CS5352, Distributed Computing, Summer II, 2020

Lecture 1: Introduction

(07-06-2020)

Lecture Outline

```
1. History of computer systems
2. Characteristics of distributed computing
3. Advantages of distributed systems
4. Distributed systems vs. distributed computing
5. General issues in distributed computing
6. Goals of distributed computing
7. Distributed computing vs. cloud computing
```

1. History of computer systems

   (1) Dedicated systems

   (2) Batching systems

   (3) Time-sharing systems

   (4) Distributed systems

A dedicated system is a computer system capable of performing 1 specific task an embedded system is a computer system within a larger system that performs a specific task.

The users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group.

Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

2. Characteristics of distributed computing

   (1) Computer networks

   (2) Multiprocessor systems with shared memory

   (3) The essential features of distributed systems

      a. Multiple processing elements
      b. Intercommunication facilities
      c. Transparency to the users
      d. Fault tolerance
         i. Processing elements fail independently
         ii. Single failure will not bring down the whole systems/computing

3. Advantages of distributed systems

   (1) Convenience

   (2) Low ratio of cost/performance

   (3) Load sharing

(4) Resource sharing

(5) Availability

(6) Reliability

(7) Scalability

A distributed system is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another.

Three aspects: concurrency of components, lack of a global clock, and independent failure of components

In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers, which communicate with each other via message passing.

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly.

4. Distributed systems vs. distributed computing

(1) Distributed systems are special cases of general distributed computing. In fact distributed systems can be regarded as one of the most important applications of distributed computing.

(2) A distributed system is characterized by the existence of a distributed operating system that provide transparent distributed services to members of the distributed system.

5. General issues in distributed computing

(1) The enormous complexity

a. **Interference**. The behavior of the combination of individually well-understood components become difficult to understand.

b. Propagation of effects is affected by distributedness.

(2) Distributed control and coordination

(3) Intercommunication. Synchronization, time and clocks. Implementation of remote procedure calls.

(4) Distributed agreement and fault tolerance

(5) Managements of replicated data

(6) Naming and security. Authentication.

6. Goals of distributed computing: achieve the highest level of the following types of of transparencies

(1) **Access transparency** enables local and remote files and other objects to be accessed using identical operations.

(2) **Location transparency** enables objects to be accessed without knowing their location.

(3) **Concurrency transparency** enables several users or application programs to execute concurrently on shared data without interference with each other.

(4) **Replication transparency** enables multiple instances of files and other data to be used to increase reliability and performance without knowledge of the replicas by users or application programs.

(5) **Failure transparency** enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.

(6) **Migration transparency** allows the movements of objects within a system without affecting the operation of users or application programs.

(7) **Performance transparency** allows the system to be reconfigured to improve performance as loads vary.

(8) **Scaling transparency** allows the system and applications to expand in scale without change to the system structure or the application programs.

7. Distributed computing vs. cloud computing

(1) **Cloud computing**: a new technique based on decades of research and advances of many fields of computing technology: distributed computing, utility computing, grid computing, networking techniques, web and software services, cyberinfrastructure, virtualization

(2) Essentials of cloud computing

a. Service oriented architecture: on-demand services

b. Component based system engineering

c. Virtualiztion: resources are virtualized - reduced information technology overhead

Cloud computing is a style of computing where massively scalable and flexible IT-related capabilities are delivered as a service to the users using Internet technologies, services may include: infrastructure, platform, applications, and storage space. The users pay for these services, resources they actually use. They do not need to build infrastructure of their own.

Distributed Computing can be defined as the use of a distributed system to solve a single large problem by breaking it down into several tasks where each task is computed in the individual computers of the distributed system.

Lecture 2
Elements of Operating Systems
(07-07-2020)

Lecture Outline

```
1. Sequential Processes
2. Concurrent Processes
3. Specification of Concurrency
4. Relationships among the Processes
5. Deadlocks
6. The Critical Section Problem
7. Semaphores
```

1. Sequential Processes

(1) *A sequential process is a sequential program in execution*

a. A process is an active entity, while a program is a passive entity

b. The execution of a process is sequential, i.e. following the control flow of the program

(2) State of a process

- **Running**. The instructions of the process are being executed
- **Blocked**. The process is waiting for some event to occur (such as I/O completion, communication)
- **Ready**. The process is waiting to be assigned to a processor
- **Deadlocked**. The process is waiting for some event that will never occur

2. Concurrent Processes

(0) *Concurrent* vs *parallel*: the latter implies *at the same time physically*, while the former implies *at the same time logically*.

(1) *Sequential programs might be executed concurrently*

(2) **Example 1.** Consider the following program:

$a := x + y;$
$b := z + 1;$
$c := a - b;$
$w := c + 1;$

The statement $a := x + y$ and $b := z + 1$ can be executed concurrently
This example illustrates that, within a single program:
• some statements can be executed concurrently
• there exists *precedence constraints* among the various statements

(3) **Precedence graph.** A precedence graph for a sequential program is a directed acyclic graph whose nodes correspond to individual statements in the program. An edge from node S_i to node S_j means that statement S_j can be executed only after statement S_i has completed execution.

Example 2. Consider the precedence graph below,

- S_2 and S_3 can be executed (concurrently) after S_1 completes
- S_4 can be executed after S_2 completes
- S_5 and S_6 can be executed (concurrently) after S_4 completes
- S_7 can be executed after S_5, S_6, and S_3 complete

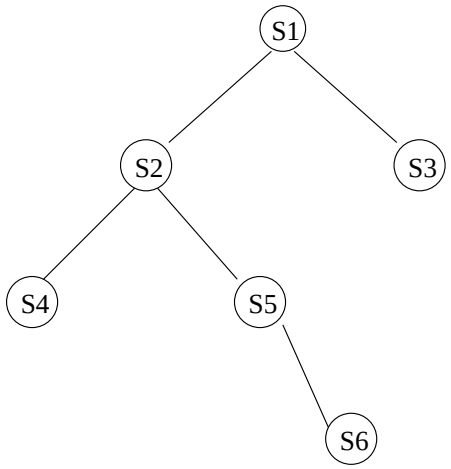

Figure 1: Process Precedence Graph

3. Specification of Concurrency

(1) The fork and join constructs

- The **fork** L instruction produces two concurrent executions in a program. One execution starts at the statement labeled L, the other is the continuation of the execution at the statement following the **fork** statement.

$$\begin{array}{ll} & S_1; \\ & \textbf{fork } L; \\ & \cdots\cdots \\ L: & S_3; \end{array}$$

- The **join** instruction combines several concurrent executions into one. All but the last computation that execute the join instruction will terminate, and the last computation will continue.

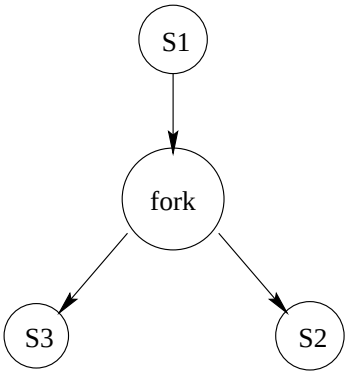

Figure 2: Effect of the Fork Statement

$count := count - 1$;
if $count \neq 0$ **then** quit;

$count := 2$;
fork L_1;
. . . . . .
S_1;
goto L_2;
$L_1 : S_2$;
L_2 : **join** $count$;

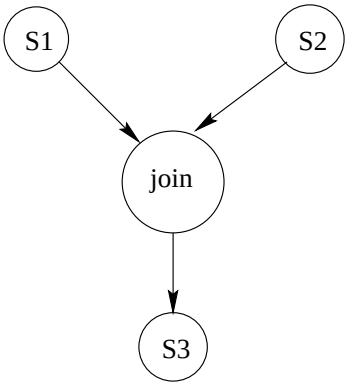

Figure 3: Effect of the join Statement

(2) The concurrent statement

parbegin $S_1; S_2; \cdots; S_n$ **parend**;

Example 3. The program in Example 1 can be expressed by

parbegin

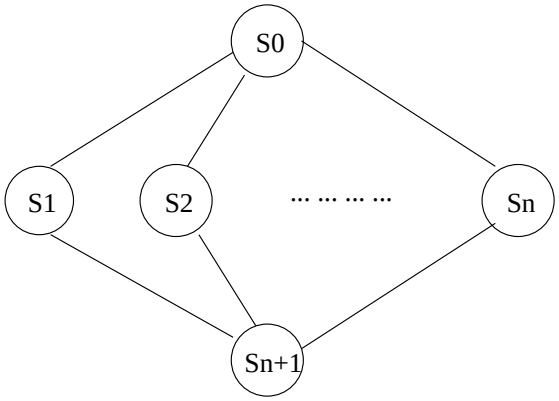

Figure 4: Illustration of the parbegin-parend Statement

$a := x + y;$
$b := z + 1;$
parend;
$c := a - b;$
$w := c + 1;$

(3) Comparison: the **fork** and **join** constructs are more powerful than the concurrent statements. The following precedence graph can be represented by the **fork** and **join** constructs, but can not be represented by concurrent statements.

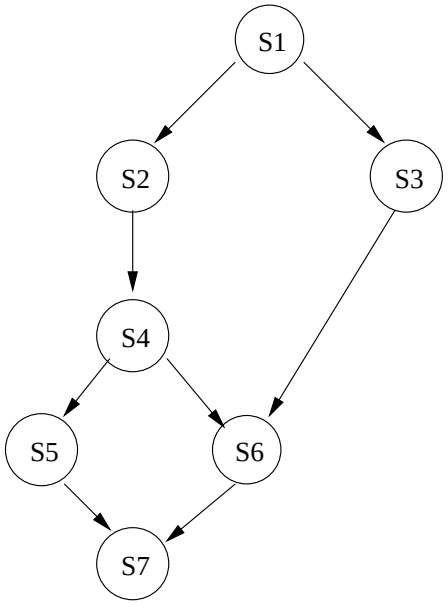

Figure 5: Comparison of fork-join and parbegin-parend Statement

4. Relationships among the Processes

(1) **Process graph.** A process graph is a directed rooted tree, whose nodes correspond to processes. An edge from node P_i to P_j means that process P_i created process P_j. P_i is called the *parent* of P_j, and P_j is called the *child* of P_i.

(2) Operations on a process

a. Process creation. When a process is created, several possible implementations are:

i. **Execution.** Concurrent versus sequential
(a) The parent continue to execute concurrently with its child
(b) The parent waits until all of its children have terminated

ii. **Sharing.** All versus partial
(a) The parent and the children share all variables in common
(b) The children share only a subset of their parent's variables

Option i.(a) and ii.(a) are adopted by the **fork** and **join** constructs. Option i.(b) and ii.(a) are adopted by the concurrent statements
Option ii.(b) is adopted by the UNIX system.

b. Process termination.
A process may terminate when it finishes its execution.
It can also be killed by its parent.

$id :=$ **fork** L;
kill id;

5. Deadlocks

(1) **Definition.** A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the set.

(2) **Example 4.** Two processes P_1 and P_2. Process P_1 is holding the resource printer, waiting for the resource disk drive. Process P_2 is holding the resource disk drive, waiting for the printer. P_1 and P_2 are in a deadlock state.

(3) Necessary conditions of deadlocks
A deadlock can arise if and only if the following four conditions hold simultaneously.

– **Mutual Exclusion.** At least one resource is being held in a non-sharable mode.
– **Hold and Wait.** There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are being held by other processes.
– **No Preemption.** Resource cannot be preempted; i.e. a resource can only be released voluntarily by the process holding it.

– **Circular Wait.** There must exist a set $\{P_0, P_1, \cdots, P_n\}$ of waiting processes, such that P_0 is waiting for a resource held by P_1, P_1 waiting for a resource held by P_2, and P_n waiting for a resource held by P_0.

6. The Critical Section Problem

(1) Problem description. A computation consists of n cooperating processes $\{P_1, \cdots, P_n\}$. Each process has a common section of code, called *critical section*, in which a process might read or write common variables, tables, or files. The execution of critical section must be *mutually exclusive*, in the sense that at any time at most one process is allowed to execute in the critical section. The critical section problem is to design a protocol which the processes may use to cooperate.
The code for each process can be divided into four sections: *entry section*, *critical section*, *exit section*, and *remainder section*. A solution to the *cs* problem must satisfy the following three requirements:

a. **Mutual Exclusion.** If process P_i is executing its critical section, then no other process can be executing in its *cs*.

b. **Progress.** If no process is executing in its critical section and there exists some process that wishes to enter its *cs*, then only those processes that are not executing in their remainder section can participate in the decision as to who will enter the critical section, and and decision can not be postponed indefinitely.

c. **Bounded Waiting.** There must exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

(2) An Example
Example 5. The readers/writers problem.
A data object is to be shared among several concurrent processes. Some of the processes (called readers) may want only read the content of the shared object, while others (called writers) may want to update the object.
A reader process and a writer process or two writer processes are not allowed to access to the shared object simultaneously, otherwise inconsistency might occur. However, two reader processes are allowed to read the object simultaneously.
Assume that initially $b = 3$.

Process P_1	Process P_2
$a := b + 1;$	$b := 2;$

Example 6. Dinning philosophers problem

7. Semaphores

(1) A semaphore S is an integer variable that can only be accessed through two *atomic* operations: P and V (P stands for dutch passeren (to pass), V stands for dutch vrygeven (to release)).

$P(S)$:**while** $S \leq 0$ **do** skip;
$S := S - 1$;

$V(S)$:$S := S + 1$;

(2) Usage

a. Solving critical section problem
$mutex := 1$;
repeat
$P(mutex)$;
critical section;

$V(mutex)$;
remainder section;
until false

b. Synchronization