Operating Systems
Lecture #3

CONCURRENT PROCESSES

1. PROCESS CONCEPT

1) INTRODUCTION

Concurrent processing is basis of multiprogrammed operating systems, Process which is the unit of work in operating system is a program in execution. Process will need certain resources such as CPU time, memory, files, I/O devices, etc., to accomplish its task. These resources are given to the process when it is created.

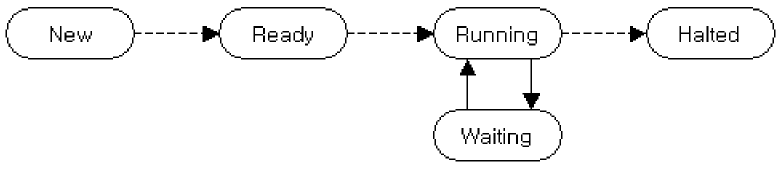
There are several motivations for allowing concurrent execution:

a) Physical resource sharing: multiuser environment since hardware resource are limited.
b) Logical resource sharing: shared file (same piece of information.
c) Computation speedup: parallel execution.
d) Modularity: divide system functions into separation processes.

2) PROCESS STATE

As process executes, it changes state. The state of a process is defined by its current activity.

Process state diagram



Since number of processes running in the system is potentially large, the CPU must be shared among several processes.

A sequential process may be in one of the following three. states

a) Running: instruction are being executed.
b) Blocked: the process is waiting for some event to occur (I70 completion).
c) Ready : the process is waiting to be assigned to a processor.

3) PROCESS CONTROL BLOCK (PCB)

Each process is represented in the operating system by its own process control block. A PCB is a data block or record containing many pieces of the information associated with a specific process, including:

a) Process state: new, ready, running waiting, halted
b) Program counter: indicates the address of the next instruction to be executed.
c) CPU registers: AC, index register, general purpose register. PC (save state information when an interrupt occurs).
d) Memory management information: I/O requests. I/O device allocated to this process.
e) I/O status information: process priority, pointer Io scheduling queues.
f) CPU scheduling information

4) OPERATION ON A PROCESS

Most of systems support at least two types of operations thai can be invoked on a process; process creation and process deletion.

a) process creation

Creating process: parent process
New processes: children of that process
When a process mates a new process several possible implementations exist:

i) Execution: concurrent versus sequential
- The parent continues to execute concurrently with its children.
- The parent waits until all of its children have terminated.

ii) Sharing : All Versus partial
- The parent and children sham all resources in common.
- The children share only a subset of their parent's resources.
- The parent and children share no resources in common.

b) Processes termination

When process terminate, it returns some data to its parent process.

A parent may terminate the execution of one of its children for a following reasons;

i) The child has exceeded its usage of some of the resources it has been allocated.
ii) The task assigned to the child is no longer required.

Cascading termination: do not allow a child to exist if its parent has terminated parent process terminate: its children must be terminated.

5) RELATION BETWEEN PROCESSES

The processes executing in the operating system may be either independent processes or cooperating processes.

a) Independent processes

* Its state is not shared in any way by any other process.
* Its execution is deterministic; that is, the result of execution depends only on the input state.
* Its execution is reproducible; that is, the result of the execution will always be the same for the same input.
* Its execution can be stopped and restarted without causing ill effects.

b) Cooperating processes

* Its state is sha/ed along other processes.
* Its execution is nondeterministics; that is, the result of the execution depends on relative execution sequence and cannot be predicted in advanced.
* Its execution in irreproducible; that is, the result of the execution will not always be the same for the same input.

2. PRECEDENCE GRAPHS

1) DEFINITION

A precedence graph is a directed acyclic graph whose nodes correspond to individual statements.

a) Example)

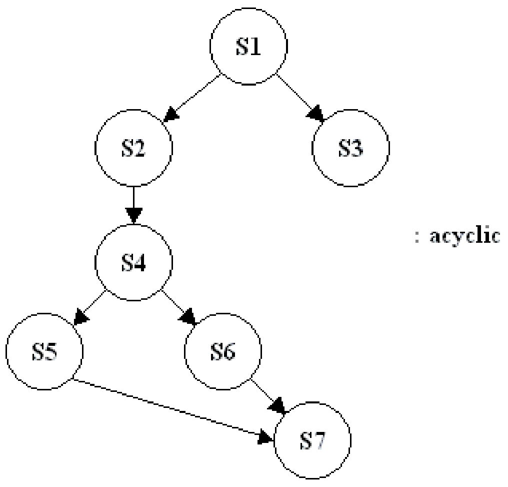
Consider the following program segment;

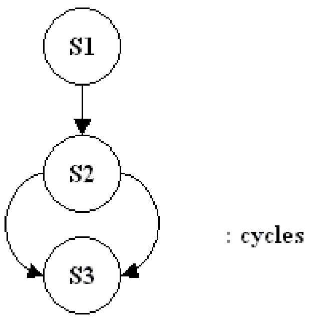
```
a := x + y;
b := z + 1;
c := a - b;
w := c + l;
```

Suppose we want to execute some of these statements concurrently.
The statement c := a - b cannot be executed before both a and b have been assigned values. Similarly, w := c + 1 cannot be executed before the new values of c has been computed. The statements a := x + y and b := z + 1 could be executed concurrently since neither depends upon the other.

b) In the precedence graph, the following precedence relations exist (fig.-1);

* S2 and S3 can executed after S1 completes.
* S4 can be executed after S2 completes.
* S5 and S6 can be executed after S4 completes.
* S7 can executed only after S5, S6, and S3 complete.





2) CONCURRENCY CONDITIONS

A precedence graph is a directed acyclic graph whose nodes correspond to individual statements. In other words, A edge from node Si to node Sj => Sj can be executed only after statement Si has completed execution.

a) Define notation

* R(Si) = {al, a2, a3 ....., an}, the read set for Si, is the set of all variables whose values are referenced in statement Si during the execution.

* W(Si) = {b1, b2, ..., bn}. the write set for Si, is the set of all variables whose values are changed (written) by the execution of statement Si.

b) Examples

* The statement c:= a - b;
  R(c := a - b) = {a, b}
  W(c := a - b) = {c}

* The statement w := c + 1;
  R(w := c + 1) = {c}
  W(w := c + 1) = {w}

* The statement x := x + 1;
  R(x := x + 2) = W(x := x + 2) = {x}
  (the intersection of R(Si) and W(Si) need not be null)

* The statement read(a)
  R(read(a)) = { }
  W(read(a)) = {a}

c) Bernstein's conditions

Following three conditions must be satisfied for Two successive statements S1 and S2 to be executed concurrently and still produce the same result.

i) R(S1) ∩ W(S2) = { }
ii) W(S1) ∩ R(S2) = { }
iii) W(S1) ∩ W(S2) = { }

* Questions: - two statement in a program be executed concurrently then produce same result?
  - precedencc graph can be correspond to program?

d) Examples of Bernstein's conditions

* S1 **:** a := x + y and S2 **:** b := z + 1

  R(S1) = {x, y}
  R(S2) = {z}
  W(S1) = {a}
  W(S2) = {b}

S1 and S2 can be executed concurrently (satisfy the Bernstein's conditions)

* S2 cannot be executed concurrently with S3: c:= a - b, since

$W(S2) \cap R(S3) = \{b\}$

$R(S2) = \{z\}$
$R(S3) = \{a, b\}$
$W(S2) = \{b\}$
$W(S3) = \{w\}$

3. SPECIFICATION

Limitation of precedence graph lies on its difficulty of using a programming language since it is a two dimensional object.

1) THE FORK AND JOIN CONSTRUCTS

a) Introduction

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

The join instruction provides the means to recombine two concurrent computations into one. The join instruction has parameter (count) to specify the number of computations which are to be joined.

```
count := count - 1;
IF count <> THEN quit;
```

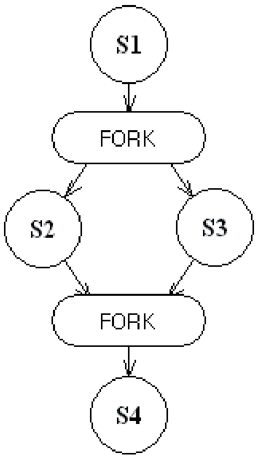
where count is a non-negative integer variable, and quit is an instruction which results in the termination of the execution.

Since computations may execute at different speeds, one may execute the join before the other.

Join instruction: the join instruction must be executed automatically (two join statements is equivalent to the serial execution of these two statements)

b) Example 1 :

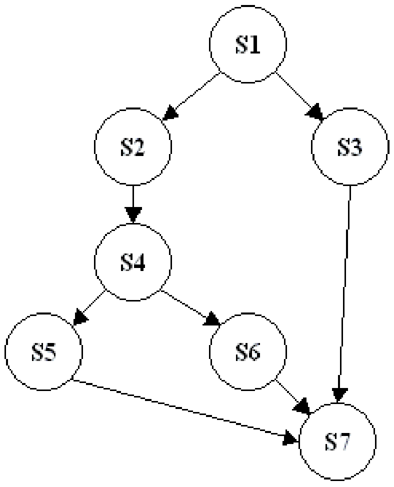
```
    count := 2;
    S1;
    FORK L1
    S2
    .
    .
    GO TO L2;
L1: S3;
L2: JOIN count; [* count := 2
    A4;
```



Note: S2 and S3 can execute concurrently

d) Exercise: construct a precedence graph using a following FORK/JOIN program.

```
      S1;
      count := 3;
      FORK L1;
      S2;
      S4;
      FORK L2;
      S5;
      GO TO L3;
  L2: S6;
      GO TO L3;
  L1: S3;
  L3: JOIN count;
      S7;
```



Note: statement "L1: S3" is not necessary GOTO L3 since L3 comes next step
count:= 3 (in degree of 3) at S7

Example 2:

Copies from a sequential file "f" to another file "g". By using double buffering with "r" and "s", this program can read from "f' concurrently with writing "g".

```
VAR f, g: file of T;
    r. s: T;
    count: integer

BEGIN
    reset (f);
    READ (f, r);
    WHILE not EOF (f) DO
         BEGIN
             count := 2;
             s := r
             FORK L1;
             WRITE (g, s);
             GOTO L2;
             L1: READ (f, r);
             L2: JOIN count;
         END
    WRITE (g. r)
END.
```

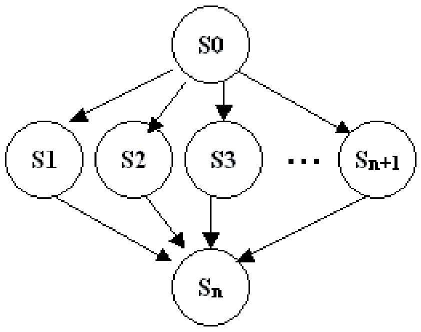
```
VAR f, g: file of T;
    r, s: T;

BEGIN
    reset (f);
    READ (f, r);
    WHILE not EOF (f) DO
       BEGIN
          s := r;
          PARBEGIN
              WRITE (g, s);
              READ (f, r);
          PAREND;
       END
    WRITE (g. r)
END.
```

3) THE CONCURRENT STATEMENT

a) Introduction

Limitation of FORK/JOIN: program has an awkward control structure (GO-TO statement has undesirable effects)

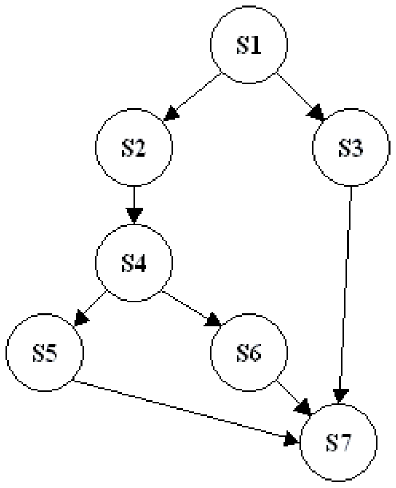
A higher-level language construct for specifying concurrency is the PARBEGIN/PAREND statement which has the following form:
PARBEGIN S1; S2; ...; Sn PAREND;



All statements enclosed between PARBEGIN and PAREND can be executed concurrently.

b) Example:

```
S1;
PARBEGIN
    S3;
    BEGIN
        S2;
        S4;
        PARBEGIN            <==>
            S5;
            S6;
        PAREND;
    END;
PAREND;
S7;
```



c) Strength of concurrent statement (PARBEGIN/PAP. END):

Early added to a modem block-structured high-level language and have the advantages of structured control statements (add other mechanisms such as semaphores).

3) COMPARISON

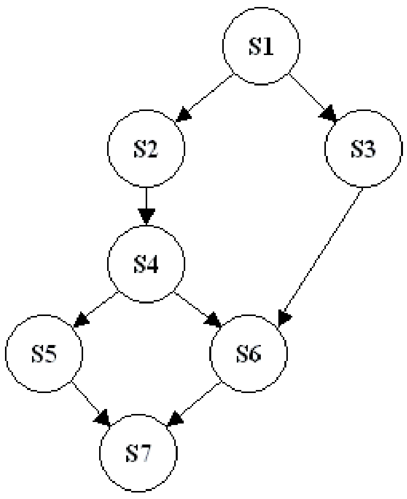
a) Introduction

The concurrent statement (PARBEGIN/PAREND) is not powerful enough to model all possible precedence graph. On the other hand, in terms of modeling precedence graphs, the FORK/JOIN construct is more powerful than the concurrent statement.

b) Example:

Can you construct an equivalent program using PARBEGIN/PARENT) to the following precedence graph ?

* Precedence graph with no corresponding concurrent statement



* FORK/JOIN construct for above precedence graph

```
     S1;
     countl := 2;
     FORK L1;
     S2;
     S4;
     count2 := 2;
     FORK L2;
     S5;
     GO TO L3;
L1:  S3;
L2:  JOIN countl;
     S6;
L3:  JOIN count2;
     S7:
```