

CONCURRENT SYSTEMS

- Concurrency is a ubiquitous phenomenon. Concurrency is unavoidable in
 - nature
 - daily life
 - computer systems
 - current and future computing practices.
- The intuitive definition of concurrency is the performance of multiple tasks at the same time.
- However, these tasks are not always independent and they need to interact with each other.

Course Structure:

Part 1

- Significance of concurrency
- Realising concurrency in computer hardware
- Describing concurrent systems

Part 2 (A/Prof Amitava Datta)

- Programming for concurrency
- Communication and concurrency in Java

An example

Consider the cars on a highway at any given time:

- We can think of each car as a *task* which is getting executed independently.
- Each car has a starting point and a destination.
- The task of each car is to reach its destination.
- However, the tasks in this case are not completely independent.
- The tasks must *interact* with each other, otherwise, their execution may not complete *correctly*.
- How do they interact? The need to *communicate*.
- What does *correctly* mean? No two cars try to occupy the *same section* of road at the *same time*, otherwise they crash!
- The cars *and* the roadway form a *concurrent system*.

An example : a resource shared by Concurrent “agents”

One important point stands out from our example:

- Concurrent processes often must *communicate* with each other. We then have a *system of communicating processes*.
- In our example, the cars communicate by signaling to each other and by others observing this signal. So there are two parts to communication, namely *generating* a signal (an output event) and *receiving* a signal (an input event).
- We can say that the communication is through message passing as each signal from one car is a message for the other cars.
- Concurrent processes often execute the same code on different input data.
- In our example, the code corresponds to rules-of-the-road and the data may be the source and destination of a car.

An example

- Only one car can be on a specific stretch of the road at a time. So cars *share* a common resource, namely the roadway.
- No two cars can be on the same part of the road at the same time.
- Consider the case when a car is trying to change its lane. It must signal to the other cars that it wants to change lane and occupy a certain part of the next lane.
- In concurrent programming this is often called the *mutual exclusion* problem. That is where a particular part of the roadway can only be occupied by one car at a time, the cars must *communicate* to successfully *share* the road section.

What do we study in this unit?

Part 1

- We will study the basic concepts found in concurrent systems.
- We will introduce a modelling language based on *interacting automata*, with which to abstractly describe concurrent systems.
- We will illustrate the important concepts of concurrency via examples described using this interacting automata language.
- We will see the different ways of implementing mutual exclusion.

Part 2

- We will program in Java. The main aim will be to use *multiple threads* to understand concurrent execution of these threads.
- We will study how Java programs can communicate with other Java programs across machines. We will do this through Remote Method Invocation (RMI).
- We will see how a particular *middleware* called Common Object Request Broker Architecture (CORBA) facilitates communication among distributed objects. These distributed objects may be written in different languages.

Concurrent Programming

- Today's computers are fast approaching the physical limits of speed. Computer chips are made of Silicon. The length of the covalent bonds between two adjacent Silicon atoms is 100A, $1\text{A} = 10^{-8}\text{cm}$.
- This length is probably the smallest possible size of a transistor on an integrated circuit.
- Hence the miniaturization will eventually approach the physical limits.
- On the other hand, the demand of computing speed is increasing at a high rate. For example, there are problems in fluid dynamics that require months of computing by the fastest supercomputers.
- Hence, we need to look at other approaches.

An example : lessening time to compute

- Suppose that you are asked to add 32 numbers. If you are doing it alone, you have to read each of the 32 numbers one after another and add them to the sum you have computed so far giving 32 additions, each taking a unit of *time*. But in reality $32-1$.
- If you have a friend to help you, each of you can take responsibility for adding 16 numbers and at the end add the two sums you obtain. If both of you are doing the addition at the same time, you can reduce the time requirement by half (to 16).
- This process of dividing a given problem into smaller parts and solving the smaller parts simultaneously can be continued.

- In our example this will result in a *binary tree* of depth 5 and one can compute the sum in 5 time steps. There are $2^5 - 1$ additions to process.
- Maximum concurrency (16 people in this example) then pass results to 2^3 people (8 out of the 16), and then to 2^2 (4) and so on. Finally the result is obtained by the last “adder”.
- *Concurrent computation* with different degrees of concurrency, results in shorter computation time.

An example

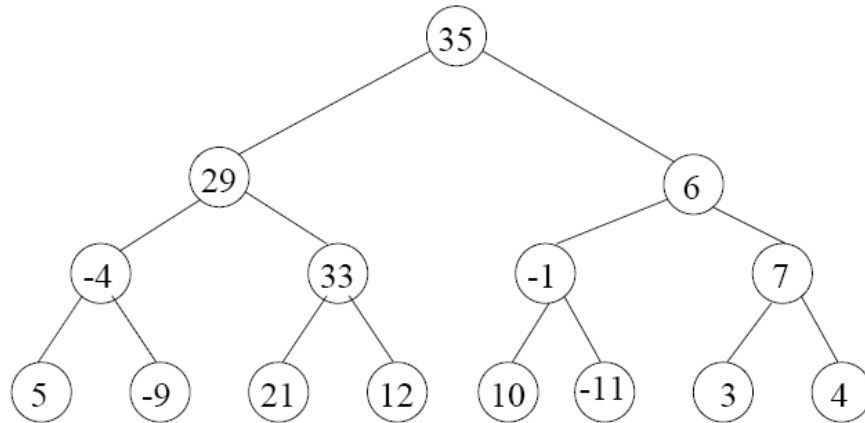


Figure 1: Adding 8 numbers concurrently : 3 time steps

Concurrent Programming

- There are, however, many problems if you try to use this technique.
- Instead of you and your friends, let us assume that we have several *processors* to do the computing for us.
- We have to ensure that each processor performs the computing on *separate data*. Otherwise, the final result will be erroneous as we cannot simultaneously modify the same variables. This problem is one of *mutual exclusion*.
- We have to ensure that each processor roughly does equal amount of work, this is called *load balancing*. This is for performance reasons.
- We have to ensure that there is some type of *operating system* that supports this kind of computing.

Concurrent Programming

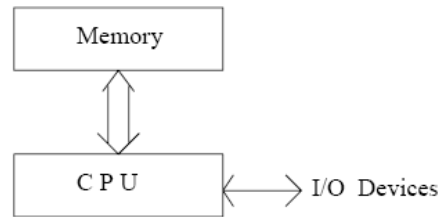


Figure 2: A simple machine model.

- Concurrent programming is the simultaneous execution of multiple processes either in a single processor or in a multi-processor machine.
- First, let us clearly specify what we mean by a machine.
- At each *clock cycle*, the CPU brings an instruction or data from the memory. The communication between the CPU and the memory is through the *memory bus*.
- A program (in a high level language) is converted into machine language by the *compiler* and loaded in the memory by the *loader*.

Structure of the CPU

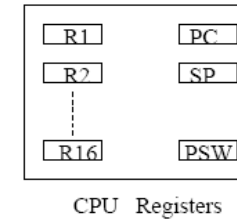


Figure 3: An overview of CPU registers.

Let us first look at a simplified structure of the CPU:

- The registers *R1* to *R16* are called *general purpose registers*. They are used for storing data items, intermediate results of computations and other computational information related to the program currently being executed.
- A special *PC* register is called the *program counter*. Each instruction in a program has an *address*. (i.e. where in memory it is located). The *PC stores* the address of the next instruction to be executed.

Structure of Memory

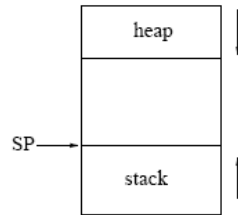


Figure 4: The memory segment of a program or process.

- The register *SP* (Stack Pointer) points to the current top of the *execution stack* of the machine code program.
- The memory allocated to a program is divided into two parts, a *stack* and a *heap*.
- The stack is used to store the activation records of the functions/ procedures currently in execution.
- The heap is used for allocating dynamic memory, e.g. through the **new** statement of PASCAL.
- The sizes of both stack and heap grow and shrink during the lifetime of a program (i.e. when the CPU is executing the program).

Structure of the CPU and Memory

- The register PSW (Program Status Word) stores different status information related to the current instruction. For example, if the result of the current instruction was negative, whether a divide by zero exception occurred etc.
- There is other information related to a program in execution. For example, the files currently opened by the program, the different access permissions, the priority level of the program etc.
- A *process* is a **program in execution**. In addition to the code (the machine code) of the program, the CPU needs a lot more information to run a program correctly. This information includes the status of all the registers and all information related to files etc.

Continued...

- From now on, we will talk about **processes** and not *programs*.
- Think of a program as the syntax (i.e. the written code) and a process as the program running on the processor.
- CPU accesses data from memory. This data includes instructions to be executed (or run) – i.e. the program code – and data to be manipulated in the CPU under instructions from the program, usually identified by variable names.

A Time-sharing System

- In a *time-sharing system* there are many user processes running simultaneously, more than the number of available processes. The CPU allocates time to each processors (let us assume) in a round-robin fashion. (i.e. one after the other and back to the beginning again.) A typical time quantum may be about 100 milli seconds.
- A process may or may not complete execution within a single time quantum. If a process is still unfinished, the CPU saves all the information associated with that process before switching over to another process. This is stored in a data structure called a **process table** where each process has an entry.
- When an unfinished process is executed again later, the CPU loads all the information (e.g. register contents) from the process table entry to the CPU registers. The execution of the process can restart after that.
- “time slicing” via interup from the scheduler. May be called *multitasking*.

Process States

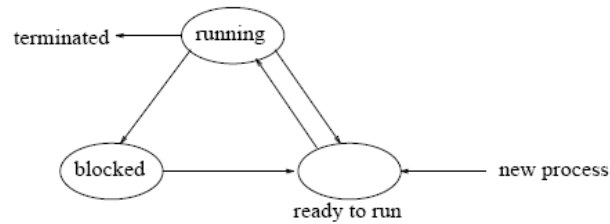


Figure 5: Different states of a process.

A process may exist in several states during its lifetime in a time-sharing system. We will take a simplified view of different process *states* and the *events* which occur as a process changes state:

Current State $\xrightarrow{\text{event}}$ **New State**

ready to run \rightarrow **running**: The CPU scheduler chooses the process for running (gives a CPU time quantum).

running \rightarrow **ready to run**: The time quantum is over. The scheduler stops this process and chooses another process.

running \rightarrow **blocked**: The process is *waiting* for some I/O to be completed. There is no point in running this process now, so the process is blocked.

blocked \rightarrow **ready to run**: The I/O is *complete* and the process is *ready* for CPU time.

How process switching is handled by the Operating System

- When the time quantum of a process is over or the process is blocked for I/O, the OS saves all *process states* in the corresponding *process table entry*. This includes the values or *contents* of the PC, SP, PSW and all other registers.
- The scheduler chooses another process for running by the CPU. The CPU loads all its registers from the *process table entry* of this new process. Once the program counter (PC) is loaded, the CPU starts execution of the new process.
- There is a limit on the total number of processes which can exist at a time in the system. If there are too many processes, the throughput of the system may become low. Why? On the other hand, if there are only a few processes, the CPU may remain idle for longer periods of time. This becomes an *optimisation* issue for the operating system

Examples of Processes

User processes: The programs you write and execute. As soon as you start executing your program, the operating system creates a process table entry for your program. All the information related to your program is kept in this process table entry until the execution is complete. You can have multiple processes running if you execute different programs in different windows or create other processes from within your process.

Operating system processes: As soon as you switch on or boot your machine, the operating system starts many different processes.

For example:

- The same machine may have many log-in windows at different terminals. These are processes created by the operating system so that different users can login to the machine.
- When you are working at a window, the shell process (in linux) is always waiting to interpret the command you are typing.
- There are **daemon** processes like **mail** which is always checking to see whether new mail has arrived for you.
- Managed by operating system in contrast to **multithreading**, which is performed from *within* a program.

Processes and Threads

- The address spaces (allocated memory) of different processes are *disjoint*. i.e. they are physically separate in memory. No process can access the address space of another process.
- Each process has a single **thread** of control.

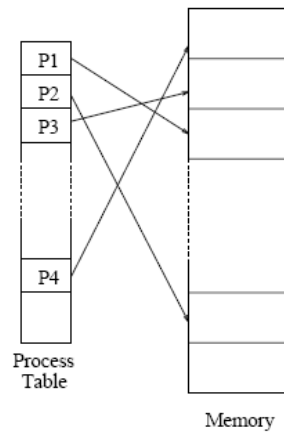


Figure 6: Address spaces of different processes.

Processes and Threads

- A single process may have multiple threads of control, but all of them share the same address space.
- Each thread has its own **program counter (PC)** and other register states which are saved in a **thread table**. Threads like processes can be in *running*, *ready* and *blocked* states.

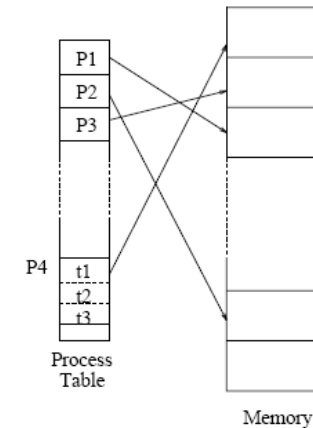


Figure 7: Address spaces of different processes having multiple threads.

Why threads are useful?

- Consider a web page with several different images in it. Suppose a browser is trying to load the web page from the other end of the world into its local machine.
- If we use a single process to do this, it has to load the images one by one into the local machine. Multiple processes won't help, as they will load the images in different address spaces and as a result, they will need separate connections to the remote machine *one after another* and this is time consuming.
- A better solution is to have multiple threads. Different threads can load the images in the same address space and as a result, the same connection to the remote machine can be used for loading *all* the images.
- Interactive program: one thread to handle events from user interface run concurrently with thread for computation.