Murdoch
UNIVERSITY

Topic 7: Multithreading

ICT373: Software Architectures

Recap

- Graphical User Interfaces (GUI).
- Finite State Machine (FSM) to describe the dynamic behaviour of the GUI.
- Event handling.

Outline

- Threads (concurrency) in Java
- Designing for Multithreading: Petri Nets
- Synchronization and Protocols
- Deadlock and other Design Issues

Reading:

Textbook (11th Ed) Chapter 23

Objectives

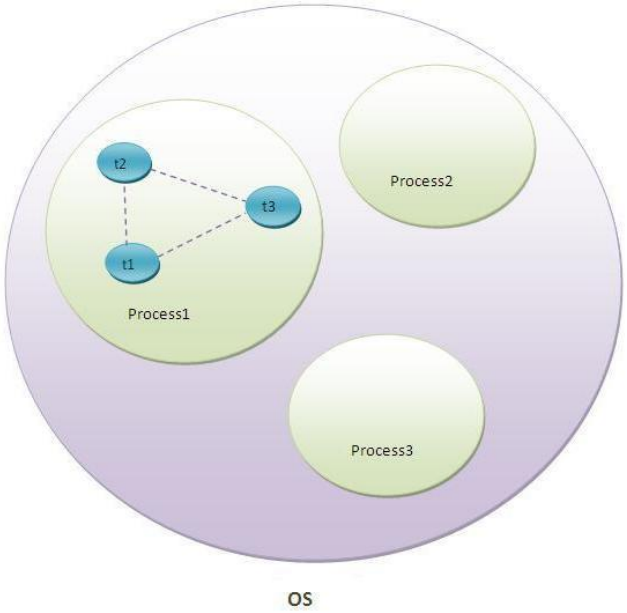
- Explain what is meant by a process and a thread.
- Describe main uses of multithreading.
- Be able to create a thread in Java.
- Describe the use of the Runnable interface in Java.
- Explain the concept of a Daemon.
- Describe some of the main design issues concerned with multithreading.
- Explain what a Petri Net is and what it is for.
- Explain synchronization.
- Explain the use of Monitor locks in java multithreading.

Objectives

- Describe a protocol for ensuring that only one thread is allowed into a critical region at a time.
- Describe the states that a thread may be in (life cycle of a thread).
- Explain what may cause deadlock.
- Describe the use of priorities and sleep in Java multithreading.
- Explain the advantages and disadvantages of using multithreading.

Multitasking - Threads

- A ***process*** is a self-contained running program.
- A ***multitasking*** operating system is capable of running more than one process at a time. (By giving them all turns to proceed in quick succession).
- A ***thread*** = a separate independently running subtask within a process. It has its own flow of control.
  - A thread itself is not a program (process). It cannot run on its own, rather it runs within a program.



Murdoch UNIVERSITY

Process and thread

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- A ***multithreaded*** (concurrent) programming language (like Java) supports multiple threads within a program (process).
  - The JVM allows a program to have multiple threads running concurrently and performing different tasks.
  - Each thread has its own method-call stack and program counter
  - Each thread can execute concurrently with other threads while sharing application-wide resources such as memory and files
  - This capability is called multithreading (or concurrent programming)

Threads: Uses

- It allows a program to get on with some work while at the same time waiting for and/or monitoring one or more events or inputs from the outside world (user, network connections or peripheral devices).
- This is crucial for responsive GUIs as well as many other applications. This is by far the most popular use: users expect a lot from a GUI. A program cannot delay responding to a button press while it finishes its current task.
- There are other uses in serious computing (e.g., game programs) where several different approaches might be available to solve a problem, so we set them all to work and wait for the first answer to come back.

Threads in Java

- class **Thread** (in java.lang package) allows us to multithread.
- The class Thread implements a generic thread that, by default, does nothing.
  - The implementation of its **run()** method is empty.
- It defines the API that lets a Runnable object provide a more interesting run() method for a thread.
- There are two ways of creating a thread of execution:
  - declare a class to be a subclass of Thread, which override the class Thread's run() method; OR
  - provide a class that implements the Runnable interface.
- Every thread has a priority.
  - Threads with higher priority are executed in preference to threads with lower priority.
  - When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread.

Thread in Java

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

```
PrimeThread p = new PrimeThread(143);
p.start();
```

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

Murdoch UNIVERSITY

Subclassing Thread

- This method of multithreading requires the programmer to supply their own subclass of **Thread** which overrides the **run()** method.
  - The subclass's run() method implements the thread's action (its behaviour).
- The programmer then creates a new object of that class and calls its **start()** method which is a method of class **Thread** which is not overridden.
- The JVM will start a new independent flow of control following the code in the **run()** method supplied and running separately from the flow of control in the rest of the program which continues on after the call to **start()**.

Subclassing Thread

- There are other useful Thread methods in java.lang. For example, **sleep(n)** puts the thread to sleep for n milliseconds.
- To get a reference to the thread which is the main thread of execution in a program (which always exists) use **Thread.currentThread()** in a main part of the code. (This returns a reference to whichever thread is currently executing).
- When threads are alive (i.e. before they reach the end of their **run** method) they will not be garbage collected even if your program retains no explicit reference to their Thread object.

Example

- SimpleThread.java (threadexample)
- Notice how unpredictable the scheduling of turns is.

Runnable interface (1/2)

- The other way to make a separate thread is to provide a class that implements the Runnable interface.
- That class then implements the run() method: so you just need to supply a body for that.
- A Runnable object provides the run() method to the thread.
- The run method can then immediately access all the other fields and methods of the object in your own class.
- To run the run() method you need to hand a reference X say, to your Runnable object over to a thread constructor:

  Thread t = new Thread(X);

  and then start the thread like this

  t.start();

Runnable interface (2/2)

- This method is useful if you have a class which has to be a subclass of some other class (like a JApplet or a JFrame) but you also want it to have its own run() method. If you get this class to implement Runnable then you can run it as a thread without having to explicitly program an extra thread object around which just runs your object and might have trouble accessing all the parts of your object.
- Java provides built-in classes and methods (in java.util.concurrent package) to create threads that execute the Runnables. These use the **Executor** interface which declares a single method named **execute**. An Executor object creates and manages a group of threads called a **thread pool**.
- Another interface called **ExecutorService** which is a sub-interface of Executor declares a number of other methods for managing the life cycle of the Executor.

Example

Codes// PrintTask.java from Deitel

// PrintTask class sleeps for a random time from 0 to 5

// seconds

// TaskExecutor.java from Deitel

// Multiple threads printing at different intervals.

// Using an Executor Service to execute Runnables.

More examples on LMS

Runnable interface vs Thread

- The most common difference is:

When you extend Thread class, you can't extend any other class which you require. (As you know, Java does not allow inheriting more than one class). When you implement Runnable, you can save a space for your class to extend any other class in future or now.

- However, the significant difference is.

When you extends Thread class, each of your thread creates unique object and associate with it. When you implements Runnable, it shares the same object to multiple threads.

- Example: ThreadVsRunnable

run() and start()

```
Thread one = new Thread();
Thread two = new Thread();
one.run();
two.run();
```

```
Thread one = new Thread();
Thread two = new Thread();
one.start();
two.start();
```

- Calling run() function as equivalent calling an ordinary function on some object
  - Example on the left-side: run two threads sequentially!!
- Example (right): on calling of start() function on thread one, JVM will call run() of thread one asynchronously and start() function return.
  - start() function just used to signal to JVM to call run() method on same object of thread.
  - start() function return for thread one;
  - then start() function of thread two get call, both thread one and two will run asynchronously.

Murdoch UNIVERSITY

Daemons

- A daemon thread supplies an extra non-core service and is supposed to run in the background just while more important threads are executing.
  - The idea is that having just daemons running is not sufficient reason for a program to continue executing.
  - Daemon threads are run for the benefit of other threads
  - they do not prevent the program from terminating, eg, the garbage collector is a daemon thread.
  - **isDaemon()** tells you whether a thread is a daemon.
  - **setDaemon()** turns the daemonhood of a thread on and off.
- If daemons create other threads then they will be daemons too.
  - E.g., if a program has a long calculation to do, you might set up a daemon to allow the user to press a button to view the current state of the calculation.

Designing for Multithreading

- ... is one of the most technically difficult problems in architecture.
  - It is important to have a "picture" of how many threads are alive at any particular time and of what they are doing.
  - Serious problems can arise with ***collisions*** or ***racing*** (when two or more threads try to work on the same object) and ***deadlock*** (when the program grinds to a halt as all threads stop to wait for certain reasons).
  - The behaviour of a multithreaded program can often surprise the designer so it is important to be extra careful.
- In the case of important software and particularly in the case of safety-critical systems (eg, aerospace, nuclear control, train/traffic signalling), formal methods are often used (and sometimes by law they must be used) to check the correctness of the design. Such software is almost always multithreaded.

Designing for Multithreading

- If we have a formal representation of a design, eg a description in a formal logical language, or a mathematical graph representation, then automated verification methods can be used to check for problems such as collisions or deadlock.
- We will have a brief look at a graphical approach.

Murdoch
UNIVERSITY

Petri Nets

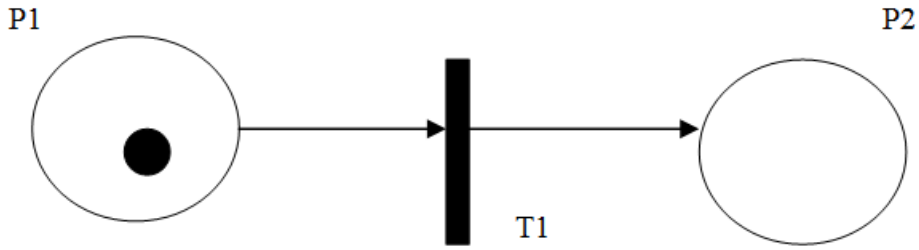
Petri Nets

- FSMs are appropriate for representing single processes. Petri Nets, which look a bit similar, are used where a number of independent but cooperating processes (or threads within the same program) require synchronization or coordination.
- Although introduced by Petri in 1962, as with most methodologies, a number of different definitions and notations have been developed.
- The basic requirement for their use is an understanding of the underlying concepts (the semantics) without allowing confusion to arise from a number of different notations (syntaxes).

Components of a basic Petri Net

- **places** = states that threads might be in;
- **transitions** = events producing changes of state(s);
- **arrows** = connections representing possible transitions between states;
- **guards** = extra conditions/results of events;
- **tokens** = temporary indicators of state.
- The places/states are modelled as circles.
- The transitions/events are denoted by bars (or rectangles/squares).
- Arrows (directed arcs) interconnect places and transitions (ie represent transition inputs and outputs.
- Tokens (dots) move from place to place within the system.

Components of a basic Petri Net



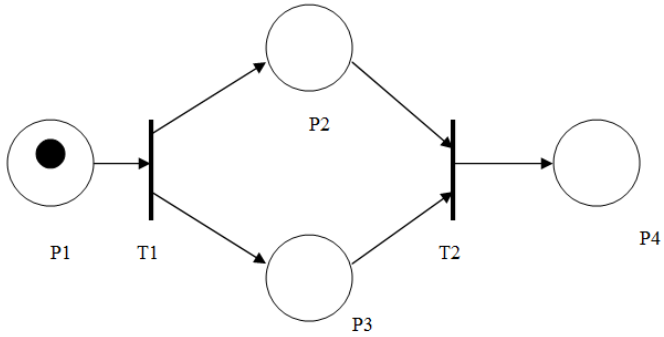
- P1 and P2 are places. T1 is a transition. Some thread is currently in the state represented by P1.
- The arrow leading into the transition represents preconditions (input places) for the corresponding event, whilst the arrow leaving the transition represents the post conditions (output places) of the event.
- Investigating the behaviour of a system modelled with Petri nets is done by executing the Petri Net – for this we need to define the actions of tokens.

Executing the Petri Net

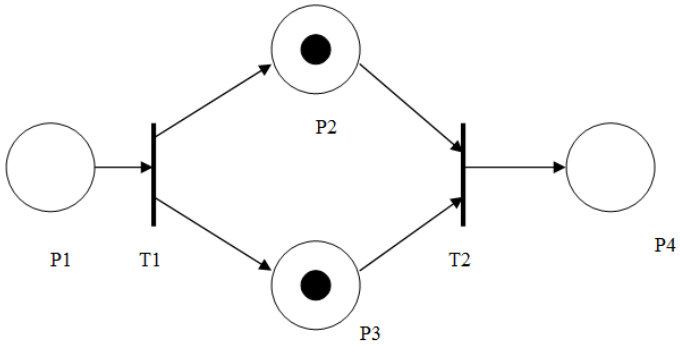
- The execution of a Petri Net is controlled by the number and distribution of tokens.
- Tokens move from place to place within the system indicating changes of state. There may be many tokens on the net at any one time.
- There will be some starting arrangement of tokens at places. The execution process is as follows:
  - At each tick of a clock we must check whether each transition *fires* or not.
  - A transition will fire if there is a token at each of the places which have an arrow leading into the transition. We say that the transition is *enabled*.
  - If a transition fires then a token is removed from each of its input places and a token is placed at each of the transition's output places (i.e. the places at ends of arrows coming out of the transition).
  - You consider each transition in some order to see if they fire. Transition firing continues as long as there is an enabled transition and then the model halts.

Multithreading example

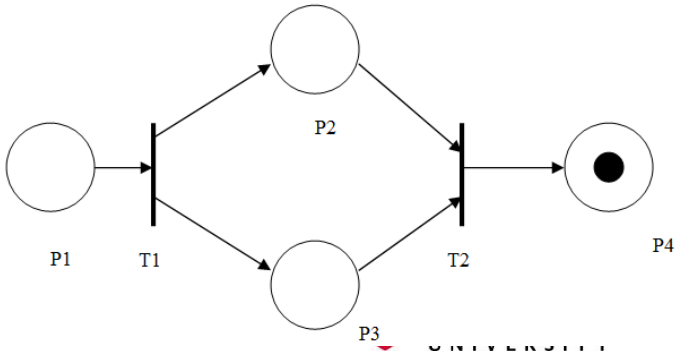
- Transition T1 is enabled, transition T2 is not.
- TICK: on receipt of a clock pulse the net fires, with the result



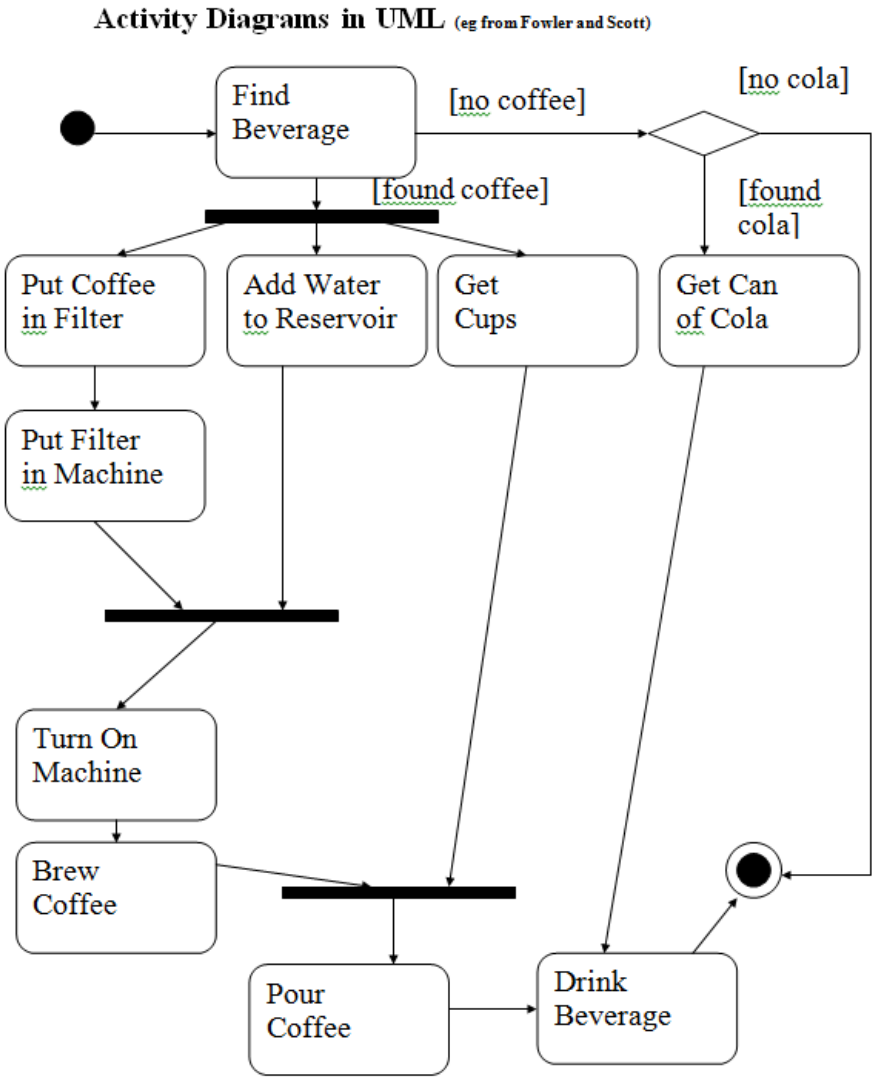
- T2 is enabled now, T1 is not. In this particular system fan-out produces a token “cloning” or replication.
- TICK:



- Tokens P2 and P3 have been merged intc one token at P4.
- Here we see multithreading.



Components of a basic Petri Net



Murdoch UNIVERSITY

Murdoch
UNIVERSITY

Synchronization

Mur
UNIV

Synchronization

- Synchronization (in a multithreaded program) means arranging for some thread(s) to wait for some others to do something before continuing.
- Why?
  - For example, several prerequisite conditions may need to be reached before a certain event can occur and the conditions may be the responsibility of several threads.
  - More importantly, in a multithreaded program there is potential for errors to arise due to **collisions**. These are when several threads try to access the same (shared) data object at about the same time. Because of the unpredictable activity of the scheduler, this can be a source of surprising errors.
    - Eg, One thread may check to see that a bank account has at least $30 in it and then debit the account by $30.
    - However, due to bad design, in between these two steps another thread (eg one responsible for collecting tax) may reduce the account balance from $30.20 to $29.50
    - Such problems can be avoided by careful programming using Thread methods like **yield**, **wait** and **join**.

Synchronization and Monitor Locks

- The problem of sharing data in a multithreaded program is solved by allowing one thread at a time exclusive access to code that manipulates the shared object.
- Java provides a synchronization facility based on **monitors** and monitor **locks**.
- The idea is that there is a ***monitor*** and a monitor ***lock*** associated with each object in the program.
  - The monitor ensures that its object's monitor lock may be held by only one thread at a time.
- Methods or other blocks of code can be declared to be ***synchronized*** and require a thread to possess the lock of a specified object before it is allowed to execute the code. When a thread comes to such a piece of code it will take the lock (if it is available) and keep it until the code is finished. If the lock is not available then the thread is **blocked** and must wait.
- This is called ***mutual exclusion*** or ***thread synchronization***.

Synchronization

- it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
- when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Synchronization Designs

- The monitor locks (for each object) idea helps with the common design approach of declaring object fields to be private.
- All requests to change the state of a particular object must then be channelled through a few public methods. If these methods are declared to be **synchronized** then there should be no **collisions** between threads trying to change the object.
- Sometimes it is useful to have more local control over synchronization. This can be achieved by declaring a block of code to be synchronized and specifying which object's monitor lock should govern entry. Eg,

  synchronized (syncObject) {block of code}

  is a statement which can appear inside any method. The variable **syncObject** determines which lock to get. A thread will be blocked on entry to the block of code until it can get hold of the specified lock.

Synchronization Designs (cont.)

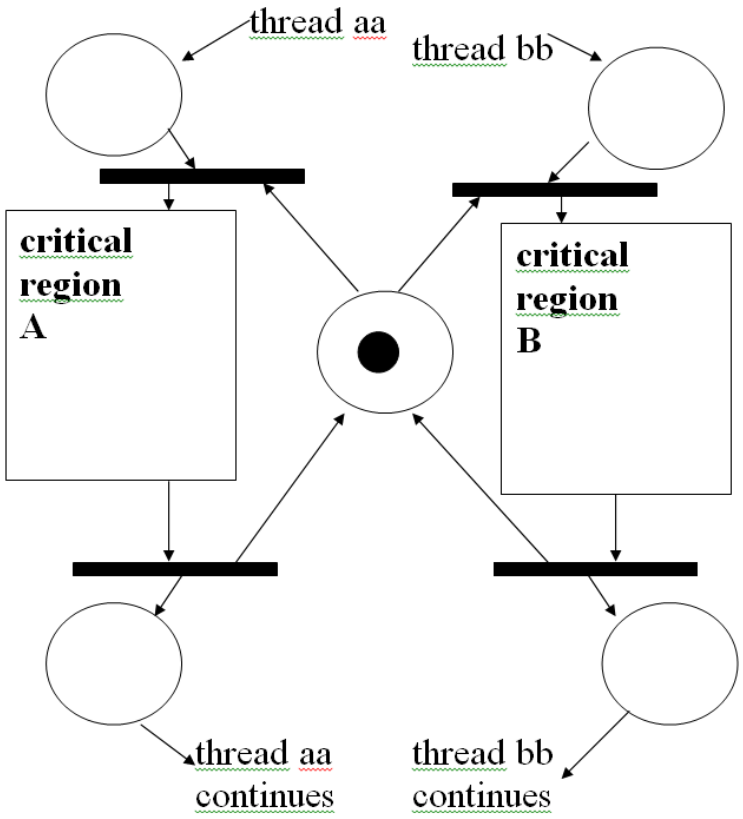
- Once again it is important to realize that the threads are being managed by having to get hold of the monitor locks. There may still be several threads executing code at once inside a synchronized block (if they didn’t have to get the same monitor lock).

Synchronization Designs (cont.)

- Even with the monitor and lock facility, it is important to be careful with multithreaded designs.
- It is inefficient to make all methods synchronized.
- However, many objects can be corrupted if their public methods are left unsynchronized. Eg, all the public methods of a Bean should be synchronized unless there are very good efficiency reasons not to.
    - Eg, the method **notifyListeners()** which loops through the list of registered listeners getting actions to be performed should not be synchronized because it might take a long time and nothing else will be able to be done with the Bean while it is going on.
    - In contrast, notifyListeners() is a dangerous method to leave unsynchronized because it has to loop through the list and the list might be changing (due to adding or removing listeners) at the same time.
    - To prevent problems here it is useful to clone the list of registered listeners (in a synchronized block) and then loop through the clone instead.

Designing for Synchronization

- Petri nets are useful for representing designs involving synchronization or the use of monitor locks.
- Eg, suppose that thread aa is not allowed in code section A at the same time that thread bb is in code section B. Here is a Petri net design to solve this problem.
- Example: TestDeadLockSolu



Murdoch UNIVERSITY

Murdoch
UNIVERSITY

Protocols

Mu
UNIV

Protocols

- *protocol* = the rules and procedures to allow communication and/or cooperation between entities.
  - Eg, in the previous Petri diagram we represented a protocol to allow two threads to proceed without something nasty happening.
- Being clear about protocols (eg, providing and checking Petri diagrams) is very important for designers of systems of communication involving distributed networks where some components/machines/users may be not well-known or not even trusted.
- Many useful protocols have been developed for such applications.
- Despite the different situation, a designer of a multithreaded program, may have to face the same issues. They may, in theory, know what each thread does individually but it is best, given the unpredictable nature of scheduling, to not trust any assumptions about what objects are being used by what threads at any time.
- Thus many of the existing communication protocols may be worth adapting for use within a multithreaded program.

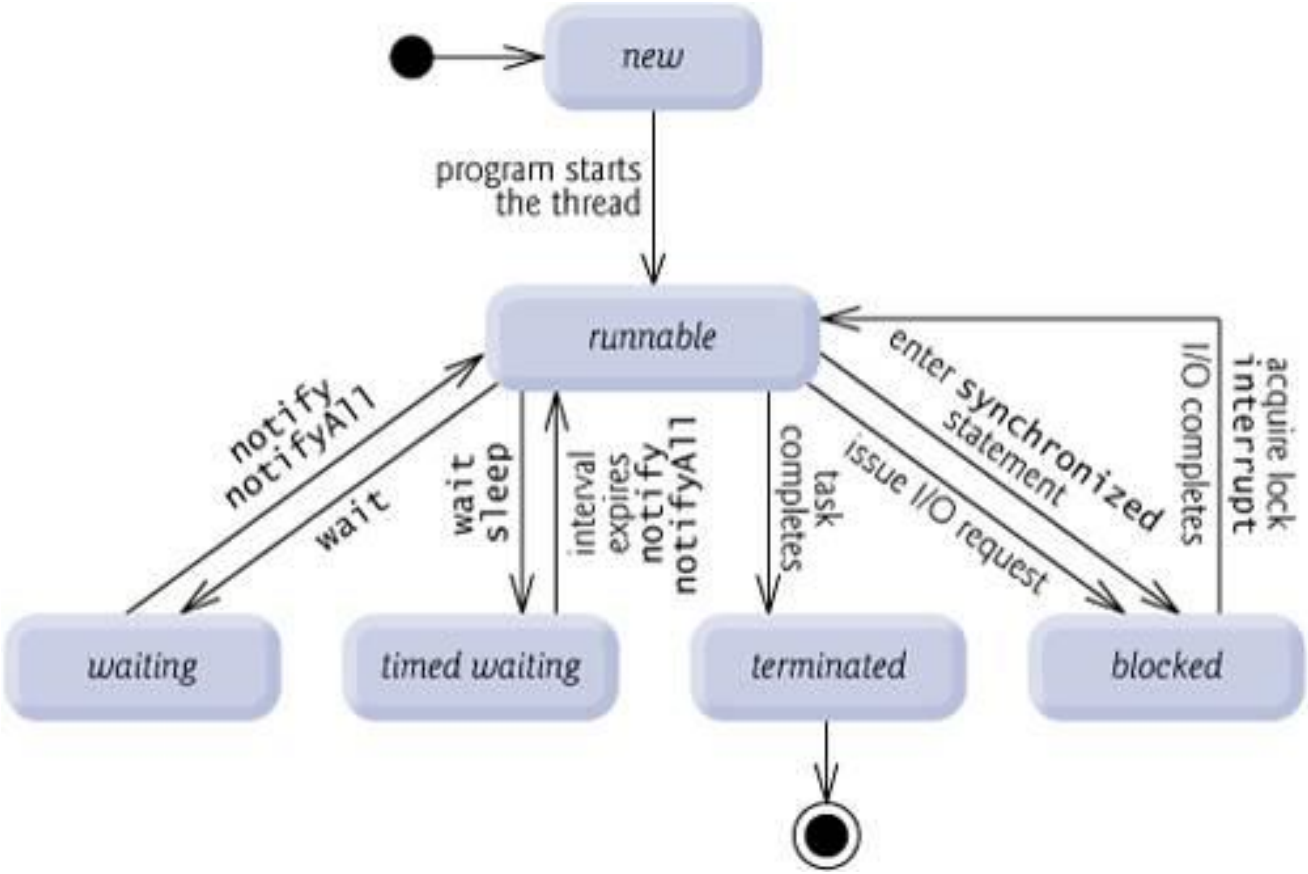
Pros and Cons of Multithreading

- Pros
  - Improved performance and concurrency.
  - Simplified coding of remote procedure calls and conversations
  - Simultaneous access to multiple applications.
  - Reduced number of required servers.
- Cons:
  - Difficulty of writing code (complexity)
  - Difficulty of managing concurrency
    - the designer has to be careful of racing, deadlock etc
  - Difficulty of porting existing code
  - slow down due to synchronized waiting for shared resources
  - slow down due to CPU overhead of thread management
  - the designer/implementer has to be able to think in a multithreaded way.
- So programming concurrent applications (multithreading) is complicated, error prone and requires careful design.

Pros and Cons of Multithreading

- Java designers recommend that the vast majority of programmers should use existing collection classes and interfaces from the concurrency APIs that manage synchronization for you.
- For advanced programmers who want to control synchronization, use the **synchronized** keyword and Object methods **wait**, **notify** and **notifyAll**.
- Only the most advanced programmers should use **Locks** and **Conditions**

Thread states and life cycle



Murdoch UNIVERSITY

Thread states and life cycle

- A thread may be in one of the following seven states:
- **New (or born) state**: the thread object has been created but not started yet.
- **Ready (or Runnable) state**: the thread moves into ready state when the thread method **start** is invoked. The thread can do something now if the scheduler (time-slicer) gives it a turn. The newly started thread and any other threads in the program run concurrently.
- **Running state**: The thread is assigned a processor (by the scheduler/operating system - also known as *dispatching the thread*) and is executing its task. Typically, each thread is given a *quantum* or *timeslice* in which to perform its task. The process that an operating system uses to determine which thread to dispatch is called *thread scheduling*.

Thread states and life cycle

- **Waiting state**: A running thread can transition to the waiting state while it waits for another thread to perform a task. It can transition back to the running/ready state only when another thread notifies it to continue executing.
- **Timed waiting state**: A running thread can enter the timed waiting state for a specified interval of time. It transitions back to the ready state when that time interval expires or when the event it is waiting for occurs. A sleeping thread remains in the timed waiting state for a designated period of time (called a sleep interval), after which it returns to the ready state.
- **Blocked state**: A running thread transitions to the blocked state when it attempts to perform a task that cannot be completed immediately and it must temporarily wait until that task completes, eg it is waiting for an I/O to complete or it is waiting for a lock.

Thread states and life cycle

- **Terminated (or dead) state**: A running thread enters the terminated state when it successfully completes its task (eg, it has finished its run() method) or otherwise terminates (perhaps due to an error). It is then disposed of by the system.

JAVA: Thread

- See the Java library (java.lang) classes **Thread** and **Object** for the various methods available. Some are:
- **start() – causes** this thread to begin execution – JVM calls the run() method of this thread.
- **sleep(long ms)** – causes the currently executing thread to temporarily cease execution (for ms millisecs) and allow other threads to execute.
- **yield()** – causes the currently executing thread to temporarily pause and allow other threads to execute.
- **wait()** – causes the current thread to wait until another thread invokes the **notify()** method or the **notifyAll()** method for this object. The **notify()** method wakes up a single thread that is waiting on this object's monitor.
- **join()** – waits for this thread to die (i.e. cease execution).

Deadlock

- In a multithreaded program, *deadlock* = a situation in which a group of threads are all blocked waiting for each other.
- Allowing deadlock is a design error.
- Note that this may happen to some of the threads in the program (in which case you might not notice that it is behaving incorrectly) or to all the threads in the program (in which case the program will not do anything).

- See the company ownership example, which follows.

Priorities

- It is sometimes useful to specify that certain threads are more important than others. In Java we can do this via priorities.
- At any moment if thread A has a higher priority than thread B then, if A is runnable (i.e. not blocked), then B will not get any time allocated. Time is only scheduled to the runnable threads with top priority. The way time is scheduled between equal top priority runnable threads depends on which scheduling mechanism the JVM is using (this varies: see text book for details).
- You can find out and change priorities with: **getPriority()** and **setPriority();**

Priorities (cont.)

- There are fixed constants in the Thread class called

  Thread.MAX_PRIORITY (a constant of 10) and
  Thread.MIN_PRIORITY (a constant of 1) which indicate limits on the priority values.

- By default, each thread starts out with the same priority value, which is **Thread.NORM_PRIORITY** (a constant of 5). Each new thread inherits the priority of its parent thread. Priorities are used to make sure that important tasks are immediately acted upon when they can proceed.
- If you want a rough way to control the speed of animations and clock-like objects etc, and a way of making sure that some sorts of user inputs are monitored much more carefully (and speedily) than others don't use priorities, use the **sleep** method.

Too Many Threads

- In most real-life applications multithreading adds convenience by allowing a fixed small number of different threads (i.e. different things) to be going on at once.
- The extra computing overhead of the scheduling will not be noticed.
- However, it is very easy for a programmer to overload the multithreading facility in a program by deliberately setting out to do this or by being extremely careless with the creation of threads. The extra scheduling computations will definitely begin to show up when a program has 100s of threads.
- When you are aware of this limit, and realise the extra work that multithreading requires, you should realise that such a large number of threads is a bad design.

Too Many Threads

- Remember that multithreading does not give you parallel computing: it just might look a bit like parallel computing if used sparingly. In fact all the work is being done one step at a time as usual but there are extra things being done as well. So for doing a big calculation (eg, involving each element of an array), it is usually better to put it in one thread.
- Even a GUI with a lot of components should be treated carefully.

Adding Sleep to speed things up

- When multithreading does seem to be getting bogged down, then it is often possible to speed things up by making sure that the threads go to sleep as much as possible.
- Knowing that other threads are asleep allows the scheduler to keep on working through one thread and avoid all the time-consuming business of changing threads. The scheduler will be notified when a thread wakes up.

JavaFX & Multithreading

JavaFX Scene Graph Is Not Thread-safe

- It is modeled to execute on the single JavaFX Application Thread. The constructor and the initialization method *init()* is called in the JavaFX-Launcher thread.
    - Implementing long-running tasks on the JavaFX Application thread inevitably makes an application UI unresponsive.
    - A best practice is to do these tasks on one or more background threads and let the JavaFX Application thread process user events

JavaFX & Multithreading

Thread Confinement

- It is a technique that allows only one thread to access the thread-unsafe part of the code, thus ensuring a simple way to achieve thread safety.
- This, however, is a common technique applied in many other areas of Java programming.
- Any data in thread confinement is also called *thread local*. It is thread-safe because races cannot exist in a thread local environment and the data does not need to be locked.

Implementing Multithreading in JavaFX

The *runLater* Method

- One technique is to invoke the *static void runLater (Runnable runnable)* method defined in the *Platform* class.
- this method runs *the specified Runnable on the JavaFX Application Thread at some unspecified time in the future.*
- should perform only small updates to allay the risk.

Using *Task* and *Worker*

- There is an interface called *Worker*, an abstract class called *Task*, and *ScheduledService* for this purpose.
- The *Task* is basically a *Worker* implementation, ideal for implementing long running computation. The *Task* class extends *FutureTask*, and, as a result, supports *Runnable*, *Future*, and *RunnableFuture* interfaces as well.
- Due to its legacy/inheritance, this class can be used in various ways.
- Example: JavaFXThread1/2/3; FXMultithreading

Murdoch UNIVERSITY

Murdoch
UNIVERSITY

Summary

Summary

- This topic looked at splitting a processing task (a process) into a number of separate threads each of which do part of the task and can share the processing time.
- The threads should be loosely coupled so they can work independently unless they need to share some resource such as an object in memory.
- When sharing resources synchronization can be used to ensure the threads preserve the integrity of the system.
- Petri nets and UML can be used to describe thread interaction.

Murdoch

UNIVERSITY