Comp 104: Operating Systems Concepts

Concurrent Programming & Threads

Today

- Introduction to Concurrent Programming
  - – Threads
  - – Java Threads and Realisation

Concurrent Programming

- Consider a program that resolves an arithmetic expression
- The steps in the calculation are performed serially: instructions are executed one step at a time
  - – every operation within the expression is evaluated in sequence following the order dictated by the programmer (and compiler)
- However, it may be possible to evaluate numerous sub-expressions at the same time in a multiprocessing system

Concurrent Programming

- Consider formula to find one root of a quadratic:

$$x = (-b + \sqrt{(b^2 - 4ac)}) / 2a$$

- This can be split into several operations:

Concurrent operations	*Serial operations*
1. t1 = -b	5. t5 = t3*c
2. t2 = b*b	6. t5 = t2 - t5
3. t3 = 4*a	7. t5 = √t5
4. t4 = 2*a	8. t5 = t1 + t5
	9. x = t5/t4

- In Java, parallel execution is achieved with threads

Exercise

- Identify the parallelism in the following:

1)
```
for i = 1 to 10
        a[i] = a[i] + 1
```

2)
```
for i = 1 to 10
        a[i] = a[i] + a[i – 1]
```

5

Question

- In calculating the formula **ut + ½at²** using maximal concurrency, which of the operations might be computed in parallel?

a) u*t; a/2; t*t
b) u*t; t+½; a*t
c) u+a; t*t
d) u+a; t*t; ½
e) no parallelism is possible for this formula

Answer: a
*u*t; a/2; and t*t – i.e. only those parts of the formula that have no dependencies on other parts of the formula can be run concurrently. Think how the formula could be written in 3-code…*

6

Threads

- A thread can be thought of as a lightweight process
- Threads are created *within* a normal (heavyweight) process
- Example 1: a Web browser
  – one thread for retrieving data from Internet
  – another thread displays text and images
- Example 2: a word processor
  – one thread for display
  – one for reading keystrokes
  – one for spell checking

7

Thread Benefits

- Four major categories:
- Responsiveness: In a multithreaded interactive application a program may be able to continue executing, even if part of it is blocked
  – e.g. in a web browser: user waiting for image to download, but can still interact with another part of the page
- Resource sharing: Threads share memory and resources of the process they belong to, thus we have several threads all within the same address space
- Economy: Threads are more economical to create and context switch as they share the resources of the processes to which they belong
- Utilisation of multiprocessor architectures: In a multiprocessor architecture, where each thread may be running in parallel on a different processor, the benefits of multithreading can be increased

8

Thread Types

- Support for threads may be provided either at the user level, for user threads, or at the kernel level, for kernel threads
- User level: Threads are supported above the kernel and implemented at the user level by a thread library
  - Library provides support for thread creation, scheduling and management, with no support from the kernel
  - User level threads are fast to create and manage
  - But, if kernel is single threaded, one thread performing a blocking system call will cause the entire process to block, even if other threads within the process can run

9

Thread Types

- Kernel level: Threads are supported directly by the OS
  - Thread creation, scheduling and management done by the kernel in the kernel space
  - Slow to create and manage
  - But, since the threads are managed by the kernel, if one thread performs a blocking system call, the kernel can schedule another thread within the application to run
  - In a multiprocessor system, kernel can schedule threads on different processors

10

Java Thread Creation

- When a Java program starts, a single thread is created
  - JVM also has own threads for garbage collection, screen updates, event handling etc.
- New threads may be created by extending the **Thread** class
- Again, threads may be managed directly by kernel, or implemented at user level by a library

The Java Thread Class

- public class Thread extends Object implements Runnable
- A Thread describes a “Run” method that defines what processing will be carried out during the Thread’s lifetime.
- Threads may be started within main(), and run simultaneously, sharing variables, etc.

A Basic Java Thread Class

```
class TwoChar extends Thread {
   private char[2] Out;
   public TwoChar(char First,Second) {
      Out[0]=First; Out[1]=Second;
    };
    public void run() {
       System.out.println(Out[0]);
       System.out.println(Out[1]);
    };
}
```

and how it might be used

```
public class ThreadEx {
  public static void main(String args[]) {
     // Thread declaration
     TwoChar LET = new TwoChar('A','B');
     TwoChar DIG = new TwoChar('1','2');
     LET.start(); DIG.start();
   };
}
```

Thread Methods I

ThreadName.start()

Causes the Thread, Threadname, to begin executing (ie calls the run() method in its specification)

1. There is no limit (other than machine resource) on the total number of Threads that may simultaneously run.
2. Concurrently running threads may access and alter common variables.
3. Because of the potential for undesirable side-effects from (2), support is offered to allow this to happen in a "controlled" style.

Thread Methods II

ThreadName.sleep(int millis)

Causes the Thread, Threadname, to sleep (ie ***temporalily*** stop) executing for millis milliseconds.

Execution resumes from exactly the point where the thread suspended: ie if X.run() contains

y++; sleep(5000); z++;

after y++ and suspension z++ is the next operation performed (the run() method does not restart)

Problem

- Suppose we have an object (called 'thing') which has the following method:

```
public void inc() {
  count = count + 1;
}
```

- Count is private to 'thing', and is initially zero
- Two threads, T1 and T2, both execute the following:

```
thing.inc();
```

Question!!!

- What value will 'count' have afterwards?

Answer

- We don't know!
- This is called indeterminacy
- If T1 executes assignment before T2, or vice-versa, then count will have value 2
- Similarly: what will be the output produced by running ThreadEx the example multi-thread earlier?

Question

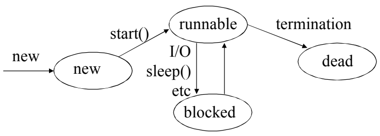
- Which of the following statements about threads is FALSE?

a) A Java program need not necessarily lead to the creation of any threads
b) A thread is sometimes referred to as a lightweight process
c) Threads share code and data access
d) Threads share access to open files
e) Threads are usually more efficient than conventional processes

Answer: a
Every Java program starts as a thread! The rest of the statements are true…

20

Java Thread States



- All threads capable of execution are in the runnable state
  – Includes currently executing thread

Java Thread States

- A Java thread can be in one of four possible states:
- New: when an object for the thread is created (i.e. through use of the ‘new’ statement)
- Runnable: when the thread’s run() method is invoked it moves from the *new* state to the *runnable* state, where it is eligible to be run by the JVM
- Blocked: when performing I/O the thread becomes blocked, and also when it invokes specific Thread methods, such as sleep() (or, as a consequence of invoking suspend(): a method now deprecated)
- Dead: when the thread’s run() method terminates (or when its stop() method is called – stop() is also deprecated), the thread moves to the dead state.

Question

- A Java object called ‘helper’ contains the two methods opposite, where num is an integer variable that is private to helper. Its value is initially 100.
- One thread makes the call
  – `helper.addone();`
- At the same time, another thread makes the call
  – `helper.subone();`
- What value will num have afterwards?

```
public void addone() {
   num = num + 1;
}

public void subone() {
   num = num - 1;
}
```

a) 100
b) 99
c) 101
d) either 99 or 101, but not 100
e) the value of num is undefined

Answer: d
either 99 or 101, but not 100 – if the two threads are run simultaneously, then it depends on the order in which the threads are executed by the ready queue. However, as “num” is not protected by a semaphore, its final value could be either value

23

Comp 104: Operating Systems Concepts

Synchronisation

Today

- Mutual Exclusion
- Synchronisation methods:
  - Semaphores
- Classic synchronisation problems
  - The readers-writers (Producer-Consumer) Problem
  - The Dining Philosophers Problem

Problem

- Suppose we have an object (called ‘thing’) which has the following method:

```
public void inc() {
  count = count + 1;
}
```

- Count is private to ‘thing’, and is initially zero
- Two threads, T1 and T2, both execute the following:

```
thing.inc();
```

Mutual Exclusion

- Indeterminacy arises because of possible simultaneous access to a shared resource
  - The variable ‘count’ in the example
- Solution is to allow only one thread to access ‘count’ at any one time; all others must be excluded
- To control access to such a shared resource we declare the section of code in which the thread/process accesses the resource to be the critical region/section
- We can then regulate access to the critical region
  - When one thread is executing in its critical region, no other thread/process is allowed to execute in its critical region
  - This is known as mutual exclusion

Semaphores

- A semaphore is an integer-valued variable that is used as a flag to signal when a resource is free and can be accessed
- Only two operations possible: wait(S) also called P and signal(S) also called V (from Dutch, *proberen* and *verhogen* – proposed by the late Dutch computer scientist Edsgar Dijkstra)

```
wait(S) {                     signal(S) {
   while (S<=0)                S++;
     ; //null                 }
   S--;
}
```

Semaphores

- wait() and signal() are indivisible
  - When one thread/process modifies the semaphore, no other thread/process can modify that same semaphore
- They can be used to enforce mutual exclusion by enclosing critical regions

```
T1                              T2
wait();                         wait();
// T1/T2 cannot access CR until they control s
    critical region             critical region
signal();               signal();
// once complete "lock" on s must be released
```

Semaphores

- A semaphore that can only take values 0 or 1 is a binary semaphore
  - unrestricted ones are counting semaphores
- When a process/task/thread is in its critical region (controlled by s), no other process (needing s) can enter theirs
  - hence, keep critical regions as small as possible
- Use of semaphores requires care

Question

- The value of a semaphore s is initially 1. What could happen in the following situation?

```
T1                    T2
wait(s);              signal(s);
  critical region       critical region
signal(s);            wait(s);
```

a) Deadlock will ensue
b) T1 and T2 can both enter their critical regions simultaneously
c) Neither T1 nor T2 can enter its critical region
d) T1 can never enter its critical region, but T2 can enter its own
e) T1 can enter its critical region, but T2 can never enter its own

Answer: b
If T1 executes first, then it acquires the semaphore, which is immediately released by T2. Both then execute the critical region.
If T2 executes first, it releases a semaphore it does not have, which can be acquired by T1. Again, both can execute the critical region.

31

Classic Synchronisation Problems

- There are a number of famous problems that characterise the general issue of concurrency control
- These problems are used to test synchronisation schemes
- We will look at two such problems that involve synchronisation issues.

The Producer Consumer

- Synchronisation: The Producer-Consumer Problem
  - Definition
  - Java implementation
  - Issues

The Producer-Consumer Problem

- A producer process (eg secretary) and a consumer process (eg manager) communicate via a buffer (letter tray)
- Producer cycle:
  - produce item (type letter say)
  - deposit in buffer (eg put in tray)
- Consumer cycle
  - extract item from buffer (eg take letter)
  - consume item (eg sign it)
- May have many producers & consumers

Problems to solve

- We have to ensure that:
  - producer cannot put items in buffer if it is full
  - consumer cannot extract items from buffer if it is empty
  - buffer is not accessed by two threads simultaneously

35

Further potential problems

- Deadlock can arise – suppose lock is given to a process “unthinkingly”:
- If the consumer tries to remove an item from an empty buffer, it will have to wait for the buffer to be filled by the producer.
- But the buffer will not be filled as the consumer has the lock.
- Similarly for the producer.

36

Solution by Semaphores

```
class Buffer {
 private int NumberIn=0;
 private boolean full=(Numberin==20);
 private boolean empty=(NumberIn==0)

 public synchronized void insert() {
   while (full) {
     try {
       wait();
     }
     catch (InterruptedException e) {}
   }
   NumberIn++; full=(NumberIn==20);
   empty=false;
   notify();
 }

 // Similarly for remove()
```

Solution by Semaphores

```
public synchronized void remove(){
   while (empty) {
     try {
       wait();
     }
     catch (InterruptedException e) {}
   }
   NumberIn--; empty=(NumberIn==0);
   full=false;
   notify();
  }
}
```

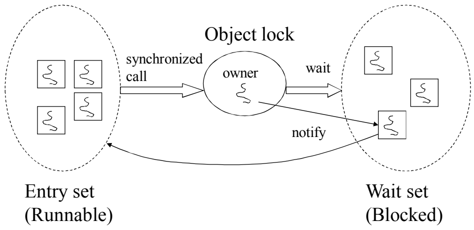
wait(), notify(), notifyAll()

- These are methods, like sleep(mils), that are available to the Thread class.
- The wait() call
  - releases the lock
  - moves the calling thread to the 'wait set'
- The notify() call
  - moves an arbitrary thread from the wait set back to the entry set (this provide implementation of signal()).
- Can use notifyAll() to move all waiting threads back to entry set

synchronized ??

- The insert() and remove() methods are specified as
  public synchronized void
- What does this mean??
- If a method is define as synchronized in Java, then
  AT MOST 1 THREAD CAN ACCESS IT AT ANY TIME
- Hence, if T1 is executing such a method S, then T1 effectively "locks out" any other threads that may invoke S until T1 "releases" it.

Entry and Wait Sets



The Dining Philosophers

- The Producer-Consumer problem models a synchronization environment in which processes with ***distinct roles*** have to coordinate access to a shared facility.
- For example: Managers behave differently to Secretaries; the "shared facility" is the Letter Tray.
- The Dining Philosophers problem models an environment in which all processes have ***identical roles***. Again coordinated access to shared facilities must be arranged.

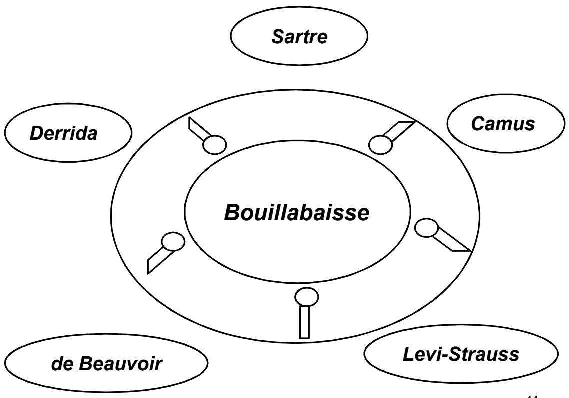
42

General Setting

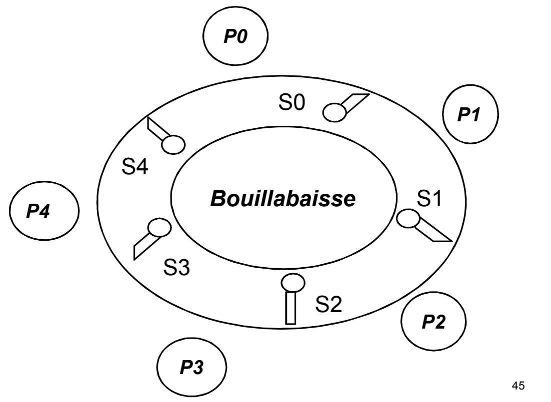
- *n* "philosophers" spend their time seated round a table going through a routine of

  ... – Eat – Think – Eat – Think – Eat – ...
- Philosophers need nothing in order to Think, BUT
- In order to Eat a philosopher must use TWO items of cutlery. (eg 2 spoons).
- Only *n* spoons, however, are provided.
- This means that if a philosopher is hungry BUT either neighbour is eating then she must wait until BOTH spoons are available.

43



44



Solution Approach I

- An abstract setting is on the previous slide.
- Each Philosopher has a unique ***index value*** – {0,1,2,3,4}.
- Similarly each Spoon is indexed from {0,1,2,3,4}.
- The Philosopher with index k must be able to use both the spoons

  k and (k-1)mod 5

  in order to Eat.
- In summary,

46

Solution Approach II

A. P0 uses S0 and S4.
B. P1 uses S1 and S0.
C. P2 uses S2 and S1.
D. P3 uses S3 and S2.
E. P4 uses S4 and S3.

- Therefore: if

1. (P0 & P1) or (P0&P4) are hungry EITHER P0 can eat OR *at most one of* {P1,P4} can.
2. (P1& P0) or (P1&P2) $\Rightarrow$ P1 or $\leq$1 of {P0,P2}
3. (P2&P1) or (P2&P3) $\Rightarrow$ P2 or $\leq$1 of {P1,P3}
4. (P3&P2) or (P3&P4) $\Rightarrow$ P3 or $\leq$1 of {P2,P4}
5. (P4&P3) or (P4&P0) $\Rightarrow$ P4 or $\leq$1 of {P0,P3}

47

Solution by Semaphores

- Associate each Spoon with its own semaphore.
- If s[i] is the (binary) semaphore controlling access to Si, then each philosopher carries out the same basic sequence of actions –

48

Solution by Semaphores

```
public void run() {
   while (alive) {
       spoon[i].get(); spoon[(i-1)%5].get();
       try { sleep(eating_time); } catch
   ....;
             spoon[i].put_down(); spoon[(i-
   1)%5].put_down();
             try { sleep(thinking_time); }
   catch ....; }; }
```

- The methods get() and put_down() are **synchronized** methods in a class associated with spoon.
- Thus,

49

Spoon Class

```
class spoon {
   private boolean in_use = false;
   public synchronized get() {
       while (in_use) {
              try { wait(); } catch {...};
         in_use = true;   // prevent any
   one else accessing         };
      public synchronized put_down() {
         in_use = false;
         notify();          // let anyone
   waiting know spoon free
        };
```

50

Some Problems

Qn. What happens if all philosophers manage to pick the spoon on their left ***simultaneously***?

An. The system will become ***deadlocked***. The spoon on the right will already be taken and never released.

- Can this situation be prevented "cleanly"?
- Yes. A number of approaches are possible.

1. Allow only *n*-1 philosophers to dine at a table with *n* places.
2. Asymmetry: even indices try to pick up spoons using order Right then Left; odd indices use order Left then Right.

51

& Some More Problems

- With some deadlock free solutions (such as schemes which require ***both*** spoons to be picked up ***simultaneously*** or ***neither*** can be used) there may be a further problem: – some philosophers may never eat. Suppose in "both or neither" methods we have:
  - P0: neither S4 nor S0
  - P1: neither S0 nor S1
  - P2: both S1 and S2
  - P3: neither S2 nor S3
  - P4: both S3 and S4

52

Starvation

- When P2 is finished eating releases (S1&S2).
- When P4 is finished (S3&S4) are put down.
- P0 can now pickup (S0&S4)
- P3 can now pick up (S2&S3) – and now
  P0: both S0 and S4 | P1: neither S0 nor S1
  P2: neither S1 nor S2 | P3: both S2 and S3
  P4: neither S3 nor S4
- Notice S1 is unused, but if P2 grabs it as soon as P3 releases S2 then P1 cannot eat.
- In total P1 may starve since either P2 has S2 or P0 has S0 always results.

53

Today

- Deadlock
  - Definition
  - Resource allocation graphs
  - Detecting and dealing with deadlock

54

Deadlock

"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."
-- Kansas law

- A set of processes is deadlocked (in **deadly embrace**) if each process in the set is waiting for an event only another process in the set can cause.
- These events usually relate to resource allocation

55

Resource Allocation

- OS must allocate and share resources sensibly
- Resources may be
  - CPUs
  - Peripheral devices (printers etc.)
  - Memory
  - Files
  - Data
  - Programming objects such as semaphores, object locks etc.
- Usual process/thread sequence is request-use-release
  - Often via system calls

56

Creating Deadlock

- In its simplest form, deadlock will occur in the following situation:
  - process A is granted resource X and then requests resource Y
  - process B is granted resource Y and then requests resource X
  - both resources are non-shareable (e.g. tape drive, printer)
  - both resources are non-preemptible (i.e. cannot be taken away from their owner processes)

57

Question

- Consider the following situation regarding two processes (A and B), and two resources (X and Y):
  - Process A is granted resource X and then requests resource Y.
  - Process B is granted resource Y and then requests resource X.
- Which of the following is (are) true about the potential for deadlock?

I. Deadlock can be avoided by sharing resource Y between the two processes
II. Deadlock can be avoided by taking resource X away from process A
III. Deadlock can be avoided by process B voluntarily giving up its control of resource Y

a) I only
b) I and II only
c) I and III only
d) II and III only
e) I, II and III

Answer: e
I, II and III – as all three options will avoid exclusive ownership of the resources.

58

Resource Allocation Graphs

- Consist of a set of vertices V and a set of edges E
  - V is partitioned into two types:
    - Set of processes, $P = \{P_1, P_2, \dots, P_n\}$
    - Set of resource types, $R = \{R_1, R_2, \dots, R_m\}$
      - e.g. printers
      - Include instances of each type

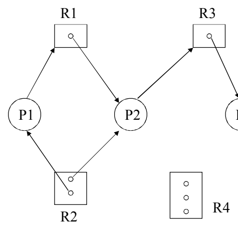
59

Resource Allocation Graphs

- E is a set of directed edges
  - Request edge – from process to resource type, denoted $P_i \rightarrow R_j$
    - States that process P_i has requested an instance of resource type R_j and is currently waiting for it
  - Assignment edge – from resource instance to process, denoted $R_j \rightarrow P_i$
    - States that an instance of a resource type R_j has been allocated to process P_i
  - Request edges are transformed to assignment edges when request satisfied

60

Example Graph



No cycles, so no deadlock.

Example Graph

- The previous diagram depicts the following:
- Processes, resource types, edges
  - $P = \{P_1, P_2, P_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Resource instances:
  - One instance of resource type R_1
  - Two instances of resource type R_2
  - One instance of resource type R_3
  - Three instances of resource type R_4

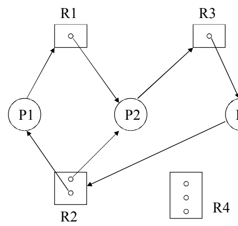
Example Graph

- Process states:
  - Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1
  - Process P_2 is holding an instance of resource type R_1 and R_2 and is waiting for an instance of resource type R_3
  - Process P_3 is holding an instance of resource type R_3

Resource Allocation Graphs

- In resource allocation graphs we can show that deadlock has not occurred if there are no cycles in the graph
- If cycles do exist in the graph, this indicates that deadlock *may* be present
  - If each resource type consists of exactly one instance, a cycle indicates that deadlock has occurred
  - If each resource type consists of several instances, a cycle does not necessarily indicate that deadlock has occurred
- Example: On previous graph, suppose P_3 now requests R_2…

Example Graph (2)



In general, a cycle indicates there *may* be deadlock.

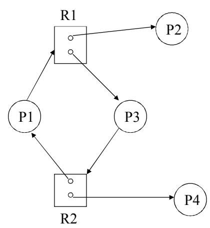
Cycles

- Suppose P_3 now requests R_2…
  - a request edge $P_3 \rightarrow R_2$ is added to the previous graph to show this
- There are now two cycles in the system:

  $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
  $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

- From this we can see that P_1, P_2, and P_3 are deadlocked
- Now consider the following the resource allocation graph…

Another Example



Deadlock?

Dealing with Deadlock

- Prevention
  - Devise a system in which deadlock cannot possibly occur
- Avoidance
  - Make decisions dynamically as to which resource requests can be granted
- Detection and recovery
  - Allow deadlock to occur, then cure it
- Ignore the problem
  - Common approach (e.g. UNIX, JVM)

Exercise

- Why might ignoring the problem of deadlock be a useful approach?

69

Deadlock Prevention

- Techniques
  - Force processes to claim all resources in one operation
    - Problem of under-utilisation
  - Require processes to claim resources in pre-defined order
    - e.g. tape drive before printer always
  - Grant request only if all allocated resources released first
    - e.g. transferring file from tape to disk, then disk to printer

70

Deadlock Avoidance

- Requires information about which resources a process plans to use
- When a request made, system analyses allocation graph to see if it may lead to deadlock
  - If so, process forced to wait
    - Problems of reduced throughput and process starvation

71

Deadlock Avoidance: Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves it in a safe state
- System is in such a state if for the sequence of processes $\langle P_1, P_2, \ldots, P_n \rangle$, for each P_i, the resources that P_i can still request can be satisfied by currently available resources plus the resources held by all the P_j, with $j < i$.
- Thus:
  - If P_i resource requirements are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain its required resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its required resources,
  - … and so on

72

Deadlock Avoidance: Safe State

- If the system is in a safe state there are no deadlocks
- If the system is in an unsafe state, there is the possibility of deadlock
  - an unsafe state may lead to it
- Deadlock avoidance: ensure that the system will never enter an unsafe state
  - Avoidance algorithms make use of this concept of a safe state by ensuring that the system always remains in it

73

Detection and Recovery

- Systems that do not have deadlock prevention or avoidance mechanisms and do not want to ignore the problem must provide the following to deal with deadlock:
  - An algorithm to analyse the state of the system to see if deadlock has occurred
  - A recovery scheme
- Method depends upon whether or not there are multiple instances of each resource type…

74

Detection and Recovery

- If there are multiple instances of a resource type detection algorithms can be used that track:
  - the number of available resources of each type
  - the number of resources of each type allocated to each process
  - the current requests of each process
- If all resources have only a single instance, can make use of a wait-for graph
  - Variant of a resource-allocation graph
  - Obtained from resource allocation graph by removing nodes of type resource and collapsing the appropriate edges

75