UNIVERSITY OF
TORONTO
MISSISSAUGA

CSC207

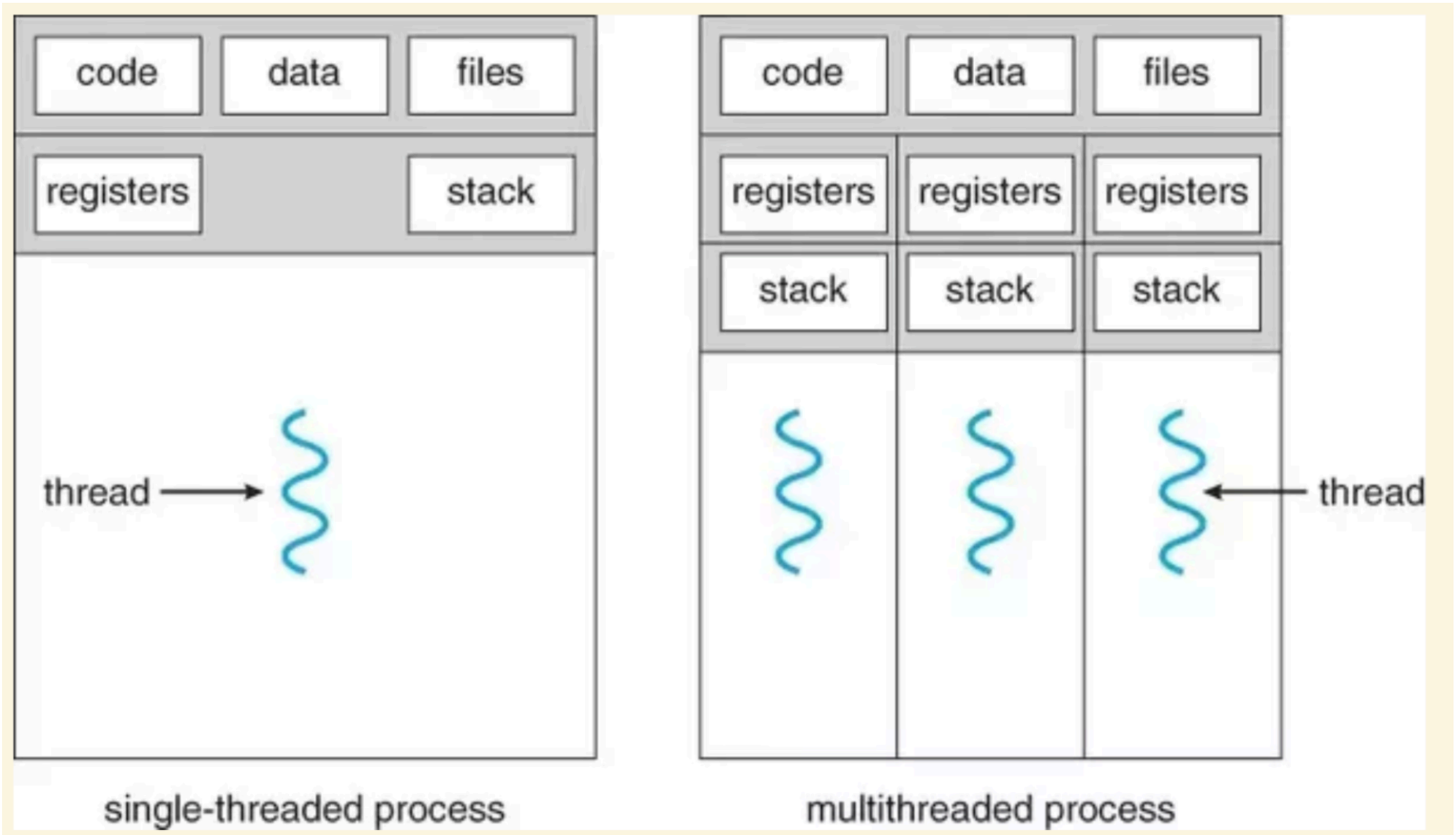
Threading, Concurrency Patterns

Learning Objectives for Today

- What is a thread?
- Thread handling

UNIVERSITY OF TORONTO MISSISSAUGA

Threads



Single Threads

The CPU has a collection of registers, including a program counter.

Each program has its own call stack.

- It fetches instructions from memory (based on the program counter) and carries them out.

- To carry out an instruction, it may fetch data into registers, modify registers and then store data back into memory.

Multiple Threads

Each thread has a collection of registers, including a program counter.

Each thread has its own call stack.

When a Thread is running, the CPU loads the Thread registers, including the program counter.

- The CPU fetches instructions from memory and carries them out.
- To carry out an instruction, it may fetch data into the threads registers, modify the threads registers and then store data back into memory.
- Code and Data are shared between threads.

Why Use Multiple Threads?

Want to logically handle multiple independent threads of execution.

Example: GUI should remain responsive, even while application is processing images.

One thread for GUI, one for processing images.

Example: Handle multiple simultaneous TCP connections 'at the same time'.

CPU has multiple cores, so we can improve performance by applying them to one task. Decrease latency, increase throughput.

The Runnable Interface

A *Runnable object* represents a *task* that can execute concurrently with other tasks.

The *Runnable interface* declares the single method *run*, which contains the code that defines the task that a Runnable object should perform.

When a thread executing a Runnable is created and started, the thread calls the Runnable object's run method, which executes in the new thread.

An Example

We compute sum of numbers from 0 to 100000000000L-1

On single thread, it takes about 25 sec on a Mac laptop with 16 GB RAM.

On two threads, it takes about 14 sec.

A Thread's lifecycle

A new thread begins its life cycle in the new state.

When started (to *runnable state*) executes its task.

A *runnable* thread can transition to the *waiting state* waiting for other threads.

Transitions back to the runnable when notified by other thread(s).

A Thread's lifecycle

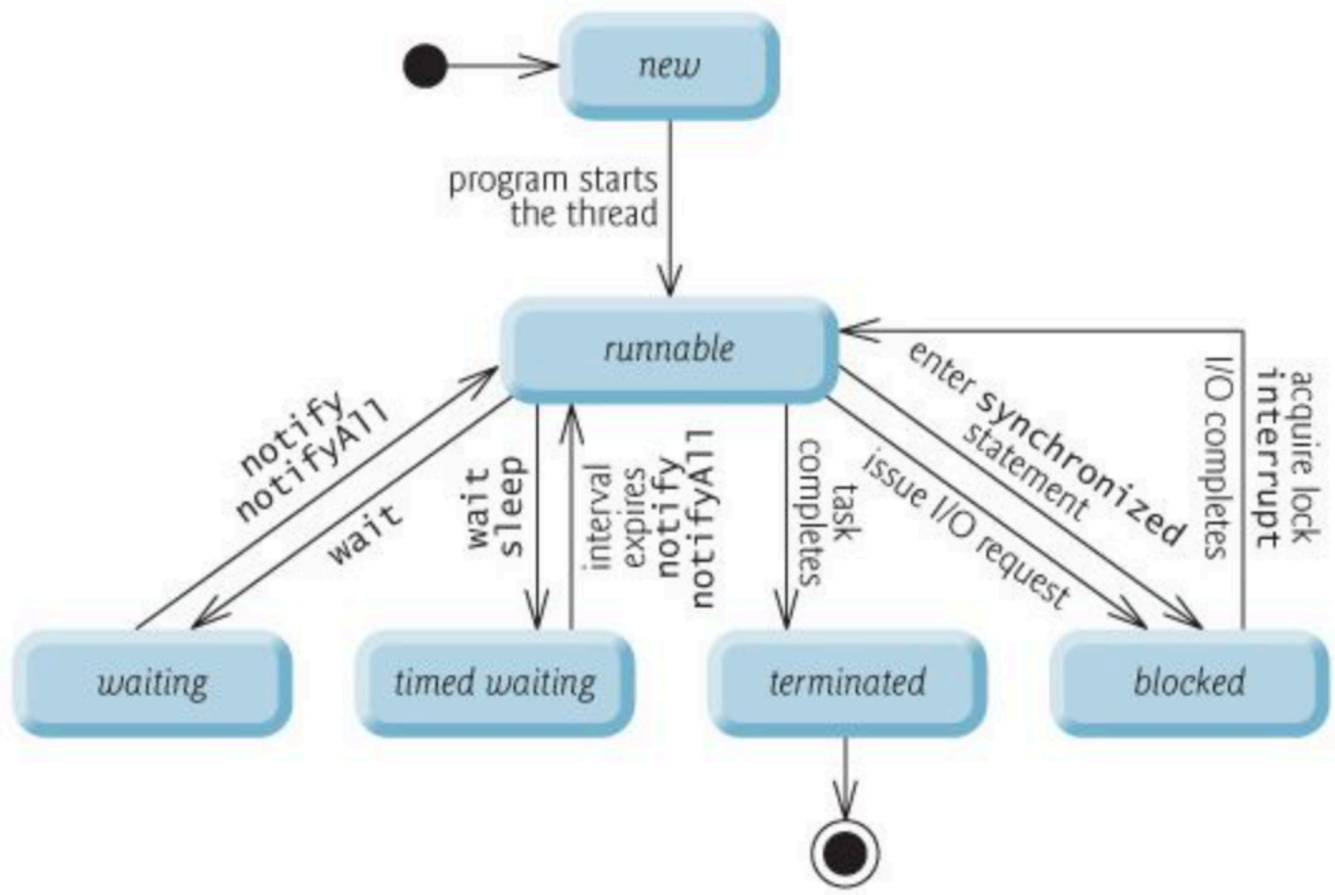
A *runnable* thread can enter the *timed waiting state* for a specified interval of time.

- Goes back to the runnable state only if time expires.
- Cannot use a processor, even if one is available.

A runnable 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.

A runnable thread enters the *terminated state* when it completes its task or otherwise terminates.

A Thread's lifecycle



OS view

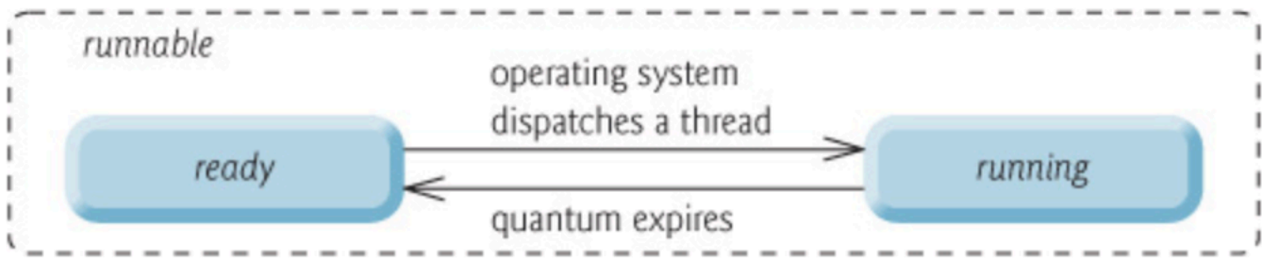
At the operating-system level, each Java runnable is executed in two phases.

When a thread begins to run, it starts in the **ready** state.

A **ready** thread enters the **running** state when the OS assigns it to a process.

Each thread is given a time slice to perform its task.

OS view



Synchronization

Threads communicate via shared data.

There are problems with this though!!

When multiple threads share an object and it is modified by one or more of them, indeterminate results may occur unless access to the shared object is managed properly.

Example

```
Withdraw(account, amount):
1    balance = get_balance(account)
2    balance = balance - amount
3    put_balance(account, balance)
4    return balance
```

```
Deposit(account, amount):
5    balance = get_balance(account)
6    balance = balance + amount
7    put_balance(account, balance)
8    return balance
```

Let's say I have **$1000** originally, then I called

`Withdraw(acc, 100)`

and

`Deposit(acc, 100)`

concurrently

How much money do I have now?

Example

Possible schedule A

```
1    balance = get_balance(acc)
2    balance = balance - 100
```

```
5    balance = get_balance(acc)
6    balance = balance + 100
7    put_balance(acc, balance)
```

```
3    put_balance(acc, balance)
```

Possible schedule B

```
1    balance = get_balance(acc)
2    balance = balance - 100
```

```
5    balance = get_balance(acc)
6    balance = balance + 100
```

```
3    put_balance(acc, balance)
```

```
7    put_balance(acc, balance)
```

Final Balance with Schedule A?

Final Balance with Schedule B?

UNIVERSITY OF TORONTO MISSISSAUGA

Thread Synchronization

The problem can be solved by giving only one thread at a time exclusive access to code that accesses the shared object.

During that time, other threads desiring to access the object are kept waiting.

When the thread with exclusive access finishes accessing the object, one of the waiting threads is allowed to proceed.

Thread Synchronization

This process, called thread synchronization, coordinates access to shared data by multiple concurrent threads.

Ensures that each thread accessing a shared object; excludes all other threads from doing so simultaneously - this is called mutual exclusion.

Thread Synchronization

- In the example threading code attached to the Quercus website, all threads share access to the object *Sum*.
  - When we ask the Sum to “add”, the CPU has to ...
    - Retrieve a value into a register
    - Retrieve another value into a register
      - Add the two registers
      - Store the incremented value
    - If two threads are executing these steps at the same time, wrong values may be stored.

Monitors

A common way to perform synchronization is to use Java's built-in monitors.

Every object has a monitor and a monitor lock (or intrinsic lock).

- Can be held by a maximum of only one thread at any time.
- A thread must acquire the lock before proceeding with the operation.
- Other threads attempting to perform an operation that requires the same lock will be blocked.

Monitors

To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a synchronized statement.

This is guarded by the monitor lock

```
synchronized (object)
{ statements }
// end synchronized statement
```

When a synchronized statement finishes executing, the object's monitor lock is released.

Concurrency Pattern: Producer/Consumer Pattern

In a producer/consumer relationship, the producer portion of an application generates data and stores it in a shared object, and the consumer portion of the application reads data from the shared object.

A producer thread generates data and places it in a shared object called a buffer.

A consumer thread reads data from the buffer.

Concurrency Pattern: Producer/Consumer Pattern

This relationship requires synchronization to ensure that values are produced and consumed properly.

Operations on the buffer data shared by a producer and consumer thread are also state dependent - the operations should proceed only if the buffer is in the correct state.

If the buffer is in a not-full state, the producer may produce.

If the buffer is in a not-empty state, the consumer may consume.

Concurrency Pattern: Producer/Consumer Pattern