Programming Languages

Concurrency

CSCI-GA.2110-001
Spring 2023

Concurrent programming

- refers to the handling of multiple independent activities
  - contrast to *parallelism*—simultaneous execution of independent activities.
- a *task* (Ada) or *thread* (Java,C++,C#) is an independent execution of the same static code, having a stack, program counter and local environment, but shared data.
- Ada tasks communicate through
  - rendezvous (think "meeting someone for a date")
  - shared variables
  - protected objects
- Java, C++, C# threads communicate through shared objects.
- C++ also supports promises and futures (C++11).
- C# employs asynchronous programming on a single thread (it also supports multi-threading).

Task Declarations (Ada)

A task type is a limited type

```
task type Worker;         -- declaration;
                          --  public interface

type Worker_Id is access Worker;

task body Worker is       -- actions performed in lifetime
begin
  loop                    -- Runs forever;
    compute;              --  will be shutdown
  end loop;               --  from the outside.
end Worker;
```

More Task Declarations

- a task type can be a component of a composite
- number of tasks in a program is not fixed at compile-time.

```
W1, W2: Worker;  -- two individual tasks

type Crew is array (Integer range <>) of Worker;

First_Shift: Crew (1 .. 10); -- group of tasks

type Monitored is record
  Counter: Integer;
  Agent: Worker;
end record;
```

Task Activation

When does a task start running?

- if statically allocated $\Longrightarrow$ at the next `begin`
- if dynamically allocated $\Longrightarrow$ at the point of allocation

```
declare
   W1, W2: Worker;
   Joe: Worker_Id := new Worker; -- Starts working now
   Third_Shift: Crew(1..N);      -- N tasks
begin       -- activate W1, W2, and the Third_Shift
   ...
end;        -- wait for them to complete
            -- Joe will keep running
```

Task Services

- a task can perform some actions on request from another task
- the interface (declaration) of the task specifies the available actions (entries)
- a task can also execute some actions on its own behalf, without external requests or communication

```
task type Device is
  entry Read (X: out Integer);
  entry Write (X: Integer);
end Device;
```

Synchronization: The Rendezvous

- caller makes explicit request: *entry call*
- callee (server) states its availability: *accept statement*
- if server is not available, caller blocks and queues up on the entry for later service
- if both present and ready, parameters are transmitted to server
- server performs action
- `out` parameters are transmitted to caller
- caller and server continue execution independently

Example: semaphore

Simple mechanism to prevent simultaneous access to a *critical section*: code that cannot be executed by more than one task at a time

```
task type semaphore is
  entry P;    -- Dijkstra's terminology
  entry V;    -- from the Dutch
  -- Proberen te verlangen (wait) [P];
  -- verhogen [V] (post when done)
end semaphore;

task body semaphore is
begin
  loop
    accept P;
      -- won't accept another P
      --  until a caller asks for V
    accept V;
  end loop;
end semaphore;
```

Using a semaphore

- A task that needs exclusive access to the critical section executes:

```
Sema : semaphore;
...
Sema.P;
-- critical section code
Sema.V;
```

- If in the meantime another task calls `Sema.P`, it blocks, because the semaphore does not accept a call to `P` until after the next call to `V`: the other task is blocked until the current one releases by making an entry call to `V`.
- programming hazards:
  - someone else may call `V` $\Longrightarrow$ race condition
  - no one calls `V` $\Longrightarrow$ other callers are *livelocked*

Delays and Time

- A `delay` statement can be executed anywhere at any time, to make current task quiescent for a stated interval:

```
delay 0.2;   -- type is Duration, unit is seconds
```

- We can also specify that the task stop until a certain specified time:

```
delay until Noon;   -- Noon defined elsewhere
```

Conditional Communication

- need to protect against excessive delays, deadlock, starvation, caused by missing or malfunctioning tasks
- timed entry call: caller waits for rendezvous a stated amount of time:

```
select
  Disk.Write(Value => 12,
             Track => 123); -- Disk is a task
or
  delay 0.2;
end select;
```

- if `Disk` does not accept within 0.2 seconds, go do something else

Conditional Communication (ii)

- conditional entry call: caller ready for rendezvous only if no one else is queued, and rendezvous can begin at once:

```
select
  Disk.Write(Value => 12, Track => 123);
else
  Put_Line("device busy");
end select;
```

- print message if call cannot be accepted immediately

Conditional communication (iii)

- the server may accept a call only if the internal state of the task is appropriate:

```
select
  when not Full =>
    accept Write (Val: Integer) do ... end;
or
  when not Empty =>
    accept Read (Var: out Integer) do ... end;
or
  delay 0.2;  -- maybe something will happen
end select;
```

- if several guards are open and callers are present, any one of the calls may be accepted – non-determinism

Concurrency in Java

- Two notions
  - `class Thread`
  - `interface Runnable`
- An object of class `Thread` is mapped into an operating system primitive

```
interface Runnable {
  public void run ();
}
```

- Any class can become a thread of control by supplying a `run` method

```
class R implements Runnable { ... }
```

```
Thread t = new Thread(new R(...));
t.start();
```

Threads at work

```java
class PingPong extends Thread {
  private String word;
  private int delay;
  PingPong (String whatToSay, int delayTime) {
    word = whatToSay;  delay = delayTime;
  }

  public void run () {
    try {
      for (;;) { // infinite loop
        System.out.print(word + " ");
        sleep(delay); // yield processor
      }
    } catch (InterruptedException e) {
      return; // terminate thread
    }
  }
}
```

Activation and execution

```
public static void main (String[] args) {
  new PingPong("ping", 33).start();  // activate
  new PingPong("pong", 100).start(); // activate
}
```

- call to `start` activates thread, which executes `run` method
- Do not call `run` directly: it will execute like an ordinary method with no new thread.
- threads can communicate through shared objects
- classes can have synchronized methods to enforce critical sections

Volatile variables

Consider the following fragments executing in 2 threads:

```
while (keepgoing)
{
 // processing loop
}
```

```
...
if (condition)
  keepgoing = false;
...
```

- Concurrency + optimization complicates things:
  - One thread may maintain a variable in a hardware register while the other reads/writes from memory.
  - Can result in dirty reads and data corruption
  - E.g., loop above may never terminate
- Keyword `volatile` prevents the compiler from optimizing.
  - `private volatile bool keepgoing;`
- Works with primitives and non-primitives (unlike `synchronized`).
- Still need to synchronize—`volatile` does not provide atomic locking.
- Seen in many languages [**Java**, **C**, **C++**, **C#**].

Mutual exclusion

- Declare a method as `synchronized`
  - Locks the object whose `synchronized` method is running.
  - *All* other `synchronized` regions for “this” object will block.

```
public synchronized void foo(...) {
  // entire method protected
}
```

- Declare a defined region as `synchronized`
  - Similar to above, but programmer specifies the exact region to protect.
  - Programmer specifies the lock object.

```
synchronized (obj) {
  // protected region here
}
```

- Fairness is **not** guaranteed with `synchronized`. Use `ReentrantLock`.

Java notify/wait pattern

Used to achieve synchronization on an object across threads.

- `wait` releases the lock of a specified object and blocks until a `notify` event from another thread. Then reacquires the lock.
- `notify` wakes up an arbitrary waiting thread. No fairness.
- `notifyAll` wakes up all waiting threads.

Typical notify pattern:

```
synchronized (obj)
{
    // set the condition
    readyToConsume = true;

    obj.notify();
}
```

More Java notify/wait

Typical wait pattern:

```
synchronized (obj)
{
    while( ! readyToConsume )
    {
        obj.wait();
    }

    // perform sync action here
}
```

- `synchronized` prevents race conditions (all threads must agree on the state of the predicate).
- Condition is needed to address spurious `wait` wake ups.
- Condition variables should be `volatile`.
- Easy to introduce bugs by not following the proper pattern.
- Pattern seen in many languages [**Java**,**C++**,**Scala**,**PHP**]

Threads in C++11

- C++ didn't have native thread support until C++11.
- Previously had to use external libraries like `pthreads`, `Boost` `OpenThreads`, etc.
- Full state-of-the-art thread support now included in C++.
- One-to-one mapping to operating system threads.
- Based on the Boost thread library.

Example Thread Class

```
 class Runnable
{
   std::thread mthread;
   Runnable(Runnable const&) = delete;
   Runnable& operator =(Runnable const&) = delete;

public:
    virtual ~Runnable() { try { stop(); }
                   catch(...) { /* clean up */ } }

    virtual void run() = 0;
    void stop() { mthread.join(); }
    void start()
     { mthread = std::thread(&Runnable::run, *this); }
};
```

Use of Thread Class

```
class myThread : public Runnable
{
  protected:
   void run() { /* do something */ }
};
```

Mutual exclusion can be acheived as follows:

```
static std::mutex pmm;

void mySynchronizedFunction() {
   std::lock_guard<std::mutex> myLock(pmm);
   // critical area
   // unlocked automatically on return
}
```

Automatic Threads & Futures

```
int main()
{
 std::future<bool> sol=std::launch::async(hamcycle);
 do_other_stuff(); // while hamcycle is computing
 std::cout<<"There exists a hamcycle: "
           << sol.get() << std::endl;
}
```

Variable `sol` is called a *future* (a promise to deliver a result in the future). Method `get` blocks until the future returns.

Invocation of the asynchronous thread, synchronization and communication between main and asynchronous threads all happen automatically.

Replacing `async` with `sync` will cause `hamcycle` to become a *deferred function*, which runs entirely during the call to `get`.

C++ Thread Summary

- *Future*: an object held by the *receiver* of a communication.
- To get the value from a future, call `future::get`.
- Function `future::get` will block until the value is available.
- Can also call `future::has_value` which checks for a waiting result without blocking.

- *Promise*: a channel through which a value is communicated to a future.
- The promise object (if any) is handled by the communication *sender*.
- Promises can be implicit or explicit.
- Values are sent through promises implicitly when the thread returns.
- Values are sent explicitly ordinarily using `promise::set_value`.
- Explicit normally used for manual thread management (e.g., multiple values must be communicated during the lifetime of a thread.)

Asynchronous programming in C#

Observations:

- I/O-bound operations result in CPU under-utilization.
- Examples: disk read, send data over a network, query a database.
- CPU has to idle waiting for an external resource.
- Multithreading solves responsiveness but does not solve utilization.
- **async** and **await** are constructs to increase CPU utilization.

Start with a time-consuming I/O bound method:

```
 static int GetTotal()
{
  int total = GetStartValue();
  Thread.Sleep(3000);  // I/O bound operation
  int increment = GetIncr();
  return total + increment;
}
```

Asynchronous programming in C#

How can we make this more CPU-efficient?

- Replace ordinary I/O bound operation calls with special **async** library routines. `Thread.Sleep` ⟶ `Task.Delay`

```
 static async Task<int> GetTotalAsync()
{
  int total = GetStartValue();
  await Task.Delay(3000);  // replaced
  int increment = GetIncr();
  return total + increment;
}
```

- Special **async** methods like `Task.Delay` return a future (called a “Task”).
- `GetTotalAsync` is also **async** and also returns a future.
- Executing **await**:
  - Causes a continuation to be captured and the method immediately exits.
  - Continuation is executed when the future eventually produces a value.

Asynchronous programming in C#

Now let's call GetTotalAsync:

```
private async static Task<int> DoComputation()
{
  int startVal = GetStartValue();
  int increment = await GetTotalAsync();
  return total + increment;
}
```

This is (roughly) shorthand for:

```
private static Task<int> DoComputation()
{
  int startVal = GetStartValue();
  return GetTotalAsync().ContinueWith((incrementTask) => {
     int increment = incrementTask.Result;
     return total + increment;
    });
}
```

Asynchronous programming in C#

- Summary: maximize thread CPU utilization: rather than block the CPU, execute both I/O-bound and CPU-bound activities simultaneously.
- `async` keyword must be used in method signature when `await` is called in the body.
- Presence of `async` creates a state machine for special handling.
- When `await` is reached, control returns to caller immediately.
  - Rest of the method body is wrapped in a continuation and deferred.
- Use of `async`/`await` does *not* require multi-threading.
- Contrast to multi-threading:
  - Increases CPU utilization, like threads. But...
  - No need to create and destroy threads.
  - No chance of deadlocking.
  - No prioritization headaches.
- Similar `async`/`await` concept available in other languages [**JavaScript**,**Python**,**Hack**, **Dart**].