CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Fall 2025 L12
Synchronization (Chap 6, 7)

Slides based on

- **Text by Silberschatz, Galvin, Gagne**
- **Various sources**

Monitors

Monitor: A high-level abstraction that provides a convenient and effective mechanism for process synchronization

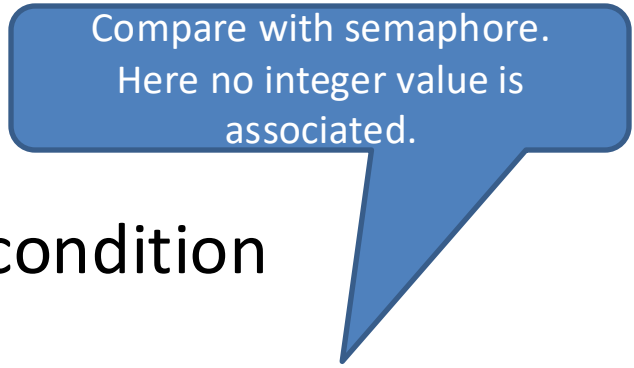
- Abstract data type, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
  - Automatically provide mutual exclusion
  - Implement waiting for conditions
- Queues:
  - for entry
  - for each condition
- Originally proposed for Concurrent Pascal 1975
- Directly supported by Java (see self exercise) but not C

Condition Variables

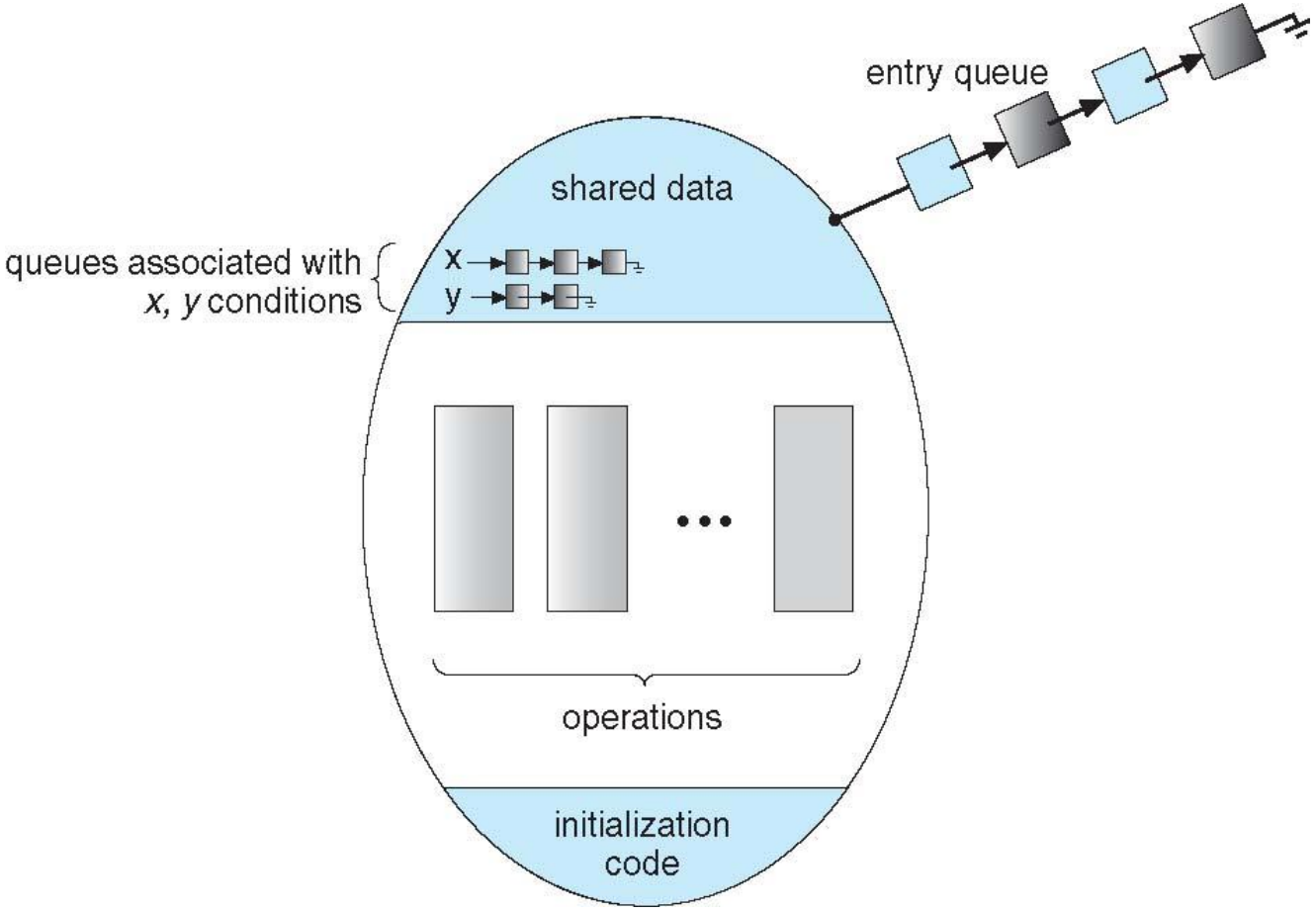
Some actions need some conditions to go ahead.

The **condition** construct

- **condition x, y;**
- Two operations are allowed on a condition variable:
  - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
  - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
    - If no **x.wait()** on the condition variable, then it has no effect on the variable. *Signal is lost.*



Monitor with Condition Variables



Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal ('75) compromise
    - P executing signal immediately leaves the monitor, Q is resumed
    - Implemented in other languages including C#, Java

Monitor Solution to Dining Philosophers: Deadlock-free

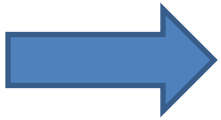
```
enum {THINKING,HUNGRY,EATING} state[5];
```

- `state[i] = EATING` only if
  - `state[(i+4)%5] != EATING && state[(i+1)%5] != EATING`
- `condition self[5]`
  - Delay self when **HUNGRY but unable** to get chopsticks

Sequence of actions

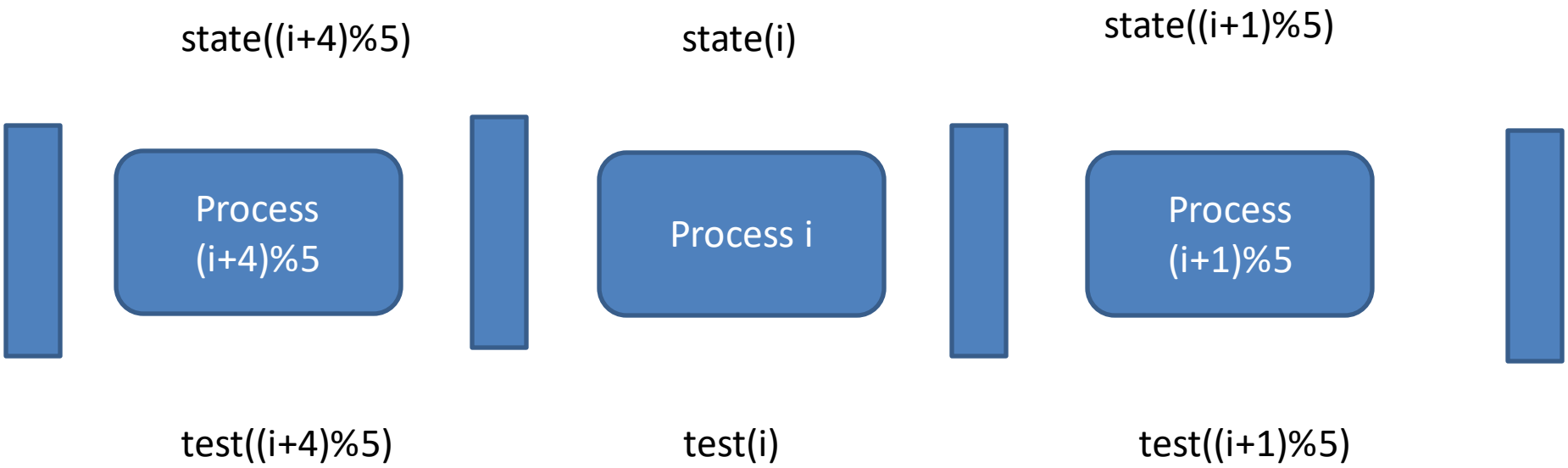
- Before eating, must invoke pickup()
  - May result in suspension of philosopher process
  - After completion of operation, philosopher may eat

```
think
DiningPhilosophers.pickup(i);
eat
DiningPhilosophers.putdown(i);
think
```



Monitor Solution to Dining Philosophers: Deadlock-free

```
enum {THINKING,HUNGRY,EATING} state[5];
```



The pickup() and putdown() operations

```
monitor DiningPhilosophers
{
    enum { THINKING, HUNGRY, EATING} state [5] ;
    condition self [5];

    void pickup (int i) {
            state[i] = HUNGRY;
            test(i);    //below
            if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
            state[i] = THINKING;
                    // test left and right neighbors
             test((i + 4) % 5);
             test((i + 1) % 5);
    }
```

Suspend self if unable to acquire chopstick

Check to see if person on left or right can use the chopstick

```
    void test (int i) {
            if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING ;
               self[i].signal () ;
            }
    }

        initialization_code() {
            for (int i = 0; i < 5; i++)
            state[i] = THINKING;
          }
}
```

Eat only if HUNGRY and Person on Left AND Right are not eating

Signal a process that was suspended while trying to eat

Possibility of starvation

- Philosopher i can starve if eating periods of philosophers on left and right overlap
- Possible solution
  - Introduce new state: STARVING
  - Chopsticks can be picked up if no neighbor is starving
    - Effectively wait for neighbor's neighbor to stop eating
    - REDUCES concurrency!

Monitor Implementation of Mutual Exclusion

For each monitor

- Semaphore mutex initialized to 1
- Process must execute
  - wait(mutex) : Before entering the monitor
  - signal(mutex): Before leaving the monitor

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() is executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next

Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

```
R.acquire(t);
        ...
access the resource;
        ...
R.release;
```

- Where R is an instance of type **ResourceAllocator**
- **A monitor based solution next.**

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
            if (busy)
               x.wait(time);
            busy = TRUE;
   }

   void release() {
            busy = FALSE;
            x.signal();
   }
    initialization code() {
    busy = FALSE;
   }
}
```

Sleep, Time used to prioritize waiting processes

Wakes up one of the processes

Java Synchronization

- For simple synchronization, Java provides the synchronized keyword
  - synchronizing methods

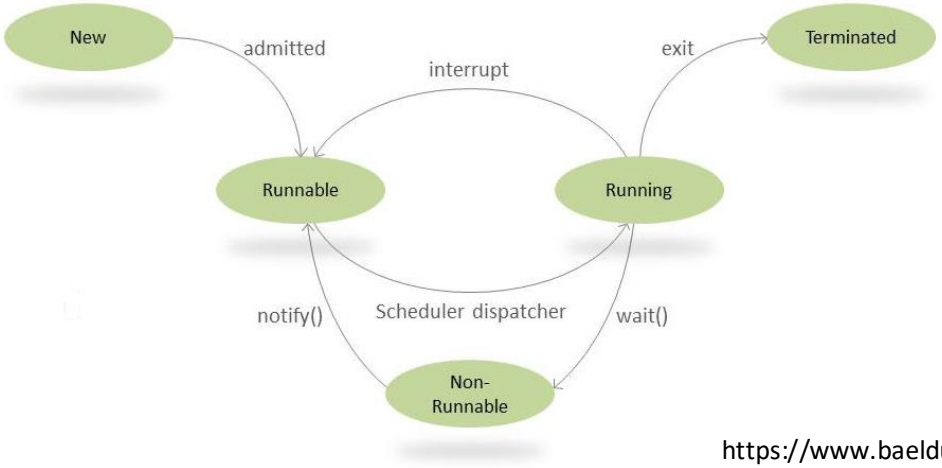
```
public synchronized void increment( ) { c++; }
```

  - synchronizing blocks

```
synchronized(this) {
        lastName = name;
        nameCount++;
}
```

- wait() and notify() allows a thread to wait for an event. A call to notifyAll() allows all threads that are on wait() with the same lock to be notified.
- notify() notifies one thread from a pool of identical threads, notifyAll() when threads have different purposes
- For more sophisticated locking mechanisms, starting from Java 5, the package java.concurrent.locks provides additional capabilities.

Java Synchronization



https://www.baeldung.com/java-wait-notify

Each object automatically has a monitor (mutex) associated with it

- When a method is synchronized, the runtime must obtain the lock on the object's monitor before execution of that method begins (and must release the lock before control returns to the calling code)

wait() and notify() allows a thread to wait for an event.

- wait( ): Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
- notify(): Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened.
- A call to notifyall() allows all threads that are on wait() with the same lock to be released, they will run in sequence according to priority.

Java Synchronization: Dining Philosophers

```
public synchronized void pickup(int i)
   throws InterruptedException {
      setState(i, State.HUNGRY);
      test(i);
      while (state[i] != State.EATING) {
         this.wait();
         // Recheck condition in loop,
         // since we might have been notified
         // when we were still hungry
      }
   }
```

```
private synchronized void test(int i) {
      if (state[left(i)] != State.EATING &&
         state[right(i)] != State.EATING &&
         state[i] == State.HUNGRY)
      {
         setState(i, State.EATING);
         // Wake up all waiting threads
         this.notifyAll();
      }
   }
```

```
public synchronized void putdown(int i) {
      setState(i, State.THINKING);
      test(right(i));
      test(left(i));
   }
```

Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic operations on integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

Transactional Memory

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically without the use of locks.

```
void update(){
        atomic{
        /* modify shared data*/
        }
  }
```

May be implemented by hardware or software.

OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```
void update(int value)
{
      #pragma omp critical
      {
        count += value
      }
}
```

The code contained within the **`#pragma omp critical`** directive is treated as a critical section and performed atomically.

CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Deadlock

Slides based on

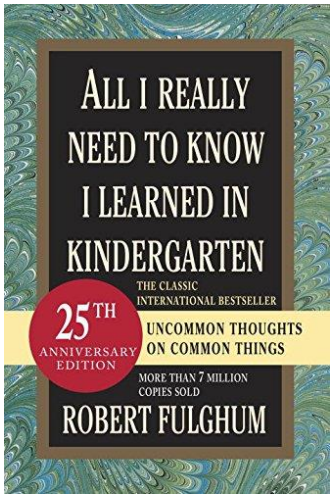
- Text by Silberschatz, Galvin, Gagne
- Various sources

Chapter 8: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance resource-allocation
  - Deadlock Detection
  - Recovery from Deadlock

A Kansas Law

- Early 20th century Kansas Law
  - "When *two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone"*
- *Story of the two silly goats: Aesop 6th cent BCE?*



Colorado State University

A contemporary example

Colorado State University

Deadlock Characterization

Deadlock **can** arise if these **four conditions** hold simultaneously.

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc.
- See example
  - Dining Philosophers: each get the right chopstick first
  - we saw this example earlier

Let *s* and *Q* be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(Q);`	`signal(S);`
`signal(S);`	`signal(Q);`

P0 executes wait(s), P1 executes wait(Q)

P0 must wait till P1 executes signal(Q)

P1 must wait till P0 executes signal(S) Deadlock!

System Model

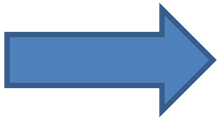
- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Resource-Allocation Graph

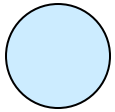
A set of vertices V and a set of edges E.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

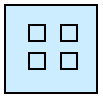


Resource-Allocation Graph (Cont.)

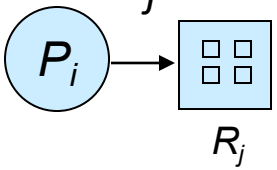
- Process



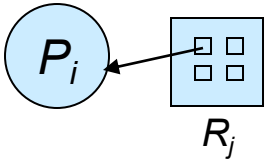
- Resource Type with 4 instances



- P_i requests instance of R_j



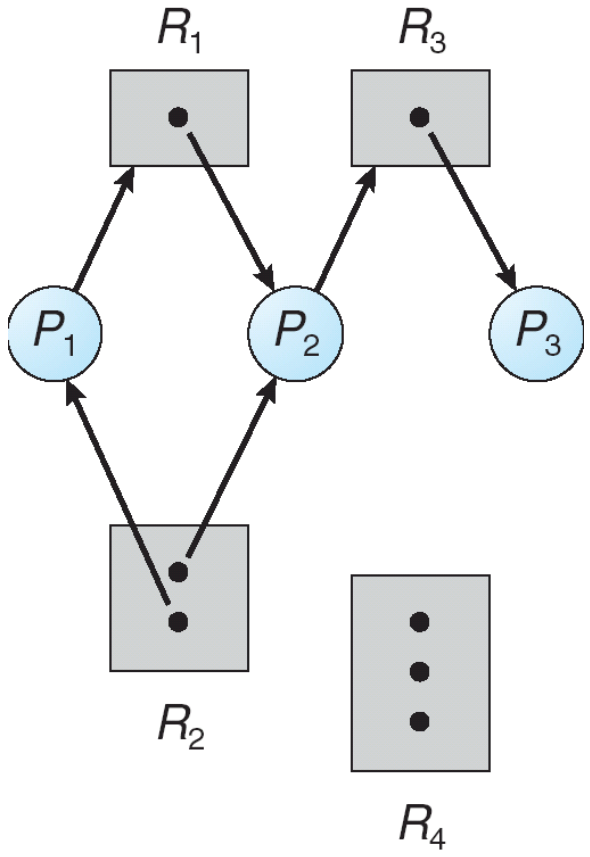
- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

P1 holds an instance of R2, and is requesting R1 ..

If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

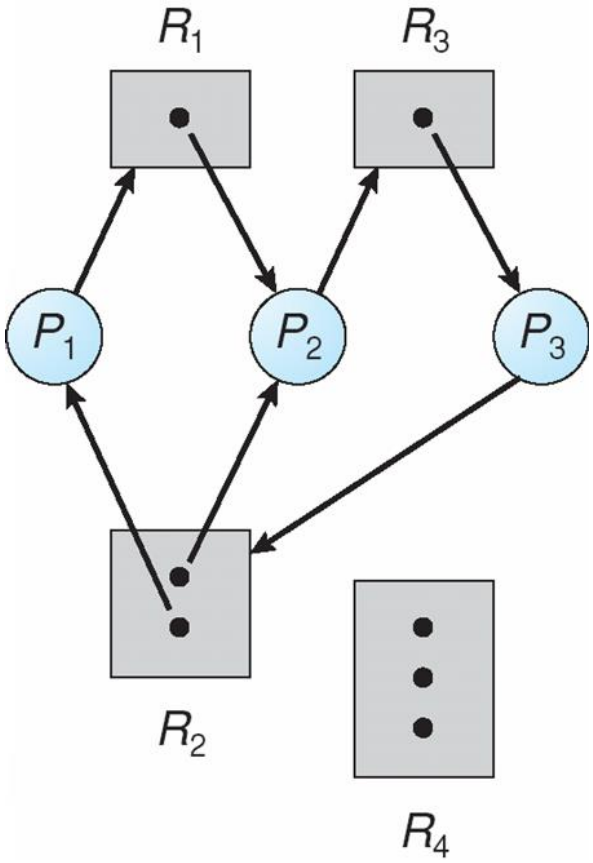


Does a deadlock exist here?

P3 will eventually be done with R3, letting P2 use it.

Thus P2 will be eventually done, releasing R1. …

Resource Allocation Graph With A Deadlock



Does a deadlock exist?

At this point, two minimal cycles exist in the system:

P1→ R1→ P2→ R3→ P3→ R2→ P1

P2→ R3→ P3→ R2→ P2

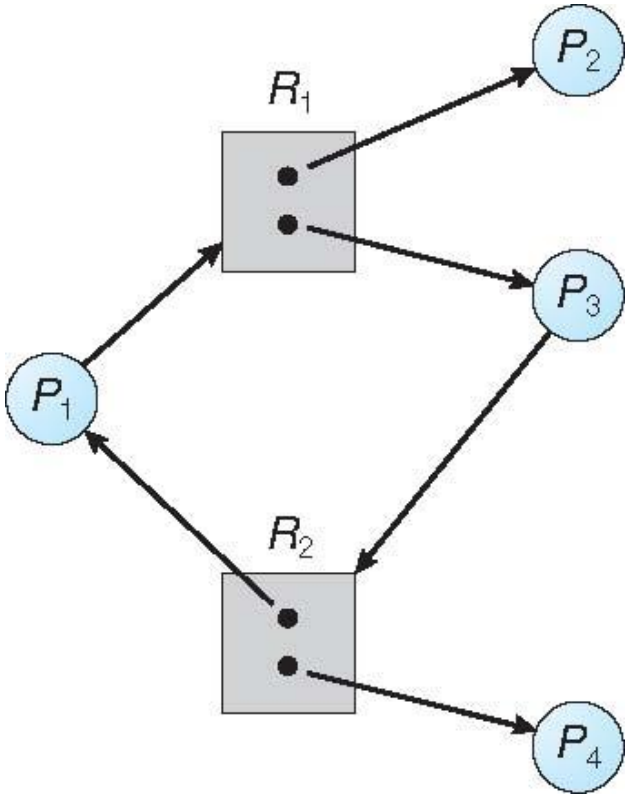
Processes *P1, P2,* and *P3* are deadlocked.

Graph With A Cycle But No Deadlock

Is there a deadlock?

P4 will release its instance of resource type *R2* . That resource can then be allocated to *P3* , breaking the cycle. Thus, there is no deadlock.

If a resource-allocation graph does not have a cycle, then the system is ***not*** in a deadlocked state.
If there is a cycle, then the system may or may not be in a deadlocked state.



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Related classes

Classes that follow CS370

- CS455 Distributed Systems Spring
- CS457 Networks Fall
- CS470 Computer Architecture Spring
- CS475 Parallel Programming Fall
- CS435: Introduction to Big Data Spring

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
    - ensuring that at least one of the 4 conditions cannot hold
  - Deadlock avoidance
    - Dynamically examines the resource-allocation state to ensure that it will never enter an unsafe state, and thus there can never be a circular-wait condition
- Allow the system to enter a deadlock state
  - Detection: detect and then recover. Hope is that it happens rarely.
- Ignore the problem and pretend that deadlocks never occur in the system; used by *most* operating systems, including UNIX. However..

Methods for Handling Deadlocks

- Deterministic: Ensure that the system will ***never*** enter a deadlock state at any cost
- Probabilistic view: Hope it happens rarely. Handle if it happens: Allow the system to enter a deadlock state and then recover.

Methods for Handling Deadlocks

Approach	Resource allocation policy	Scheme	Notes
Prevention	Conservative, undercommits resources	Requesting all resources at once	Good for processes with a single burst of activity
		Preemption	Good when preemption cost is small
		Resource ordering	Compile time enforcement possible
Avoidance	midway	Find at least one safe path (dynamic)	Future max requirement must be known
Detection	Liberal	Invoked periodically	Preemption may be needed

Ostrich algorithm

Ostrich algorithm: Stick your head in the sand; pretend there is no problem at all .

Advantages:

- Cheaper, rarely needed anyway
- Prevention, avoidance, detection and recovery
  - Need to run constantly

Disadvantages:

- Resources held by processes that cannot run
- More and more processes enter deadlocked state
  - When they request more resources
- Deterioration in system performance
  - Requires restart

To be fair to the ostriches, let me say that …

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can ***prevent*** the occurrence of a deadlock.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes that are circularly waiting.

Colorado State University