

2. Processes and Interactions

2.1 The Process Notion

2.2 Defining and Instantiating Processes

- Precedence Relations
- Implicit Process Creation
- Dynamic Creation With fork And join
- Explicit Process Declarations

2.3 Basic Process Interactions

- Competition: The Critical Problem
- Cooperation

2.4 Semaphores

- Semaphore Operations and Data
- Mutual Exclusion
- Producer/Consumer Situations

2.5 Event Synchronization

ICS 143

1

Processes

- process is the activity of executing a program on a CPU
- conceptually each process has its own CPU
- all processes are running concurrently
- physical concurrency = parallelism
 - requires multiple CPUs
- logical concurrency = time-shared CPU
- processes compete or cooperate

ICS 143

2

Why process structure?

- hardware-independent solutions
 - processes cooperate and compete correctly regardless of the number of CPUs
- structuring mechanism
 - tasks are isolated with well-defined interfaces

ICS 143

3

Defining/instantiating processes

- examples of precedence relationships:

Figure 2.1

ICS 143

4

Process flow graphs

- serial execution is expressed as:
 $S(p1, p2, \dots)$
- parallel execution is expressed as:
 $P(p1, p2, \dots)$
- Example 1:
 $S(p1, P(p2, S(p3, P(p4, p5))), p6, P(p7, p8))$

= Figure 2.1(c)

ICS 143

5

Process flow graphs

- Example 2:
Expression: $(a + b) * (c + d) - (e / f)$

Figure 2.2

Exercise: Derive process flow graph

ICS 143

6

Process flow graphs

- Compare:

Figure 2.1(c) Figure 2.1(d)

- we have seen expression for (c) using S/P
- (d) cannot be expressed using S/P -- *not properly nested*

ICS 143

7

Implicit process creation

- processes are created dynamically using language constructs; no process declaration
- cobegin/coend
 - syntax: cobegin C1 // C2 // ... // Cn coend
 - meaning:
 - all Ci may proceed concurrently
 - when all terminate, statement following cobegin/coend continues

ICS 143

8

cobegin/coend

- cobegin/coend have the same expressive power as S/P notation
 - $S(a,b) \equiv a; b$ (sequential execution by default)
 - $P(a,b) \equiv \text{cobegin } a \text{ // } b \text{ coend}$

ICS 143

9

cobegin/coend

- Example:

```
cobegin
Time_Date // Mail //
  Edit; cobegin
    Compile; Load; Execute //
    Edit; cobegin Print // Web coend
  coend;
coend;
```

- What does the process flow graph look like?

ICS 143

10

Data parallelism

- same code is applied to different data
- the *forall* statement
 - syntax: forall (parameters) statements
 - meaning:
 - parameters specify set of data items
 - statements are executed for each item concurrently

ICS 143

11

The *forall* statement

- Example: matrix multiply

```
forall ( i:1..n, j:1..m )
  A[i][j] = 0;
  for ( k=1; k<=r; ++k )
    A[i][j] = A[i][j] +
      B[i][k]*C[k][j];
```

- each inner product is computed sequentially
- all inner products are computed in parallel

ICS 143

12

The *fork* and *join* primitives

- cobegin/coend are limited to properly nested graphs
- forall is limited to data parallelism
- fork/join can express arbitrary functional parallelism (any process flow graph)

ICS 143

13

The *fork* and *join* primitives

- syntax: fork x
- meaning: create new process that begins executing at label x
- syntax: join t,y
- meaning:
 - t = t-1;
 - if (t==0) goto y;
- the operation must be indivisible (why?)

ICS 143

14

The *fork* and *join* primitives

- Example: graph in Figure 2-1(d)

```
t1 = 2; t2 = 3;
p1; fork L2; fork L5; fork L7; quit;
L2: p2; fork L3; fork L4; quit;
L5: p5; join t1,L6; quit;
L7: p7; join t2,L8; quit;
L3: p3; join t2,L8; quit;
L4: p4; join t1,L6; quit;
L6: p6; join t2,L8; quit;
L8: p8; quit;
```

ICS 143

15

The Unix *fork*

- `procid = fork()`
- replicates calling process
- parent and child are identical except for `procid`
- use `procid` to diverge parent and child:

```
if (procid==0)do_child_processing
else do_parent_processing
```

ICS 143

16

Explicit process declarations

- designate piece of code as a unit of execution
 - facilitates program structuring
- instantiate:
 - statically (like `cobegin`) or
 - dynamically (like `fork`)

ICS 143

17

Explicit process declarations

```
process p

process p1
  declarations_for_p1
begin ... end

process type p2
  declarations_for_p2
begin ... end

begin
  ...
  q = new p2;
  ...
end
```

ICS 143

18

Process interactions

- Competition: the Critical Problem

```
cobegin
p1: ...
   x = x + 1;
   ...
   //
p2: ...
   x = x + 1;
   ...
coend
```

- x should be 2 after both processes execute

ICS 143

19

The critical problem

- interleaved execution (due to parallel processing or context switching):

```
p1: R1 = x;          p2: ...
   R1 = R1 + 1;      R2 = x;
   x = R1 ;          R2 = R2 + 1;
   ...              x = R2;
```

- x has only been incremented once -- the first update (x=R1) is lost;

ICS 143

20

The critical problem

- problem statement:

```
cobegin
p1: while (1) CS1; program1;
   //
p2: while (1) CS2; program2;
   //
   ...
   //
pn: while (1) CSn; programn;
coend
```

- guarantee **mutual exclusion**: only one process executed within its CSi

ICS 143

21

Software solutions

- in addition to mutual exclusion, prevent **mutual blocking**:
 1. process outside of its CS must not prevent other processes from entering
 2. process must not be able to repeatedly reenter its CS and **starve** other processes (fairness)
 3. processes must not block forever (**deadlock**)
 4. processes must not repeatedly yield to each other (“after you”--“after you” **livelock**)

ICS 143

22

Algorithm 1

- use a single turn variable

```
int turn = 2;
cobegin
p1: while (1)
    while (turn==2) ; /* wait */
    CS1; turn = 2; program1;

    // ...
```

- violates blocking requirement (1)

ICS 143

23

Algorithm 2

- use two variables to indicate intent

```
int c1 = 1, c2 = 1;
cobegin
p1: while (1)
    c1 = 0;
    while (!c2) ; /* wait */
    CS1; c1 = 1; program1;

    // ...
```

- violates blocking requirement (3) -- processes wait forever

ICS 143

24

Algorithm 3

- like #2 but reset intent variable each time

```
int c1 = 1, c2 = 1;
cobegin
p1: while (1)
    c1 = 0;
    if (!c2) c1 = 1;
    else CS1; c1 = 1; program1;

// ...
```

- violates blocking requirements (2) and (4)

Algorithm 4 (Peterson)

- like #2 but use a turn variable to break a tie

```
int c1 = 0, c2 = 0, will_wait;
cobegin
p1: while (1)
    c1 = 1;
    will_wait = 1;
    while (c2 && (will_wait==1) ;
        /*wait loop*/
    CS1; c1 = 0; program1;

// ...
```

- guarantees mutual exclusion *and* no blocking

Cooperation

- problems with software solutions
 - difficult to program and verify
 - processes loop while waiting (busy-wait)
 - applicable to only to critical problem: *competition* for a resource
- *cooperating* processes also need to synchronize
- generic scenario:
producer → buffer → consumer

Semaphores

- a semaphore s is a nonnegative integer
- P and V operate on s
- semantics:
 - V(s): increment s by 1
 - P(s): if $s > 0$ then decrement s , else wait until $s > 0$
- the waiting can be implemented by
 - blocking the process, or
 - busy-waiting (see Chapter 4)
- P and V are indivisible operations (atomic)

ICS 143

28

Mutual exclusion w/ semaphores

```
semaphore mutex = 1;
cobegin
...
//
p1: while (1)
    P(mutex); CSi; V(mutex); programi;

//
...
coend;
```

ICS 143

29

Signal/wait with semaphores

```
semaphore s = 0;
cobegin
p1: ...
    P(s); /* wait for signal */
    ...

//
p2: ...
    V(s); /* send signal */
    ...

...
coend;
```

ICS 143

30

Bounded buffer problem

```
semaphore e = n, f = 0, b = 1;
cobegin
  Producer: while (1)
    Produce_next_record;
    P(e); P(b); Add_to_buf; V(b); V(f);

  //
  Consumer: while (1)
    P(f); P(b); Take_from_buf; V(b); V(e);
    Process_record;

coend
```

ICS 143

31

Event synchronization

- synchronous event (e.g. I/O completion)
 - process waits for it explicitly
 - constructs: E.wait, E.post
- asynchronous event (e.g. arithmetic error)
 - process provides event handler
 - invoked whenever event is posted

ICS 143

32

Case study: Event synch.

- UNIX signals
 - kill(pid, sig) send signal (SIGILL, SIGKILL, ...)
 - process may catch signal
 - process may ignore signal
 - default: kill process

ICS 143

33

Case study: Event synch. (cont)

- Windows 2000
 - WaitForSingleObject or WaitForMultipleObjects
 - process blocks until object is signaled

object type	signaled when:
process	all threads complete
thread	terminates
semaphore	incremented
mutex	released
event	posted
timer	expires
file	I/O operation terminates
queue	item placed on queue

ICS 143

34
