Review: Concurrent Processes

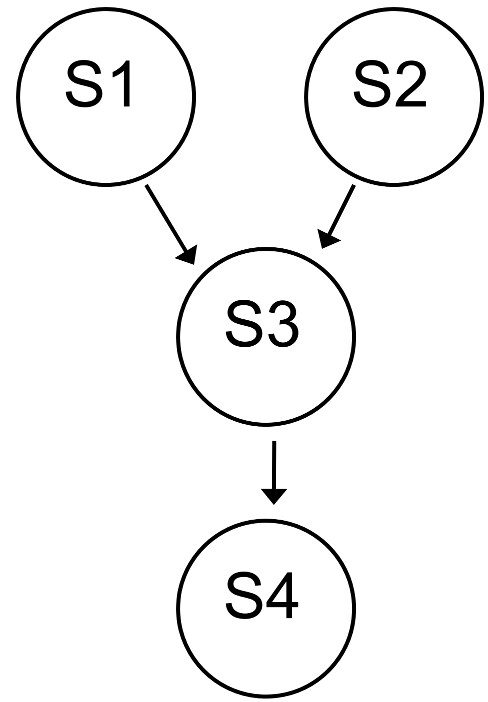
Outline

1. Review basic issues in concurrency
2. The critical section problem
3. Hardware support for critical sections

Concurrent Processes

Consider the following program:

```
S1:  a := x + y;
S2:  b := z + 1;
S3:  c := a - b;
S4:  w := c + 1;
```



Which can be executed concurrently?

Which must precede others?

Let each node be a process

Then each FORK initiates a new process

Each JOIN kills all but one process

S1 must precede S3

S2 must precede S3

S3 must precede S4

How to Determine Precedence

Fundamental Property: If statements are not explicitly constrained, then any order is possible. ***>> would reordering change the outcome?***

→ True for serial as well as parallel systems!

Q: If an instruction precedes another, when can this precedence be removed without changing the intended result of the program execution?

A: When three conditions hold:

a) No Read after Write (data/true dependence)

```
a := b + c;
d := a + e;
```

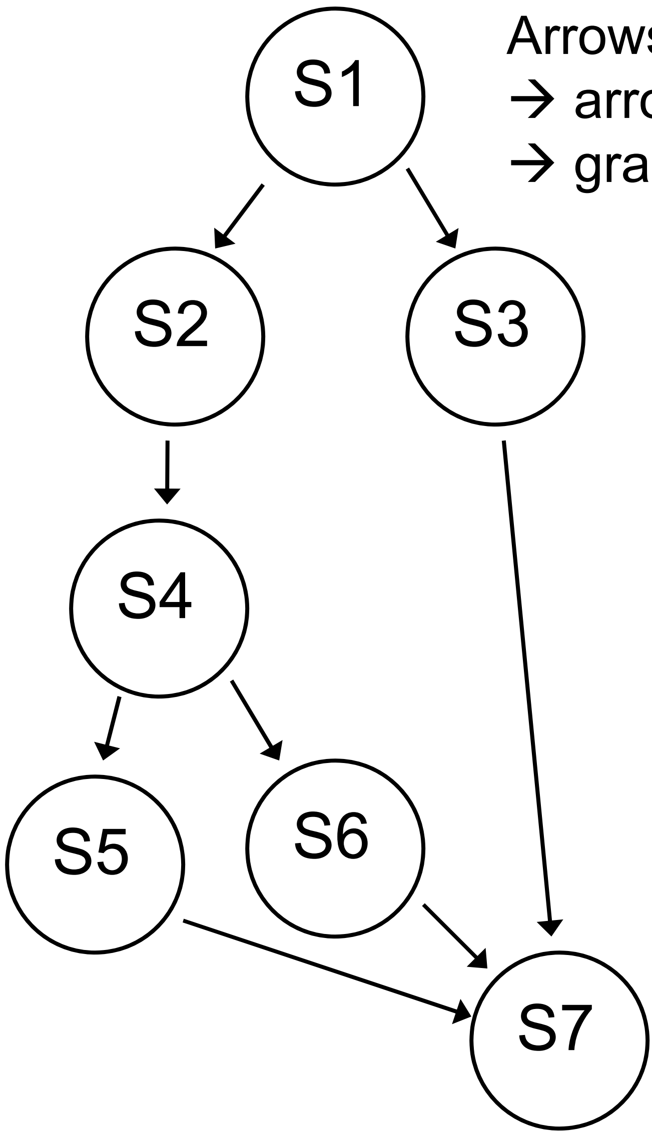
b) No Write after Read (anti-dependence)

```
d := a + e;
a := b + c;
```

c) No Write after Write (output dependence)

```
a := b + c;
a := d + e;
```

Precedence Graphs



Arrows represent orderings
→ arrows are transitive
→ graphs must be acyclic

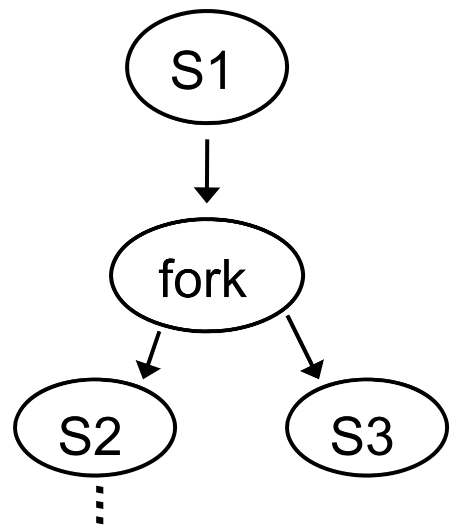
<u>Handmade precedence graphs</u>

forkL ≡ produces two concurrent executions

1) continuation after fork
2) execution beginning at label L

Example:

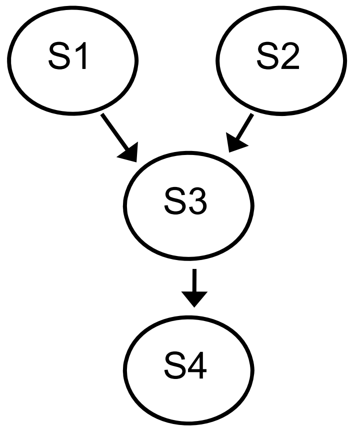
```
S1;
fork L;
S2;
.
.
L:   S3;
```



join count ≡ recombine multiple concurrent threads into one

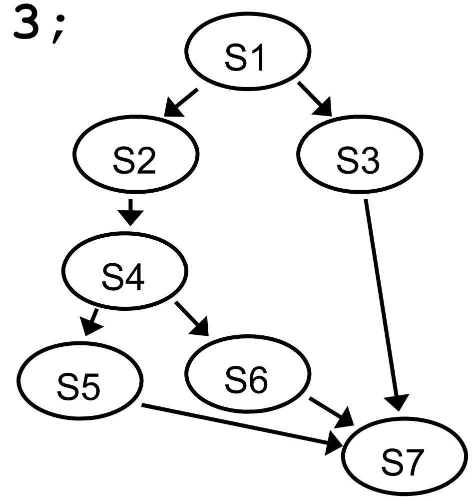
Example 2:

```
    count := 2;
    fork L1;
    a := x + y;
    goto  L2;
L1: b := z + 1;
L2: join count
    c := a - b;
    w := c + 1;
```



Example 3:

```
    S1;
    count := 3;
    fork L1;
    S2;
    S4;
    fork L2;
    S5;
    goto L3;
L2: S6;
    goto L3;
L1: S3;
L3: join count;
    S7;
```

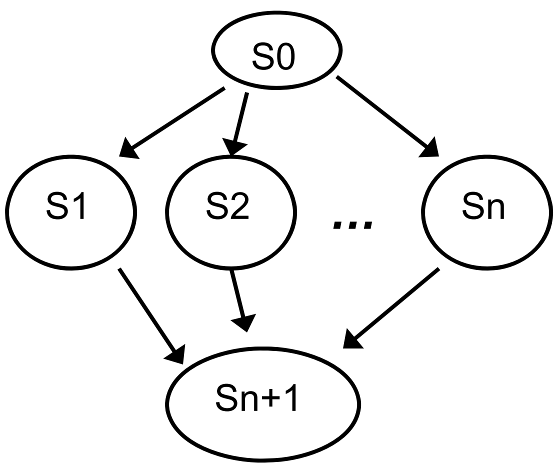


Problem: Too much flexibility -- makes efficient compilation and code maintenance very difficult

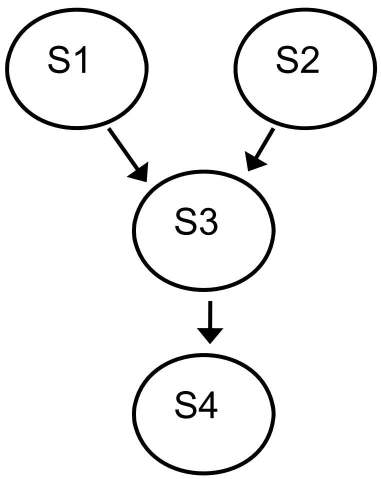
Specification Alternative: Parbegin/Parend

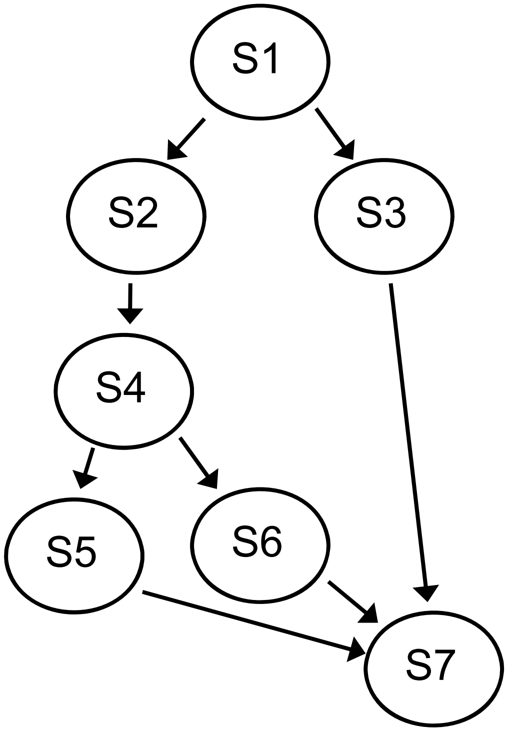
```
S0;
Parbegin S1; S2; … Sn; Parend
Sn+1;
```

≡ All statements Si between Parbegin and Parend can be executed simultaneously




```
# Original example
Parbegin
  a := x + y;   # S1
  b := z + 1;   # S2
Parend
c := a - b;     # S3
w := c + 1;     # S4
```

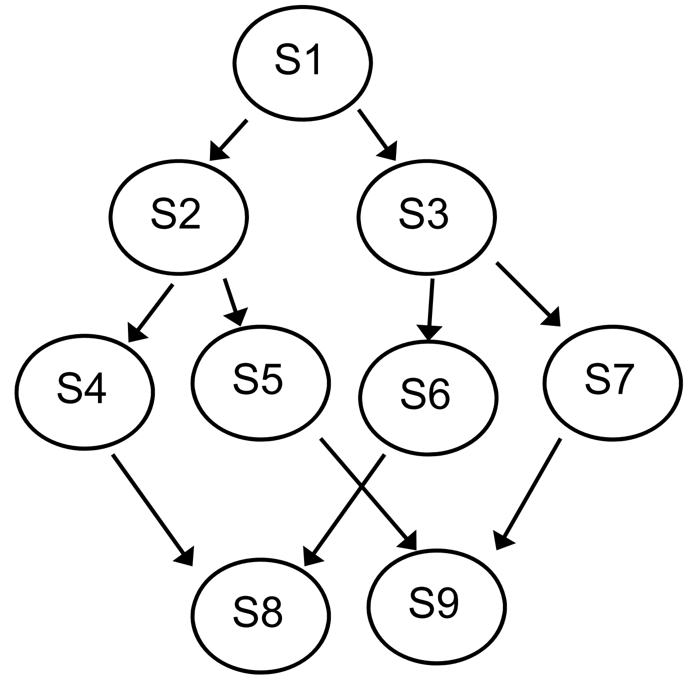





```
# 2nd Example
# Note block
#  structured style
S1;
Parbegin
  S3;
  Begin
    S2;
    S4;
    Parbegin
      S5;
      S6;
    Parend
  End
Parend
S7;
```

Another example:

easy with fork/join but impossible to implement with parbegin/parend



The Critical Section Problem

Let P1 and P2 be two processes …

Let Count = 5

```
Begin P1
    .
    .
   Count := Count + 1;
    .
End P1
```

```
Begin P2
    .
   Count := Count - 1;
    .
    .
End P2
```

What is the value of Count when P1 and P2 finish?

```
Begin P1
   .
   .
   Count := Count + 1;
   .
End P1

# in assembly language
    LD   R1,Count
    ADD  R1,R1,1
    ST   R1,Count
```

```
Begin P2
   .
   Count := Count - 1;
   .
   .
End P2

# in assembly language
    LD   R2,Count
    SUB  R2,R2,1
    ST   R2,Count
```

```
# One possible interleaving:
# (Why are multiple different interleavings
#   possible even on a single core?)

    LD   R1,Count       # P1
    ADD  R1,R1,1        # P1
    LD   R2,Count       # P2
    SUB  R2,R1,1        # P2
    ST   R2,Count       # P2
    ST   R1,Count       # P1

# PROBLEM:  Count must be updated ATOMICALLY!
```

Solution Requirements

3 Issues

1. Mutual Exclusion: If P_i is in the critical section, then no other P_j is allowed in.

2. Progress: Only processes waiting to enter the critical section can participate in the decision. Decision must be made in finite time.

3. Bounded Waiting: All processes are allowed in the critical section only a finite # of times before every process has access.

Critical Section Problem: SW Solution

```
while (   ) do
   begin
      parbegin
         P0;
         P1;
      parend
   end
end
```

```
Process Pi
   repeat

      entry section

      critical section

      exit section

      remainder section

   until false
```

Algorithm 1

```
BOOL TURN = RANDOM();
// Two processes:  P0, P1
FOR Pi Initialize TURN to 0 or 1
REPEAT
    entry section
  WHILE TURN ≠ i DO SKIP;

    critical section

  TURN := j;
    remainder section
UNTIL FALSE;
```

Allows alternating execution
What's the problem?

Algorithm 2

```
// Two processes:  P0, P1
flag[i] := FALSE; flag[j] := FALSE;
REPEAT
    entry section
  WHILE flag[j] DO SKIP;
  flag[i] := TRUE;

    critical section

  flag[i] := FALSE;
    remainder section
UNTIL FALSE;
```

What's the problem?

```
T0:  P0 executes `while' and finds flag[1] == F

T1:  P1 executes `while' and finds flag[0] == F

T2:  P1 sets flag[1] ← T

T3:  P0 sets flag[0] ← T
```

Now BOTH P0,P1 are in the critical section!

Q: Can we make

while … *flag …*

another critical section?

Algorithm 3

```
// Two processes:  P0, P1
REPEAT
    entry section
  flag[i] := TRUE;
  WHILE flag[j] DO SKIP;

    critical section

  flag[i] := FALSE;
    remainder section
UNTIL FALSE;
```

What's the problem?

Solution

```
// Two processes:  P0, P1
flag[0], flag[1];  // initialize to FALSE
turn;              // initialize to 0 or 1
REPEAT
    entry section
  flag[i] := TRUE; // I'd like to enter
  turn := j;       // but it's yours if you want it
  // I'll wait while you want it or it's your turn
  WHILE (flag[j] AND turn = j) DO SKIP;

    critical section

  flag[i] := FALSE;  // I don't want it anymore
    remainder section
UNTIL FALSE;
```

Why does it work?

→ setting **`flag[i] = FALSE`** when leaving the critical section ensures that this process won't prevent entry

→ **`turn`** ensures that only one process will succeed

a) either it is **`Pi`**'s turn or **`Pj`** does not care

b) c) only loop if **`flag[j] = TRUE AND turn = j`**

This only happens if **`Pj`** *is in the critical section*

What about crash in the critical section?
… mendacity??

Hardware Solutions

Basic Technique: Multiple operations are performed on one memory location ***atomically.***

Two Methods:

```
int TEST-AND-SET(target)
   TEST-AND-SET := target;
   target := TRUE

VOID SWAP(a,b)
   temp := a;
   a := b;
   b := temp;
```

Critical Section w/ atomic primitives

```
// Use TEST-AND-SET
// Let "target" be lock
// Let lock be FALSE
REPEAT
    entry section
  WHILE TEST-AND-SET(lock)
   DO SKIP;

    critical section

  lock := FALSE;
    remainder section
UNTIL FALSE;
```

```
// Use SWAP
// Let "lock" be a global Boolean
// Let "key" be a local Boolean
// Initialize lock to FALSE
REPEAT
    entry section
  key := TRUE;
  REPEAT
    SWAP(lock,key);
  UNTIL key = FALSE;

    critical section

  lock := FALSE;
    remainder section
UNTIL FALSE;
```

Critical Section w/ multiple processes

```
// Use TEST-AND-SET
// Common data structures, initialized to false
 var waiting: array [0.. n-1] of boolean
     lock: boolean

// For process Pi:
 var j: 0..n-1;
     key: boolean;

repeat
    entry section

 waiting[i] := true;
 key := true;
 while waiting[i] and key do key := TEST-AND-SET(lock);
 waiting[i] := false;

    critical section

 j := i+1 mod n;
 while (j != i) and (not waiting[j]) do j := j+1 mod n;
 if j = i then lock := false;
          else waiting[j] := false;

    remainder section
until false;
```

Semaphores

Semaphore S ≡ an integer variable
only accessible through two
atomic operations **P** and **V**

```
P(S): while S <= 0 do skip;
          S := S - 1;
```

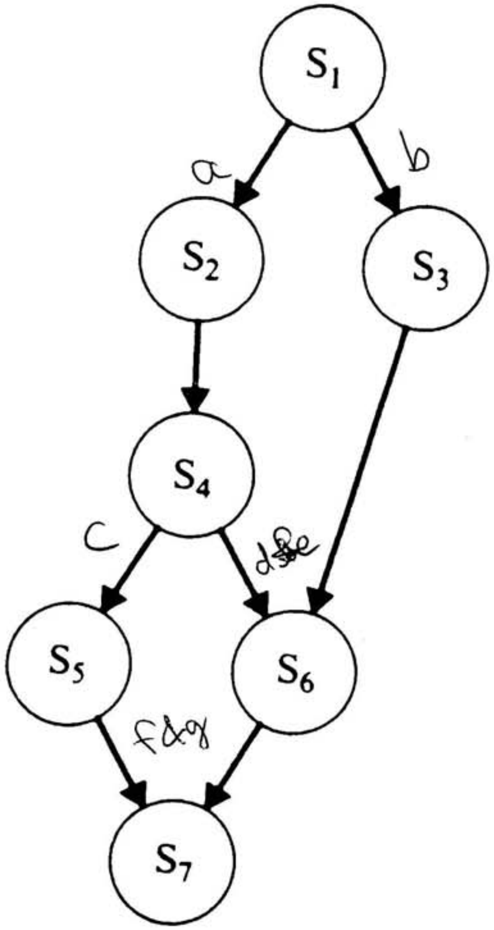
```
V(S):  S := S + 1;
```

Example: Let n processes share **lock**

```
// Let lock be initialized to 1
// Then code for Pi →

repeat
   P(lock);
       critical section
   V(lock);
       remainder section
until false;

// this is similar to
//   test-and-set
```

Ex: Semaphores and Synchronization

```
sync := 0;     // not 1

     P1                  P2
      .                   .
      .                   .
     S1;                P(sync);
     V(sync);           S2;
      .                   .
      .                   .
// what are the P and V doing?
```




```
var a, b, c, d, e, f, g: semaphores;
(* initial value of all semaphores is 0 *)
    begin
        parbegin
            begin S1; V(a); V(b); end;
            begin P(a); S2; S4; V(c); V(d); end;
            begin P(b); S3; V(e); end;
            begin P(c); S5; V(f); end;
            begin P(d); P(e); S6; V(g); end;
            begin P(f); P(g); S7; end;
        parend;
    end;
```

all in "wait" state

modify?

Semaphore Implementation

Problem with both Peterson's SW solution and semaphore: **Busy Waiting**

Solution: Block if S is not positive

Problem: How do you restart blocked process?

Solution: Through the execution of a **V** op. by another process

```
// Semaphore has not only an integer,
// but also a list of processes

type Semaphore = record
   value = integer;
   L = list of processes;
end;
```

```
// If P finds integer not positive,
// then adds self to list and blocks

P(s):   S.value := S.value - 1;
        if S.value < 0
          then begin
            add process to S.L;
            block;
          end;
```

```
// If V finds integer negative, then it knows
//   that another process is waiting
// Note:  lock not necessarily opened after
//   V(S) completion since S.value can still
//   be <= 0

V(S):   S.value := S.value + 1;
        if S.value <= 0
          then begin
            remove a process P
                from S.L;
            wake up (P);
          end;
```

Notes

→ Integer can be negative in this implementation:

  |value| = # of processes waiting

→ Linked list can be list of PCB pointers

  → but then must be in system space

→ List should be FIFO or some priority strategy (not stack!)

→ P & V must remain atomic

  → lock out interrupts (for a uniprocessor)

  → hardware support or software solution to critical section problem (for a multiprocessor)

→ But, still busy waiting on entry into semaphore!

  → However, critical section is very short

  → And requires well thought-out system support