MODULE 4: PROGRAMMING ERRORS NOT FOUND IN SEQUENTIAL PROGRAMMING
WEEK 4

Learning Outcomes:

After completing this course you are expected to demonstrate the following:

1. Get familiar with various programming errors not found in sequential programming that helps you understand and handle data and high – level races when it occurs to avoid lack of liveness or progress.

A. Engage

Trivia

Figure 4.1

Flynn (1972) introduced a classification of computer systems that is now widely used by the computing community. He divided machines into four categories based on how the machine relates its instructions to the data being processed. The categories are ***Single Instruction stream Single Data stream, Single Instruction stream Multiple Data stream, Multiple Instruction stream Single Data stream and Multiple Instruction stream Multiple Data stream***. The term ***stream*** relates to the sequence of data or instructions as seen by the machine during execution of a program. An instruction stream is a sequence of instructions as executed by the machine and a data stream is a sequence of data including input, partial or temporary results, called for by the instruction stream.

B. Explore

Video Title**: Parallel Communication Patterns Recap – Intro to Parallel Programming**
Youtube Link:**https://youtube.com/watch?v=LjWlZHqUG8A**
Module Video Filename: **Week 4- Parallel Communication Patterns Recap – Intro to Parallel Programming**

C. Explain

The computing landscape continues to evolve at a phenomenal rate, placing new demands on hardware and software design. A decade ago, when multi core processors became commonplace, desktops (and traditional servers) accounted for the bulk of computing and performance was still the primary design objective (though energy efficiency was rapidly gaining importance). A relatively small number of software vendors typically created "shrink-wrapped" applications on desktops. Today, mobile devices have significantly expanded computing at the low end, and cloud computing at the high end. Multi core processors are being used across this spectrum. Energy efficiency has now become the primary design objective. The number of software vendors has increased dramatically, creating hundreds of

thousands of applications to run on these diverse computing devices. Thus the importance of the ease of writing programs that will run in parallel (to achieve energy efficiency and/or performance) on a diverse set of devices continues to increase. Most of the computing community has responded to the parallelism challenge by trying to increase the prevalence of parallel programming—to make parallel programming synonymous with programming.

D. Elaborate

A ***Sequential program*** specifies sequential execution of a list of statements; its execution is called a *process*. A ***concurrent program*** specifies two or more sequential programs that may be executed concurrently as ***parallel processes***. In many languages, *process* is also the name of the construct used to describe process behavior; one notable exception is Ada, which uses the name *task* for this purpose.

Distinguishing concurrent, parallel, and distributed programs
In the literature, a concurrent program is commonly discussed in the same context as parallel or distributed programs. Unfortunately, few authors give precise meanings to these terms and the meanings that are offered tend to conflict. On balance, the following definitions seem appropriate:

- A ***concurrent program*** defines actions that may be performed simultaneously.
- A ***parallel program*** is a concurrent program that is designed for execution on parallel hardware.
- A ***distributed program*** is a parallel program designed for execution on a network of autonomous processors that do not share main memory.

Non-determinism
A sequential program imposes a total ordering on the actions it specifies. A concurrent program imposes a partial ordering, which means that there is uncertainty over the precise order of occurrence of some events; this property is referred to as ***non-determinism***. A consequence of non-determinism is that when a concurrent program is executed repeatedly it may take different execution paths even when operating on the same input data.

Process Interaction
All concurrent programs involve process interaction. This occurs for two main reasons:

1. Processes compete for exclusive access to shared resources, such as physical devices or data.
2. Processes communicate to exchange data.

In both cases it is necessary for the processes concerned to synchronize their execution, either to avoid conflict, when acquiring resources, or to make contact, when exchanging data. Processes can interact in one of two ways: through shared variables, or by message passing from one to another. Process interaction may be ***explicit*** within a program description or occur *implicitly* when the program is executed. In particular, there is implicit management of machine resources, such as processor power and memory that are needed to run a program.

Resource Management
A process wishing to use a ***shared resource*** (e.g. a printer) must first ***acquire*** the resource, that is, obtain permission to access it. When the resource is no longer required, it is ***released***; that is, the process relinquishes its right of access. If a process is unable to acquire a resource, its execution is usually suspended until that resource is available. Resources should be

administered so that no process is delayed unduly. A process may require access to one or more resources simultaneously, and those resources may be of the same or different types. The resources may be defined statically within the program or created and destroyed as the program executes. Also, some resources may be acquired for exclusive use by one process while others may be shared if used in a particular way. For example, several processes may inspect a data item simultaneously but only one process at a time may modify it.

Communication

Inter-process communication is one of the following:

1. ***Synchronous*****,** meaning that processes synchronize to exchange data.
2. ***Asynchronous*****,** meaning that a process providing data may leave it for a receiving process without being delayed if the receiving process is unable to take the data immediately;

The data is held temporarily in a ***communication buffer*** (a shared data structure). Where several data items are buffered, these are made available to the receiving process in the order in which they arrive at the buffer.

Violating mutual exclusion

Some operations in a concurrent program may fail to produce the desired effect if they are performed by two or more processes simultaneously. The code that implements such operations constitutes a *critical region* or ***critical section***. If one process is in a critical region, all other processes must be excluded until the first process has finished. When constructing any concurrent program, it is essential for software developers to recognize where such *mutual exclusion* is needed and to control it accordingly. Most discussions of the need for mutual exclusion use the example of two processes attempting to execute a statement of the form:

x := x + 1

Assuming that *x* has the value *12* initially, the implementation of the statement may result in each process taking a local copy of this value, adding one to it and both returning *13* to *x* Mutual exclusion for individual memory references is usually implemented in hardware. Thus, if two processes attempt to write the values 3 and 4, respectively, to the same memory location, one access will always exclude the other in time leaving a value of 3 or 4 and not any other bit pattern.

Deadlock

A process is said to be in a state of ***deadlock*** if it is waiting for an event that will not occur. Deadlock usually involves several processes and may lead to the termination of the program. A deadlock can occur when processes communicate (e.g., two processes attempt to send messages to each other simultaneously and synchronously) but is a problem more frequently associated with resource management. In this context there are four necessary conditions for a deadlock to exist:

1. Processes must claim exclusive access to resources.
2. Processes must hold some resources while waiting for others (i.e., acquire resources in a piecemeal fashion).
3. Resources may not be removed from waiting processes.
4. A circular chain of processes exists in which each process holds one or more resources required by the next process in the chain.

Indefinite postponement (or starvation or lockout)
A process is said to be ***indefinitely postponed*** if it is delayed awaiting an event that may not occur. This situation can arise when resource requests are administered using an algorithm that makes no allowance for the waiting time of the processes involved. Systematic techniques for avoiding the problem place competing processes in a priority order such that the longer a process waits the higher its priority becomes.

Unfairness
It is generally believed that where competition exists among processes of equal status in a concurrent program, some attempt should be made to ensure that the processes concerned make even progress; that is, to ensure that there is no obvious ***unfairness*** when meeting the needs of those processes. **Fairness** in a concurrent system can be considered at both the design and system implementation levels. For the designer, it is simply a guideline to observe when developing a program; any neglect of fairness may lead to indefinite postponement, leaving the program incorrect.

Busy waiting
Regardless of the environment in which a concurrent program is executed, it is rarely acceptable for any of its processes to execute a loop awaiting a change of program state. This is known as ***busy waiting***. The state variables involved constitute a ***spin lock***. It is not in itself an error but it wastes processor power, which in turn may lead to the violation of a performance requirement. Ideally, the execution of the process concerned should be suspended and continued only when the condition for it to make progress is satisfied.

Transient errors
In the presence of non-determinism, faults in a concurrent program may appear as ***transient errors***; that is, the error may or may not occur depending on the execution path taken in a particular activation of the program. The cause of a transient error tends to be difficult to identify because the events that precede it are often not known precisely and the source of the error cannot, in general, be found by experimentation. Thus, one of the skills in designing any concurrent program is an ability to express it in a form that guarantees correct program behavior despite any uncertainty over the order in which some individual operations are performed. That is, there should be no part of the program whose correct behavior is ***time dependent*****.**

Safety
Safety properties assert what a program is allowed to do, or equivalently, what it may not do.

Examples include:
1. **Mutual exclusion:** no more than one process is ever present in a critical region.
2. **No deadlock:** no process is ever delayed awaiting an event that cannot occur.
3. **Partial correctness:** if a program terminates, the output is what is required.

A safety property is expressed as an ***invariant*** of a computation; this is a condition that is true at all points in the execution of a program. Safety properties are proved by ***induction*****.** That is, the invariant is shown to hold true for the initial state of the computation and for every transition between states of the computation.

Liveness
It is also known as the progress; properties assert what a program must do; they state what will happen (eventually) in a computation.
Examples include:

1. **Fairness (weak):** a process that can execute will be executed.
2. **Reliable communication:** a message sent by one process to another will be received.
3. **Total correctness:** a program terminates and the output is what is required.

Liveness properties are expressed as a set of liveness ***axioms*****,** and the properties are proved by verifying these axioms. Safety properties can be proved separately from liveness properties, but proofs of liveness generally build on safety proofs.

Parallel programming involves a set of components that must each be considered when developing a parallel system. This set, which we regard as the parallel programming domain, includes, among others, the following aspects of the code: sequential code, inter process communication, synchronization, and processor utilization. Understanding the issues involved with the components of this domain makes understanding the source and manifestation of errors easier. This understanding is useful for determining the approach needed to efficiently debug parallel programs. In addition, it helps determine where to focus the debugging effort, depending on which component of the domain the programmer looks for errors in.

The four components are:

1. **Partitioning.** The computation to be performed and the data which it operates on are decomposed into small tasks.

2. **Communication.** The communication required to coordinate task execution is determined, and the appropriate communication structures and algorithms are defined.

3. **Agglomeration.** The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs.

4. **Mapping.** Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs.

The two last components, agglomeration and mapping, are mostly concerned with performance issues which, while important, are outside the scope of this paper. For the first two components, partitioning and communication, we propose the following additional breakdown:

1. **Algorithmic changes.** Many parallel programs begin life as a sequential program. If parallel algorithms are based on, or derived from, existing algorithms and/or programs, then a transformation from the sequential to the parallel domain must occur. The transformation of a sequential program into a parallel program typically consists of inserting message passing calls into the code and changing the existing data layout; for example, shrinking the size of arrays as data is distributed over a number of processes. However, if the sequential algorithm is not suitable for parallel implementation, a new algorithm must be developed. For example, the pipe-and-roll matrix multiplication algorithm does not have a sequential counterpart.

2. **Data decomposition.** When a program is re-implemented, the data is distributed according to the algorithm being implemented. Whether it is the transformation of a sequential program or an implementation of a parallel algorithm from scratch, data decomposition is a nontrivial task that cannot be ignored when writing parallel programs, as not only correctness, but efficiency also greatly depends on it.

3. **Data exchange.** As parallel programs consist of a number of concurrently executing processes, the need to explicitly exchange data inevitably arises. This problem does not exist in the sequential world of programming where all the data is available in the process running the sequential program. However, in parallel programs, the need for data exchange is present. On a shared memory machine, the data can be read directly from memory by any process. There is still the problem of synchronized access to shared data to consider, but no sending and receiving of data is needed. When working with a cluster of processors, each having a separate memory, message passing becomes necessary.

4. **Protocol specification.** The protocol for a parallel system is defined as the content, order, and overall structure of the message passing between communicating processes. Along with the data exchange, the communication protocol of the program is a new concept that has been introduced by parallelizing the algorithm.

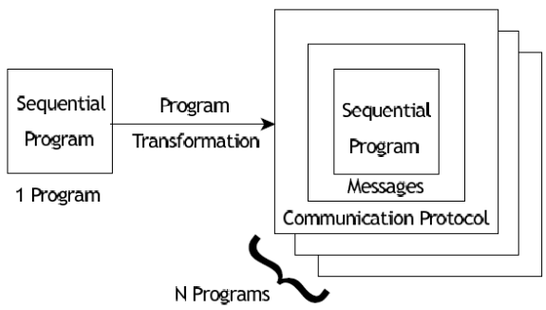

Figure 4.2 The Sequential Versus The Parallel Programming Domain

Figure 4.1 shows a stylized representation of a sequential and a parallel program. As shown, a sequential program is depicted as a single box, representing the sequential code of the program. The parallel program is represented as a number of boxes, each consisting of three nested boxes. The innermost of these boxes represents the sequential program that each process in the parallel program executes. The sequential code of the parallel program can either be an adaption of the existing sequential program, or a completely rewritten piece of code. The middle box represents the messages being sent and received in the system (the data exchange), and the outer box represents the protocol that the communicating processes must adhere to.

The Debugging Process
A well-known approach to debugging was proposed by Araki, Furukawa and Cheng. They describe debugging as an iterative process of developing hypotheses and verifying or refuting them. They proposed the following four step process:

1. Initial hypothesis set. The programmer creates a hypothesis about the errors in the program, including the locations in the program where errors may occur, as well as a hypothesis about the cause, behavior, and modifications needed to correct them.

2. Hypothesis set modification. As the debugging task progresses, the hypothesis changes through the generation of new hypotheses, refinement, and the authentication of existing ones.

3. Hypothesis selection. Hypotheses are selected according to certain strategies, such as narrowing the search space and the significance of the error.

4. Hypothesis verification. The hypothesis is verified or discarded using one or more of the four different techniques: static analysis; dynamic analysis (executing the program); semi-dynamic analysis (hand simulation and symbolic execution) and program modification.

The Main Difference between Sequential and Parallel Programs

In all our programs, there are some execution phases in which our processes are taking sometime in an I/O operation or in a synchronization operation waiting for the system or another process to finish a task. During these phases, our programs are not progressing on their work; it is rather the operating system or some other process running on the computer the ones which are taking all the valuable CPU time.

Processes and threads

In traditional operating systems, the concept of process is described as the instance of an executing program. In these, a process can only contain a running program where there is a single execution flow, that is, the program will be executing a single instruction at any time. This means in other words that in this traditional system a process is represented as a single execution unit and that it only can be executed in a single CPU at any particular time.

These systems included a new concept known as ***thread*** that allowed a program to have more than an internal function running at the same time within the same memory space of a single process. As a result, a process could have more than a execution flow, and therefore more than a single line of the same program could be executing in parallel in two different CPUs simultaneously. Multithreading operating systems provide dedicated system calls for creating and managing execution threads as well as for creating and managing processes. In these systems, the memory address space of every process is still private for the rest of the processes, and as execution threads are attached to a unique process, they can use its whole memory space. In a multithread operating system two threads attached to a same process can use global shared variables within the process's memory space for communication. However, two threads attached to different process could not share any memory space and message passing is the only communication mechanism for them. Nowadays, all commercial and general purpose operating systems are multithreading ones.

E. Evaluate

ASSESSMENT:
Instruction: Answer the questions below using the Answer Sheet (AS) provided in this module.

CONTENT FOR ASSESSMENT:
(2-points each)

1. It specifies sequential execution of a list of statements; its execution is called a process.
2. It defines actions that may be performed simultaneously.
3. A concurrent program that is designed for execution on parallel hardware.
4. A parallel program designed for execution on a network of autonomous processors that do not share main memory.
5. The computation to be performed and the data which it operates on are decomposed into small tasks.
6. The communication required to coordinate task execution is determined, and the appropriate communication structures and algorithms are defined.
7. The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs.
8. The process in which each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs.
9. It occurs when process that can execute will be executed.
10. It is when a message sent by one process to another will be received.

References:

1. *Introduction to Parallel Computing UsingAdvanced Architectures and Algorithms, D. R. EMERSON, Head of Computational Engineering, Daresbury Laboratory, Keckwick Lane. Daresbury, Warrington WA4 4AD, Cheshire, United Kingdom*

2. *The Road to Parallelism Leads Through Sequential Programming, Gagan Gupta Srinath Sridharan Gurindar S. Sohi, Department of Computer Sciences, University of Wisconsin-Madison*

3. *Concepts ofConcurrentProgramming, David W. Bustard, University of Ulster*

4. *J.B. Pedersen / Classification of Programming Errors in Parallel Message Passing Sytems, University of Nevada, 4505 Maryland Parkway,Las Vegas, Nevada, 89154, USA*

Facilitated By:		
Name	:	
MS Teams Account (email)	:	
Smart Phone Number	:	