Process Synchronization &
Deadlocks

**Background**

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we want to provide a solution to the consumer-producer problem with a buffer of limited capacity.
- We can do so by having an integer `count` that keeps track of the number of full buffers.
  - Initially, `count` is set to 0.
  - It is incremented by the producer after it places a new item and is decremented by the consumer after it consumes an item.
  - If `count` is zero, the buffer is empty. If `count` is equal to the capacity of the buffer, then the buffer is full.
- When several processes operate on shared data concurrently and the outcome of the execution depends on the particular order in which the access takes place, we have what is called a race condition.

**Producer-Consumer**

**Producer:**
```java
while (true) {
    /* produce an item and put in nextProduced */
    ...
    while (count == BUFFER_SIZE); // do nothing while waiting for space
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

**Consumer:**
```java
while (true) {
    while (count == 0); // do nothing while waiting for items
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    /* consume the item in nextConsumed */
    ...
}
```

**race condition**

**Producer-Consumer**

**Producer:**
```java
while (true) {
    /* produce an item and put in nextProduced */
    ...
    while (count == BUFFER_SIZE); // do nothing while waiting for space
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

**Consumer:**
```java
while (true) {
    while (count == 0); // do nothing while waiting for items
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    /* consume the item in nextConsumed */
    ...
}
```
Race Condition

- `count++` is not an atomic operation, it could be implemented as:
  ```
  register1 = count
  register1 = register1 + 1
  count = register1
  ```
- `count--` could be implemented as:
  ```
  register2 = count
  register2 = register2 – 1
  count = register2
  ```
- Consider this execution interleaving with "count = 5" initially:
  ```
  S0: producer execute `register1 = count` {register1 = 5}
  S1: producer execute `register1 = register1 + 1` {register1 = 6}
  S2: consumer execute `register2 = count` {register2 = 5}
  S3: consumer execute `register2 = register2 - 1` {register2 = 4}
  S4: producer execute `count = register1` {count = 6}
  S5: consumer execute `count = register2` {count = 4}
```

Critical Section

- Assume several processes, each containing a particular portion of code in which at most a fixed number (usually one) of them can be executing concurrently.
- Those restricted portions of code are called **critical sections**.
- Note that there might be critical sections in different process which do not conflict with each other.
- A critical section is so only with respect to a conflicting critical section of another processes (or processes).

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections that conflict with the one $P_i$ is in.
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the $N$ processes

Peterson’s Solution:

- **A software-based solution to CS Problem**
  - Not guaranteed to work correctly in modern computer architectures...
  - Assume two processes: P1 and P2
- **Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.**
- **The two processes share two variables:**
  - `int turn;`
  - Boolean `flag[2]`
- **The variable `turn` indicates whose turn it is to enter the critical section.**
- **The flag array is used to indicate if a process is ready to enter the critical section.** $\text{flag}[i] = \text{true}$ implies that process $P_i$ is ready!
Algorithm for Process $P_i$

Assume both processes execute the following code. For $P_1$, the values for $i$ and $j$ are: $i=1$ and $j=2$. For $P_2$, the values for $i$ and $j$ are: $i=2$ and $j=1$.

```c
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j); // busy wait
... critical section
flag[i] = FALSE;
... remainder section
```

Synchronization Hardware

Software-only based solutions to the critical-section problem may not work in modern CPU architectures. Many systems provide hardware support for critical section code.

Uniprocessors – could disable interrupts
- Currently running code would execute without preemption
- May cause other problems, such as hanging the whole system...
- Generally too inefficient on multiprocessor systems
  * Operating systems using this not broadly scalable

Modern machines provide special atomic hardware instructions
- Atomic = non-interruptable
- Either test memory word and set value
- Or swap contents of two memory words

Using Locks to Solve Critical-section Problem

- Any solution to critical-section problem requires some sort of lock.
- The idea is: each set of mutually conflicting critical section is protected by a lock.
- The process that acquires the lock first is the only one allowed to enter critical section protected by the lock.
- Process holding a lock must release it after exit from critical section.
- A lock protects a set of mutually conflicting critical sections that may exist in different processes.
- Solutions based on locks required processes to acquire protecting lock before entering a particular critical section (to begin execution of part of it):

```
acquire lock
... critical section
release lock
```

- Locks can be implemented by combining hw and sw operations…

TestAndSet Instruction

TestAndSet instruction: Atomically sets a target variable to true and returns its current value...

```c
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```
Solution using **TestAndSet**

Shared boolean variable `lock`, initialized to `false`.
Solution:

```c
while ( TestAndSet (&lock )); // do nothing
...   critical section
lock = FALSE;
...   remainder section
```

Solution using **Swap**

Shared boolean variable `lock` initialized to `false`.
Each process has a local boolean variable `key`
Solution:

```c
key = TRUE;
while ( key == TRUE)
    Swap (&lock, &key );
...   critical section
lock = FALSE;
...   remainder section
```

**Swap Instruction**

Atomic `swap` instruction: atomically switch the value in two storage locations.

```c
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

**Semaphore**

- Synchronization tool that does not require busy waiting. Less complicated…
- Semaphore a data type such as:
  ```c
typedef struct {
    int value;   // current value of the semaphore
  } Semaphore;
```
- Two standard operations modify `S`: acquire(&S) and release(&S)
  - Originally called `P()` and `V()`
  - Also called `down()` and `up()`
- Can only be accessed via two indivisible (atomic) operations
  - `acquire(Semaphore *S)`
    ```c
    while (S->value <= 0); // no-op ... busy waiting though...
    (S->value)--;    
    }
    ```
  - `release(Semaphore *S)`
    ```c
    (S->value)++;
    ```
Semaphore

- Synchronization tool that does not require busy waiting. Less complicated…
- Semaphore a data type such as:

```c
typedef struct {
    int value;  // current value of the semaphore
} Semaphore;
```
- Two standard operations modify S: acquire(&S) and release(&S)
  - Originally called P() and V()
  - Also called down() and up()
- Can only be accessed via two indivisible (atomic) operations
  - `acquire(Semaphore *S)` {
    while (S->value <= 0);   // no-op ... busy waiting though...
    (S->value)--;
  }
  - `release(Semaphore *S)` {
    (S->value)++;
  }

Note: the atomicity here assumes that if a process checks
the while expression and determines that it is false, it
proceeds with the decrement operation right away
without being interrupted.

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1
  - can be simpler to implement
  - Also known as mutex locks
- Provides mutual exclusion:
  // following semaphore is shared by multiple processes
  Semaphore mutex;  //initialized to 1 => binary
- All processes use mutex to control entrance to critical section.

Java Example: Using a Binary Semaphore

```java
class Worker implements Runnable {
    private Semaphore sem;
    public Worker(Semaphore sem) {
        this.sem = sem;
    }
    public void run() {
        while (true) {
            sem.acquire();
            criticalSection();
            sem.release();
            remainderSection();
        }
    }
    class SemaphoreFactory {
        public static void main(...) {
            Semaphore sem = new Semaphore(1);
            for (int i=1; i<=5; i++)
                new Thread(new Worker(sem)).start();
        }
    }
}
```

Counting Semaphores for Synchronization

Dependency Graph

Example:

```
A
   /  
 B   C   D
    /     
   E       F
    /     
   G       H
```

Possible execution order: 
- A, D, H, B, C, F, E, G
- A, B, C, D, E, H, F, G
- ...

To enforce the right order, but at the same time allowing as much concurrency as possible, we can use semaphores...
Counting Semaphores for Synchronization (1)

Assume shared semaphores:
Semaphore BS = new Semaphore(0),
DS = new Semaphore(0),
ES = new Semaphore(-1);

Thread A:
A’s work
BS.release();
DS.release();

Thread B:
BS.acquire();
B’s work
ES.release();

Thread D:
DS.acquire();
D’s work
ES.release();

Thread E:
ES.acquire();
E’s work

Semaphore Implementation

• Main disadvantage of semaphore definition is: busy waiting.
• Busy waiting is a problem in a multiprogramming system: CPU waste.
• While a process holds a semaphore, other processes trying to acquire it are in a loop, waiting for the semaphore to be released.
  – This is called spinlock.
• This is not bad at all if the waiting time is small, specially, if smaller (or expected to be) than context switch time.
  – Usually employed in multiprocessor systems.
• Semaphores implementation should eliminate busy waiting.
  – By blocking any process trying to acquire an assigned semaphore.
  – Waking it up when the semaphore is released.

Semaphore Implementation Without Busy Waiting

• One alternative to remove the busy waiting is to block processes trying to get an already acquired semaphore.
• A blocked process on a particular semaphore is waken up when the semaphore is released.
• We can now define a semaphore as suggested by defining and object type containing two internal fields:
  – An integer value - the value of the process
  – A queue of PCPs - the waiting queue of processes
• In addition to the semaphore’s standard operations, it has the following internal operations:
  – block – blocks a process. The process is removed from the ready queue.
  – wakeup – removes one of processes in the waiting queue of the semaphore and places it in the ready queue.

A Semaphore Example in Java (assuming atomic operations)

```java
public class Semaphore {
    private int value;
    private Queue<PCB> waitingQueue;
    public Semaphore(int initialValue) {
        this.value = initialValue;
        waitingQueue = new waitingQueue<PCB>();
    }

    // let pcb be the PCB of the process trying to acquire/release this semaphore ...
    public void acquire() {   // assumed atomic
        value--;
        if (value < 0) {
            waitingQueue.enqueue(pcb);
            block(pcb);   // removes process from ready queue
        }
    }

    public void release() { // assumed atomic
        value++;
        if (value <= 0) {
            PCB pcb = waitingQueue.dequeue();
            wakeup(pcb);   // places process into the ready queue
        }
    }
}
```
Semaphore Implementation

- Must guarantee that no two processes can execute acquire() and release() on the same semaphore at the same time.
- Thus, the implementation of the semaphore itself contains a critical section problem - acquire and release code are placed in that critical section.
  - Could now have busy waiting to handle such critical section.
    - But it can be tolerated since implementation code is short.
    - In addition, little busy waiting is expected since this semaphore’s critical section is rarely occupied.
- Note that we are still having busy waiting, but it should not be of the same magnitude as it might be in critical sections on user’s applications.
  - User’s applications may spend lots of time in critical sections and therefore to have busy waiting there is not good in general.

Deadlock and Starvation

- Inappropriate use of semaphores is a common source of deadlocks.
- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1 - this may lead to deadlock

\[
\begin{align*}
P_0 & \quad \text{acquire}(S) \quad \text{acquire}(Q) \quad \text{release}(S) \\
P_1 & \quad \text{acquire}(Q) \quad \text{acquire}(S) \quad \text{release}(Q) \\
\end{align*}
\]

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process (see next)

Priority Inversion Problem

- The priority inversion problem occurs when a process of lower priority may come to ready state and causes another higher priority process to wait longer for some resource.
- Let P1, P2, and P3 be three process such that:
  \[\text{prty}(P1) < \text{prty}(P2) < \text{prty}(P3)\]
- Assume P1 is running, P2 is blocked, and P3 is waiting for some resource held by P1.
- P2 wakes up and goes to ready queue.
- Since P2 has higher priority than P1, P1 is preempted for P2 to enter into Running state.
- P2, who has lower priority than P3 is causing P3 to wait longer for some resource that P2 does not hold.
- The problem occurs only if there are more than two different priorities.

Priority Inversion Problem (2)

Solutions:

1. (1) allow no more than two different priorities - NOT PRACTICAL
2. (2) Implement the policy that whenever a set of processes is waiting for some resource being held by another process P of lower priority, the process P will temporarily get the same priority as the highest among the waiting processes.
Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
- These problems are classical for testing different synchronization tools.
- We shall now study about this problems and possible solutions.

Bounded-Buffer Problem

We have seen this problem before...
- let’s study a semaphore-based solution:
  - N buffers, each can hold one item (or a buffer with N slots) - N is the capacity of the whole buffer or the max number of items that it can hold simultaneously.
  - Semaphore *mutex* initialized to the value 1 - binary semaphore used to provide mutual exclusion for accesses to the pool.
  - Semaphore *full* initialized to the value 0
    - counting semaphore
    - counts the number of occupied slots in (number of items) the buffer
    - used to block not-allowed operations when the buffer is full
  - Semaphore *empty* initialized to the value N
    - counting semaphore
    - counts the number of empty (or available) slots in the buffer
    - used to block not-allowed operations when the buffer is empty.

Bounded Buffer Problem (Cont.)

A Java solution:

```java
public class BoundedBuffer<E> implements Buffer<E> {
    private static final int BUFFER_SIZE = ...; //capacity of buffer
    private E[] buffer;
    private int in, out;
    private Semaphore mutex, empty, full;

    public BoundedBuffer() {
        in = 0;
        out = 0;
        mutex = new Semaphore(1);
        empty = new Semaphore(BUFFER_SIZE);
        full = new Semaphore(0);
        buffer = (E[]) new Object[BUFFER_SIZE];
    }
    public void insert(E item) { ... next slide ... } - for producers
    public E remove() { ... next slide ... } - for consumers
}
```

Bounded Buffer Problem (Cont.)

**Producer process/thread:**

```java
// Producers call this method
public void insert(E item) {
    empty.acquire();
    mutex.acquire();
    // add an item to the buffer
    buffer[in] = item;
    in = (int + 1) % BUFFER_SIZE;
    mutex.release();
    full.release();
}
```

**Consumer process/thread:**

```java
// Consumers call this method
public E remove() {
    full.acquire();
    mutex.acquire();
    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    mutex.release();
    empty.release();
}
```
Bounded Buffer Problem (Cont.)

// producer thread
public class Producer implements Runnable {
    private Buffer<Date> buffer;
    public Producer(Buffer<Date> buffer) {
        this.buffer = buffer;
    }
    public void run() {
        Date message;
        while (true) {
            // sleep for a while
            SleepUtilities.nap();
            // produce an item & enter it into the buffer
            message = new Date();
            buffer.insert(message);
        }
    }
}

// consumer thread
public class Consumer implements Runnable {
    private Buffer<Date> buffer;
    public Consumer(Buffer<Date> buffer) {
        this.buffer = buffer;
    }
    public void run() {
        Date message;
        while (true) {
            // sleep for a while
            SleepUtilities.nap();
            // consume an item from the buffer
            message = buffer.remove();
        }
    }
}

Bounded Buffer Problem (Cont.)

// Producer/Consumer factory
public class Factory {
    public static main(String[] args) {
        Buffer<Date> buffer = new BoundedBuffer<Date>();
        // CREATE THE PRODUCER AND CONSUMER THREADS
        Thread producer = new Thread(new Producer(buffer));
        Thread consumer = new Thread(new Producer(buffer));
        producer.start();
        consumer.start();
    }
}

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Different variations:
  1. no reader shall be kept waiting unless a writer has already obtained permission to use the shared data object
     - may cause starvation to writers
  2. once a writer wants to write (is ready), that writer perform its write as soon as possible
     - may cause starvation to readers

Shared data
Readers-Writers Problem (1) Solution

Java solution for version (1).
// The database object type that is to be shared
// by the readers and writers
public class Database {
    private int readerCount;  // Number of readers currently active
    private Semaphore mutex;  // Binary semaphore to control updates to variable
            readerCount
    private Semaphore db;    // Binary semaphore to control that no writer operates in the
            // database if some other thread is already operating in it.

    public Database() {
        readerCount = 0;
        mutex = new Semaphore(1);
        db = new Semaphore(1);
        // these are shown on next slide
    }

    public void acquireReadLock() {…}
    public void releaseReadLock() {…}
    public void acquireWriteLock() {…}
    public void releaseWriteLock() {…}
}

Readers-Writers Problem (Cont.)

public class Reader implements Runnable {
    private ReadWriteLock db;
    public Reader(ReadWriteLock db) {
        this.db = db;
    }
    public void run() {
        while (true) {
            db.acquireReadLock(); // acquire reader lock
            … read from database
            db.releaseReadLock();   // release reader lock
        }
    }
}

public class Writer implements Runnable {
    private ReadWriteLock db;
    public Writer(ReadWriteLock db) {
        this.db = db;
    }
    public void run() {
        while (true) {
            db.acquireWriteLock(); // acquire writer lock
            … write to database
            db.releaseWriteLock();   // release writer lock
        }
    }
}

Readers-Writers Problem (1)(cont)

The other methods for class Database

```java
public void acquireReadLock() {
    mutex.acquire();
    // the first reader indicates that the
    // database is being read
    readerCount++;
    if (readerCount == 1)
        db.acquire();
    mutex.release();
}
```

```java
public void releaseReadLock() {
    mutex.acquire();
    // the last reader indicates that the
    // database is no longer being read
    readerCount--;
    if (readerCount == 0)
        db.release();
    mutex.release();
}
```

```java
public void acquireWriteLock() {
    db.acquire();
}
```

```java
public void releaseWriteLock() {
    db.release();
}
```

Readers-Writers Problem (Cont.)

```java
public class ReadersWritersFactory {
    private static Database db = new Database();
    public static void main(String[] args) {
        // init db …
        int nReaders, nWriters;
        Thread[] reader, writer;
        nReaders = getNumberOfReaders();
        nWriters = getNumberOfWriters();
        reader = initReaders(nReaders);
        writer = initWriters(nWriters);
    }
    private static Thread[] initReaders(int n) {
        Thread[] reader = new Thread[n];
        for (int i=1; i<=n; i++) {
            reader[i] = new Thread(new Reader(db));
            reader[i].start();
        }
        return reader;
    }
    private static Thread[] initWriters(int n) {
        Thread[] writer = new Thread[n];
        for (int i=1; i<=n; i++) {
            writer[i] = new Thread(new Writer(db));
            writer[i].start();
        }
        return writer;
    }
}
```

Readers-Writers Problem (Cont.)

…just an example…
Readers-Writers Problem (Cont.)

```java
public class ReadersWritersFactory {
    private static Database db = new Database();
    public static void main(String[] args) {
        ... init db ...
        int nReaders, nWriters;
        Thread[] reader, writer;
        nReaders = getNumberOfReaders();
        nWriters = getNumberOfWriters();
        reader = initReaders(nReaders);
        writer = initWriters(nWriters);
    }
    private static Thread[] initReaders(int n) {
        Thread[] reader = new Thread[n];
        for (int i=1; i<n; i++) {
            reader[i] = new Thread(new Reader(db));
            reader[i].start();
        }
    }
    private static Thread[] initWriters(int n) {
        Thread[] writer = new Thread[n];
        for (int i=1; i<n; i++) {
            writer[i] = new Thread(new Writer(db));
            writer[i].start();
        }
    }
}
```

Dining-Philosophers
A Possible Solution in Java

- Assume shared data in a Java program:
  Semaphore chopStick[] = new Semaphore[5];
  for (int i=0; i<5; i++)
    chopStick[i] = new Semaphore(1);

- Each Philosopher lives as a different thread, executing:
  // thread for philosopher i executes the following
  while (true) {
    // get left chopstick
    chopStick[i].acquire();
    // get right chopstick
    chopStick[(i+1)%5].acquire();
    eating();
    // return left chopstick
    chopStick[i].release();
    // return right chopstick
    chopStick[(i+1)%5].release();
  }

Dining-Philosophers Problem

- Five philosophers spend their lives thinking and eating rice.
- There are only 5 chopstick on the table, one between every two philosophers.
- In order to eat, a philosopher needs to get hold of the two chopsticks that are closest to her.
- Yes, they need to share the chopsticks.
- A philosopher cannot pick up a chopstick that is already taken by a neighbor.

WARNING
This solution has a potential deadlock situation...
Problems with Semaphores

- Although semaphores provide an effective mechanism for synchronization, it is easy to use them incorrectly.
- The programming language cannot help in blocking several possible errors.
- Incorrect use of semaphore operations:
  - Omit acquire or omit release statements.
  - Use an incorrect order of the acquire and release statements.
  - This may cause violation of mutual exclusion, etc
  - May also cause deadlocks
  - Examples: We have already seen some (e.g. slide 27)
- Better language constructs are needed.

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
  - We can think of it as a Java class...
- Only one process is allowed to be active within the monitor at a time

```java
monitor monitor-name {
    // shared variable declarations
    procedure P1 (...) { .... }
    ...
    procedure Pn (...) {......}
    Initialization code ( ....) { .... }
}
```

Condition Variables

- Monitor construct ensures only one process at the time can be active within the monitor (executing part of it).
- However, this is not powerful enough and extra synchronization mechanisms are needed.
  - One such mechanism are condition variables.
  - A new data type is required: Condition
    ```java
    Condition x, y; // declares two condition variables
    ```
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of the processes (if any) that is currently waiting because it has invoked x.wait ()
  - Has no effect if no process is waiting on x at the moment.
Condition Variables (cont)

- If a process P executes `x.signal()` and there is another process Q waiting on x, what should happened?
  *** Should P and Q both continue simultaneously within the monitor?
- Two possibilities exist:
  - **Signal and wait**: P either waits until Q leaves the monitor or waits for another condition.
  - **Signal and continue**: Q either waits until P leaves the monitor or waits for another condition.
- A compromise between the two alternatives is needed.
- Some languages implement the following policy:
  - When P executes `signal()` operation, it immediately leaves the monitor and Q is immediately resumed.

Monitor with Condition Variables

Schematic view of a monitor and condition variables.

Solution to Dining Philosophers

This solution **has the problem of possible starvation**... but is a good start...

```java
monitor DiningPhilosophers {
    enum State {THINKING, HUNGRY, EATING};
    State[] state = new State[5];
    Condition[] cond = new Condition[5];
    // initialization code ... like a Java constructor ...
    public DiningPhilosophers {
        for (int i=0; i<5; i++)
            state[i] = THINKING;
    }
    // see other methods of this monitor on next slide
}
```

Solution to Dining Philosophers

```java
void takeForks(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING)
        cond[i].wait();
}

void returnForks (int i) {
    state[i] = THINKING;
    test((i+4)%5);
    test((i+1)%5);
}

void test (int i) {
    if((state[(i+4)%5] != EATING)
        && (state[i] == HUNGRY)
        && (state[(i+1)%5] != EATING))
    {
        // none of the neighbors of i is eating
        // and i is hungry, then if this call comes
        // from returnForks, i must be waiting
        // on condition variable cond[i] - wake it!
        // If the call comes from takeForks, the
        // signal is ignored...
        state[i] = EATING;
        cond[i].signal();
    }
}```
Solution to Dining Philosophers

- Let dp be an instance of the DiningPhilosophers monitor.

- Philosopher i does the following:
  ```java
  while (true) {
    db.takeForks(i);
    eat();
    dp.returnForks(i);
  }
  ```

A Monitor to Allocate Single Resource

**Example:** Monitor to allocate a single resource to one user at a time

```java
monitor ResourceAllocator
{
  boolean busy;
  condition x;
  void acquire() {
    if (busy)
      x.wait();
    busy = TRUE;
  }
  void release() {
    busy = FALSE;
    x.signal();
  }
  initialization code() {
    busy = FALSE;
  }
}
```

Java Synchronization (1)

Java allows to declare instance methods as `synchronized`.

- Just need to add the `synchronized` keyword in the method’s header. For example:
  ```java
  public synchronized void insert(...) {...}
  ```
- Every object in a Java program has associated with it a single lock.
- Usually, that lock is not used, but if the method is synchronized, then any thread that wants to apply that particular method to an object instance needs to acquire that object’s lock first.
- If another thread already owns the lock, then the new thread as to wait - the new calling thread is placed in the entry set for the object’s lock.
- Notice that at most one synchronized method at a time can be applied to an object instance, since there is only one lock per object instance.
  - non synchronized methods are not affected

Java Synchronization (2)

The entry set of an object represents the set of threads that are waiting for the object’s lock.

- The entry set is managed by JVM.
- Usually managed as a FCFS queue, but may vary from one JVM to another.
- A thread owning the lock releases it when it exits the synchronized method that required to posses the lock.
Consider the following attempt to solve the Producer-Consumer problem with a bounded buffer.

Assume the following implementation for methods `insert` and `remove` seen earlier.

```java
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE)
        Thread.yield();
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    count++;}

public E synchronized E remove() {
    E item;
    while (count == 0)
        Thread.yield();
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    return item;
}
```

Instead of wasting CPU cycles while waiting, allow JVM to schedule another thread. The thread that yields remains in runnable state.

**Possible deadlock:**
- Consumer enters to remove but `count == 0`.
- It yields while possessing the lock.
- Producer cannot enter to insert since the lock is owned by another thread.

Possible deadlock:
- Consumer enters to remove but `count == 0`.
- It yields while possessing the lock.
- Producer cannot enter to insert since the lock is owned by another thread.
Java Synchronization (7)

Other Java synchronization methods inherited from Object:

- `wait()`: the thread executing it releases the lock for the object that the method is applied to. The thread is placed into the **waiting set** (a queue) associated with the object.
- `notify()`: wakes up one of the threads in the waiting set for the particular object it is applied to (if any thread is there). The thread is placed in the **entry set** of the object and is able to compete for the object’s lock. The selection is usually based on FCFS.
- `notifyAll()`: wakes up all of the threads in the waiting set for the particular object it is applied to (if any thread is there). The threads are placed in the **entry set** of the object.

When a thread that blocks in `wait` eventually resumes, it continues execution from the call to `wait()`.

These methods are declared within `Object`, as shown here:

```java
final void wait() throws InterruptedException
final void notify()
final void notifyAll()
```

Java Synchronization (8)

The figure depicts the two sets of threads associated with and object at any instance.

- When a thread exits a synchronized method on a particular object, it releases the lock for that object.
- That allows only threads in the entry set to compete for the object’s lock.
-Threads in the waiting set are moved to the entry set only by explicit execution of `notify` or `notifyAll`.

Threads resuming from a wait call, continue execution as if there was a return from the call to `wait()`.

Java Synchronization (9)

One solution for producer-consumer using: `wait()` and `notify()`

```java
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE)
        try {
            wait(); // places the thread in the waiting set of the object
        }
        catch (InterruptedException e) {} // places the thread in the waiting set of the object
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    notify(); // wakes up a thread waiting in waiting set (if any) and places in entry set
}

public E synchronized E remove() {
    E item;
    while (count == 0)
        try {
            wait(); // places the thread in the waiting set of the object
        }
        catch (InterruptedException e) {} // places the thread in the waiting set of the object
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    notify(); // wakes up a thread waiting in waiting set (if any) and places in entry set
    return item;
}
```

Java Synchronization (10)

When to use `notify` or `notifyAll`.

- Call to `notify()` is good if it is guaranteed that only one thread might be in the waiting set or if it really does not matter which thread from that set is picked.
- But if there might be threads that cannot continue for some property or scheme followed on the particular application, then perhaps is safer and better to call `notifyAll()` instead.
- Consider five threads sharing object with the following method:

```java
public synchronized void doWork(int myNumber) {
    while (turn != myNumber) // myNumber identifies a particular thread: 0, 1, 2, 3, 4
        try {
            wait();
        } catch (InterruptedException e) {} // myNumber identifies a particular thread: 0, 1, 2, 3, 4
    ... do some work...
    turn = (turn + 1) % 5; // set the next waiting thread that has the turn to work
    notify(); // wakes up only the thread, but that that may not be the one whose number matches the value of turn ==> DEADLOCK situation.
```
A better solution in this case would be to use `notifyAll()` instead of `notify()`.

```java
public synchronized void doWork(int myNumber) {
    while (turn != myNumber) // myNumber identifies each thread: 0, 1, 2, 3, 4
        try {
            wait();
        } catch (InterruptedException e) {} // do some work...
    turn = (turn + 1) % 5; // set the next waiting thread that has the turn to work
    notifyAll(); // wakes up all the threads
}
```

Although less efficient, the use of `notifyAll()` is safer.

---

### Java Synchronization (12)

// The database object type that is to be shared
// by the readers and writers
public class Database {
    private int readerCount; // Number of readers currently active
    private boolean dbWriting; // true if a writer is working in the database or shared data
    public Database() {
        readerCount = 0;
        dbWriting = false;
    }
    // these are shown on next slide
    public synchronized void acquireReadLock() {...}
    public synchronized void releaseReadLock() {...}
    public synchronized void acquireWriteLock() {...}
    public synchronized void releaseWriteLock() {...}
}

### Java Synchronization (13)

The other methods for class `Database (Version 2)`

- `synchronized void acquireReadLock()`
  - while (dbWriting) // wait if someone is writing
    - try {
      - wait();
    } catch (InterruptedException e) {} // do some work...
  - `readerCount++;
- `synchronized void acquireWriteLock()`
  - while(readerCount > 0 || dbWriting) {
    - try {
      - wait();
    } catch (InterruptedException e) {} // Once there are no readers or another writer working
    // on the shared db, indicate that the db is being written
    - `dbWriting = true;`
- `synchronized void releaseReadLock()`
  - `readerCount--;
    - // the last reader indicates that
      - `if (readerCount == 0)
      - notify(); // there must be at most one writer
      - // waiting`
- `synchronized void releaseWriteLock()`
  - `dbWriting = false;
    - notifyAll(); // safer - there might be many readers
    - // waiting...

---

### Java Synchronization (14)

Block synchronization in Java

- If the critical section in a particular method is a small block of instructions, and the whole method is too large, one should perhaps use synchronized blocks.
- In general, they are as illustrated next:

```java
Object mutexLock = new Object();
public void someMethod() {
    synchronized (mutexLock) {
        // method is not synchronized
        // nonCriticalSection()
        synchronized(mutexLock) {
            // acquire the lock for the mutexLock
            // enter here only is the lock is owned
            // criticalSection;
            // the lock is released automatically
            remainderSection();
        }
    }
}
```
Block synchronization in Java (cont.)

- Synchronized blocks can also apply wait and notify commands.
- See next example.

```java
Object mutexLock = new Object();
...
public void someMethod() {  // method is not synchronized
    nonCriticalSection();
    synchronized(mutexLock) {
        ...
        try {
            mutexLock.wait();
        } catch(InterruptedException e) {}
        ...
    }
}
```

To notify any waiting threads on a particular object’s lock, the following is required:

- First, acquire the object’s lock and then notify. This is achieved as follows:

```java
synchronized(mutexLock){
    mutexLock.notify();
}
```

Some important synchronization rules in Java:

- Once a thread acquires an object’s lock, it can call a synchronized method from another method while it holds the lock. This means that locks are recursive or reentrant.
- A thread can own more than one object lock at the time.
- A method that is not synchronized can be invoked at any moment by any thread.
- If waiting set for an object is empty, then the execution of notify() or notifyAll() on that object has no effect.
- Methods wait(), notify(), and notifyAll() may only be invoked from synchronized methods or blocks.

Java also allows synchronized static methods.
- In this case, it uses a class lock that every class in Java has.
- We can also use the class lock in synchronized blocks:

```java
synchronized(ClassName.class) {
    ...
    synchronized code ...
}
```

Read about `InterruptedException` in Java.
Java Synchronization (19)

**Reentrant Locks in Java**
- Java provides a locking mechanism through the class `ReentrantLock` and interface `Lock`.
- Acts like synchronized in Java, but it provides other features such as scheduling decisions for waiting threads.
- **Example**:
  ```java
  Lock key = new ReentrantLock();
  key.lock();
  try {
    // ... critical section ...
  } finally {
    key.unlock();
  }
  ```

Java Synchronization (20)

**Semaphores in Java**
- Java provides a counting semaphore: class `Semaphore`.
- It complies with the specifications for counting semaphores as we have studied.
- A new object of type `Semaphore` can be created by:
  ```java
  new Semaphore(int_value)
  ```
- **Example**:
  ```java
  Semaphore sem = new Semaphore(1);
  try{
    sem.acquire();
    // ... critical section ...
  } catch(InterruptedException e){}
  finally {
    sem.release();
  }
  ```

Synchronization Examples

- Solaris
- Windows XP
- 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:
  - locks can behave as spinlocks or as blocking-wakeup
  - selection depends on number of cores or if lock’s owner is blocked
  - or if it is active
- Uses condition variables and readers-writers locks when longer sections of code need access to data:
  - readers-writers locks allow multiple reader threads simultaneously while only one writer...
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock:
  - these are queues of waiting threads...
Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

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
  - spin locks

Pthreads Synchronization

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

Atomic Transactions

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions
System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- **Transaction** - collection of instructions or operations that performs single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of **read** and **write** operations
  - Terminated by **commit** (transaction successful) or **abort** (transaction failed) operation
  - Aborted transaction must be rolled back to undo any changes it performed

Types of Storage Media

- **Volatile storage** – information stored here does not survive system crashes
  - Example: main memory, cache
- **Nonvolatile storage** – Information usually survives crashes
  - Example: disk and tape
- **Stable storage** – Information never lost
  - Not actually possible, so approximated via replication or RAID to devices with independent failure modes

Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is **write-ahead logging**
  - Log on stable storage, each log record describes single transaction write operation, including
    - Transaction name
    - Data item name
    - Old value
    - New value
  - `<T_i` starts>` written to log when transaction T_i starts
  - `<T_i commits>` written when T_i commits
- Log entry must reach stable storage before operation on data occurs

Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
  - **Undo**(T_i) restores value of all data updated by T_i
  - **Redo**(T_i) sets values of all data in transaction T_i to new values
- **Undo**(T_i) and **redo**(T_i) must be idempotent
  - Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
  - If log contains `<T_i` starts>` without `<T_i commits>`, **undo**(T_i)
  - If log contains `<T_i` starts>` and `<T_i commits>`, **redo**(T_i)
Checkpoints

- Log could become long, and recovery could take long
- Checkpoints shorten log and recovery time.
- Checkpoint scheme:
  - Output all log records currently in volatile storage to stable storage
  - Output all modified data from volatile to stable storage
  - Output a log record <checkpoint> to the log on stable storage
- Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti. All other transactions already on stable storage

Concurrent Transactions

- Must be equivalent to serial execution – serializability
- Could perform all transactions in critical section
  - Inefficient, too restrictive
- Concurrency-control algorithms provide serializability

Serializability

- Consider two data items A and B
- Consider Transactions T₀ and T₁
- Execute T₀, T₁ atomically
- Execution sequence called schedule
- Atomically executed transaction order called serial schedule
- For N transactions, there are N! valid serial schedules

Schedule 1: T₀ then T₁

<table>
<thead>
<tr>
<th>T₀</th>
<th>T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</td>
</tr>
</tbody>
</table>
Nonserial Schedule

- **Nonserial schedule** allows overlapped execute
  - Resulting execution not necessarily incorrect
- Consider schedule $S$, operations $O_i$, $O_j$
  - **Conflict** if access same data item, with at least one write
- If $O_i$, $O_j$ consecutive and operations of different transactions & $O_i$ and $O_j$ don’t conflict
  - Then $S$’ with swapped order $O_j$, $O_i$ equivalent to $S$
- If $S$ can become $S$’ via swapping nonconflicting operations
  - $S$ is conflict serializable

Schedule 2: Concurrent Serializable Schedule

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>write($A$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($B$)</td>
</tr>
</tbody>
</table>

Locking Protocol

- Ensure serializability by associating lock with each data item
  - Follow locking protocol for access control
- Locks
  - **Shared** – $T_i$ has shared-mode lock (S) on item $Q$, $T_i$ can read $Q$ but not write $Q$
  - **Exclusive** – $T_i$ has exclusive-mode lock (X) on $Q$, $T_i$ can read and write $Q$
- Require every transaction on item $Q$ acquire appropriate lock
  - If lock already held, new request may have to wait
    - Similar to readers-writers algorithm

Two-phase Locking Protocol

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
  - **Growing** – obtaining locks
  - **Shrinking** – releasing locks
- Does not prevent deadlock
Timestamp-based Protocols

- Select order among transactions in advance – *timestamp-ordering*
- Transaction $T_i$ associated with timestamp $TS(T_i)$ before $T_i$ starts
  - $TS(T_i) < TS(T_j)$ if $T_i$ entered system before $T_j$
  - TS can be generated from system clock or as logical counter incremented at each entry of transaction
    - Timestamps determine serializability order
  - If $TS(T_i) < TS(T_j)$, system must ensure produced schedule equivalent to serial schedule where $T_i$ appears before $T_j$

Timestamp-based Protocol Implementation

- Data item $Q$ gets two timestamps
  - $W$-timestamp($Q$) – largest timestamp of any transaction that executed write($Q$) successfully
  - $R$-timestamp($Q$) – largest timestamp of successful read($Q$)
  - Updated whenever read($Q$) or write($Q$) executed
- *Timestamp-ordering protocol* assures any conflicting read and write executed in timestamp order
- Suppose $T_i$ executes read($Q$)
  - If $TS(T_i) < W$-timestamp($Q$), $T_i$ needs to read value of $Q$ that was already overwritten
    - read operation rejected and $T_i$ rolled back
  - If $TS(T_i) \geq W$-timestamp($Q$)
    - read executed, $R$-timestamp($Q$) set to max($R$-timestamp($Q$), $TS(T_i)$)
- Suppose $T_i$ executes write($Q$)
  - If $TS(T_i) < R$-timestamp($Q$), value $Q$ produced by $T_i$ was needed previously and $T_i$ assumed it would never be produced
    - Write operation rejected, $T_i$ rolled back
  - If $TS(T_i) < W$-timestamp($Q$), $T_i$ attempting to write obsolete value of $Q$
    - Write operation rejected and $T_i$ rolled back
  - Otherwise, write executed
- Any rolled back transaction $T_i$ is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock

Schedule Possible Under Timestamp Protocol

<table>
<thead>
<tr>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td>read($A$)</td>
<td>write($B$)</td>
</tr>
<tr>
<td></td>
<td>read($A$)</td>
</tr>
<tr>
<td></td>
<td>write($A$)</td>
</tr>
</tbody>
</table>
The Deadlock Problem

• A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

Example 1:
- System has 2 disk drives
- $P_1$ and $P_2$ each hold one disk drive and each needs another one

Example 2: Two processes or threads sharing two semaphores: $A$ and $B$, initialized to 1

$P_0$ $P_1$

acquire(A); acquire(B);
acquire(B); acquire(A);
...
...

Bridge Crossing Example

• Traffic only in one direction
• Each section of a bridge can be viewed as a resource
• If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
• Several cars may have to be backed up if a deadlock occurs
• Starvation is possible
• Note – Most OSes do not prevent or deal with deadlocks

System Model

• 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

Deadlock Characterization

• 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 $P_0$
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 $R_j$
- $P_j$ is holding an instance of $R_j$

Example of a Resource Allocation Graph

Resource Allocation Graph With A Deadlock
Graph With A Cycle But No Deadlock

Basic Facts

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

Java Deadlock Example

```
class A implements Runnable {
    private Lock first, second;
    public A(Lock first, Lock second) {
        this.first = first;
        this.second = second;
    }
    public void run() {
        try {
            first.lock();
            // do something
            second.lock();
            // do something else
        } finally {
            first.unlock();
            second.unlock();
        }
    }
}
```

```
class B implements Runnable {
    private Lock first, second;
    public B(Lock first, Lock second) {
        this.first = first;
        this.second = second;
    }
    public void run() {
        try {
            second.lock();
            // do something
            first.lock();
            // do something else
        } finally {
            second.unlock();
            first.unlock();
        }
    }
}
```

```
public static void main(String arg[]) {
    Lock lockX = new ReentrantLock();
    Lock lockY = new ReentrantLock();
    Thread threadA = new Thread(new A(lockX, lockY));
    Thread threadB = new Thread(new B(lockX, lockY));
    threadA.start();
    threadB.start();
}
```

Deadlock is possible if:

```
threadA --> lockY --> threadB --> lockX --> threadA
```
Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

Requires the system has *a priori* information available.

- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.
Safe, Unsafe, Deadlock State

Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a sequence \(<P_1, P_2, \ldots, P_n>\) of all the processes in the system such that for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by currently available resources + resources held by all the \(P_k\), \((k < i)\)
- That is:
  - If \(P_i\) resource needs are not immediately available, then \(P_i\) can wait until all \(P_k\) \((k < i)\) have finished.
  - When all \(P_k\) \((k < i)\) are finished, \(P_i\) can obtain needed resources, execute, return allocated resources, and terminate.
  - When \(P_i\) terminates, \(P_{i+1}\) can obtain its needed resources, and so on.

Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker’s algorithm
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process $P_j$ may request resource $R_j$; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Unsafe State In Resource-Allocation Graph

Resource-Allocation Graph Algorithm

- Suppose that process $P_i$ requests a resource $R_j$
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph
Banker’s Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for Banker’s Algorithm

Let $n$ = number of processes, and $m$ = number of resources types.

- **Available**: Vector of length $m$. If available $[j] = k$, there are $k$ instances of resource type $R_j$ available
- **Max**: $n \times m$ matrix. If $Max[i,j] = k$, then process $P_i$ may request at most $k$ instances of resource type $R_j$
- **Allocation**: $n \times m$ matrix. If $Allocation[i,j] = k$, then $P_i$ is currently allocated $k$ instances of $R_j$
- **Need**: $n \times m$ matrix. If $Need[i,j] = k$, then $P_i$ may need $k$ more instances of $R_j$ to complete its task
  
  \[ Need[i,j] = Max[i,j] – Allocation[i,j] \]

Safety Algorithm

1. Let $Work$ and $Finish$ be vectors of length $m$ and $n$, respectively. Initialize:
   \[ Work = Available \]
   \[ Finish[i] = false \text{ for } i = 0, 1, \ldots, n-1 \]
2. Find and $i$ such that both:
   (a) $Finish[i] = false$
   (b) $Need[i] \leq Work$
   If no such $i$ exists, go to step 4
3. $Work = Work + Allocation_i$, $Finish[i] = true$
   go to step 2
4. If $Finish[i] == true$ for all $i$, then the system is in a safe state

Resource-Request Algorithm for Process $P_i$

$Request = request$ vector for process $P_i$.
If $Request[j] = k$ then process $P_i$ wants $k$ instances of resource $R_j$

1. If $Request[i] \leq Need[i]$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request[i] \leq Available$, go to step 3. Otherwise $P_i$ must wait, since resources are not available
3. Pretend to allocate requested resources to $P_i$ by modifying the state as follows:
   \[ Available = Available – Request; \]
   \[ Allocation[i] = Allocation[i] + Request; \]
   \[ Need[i] = Need[i] – Request; \]
   - If safe $\implies$ the resources are allocated to $P_i$
   - If unsafe $\implies$ $P_i$ must wait, and the old resource-allocation state is restored
Example of Banker’s Algorithm

- 5 processes $P_0$ through $P_4$;
- 3 resource types: $A$ (10 instances), $B$ (5 instances), and $C$ (7 instances)

Snapshot at time $T_0$:

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Max</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>0 1 0</td>
<td>7 5 3</td>
</tr>
<tr>
<td>$P_1$</td>
<td>2 0 0</td>
<td>3 2 2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 2</td>
<td>9 0 2</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>2 2 2</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>4 3 3</td>
</tr>
</tbody>
</table>

Example (Cont.)

- The content of the matrix $Need$ is defined to be $Max - Allocation$

<table>
<thead>
<tr>
<th>Need</th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>7 4 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_1$</td>
<td>1 2 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>6 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td>4 3 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The system is in a safe state since the sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies safety criteria

Example: $P_1$ Request (1,0,2)

Check that Request $\leq$ Available (that is, $1$,$0$,$2$) $\leq$ $3$,$3$,$2$ $\Rightarrow$ true

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>0 1 0</td>
<td>7 4 3</td>
</tr>
<tr>
<td>$P_1$</td>
<td>3 0 2</td>
<td>0 2 0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3 0 1</td>
<td>6 0 0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0 0 2</td>
<td>4 3 1</td>
</tr>
</tbody>
</table>

Executing safety algorithm shows that sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement

Can request for (3,3,0) by $P_4$ be granted?
Can request for (0,2,0) by $P_0$ be granted?

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme
Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes
  - \( P_i \rightarrow P_j \) if \( P_i \) is waiting for \( P_j \)
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of \( n^2 \) operations, where \( n \) is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph

Several Instances of a Resource Type

- **Available**: A vector of length \( m \) indicates the number of available resources of each type.
- **Allocation**: An \( n \times m \) matrix defines the number of resources of each type currently allocated to each process.
- **Request**: An \( n \times m \) matrix indicates the current request of each process. If \( \text{Request}[i][j] = k \), then process \( P_i \) is requesting \( k \) more instances of resource type \( R_j \).

Detection Algorithm

1. Let \( \text{Work} \) and \( \text{Finish} \) be vectors of length \( m \) and \( n \), respectively. Initialize:
   - (a) \( \text{Work} = \text{Available} \)
   - (b) For \( i = 1, 2, \ldots, n \), if \( \text{Allocation}[i] \neq 0 \), then \( \text{Finish}[i] = \text{false} \); otherwise, \( \text{Finish}[i] = \text{true} \)

2. Find an index \( i \) such that both:
   - (a) \( \text{Finish}[i] = \text{false} \)
   - (b) \( \text{Request}[i] \leq \text{Work} \)

   If no such \( i \) exists, go to step 4
Detection Algorithm (Cont.)

3. \[ \text{Work} = \text{Work} + \text{Allocation}_i \]
   \[ \text{Finish}[i] = \text{true} \]
   go to step 2

4. If \( \text{Finish}[i] = \text{false} \), for some \( i \), \( 1 \leq i \leq n \), then the system is in deadlock state. Moreover, if \( \text{Finish}[i] = \text{false} \), then \( P_i \) is deadlocked

Example of Detection Algorithm

Five processes \( P_0 \) through \( P_4 \); three resource types A (7 instances), B (2 instances), and C (6 instances)

Snapshots at time \( T_0 \):

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>( P_0 )</td>
<td>0 1 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>2 0 0</td>
<td>2 0 2</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>3 0 3</td>
<td>0 0 0</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>2 1 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>0 0 2</td>
<td>0 0 2</td>
</tr>
</tbody>
</table>

Sequence \(<P_0, P_2, P_3, P_1, P_4>\) will result in \( \text{Finish}[i] = \text{true} \) for all \( i \)

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Example (Cont.)

\( P_2 \) requests an additional instance of type C

<table>
<thead>
<tr>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
</tr>
<tr>
<td>( P_0 )</td>
</tr>
<tr>
<td>( P_1 )</td>
</tr>
<tr>
<td>( P_2 )</td>
</tr>
<tr>
<td>( P_3 )</td>
</tr>
<tr>
<td>( P_4 )</td>
</tr>
</tbody>
</table>

State of system?
- Can reclaim resources held by process \( P_0 \), but insufficient resources to fulfill other processes; requests
- Deadlock exists, consisting of processes \( P_1 \), \( P_2 \), \( P_3 \), and \( P_4 \)
Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost
- Rollback – return to some safe state, restart process for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor

End
Bounded-waiting Mutual Exclusion with TestAndSet()

\[
\begin{align*}
\text{do } \{ \\
\quad \text{waiting}[i] &= \text{TRUE}; \\
\quad \text{key} &= \text{TRUE}; \\
\quad \text{while } (\text{waiting}[i] \text{ && key}) \\
\quad \quad \text{key} &= \text{TestAndSet}(&\text{lock}); \\
\quad \text{waiting}[i] &= \text{FALSE}; \\
\quad \ldots \text{critical section} \\
\quad j &= (i + 1) \% n; \\
\quad \text{while } (j \neq i \text{ && waiting}[j]) \\
\quad \quad j &= (j + 1) \% n; \\
\quad \text{if } (j == i) \\
\quad \quad \text{lock} &= \text{FALSE}; \\
\quad \text{else} \\
\quad \quad \text{waiting}[j] &= \text{FALSE}; \\
\quad \ldots \text{remainder section} \\
\} \text{ while (TRUE);} 
\end{align*}
\]
Monitor Implementation

Variables
Semaphore mutex; // initially = 1 : controls single access inside monitor
Semaphore next; // initially = 0 : ??????
int next_count = 0; // number of processes waiting on some
                    // condition variable (cv) inside the monitor ??????

Each procedure F is replaced by

```c
acquire(&mutex);
body of F …
release(&mutex);
```
// before exiting the monitor, check is there is any process waiting after signaling some cv
if (next_count > 0)
  // if so, let one go ahead and exit
  release(&next);
else
  // otherwise, release monitor’s semaphore and exit the monitor
  release(&mutex);
```

Mutual exclusion within a monitor is ensured.

Monitor Implementation

For each condition variable x, we have:

```
Semaphore x_sem; // (initially = 0) a semaphore to wait on variable x
int x_count = 0; // number of processes waiting on x
```

The operation `x.wait()` can be implemented as:

```
x_count++; // number of process waiting in x
if (next_count > 0) // someone waits inside the monitor after signaling some
                  // cv ??????
  release(&next); // let one of those waiting to go ahead
else // no one is waiting
  release(&mutex); // allow new processes to enter monitor
acquire(&x_sem); // wait on semaphore for x
x_count--; // decrement the number of processes waiting on x
```

The operation `x.signal()` can be implemented as:

```
if (x_count > 0) { // if some process is waiting on x ??????
```

single processor | multiple processors
--- | ---
Disable kernel preemption. | Acquire spin lock.
Enable kernel preemption. | Release spin lock.