Concurrency and Exceptions

Learning Objective

This session will introduce students to the concepts of program-level concurrency and exception handling. Students will gain an understanding of how various hardware architectures and programming languages support concurrency. Examples of the importance of exception handling will be presented, along with examples of how exceptions are handled by modern programming languages.

Overview

Early computers were only capable of running one job at a time -- in fact, the earliest computers needed to be hard-wired just to do that. "Batch" jobs were the normal way of life in the 1950's, with computers simply processing a sequence of programs that were usually submitted via punched cards.

By the early 1960's, programmers were beginning to become impatient with waiting for computers to sequentially process batch jobs. John Kemeny and Thomas Kurtz of Dartmouth University started work on a new programming language, Basic, that would allow students to simultaneously (concurrently) access a computer through terminals. The idea quickly caught on, and today virtually every computer system is capable of some level of time-sharing processing.

Another problem that quickly surfaced in the early days of computing was that of programing errors. A computer is by nature a state machine, with the state of the machine defined by the contents of all of its CPU registers and memory cells. Program instructions move the machine from one state to another. Programmers soon discovered, however, that their programs might contain "bugs" that would get the machine into a terminal state, i.e., so that it could not proceed to the next state (the machine is "locked up"). We have seen how this works with the Turing Machine. If the Turing Machine is in a given state, and there is no "rule" applicable to that state, the machine will simply stop.

As computers and their associated programs became more and more complex, the opportunity for programming errors increased dramatically. Additionally, computers became more ubiquitous, becoming embedded in more and more appliances and machines. In embedded applications, the thought of a programming error stopping an appliance's or a machine's operation became unacceptable. Consider, as an example, the engine computer in your automobile. Typically, the engine computer has many functions, such as controlling the injection of fuel into the engine, with the rate of injection based on factors such as the current speed of the car, the position of the gas pedal, the engine temperature, etc. An "illegal" combination of input parameters that would halt the engine computer is not acceptable. If such an event happens, the computer's programming language or operating system must have provisions for gracefully recovering. In modern parlance, this is called ***exception handling.***

In this session we'll be examining the concepts of concurrency and exception handling at the programming-language level.

What is Concurrency?

Concurrency is *events happening together in time or place.* In other words, concurrency is the simultaneous occurrence of more than one event or series of events.

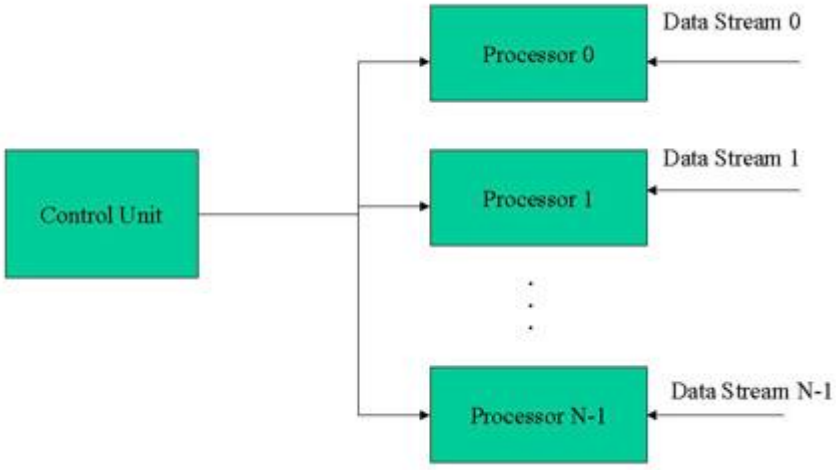
The first consideration in examining concurrency is to define what events are occurring simultaneously. At the highest level, a computing event is at the program level, i.e., a program is running, and concurrency at this level would mean that two or more programs are running simultaneously. Subprograms (e.g., procedures and functions) are the second-highest level, and concurrency at this level would mean that two or more subprograms are running simultaneously. Next is the programming-language statement level, where concurrency would mean that multiple statements are running simultaneously. The lowest level is the instruction (machine-language) level, where concurrency means that multiple machine instructions are being executed simultaneously.

At this point it is worthwhile to make the distinction between *actual* concurrency and *apparent* concurrency. Actual concurrency is a condition where multiple events are really happening simultaneously, and apparent concurrency is a condition where events are happening in such rapid sequence that humans aren't able to tell that they are not simultaneous occurrences. Apparent concurrency is also referred to as ***logical concurrency***, and actual concurrency is also called ***physical concurrency.*** The distinction is important, because it determines how concurrency is implemented in hardware.

Concurrency can be accomplished via single (for apparent concurrency) or multiple processors (for actual concurrency). Early mainframes achieved some degree of concurrency by using a CPU in conjunction with auxiliary (e.g., input/output, or I/O) processors, so that they could execute one program while doing I/O (such as printing) on another processor. Later generations incorporated clusters of full functional processors, and employed a scheduler that simply ran "jobs" on any available processor. Later generations of machines have used a wide variety of hybrid architectures.

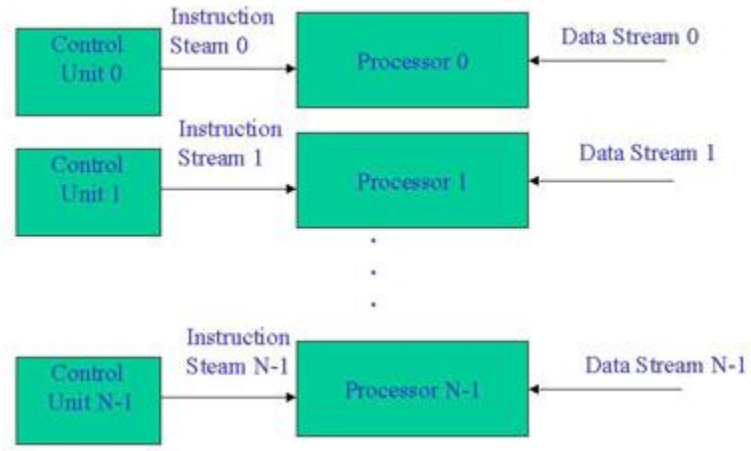
Computers have been categorized by the number of data and instruction streams that they can simultaneously process. Since there are two streams, there are four possible configurations:

- **Single-instruction, single data (SISD)**: This is nothing more than the classic von Neumann architecture.
- **Single-instruction, multiple data (SIMD)**: In this configuration, a single machine instruction controls simultaneous execution of multiple processing elements in a lockstep fashion, as shown below:



A typical application for a SIMD machine would be for vector processing, where, for example, one might want to simultaneously add all of the elements to two vectors.

- **Multiple instruction, single data (MISD)**: With this configuration, a sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence; this configuration has never been implemented.
- **Multiple instruction, multiple data (MIMD)**: Under this configuration, a set of processors simultaneously execute different instruction sequences on different data sets, as shown here:



The processors shown in the figure are not actually running independently, but are centrally controlled and communicate via a mechanism such as shared memory or through a data network.

There are at least two reasons for software engineers to study concurrency:

- Many of the applications that they write need to present the appearance of concurrency, such as calculating a spreadsheet while printing another document

- The increasing availability of physical concurrency (i.e., the availability of hardware such as MIMD that supports concurrency) makes it important for software engineers to understand how to use it efficiently.

It is important to be familiar with terms used to describe concurrency:

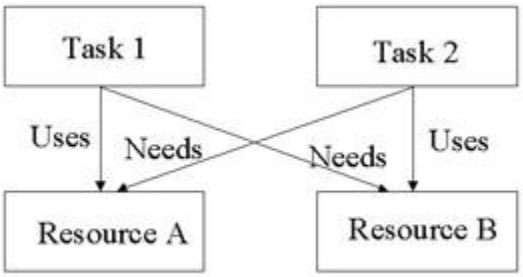
A ***task*** is a program unit that can execute concurrently with other program units; tasks are different from other subprograms in that: i) Tasks are started, subprograms are called, ii) There is no need to wait for task completion, and iii) After a task terminates, there is no need for control to return to the calling unit.
Disjoint tasks do not communicate with other tasks.

Synchronization controls the order of task execution, and comes in two "flavors:" ***cooperation synchronization*** in which multiple tasks are dependent on each other, and may need to wait for input from each other, and ***competition synchronization*** in which tasks need to share common resources, and may need to wait for resource availability. Techniques for implementing synchronization include ***semaphores, monitors***, and ***message passing***, which we'll describe later.

Now a thoughtful reader could rightfully wonder about discussing issues associated with the ordering of concurrent tasks. "Concurrent" implies "simultaneous." Why would one consider ordering things that by definition are happening at the same time? This answer is that in some cases, due to these simultaneous tasks needing common resources (or each other's results), they can't be 100% simultaneous, i.e., they aren't disjoint. It's these situations that are being addressed here.

A ***scheduler program*** manages the timesharing between tasks. Tasks can be in one of five states: *New* (created, but not yet running); *Runnable* (ready to run, but not currently running); *Running* (currently executing); *Blocked* (previously running, but waiting for something to happen); and *Dead* (previously running, but no longer active.

Liveness describes a task that is healthy and continues to execute. ***Deadlock*** describes a situation where competition for resources leads to a lack of liveness in two or more tasks, as shown here:



There are several design issues associated with concurrency: First is the issue of how ***cooperation synchronization*** is accomplished (how do different tasks communicate with each other). The second issue is how to accomplish ***competition synchronization***, so that multiple tasks can have access to limited resources such as printers. The next issue is how to schedule tasks, e.g., how to handle their different priorities. The final issue is whether tasks are created dynamically or statically.

Semaphores

Semaphores were devised in 1965 by Edsger Dijsktra to address the problem of competition synchronization. A semaphore is just a flag implemented as an integer (indicating the semaphore's state), and a task queue. Semaphores can have two states: ***wait*** ("passeren" - Dutch for "to pass"), and **release** ("vrygeren" - Dutch for "release").

Semaphores are potentially dangerous (unreliable) in that:

- a programmer can create a potential deadlock by omitting the release statement
- the operating system must ensure that the semaphore process itself is not interrupted during an update
- semaphores cannot assure fairness or prevent starvation (preventing another process from getting CPU time), or as Appleby and VandeKopple write, semaphores cannot eliminate greed!

An example from Appleby and VandeKopple (p. 216) shows how a semaphore is implemented for two processes that must share a critical resource:

```
var S : semaphore;
process P1:
    loop{forever}
        Wait(S);
            {Code requiring shared resource goes here}
         Signal(S);
            {Code not requiring shared resource goes here}
    end{loop}
end{P1};


process P2:
    loop{forever}
        Wait(S);
            {Code requiring shared resource goes here}
         Signal(S);
            {Code not requiring shared resource goes here}
    end{loop}
end{P2}
```

If process P1 requires the critical resource, it executes the procedure WAIT. If the semaphore S shows that the critical resource is available, the semaphore is reset to the wait state to block process two from using the resource, and process P1 makes use of the resource. If the call to Wait(S) indicates that the resource is in use, the procedure simply waits ("blocks") until the resource becomes available again. After P1 is finished with the resource, it calls Signal(S) to change the semaphore's state, thereby releasing the resource for use by P2.

Process P2 operates in exactly the same manner as P1; it should be evident from this example that semaphores cannot assure fairness or prevent process starvation.

Monitors

A **monitor** is an interface between concurrent user tasks. Monitors provide a set of user-callable procedures, a mechanism for scheduling calls to these procedures, and a mechanism for suspending a calling procedure until the needed resource is available. Note the similarity to data abstraction and encapsulation!

A monitor is really just an abstract data type that includes a shared data structure and associated (concurrent methods), and can be considered as a module, with its implementation details hidden from the user.

The history of monitors can be traced back to early versions of time-shared Basic; other programming languages that implement monitors include Concurrent Pascal, Modula, CSP/k, and Mesa.

Appleby and VandeKopple present the following form for a monitor:

```
monitor<MonitorName>
    var<permanent variable declarations>
    procedure<operation 1>(parameter list)
    ...
    procedure<operation N>(parameter list)
begin
    <initialization code for permanent variables>
end
```

Two operations, **delay** (corresponds to a semaphore wait) and **continue** (corresponds to a semaphore signal) are associated with each monitor; each monitor also has a queue.

Monitors act as a form of policeman for multiple cooperating processes. They are able to prevent process starvation (denying CPU time to a process), and they can prevent deadlock. Hence, monitors are considered more reliable than semaphores. In addition, they contribute to more readable code.

Message Passing

Monitors are an adequate solution for tightly coupled systems that used shared memory for inter-task communication; however, another approach must be used for loosely coupled (i.e., distributed) systems. A solution is ***message passing***.

Message passing can be either ***synchronous*** (where both the sender and receiver are ready to communicate) or ***asynchronous*** (where a task can react immediately to messages).

For a ***synchronous*** message, the actual message passing is called a ***rendezvous***. A rendezvous differs from a monitor in two ways: i) it's not a separate module, but is implemented within the tasks, and ii) a task can have several entry points for messages (each with its own queue), but a monitor can only have one queue.

Concurrency Implementation

Early implementations of concurrency were usually handled at the operating-system level, and that is often the case today. Operating systems such as Unix have system-level utilities to handle multiprocessing and communication between processes.

The more recent trend has been to incorporate concurrency capabilities into programming languages. Ada was a pioneer in programming-language level concurrency, with its capability to run concurrent tasks. C (and C++) can accomplish concurrency through system-level calls. Java has a strong capability for running ***threads***.

Threads are similar to processes running concurrently at the operating-system level, but involve less overhead. Threads make it seem like they are running in their own ***context***, i.e., that each thread has its own CPU, registers, memory, etc.

Recall that everything in Java is a member of a class. Java objects that are to run concurrently must be children of the Java class Thread. The concurrent object that is an instantiation of the class derived from Thread will always have a method (class procedure) that initiates execution of the object.

Horstmann and Cornell (volume 2, p.80) have an example of code for a game similar to "Pong," in which multiple balls ricochet around a box. In their code, each ball is controlled by a threaded object call "Ball." The implementation is sketched out below:

```
class Ball extends Thread
{...
    public void run()
    { draw();
      for (int i = i; i <= 1000; i++)
      {   move();
          try { sleep(5);}
          catch(Interrupted Execution e) {}
       }
     }
}
```

Some things to note about this code:

1. The class Ball is a child process of Thread.
2. The class Ball has a method called run.
3. The ball is initially drawn with a call to procedure draw().
4. Procedure move() then moves the ball object 1000 times
5. After each move, the thread is put in a "sleep" mode for five milliseconds, to prevent

other threads from CPU starvation.

6. The call to sleep is protected with an exception handler (try - catch).

The example shown above is typical for thread implementation in Java, i.e., the code structures are relatively straightforward (readable and writable). One drawback is that the threaded class must inherit (be a child of) from the Thread class. Occasional, the threaded subclass more naturally is a child of a Java class other than Thread. Recalling that Java does not allow multiple inheritance, other somewhat confusing Java constructs must be used to handle these situations.

Exception Handling

As mentioned in the beginning of this Session, the increasing complexity of hardware and software has made error ("exception") handling an important concern for software engineers. Run-time errors can be caused by a variety of hardware and/or software errors. A hardware-related error is exemplified by a parity error during a disk access; a software error is exemplified by an integer overflow.

Early (and many present) computing systems are good at detecting run-time errors, *and then terminating.*

Early programming languages made some rudimentary attempts at error checking, such as the FORTRAN end-of-file alert. As opposed to error checking, exception handling deals with any unusual or unexpected event (erroneous or not) that is detectable by hardware or software, and that requires special processing by an ***exception handler*** when the exception is ***raised.*** Obviously, the main motivation for using exception handlers is to increase reliability. Exception handlers, if well designed, may allow a program to recover from an error, or at least print a meaningful error message prior to terminating.

There are several advantages to having exception handlers built into a programming language. This strategy reduces program clutter, since a single exception handler can be used for multiple modules. In addition, having the capability available in a language keeps the issue of exception handling in front of developers, and helps to simplify the handling of non-erroneous but unusual conditions.

The design issues associated with exception handling are: i) the form of the exception handler (e.g., code segments versus program units), ii) the binding of exceptions to handlers (i.e., dynamic versus static), iii) the scope of exception handlers, iv) the ability to disable exception handling, and v) how processing is to continue after an exception (e.g., the Ariane rocket example).

Exception handlers were first incorporated in PL/I, and now appear in virtually all modern languages: C++, Java, Eiffel, Ada, etc.

Some Exception Examples

Ada was developed by the Department of Defense to provide concurrency and exception-handling capabilities. Ada was intended to be used extensively in embedded systems, and

therefore was required to have high reliability. Hence, exception handling was included in its design.

Ada exceptions are named, so that a specific handler can be associated with each exception. Ada has five predefined exceptions:

- Constraint_Error: e.g., bounds of an array are exceeded.
- Numeric_Error: e.g., an attempt to divide by zero.
- Program_Error: e.g., a call to a non-existent subprogram
- Storage_Error: e.g., memory space is exhausted.
- Tasking_Error: e.g., an unsuccessful rendezvous.

There are also exceptions associated with Ada library packages, such as the one used for input/output (I/O).

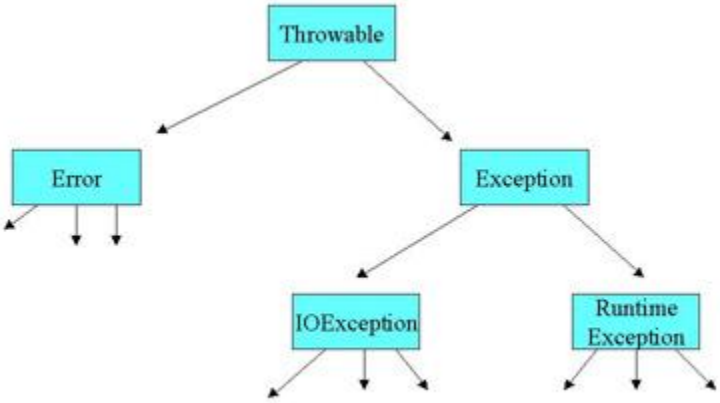
A simple example (from p.111 of Benjamin) of exception handling is shown below:

```
declare
    Data_OK : Boolean;
begin
   loop     --Prompt for integer input
      Data_OK := True;
      Put("Enter an integer: ");
      begin
     Get(Value);
      exception
          when Data_Error =>
                    Put_Line("Bad input on integer read");
                    Data_OK := False;
      when others =>
                     Put_Line("Unknown error reading data");
                     Data_OK := False;
      end;
     Skip_Line;
      exit when Data_OK;
  end loop;
end;
```

The readability of Ada makes the semantics of this example fairly apparent. The user is prompted to enter an integer, and a subprogram Get(Value)reads the integer. Data_Error, an I/O package exception, is raised if the user enters, for example, a character rather than an integer. The exception portion of the code then prints an error message and sets the boolean flag Data_OK to "false."

Ada also has a catch-all exception called **others**. This exception catches any other errors that may occur, and provides a handler for them. In the example above, the message specifying an unknown error is printed.

Java has become well known for its exception-handling capabilities. Recalling that everything in Java belongs to a class, in Java, an exception is always an instance of a class that is derived from the Java class Throwable, as shown below in the typical Java class



An example from Horstmann and Cornell that parallels the Ada example is:

```
public static String readString()
{   in ch;
    String r = " ";
     boolean done = false;
     while (!done)
     { try
      {   ch = System.in.read();
            if (ch < 0 || (char)ch == '\n') done = true;
            else r = r + (char) ch;
          }
       catch(IOException e)
      {   done = true;}
      }
      return r;
}
```

The purpose of this code is to read characters one-by-one and append them to a string 'r'. The **try** construct indicates that an exception handler will be associated with the block of code following the **try.** If an I/O exception occurs, the **catch**construct will handle the exception

Wrap Up

Concurrency and exception-handling capability are two of the most important features of modern programming languages. In this Session, we've seen how concurrency can be implemented as

either physical (actual), or logical (apparent), depending on the combination of hardware and software that is used.