B requests Q
- B is denied Q

32



Deadlock Issues

- **Prevention**
 - design a system in such a way that deadlocks cannot occur, at least with respect to serially reusable resources.
- **Avoidance**
 - impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.
- **Detection**
 - in a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved.
- **Recovery**
 - after a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying the affected processes and starting them over.

33



Deadlock Issues

- **Prevention**
 - design a system in such a way that deadlocks cannot occur, at least with respect to serially reusable resources.
- **Avoidance**
 - impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.
- **Detection**
 - in a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved.
- **Recovery**
 - after a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying the affected processes and starting them over.

34



Deadlock Detection

- Check to see if a deadlock has occurred!
- Single resource per type
 - Can use wait-for graph
 - Check for cycles

35



Deadlock Issues

- **Prevention**
 - design a system in such a way that deadlocks cannot occur, at least with respect to serially reusable resources.
- **Avoidance**
 - impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.
- **Detection**
 - in a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved.
- **Recovery**
 - after a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying the affected processes and starting them over.

36



Deadlock Issues

- **Prevention**
 - design a system in such a way that deadlocks cannot occur, at least with respect to serially reusable resources.
- **Avoidance**
 - impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.
- **Detection**
 - in a system that allows the possibility of deadlock, determine if deadlock has occurred, and which processes and resources are involved.
- **Recovery**
 - after a deadlock has been detected, clear the problem, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying the affected processes and starting them over.

37



Recovery From Deadlock

- **OPTIONS:**
 - Kill deadlocked processes and release resources
 - Kill one deadlocked process at a time and release its resources
 - Rollback all or one of the processes to a checkpoint that occurred before they requested any resources

38