Kotlin

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today's Lecture

- Thread Overview
- Thread Synchronization

Question

If the worker at the counter needs to go to the stock room to get something for a customer, what will happen to the other customers (assume only one worker at the counter)?

Worker Serving Customers

Question

If the worker at the counter needs to go to the stock room to get something for a customer, what will happen to the other customers (assume only one worker at the counter)?

ANSWER: Everything stops! All customers must wait.

Worker Serving Customers

- A better solution would be to have the customer service worker ask another worker to get the item for them.
- If this happens then the customer service worker can serve other customers and keep the customer service line moving.
- They can finish with the original customer when the other worker returns with the item.

- The one downside is that using another worker requires coordination between the two workers.

User Another Worker to Help

- Two things are being done at once in real time.
  - One worker is retrieving an item for a customer.
  - Another worker is serving customers at the custom service counter.
- If there was only one worker, then they must either retrieve the item or serve customers at the counter. They cannot do both.
- The customer service line would be "blocked" until the worker returns.

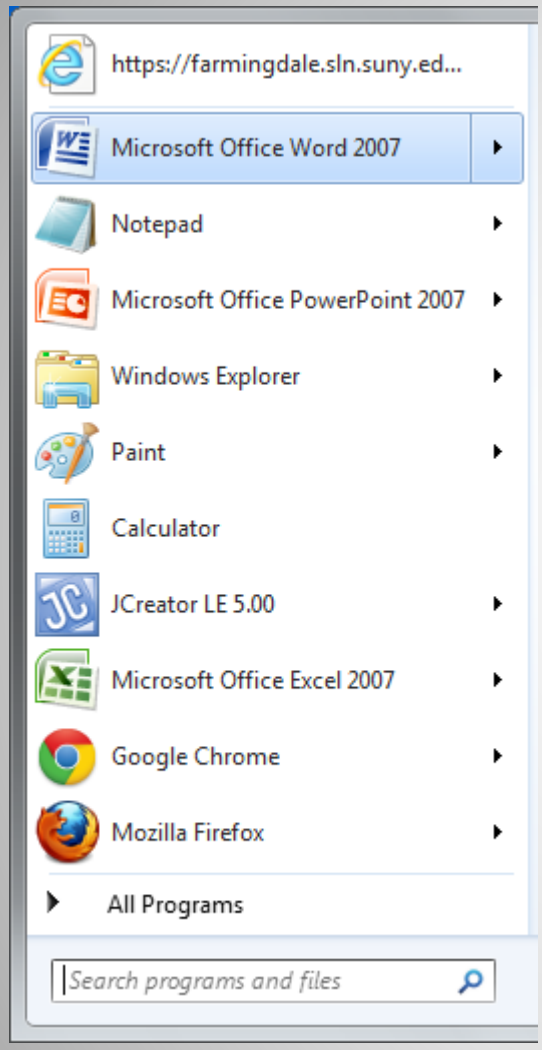
User Another Worker to Help

Threads vs Workers

- Threads in a computer program are like workers at a store.
- Similar to how we can use multiple workers to take care of something, a program can use multiple threads to take care of something.
- One downside to using multiple threads is that there must be coordination between the threads (just like workers would need to coordinate with each other in the previous example).

- Now on to OS processes…

OS Processes



- What happens when you click the icon to run an application?
- Important to understand how a program actually runs behind the scenes.

Operating Systems

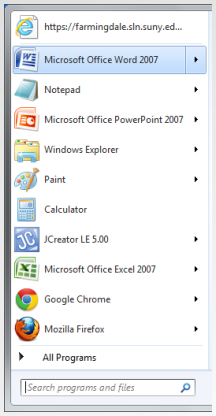
Running a Program

- When a program is started the OS does the following:
  - Creates a process for the program.
  - Copies the program from external memory (hard drive, flash drive etc…) into RAM.
- A process is an instance of a running program.

Running a Program

Hard Drive

File
Word.exe

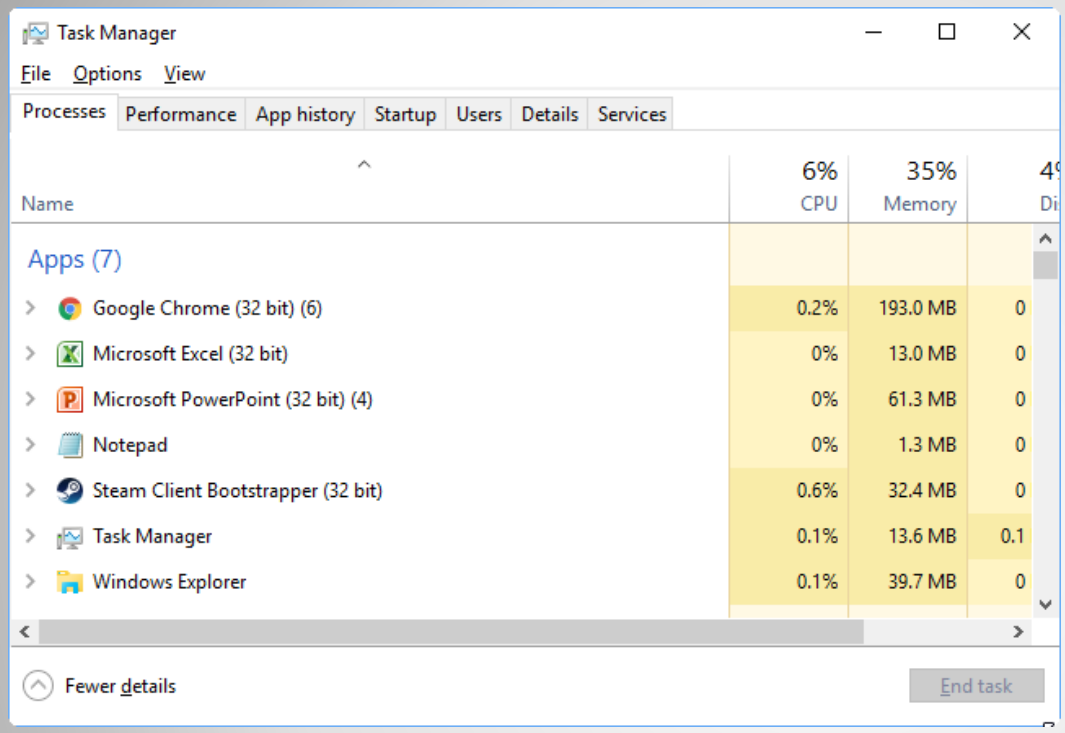


Click program to run (word.exe)

OS creates a process for the program contained in Word.exe

Memory (RAM)

Process
Code from Word.exe



Windows Task Manager shows information about the currently running processes

Note: Windows Task Manager has nothing to do with C# Tasks

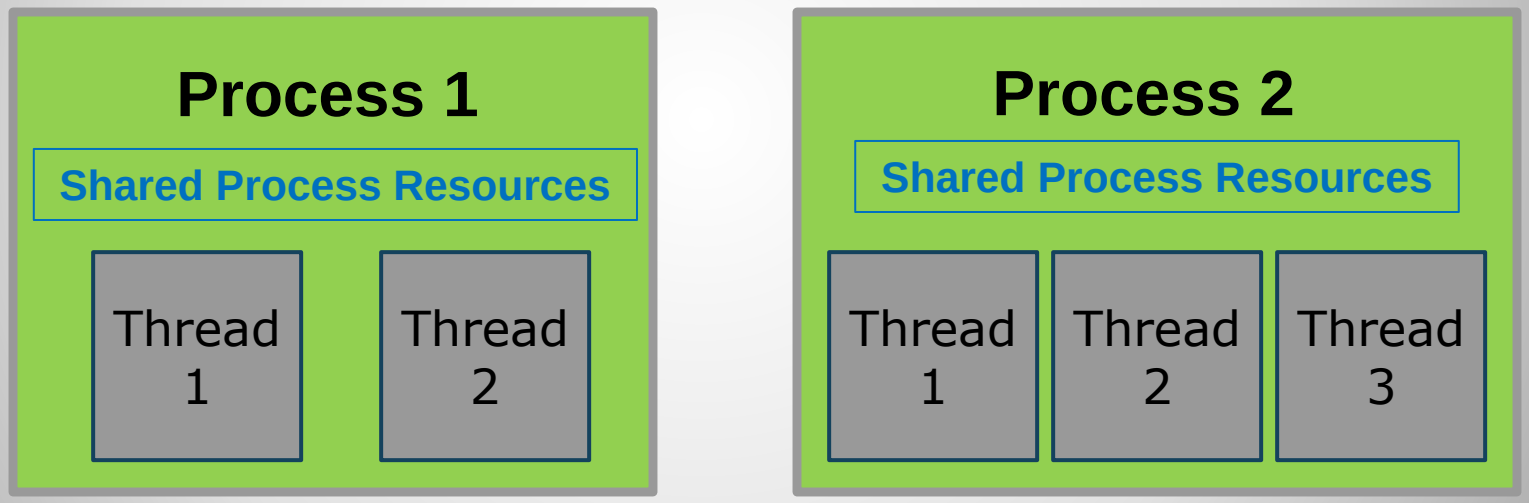
OS Controls Processes

- Now on to threads…

Threads

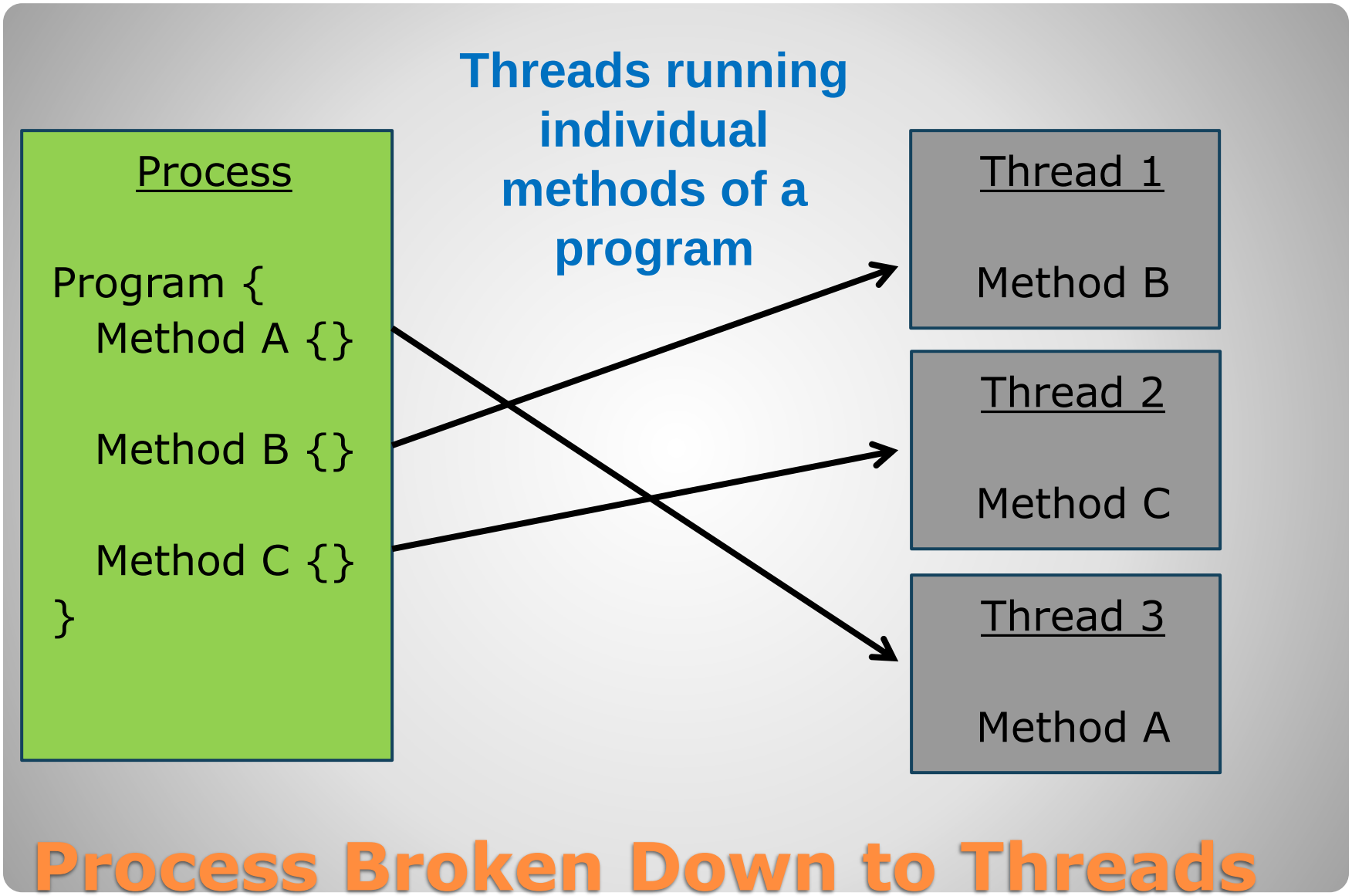
Processes and Threads

- Processes are broken down to threads.
- Each process can contain multiple threads.
- Threads can be scheduled individually.
- The number of threads in each process can vary.



Thread

- **Thread** – A smaller unit of a process which can be scheduled and executed.
- Process Resources Shared- Threads of a process share the resources of the containing process.
- Process can have more than one thread active.
- You can setup threads to run a particular method in a program.
- For example…

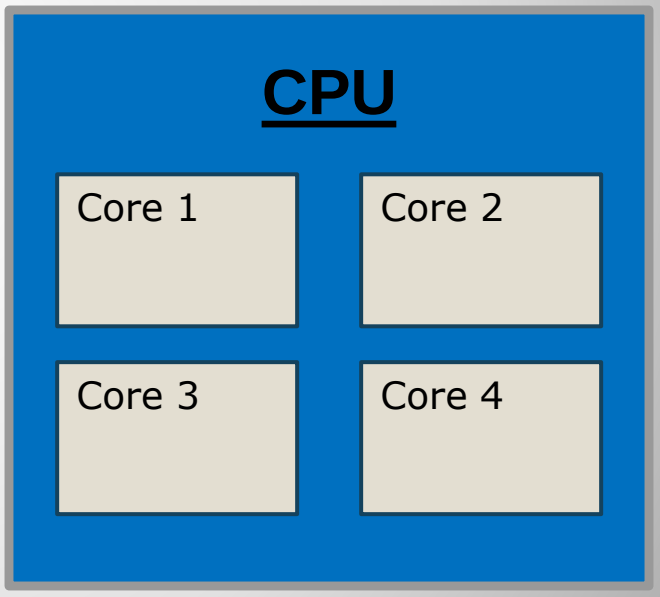


Process Broken Down to Threads

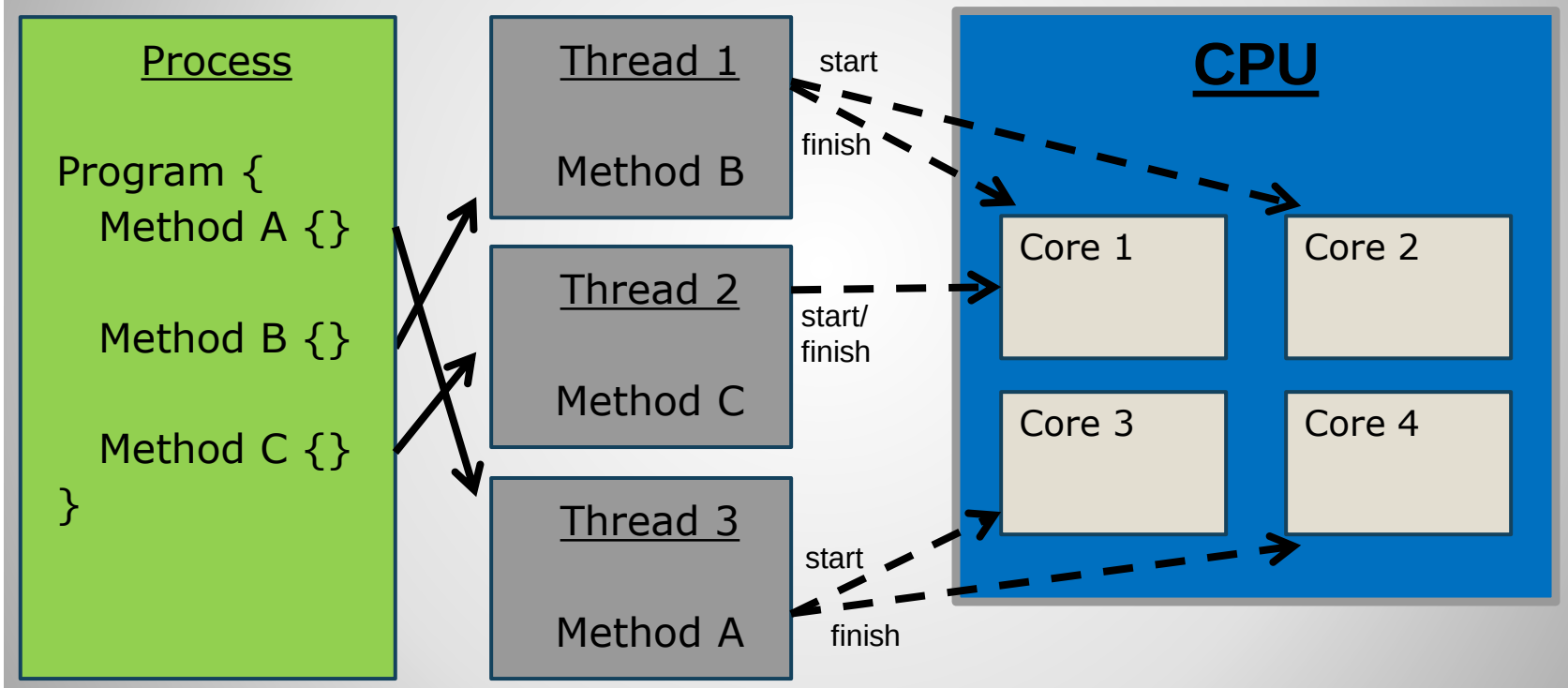
CPU with Multiple Cores

- A CPU is responsible for actually running the code contained in processes and threads.
- A CPU can have multiple cores (a core is like a mini-CPU)
- Each core can run a thread of its own.
- The cores run independently.

This CPU has 4 cores



Process can be broken down into threads. Threads can start execution on one core and finish on another core.

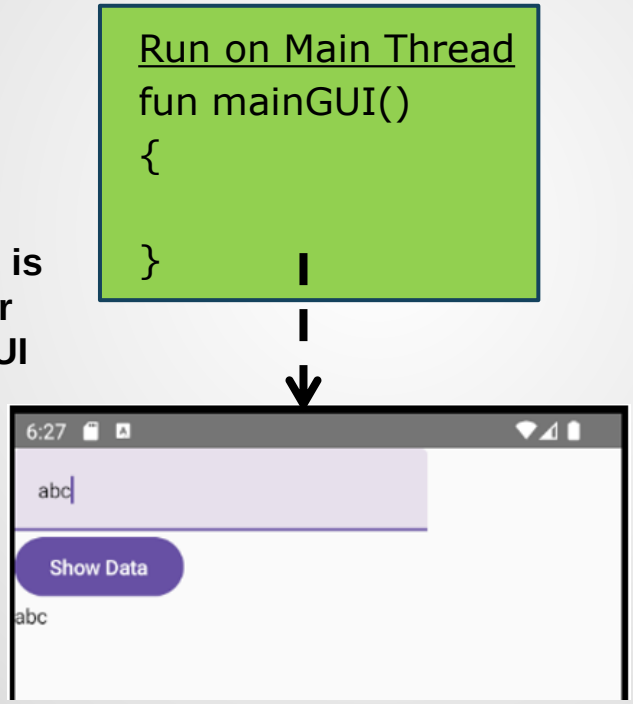


Threads Running Methods

Main Thread and GUI

- The main thread is responsible for updating the GUI.
- Long running operations should be offloaded from the main thread to other threads.
- If a long running operation is done on the main thread the user cannot interact with the GUI. The GUI "hangs" in this case.

The main thread is responsible for updating the GUI



mainGUI cannot do anything until DoSomething returns

```
Run on Main Thread
fun mainGUI()
{
   DoSomething();
}
```

Wait for DoSomething to finish

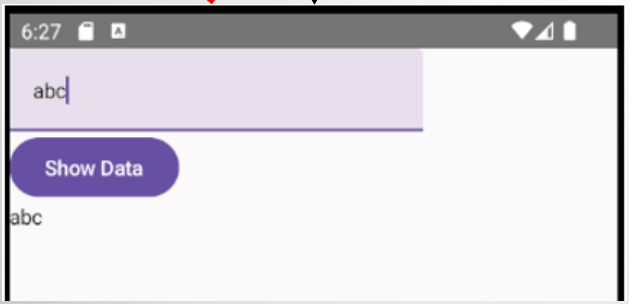
1. Main calls DoSomething synchronously

2. DoSomething runs (mainGUI is blocked)

```
Run on Main Thread
void DoSomething()
{
   // Code for a long
   // running operation
   // goes here…
}
```

3. DoSomething returns and main continues execution

mainGUI function cannot update GUI while waiting (user cannot interact)



The user will NOT be able to interact with the GUI while DoSomething is running! GUI HANGS, VERY BAD USER EXPERICENCE!

Long Running Synchronous Method Call on GUI Thread (BAD)

mainGUI continues immediately (does not wait for DoSomething to return)

Run on Main Thread

```
fun mainGUI()
{
   DoSomething();
}
```

Keep going! Do not wait for DoSomething to finish

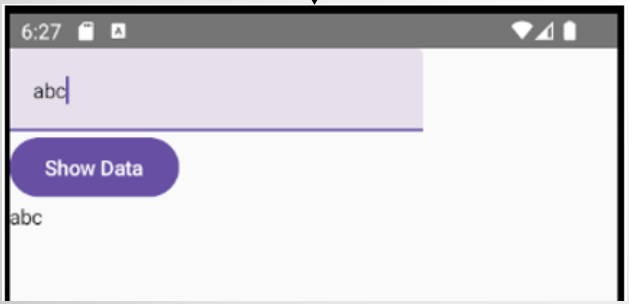
1. Main calls DoSomething on another thread

2. DoSomething runs (mainGUI is NOT blocked)

Run on Another Thread

```
void DoSomething()
{
   // Code for a long
   // running operation
   // goes here…
}
```

mainGUI function does not wait and keeps updating GUI (user can interact)



The user can interact with the GUI (not locked). Creates a good user experience.

Long Running Method Call on Another Thread (GOOD)

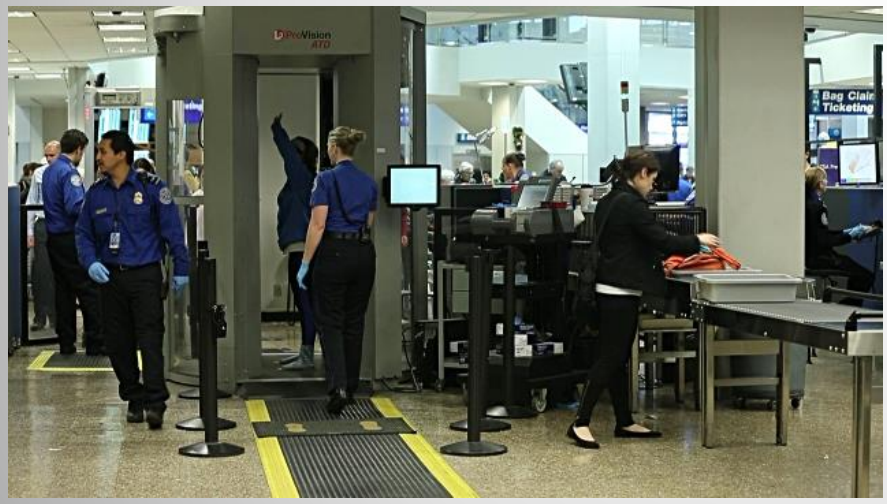
- Now on thread synchronization…

Thread Synchronization

Multithreading Issues

- Multithreading Tradeoff - A multithreaded program gives the benefit of doing two things at once, but it introduces more complexity into the program.
- A multithreaded program most likely contains timing issues that are not present in a single threaded program.
  - For example, if thread 1 needs data from thread 2 then thread 1 needs to "wait" for thread 2 to finish before it can execute.
  - These types of timing issues can become very challenging to deal with.
- Imagine a program with 20 threads and multiple timing dependencies.
- Theses types of programs are much harder to debug.

- **Critical Section** - A section of code that can only be entered by one thread at a time.
- This section of code is mutually exclusive, only one thread is allowed in.
- For example, a security check line scanner only allows one person to be scanned at a time.
- Another example is a one lane bridge (allows only one car).



Critical Section

Thread Synchronization

- **Thread synchronization** – Make sure that only one thread can access a critical section at any one moment in time.
- Once one thread leaves the critical section another can enter the critical section.
- Can use a synchronized block in Kotlin to perform thread synchronization.
- For example…

- End of Slides

End of Slides