# Chapter 12

(Sec. 12.1 – 12.4)

Concurrency – Synchronization with Semaphores and Monitors

## Thread/Process

*Thread* - A program unit that can execute concurrently with other program units.

*Heavyweight process* – Has its own address space

*Lightweight processes* – Processes that share an address space

*Task* – A well-defined unit of work performed by some thread

**** No consistent definition for these terms ****

## Concurrency Issues

*Communication*

How a thread obtains information produced by another thread

- Shared memory
- Message passing

*Synchronization*

Controlling the relative order that operations occur in different threads

- Busy-waiting (spinning)
- Blocking (scheduler-based)

## Thread Creation

Control flow

- Co-begin
- Parallel loops

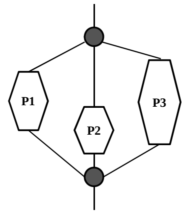
Subprograms

- Launch-at-elaboration
- Fork
- Implicit receipt
- Early reply

## Co-begin

**Co-begin** and **end** bracket statements (processes) that are all created and started in parallel

**co-begin**
  P1;
  P2;
  P3;
**end**;




## Parallel Loops

C#

```
Parallel.For(0, 3, i => {
    Console.WriteLine("Thread " + i + " here");
});
```

High Performance Fortran (HPF)

```
forall (i=1:n-1)
    A(i) = B(i) + C(i)
    A(i+1) = A(i) + A(i+1)
end forall
```

## Launch-at-Elaboration

Ada tasks are created and begin execution when the enclosing program unit starts

The program unit doesn't terminate until all enclosed tasks terminate

Each task has a single point of control

## Ada Task Example

```
with text_io; use text_io;

procedure ConcurrentWriter is

  task WriteA;
  task body WriteA is
    begin
      for j in 1..10 loop
        put('A'); new_line;
      end loop;
    end WriteA;

  task WriteB;
  task body WriteB is
    begin
      for j in 1..10 loop
        put('B'); new_line;
      end loop;
    end WriteB;

begin
  null;
end ConcurrentWriter;
```

## Subprogram Concurrency

Tasks differ from ordinary subprograms in that:

1. A task may be implicitly started
2. When a program unit starts the execution of a task, it is not necessarily suspended
3. When a task's execution is completed, control may not return to the caller

Tasks usually work together

## `fork` in C

- **fork** function is executed in a parent process and creates a child process
  - Children share their parent's code
- Child process begins executing immediately after being created; and parent resumes
- The **fork** function returns the child's process ID number to the parent and returns 0 to the child
- Processes are killed at the end of their code
- A **wait** function can be called to suspend a parent until a child terminates

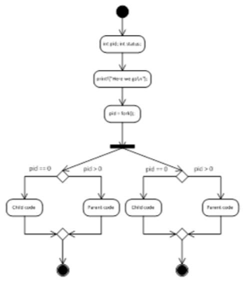
## `fork` in C

```
int childNum = fork();
if (childNum == 0)
   // child process code
else {
   // parent process code
}
```

## `fork` Example

```
#include <unistd.h>
#include <stdio.h>
int main() {
   int pid;   int status;
   printf("Here we go\n");
   pid = fork();
   if (pid == 0) { // pid == 0 in child process
       printf("I'm the child process.
                          I got %d as pid!\n", pid);
       sleep(4);
   } else { /* pid > 0 in the parent process */
       printf("I'm the parent process.
                          My child's pid = %d\n", pid);
       wait(&status); // Wait for child to terminate
       printf("Child done. status = %d\n", status);
   }
}
```

## fork Example




## Synchronization

Mechanism that controls the order in which tasks execute

Required when a task must wait for some other task to complete an activity before it can continue

Task communication is necessary for synchronization

Involves exchange of control info

## Interaction Between Tasks

**Communication**

Sharing and exchanging information between tasks

1. Parameters
2. Shared non-local variables (shared memory model - must guarantee *mutually exclusive* access)
3. Message Passing (distributed processing model)

## Interaction Between Tasks

**Synchronization** – mechanism that controls the order in which processes execute

- Competition – each process requires exclusive use of a resource
- Cooperation – two processes work on parts of the same problem

## Implementing Synchronization

Make some operation *atomic*

- Mutual exclusion – only one thread is executing a *critical section* at any given point in time

Methods for Providing Synchronization

- Semaphores
- Monitors
- Message Passing

## Semaphores

Introduced by Edsger Dijkstra in 1965

A data structure consisting of

An integer counter

A queue of suspended tasks

There are two *atomic* (indivisible) operations

**wait**

**release**

(originally called *P* and *V* – see Fig. 12-14 for detained implementation)

Can be used to provide both cooperation and competition synchronization

## Semaphore - WAIT

**procedure wait(sem)**

```
if sem.counter > 0 then
   sem.counter--
else
   put the calling task in sem.queue
   transfer control to a task from ready-list
   // Deadlock if none are ready
end
```

## Semaphore - RELEASE

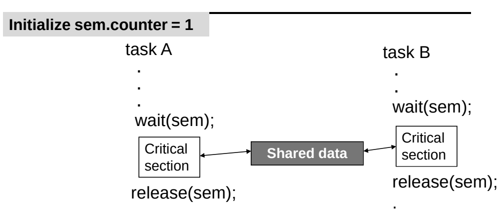
**procedure release(sem)**

```
if sem.queue.is_empty() then
  sem.counter++
else
  put calling task in ready-list
  transfer control to a task from sem.queue
end
```

## Shared Data and Semaphores

**Critical Section** – a portion of code that must be treated as an atomic unit

Access to shared data occurs in a critical section that is guarded by a semaphore




## Shared Data and Semaphores

The previous example used a *binary semaphore*

- Counter initialized to 1
- Guarantees mutual exclusion of critical section by requiring wait/release operations to occur alternately

## Cooperation Synchronization with Semaphores

*Shared buffer example*

- The buffer is implemented as an ADT with the operations **`DEPOSIT`** and **`FETCH`** as the methods to access the buffer
- Use two semaphores for cooperation: **`emptyspots`** and **`fullspots`**
- The semaphore counters are used to store the numbers of empty spots and full spots in the buffer

## DEPOSIT

**`DEPOSIT`** must first check **`emptyspots`** to see if there is room in the buffer

If there is room,
decrement **`emptyspots`** counter
and insert the value

If there is no room,
put the caller in **`emptyspots`** queue

When **`DEPOSIT`** is finished, it increments the **`fullspots`** counter

## FETCH

**FETCH** must first check **fullspots** to see if there is a value

If there is a full spot, decrement the **fullspots** counter and remove a value

If there are no values in the buffer, put the caller in the **fullspots** queue

When **FETCH** is finished, it increments the **emptyspots** counter

## Example — Cooperation

```
semaphore fullspots,
  emptyspots;
fullspots.count = 0;
emptyspots.count = BUFLEN;

task producer;
  loop
    -- produce value
    wait(emptyspots)
    DEPOSIT(value);
    release(fullspots)
  end loop;
end producer;
```

```
task consumer;
  loop
        wait(fullspots)
        FETCH(value);
        release(emptyspots)
        -- consume value
  end loop;
end consumer;
```

## Competition Synchronization with Semaphores

**fullspots** and **emptyspots** are used for cooperation

- Make sure there is *data* for the consumer
- Make sure there is *space* for the producer

A binary semaphore, named **access**, is used to control access to the shared buffer

- Prevent the producer and consumer from using the same buffer location at the same time (competition synchronization)

## Example — Cooperation/Competition

```
task producer;
loop
  -- produce value
  wait(emptyspots)
  wait(access)
  -- deposit value
  release(access)
  release(fullspots)
end loop;
end producer;
```

```
task consumer;
loop
     wait(fullspots)
wait(access)
     -- get value
release(access)
     release(emptyspots)
     -- consume value
end loop;
end consumer;
```

## Evaluation of Semaphores

Misuse of semaphores can cause failures in

1. Cooperation synchronization
   e.g., the buffer will overflow if the **wait** of **fullspots** is left out
2. Competition synchronization
   e.g., The program will deadlock if the **release** of **access** is left out

Per Brinch Hansen (1973)

*"The semaphore is an elegant synchronization tool for an ideal programmer who never makes mistakes"*

## Monitors

Introduced by Brinch Hansen in 1973

Abstract Data Type for shared data

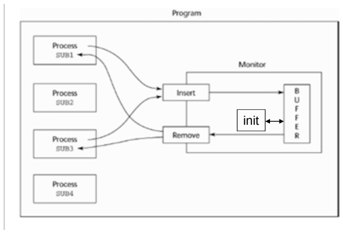
The idea: encapsulate the shared data and its operations to restrict access

Instances are statically created by declarations

Hybrid implementations in Ada, Java, C#

## Monitors – Concurrent Pascal Style

An instance is "started" by **`init`**, which allocates its local data and begins its execution




## Monitors – Competition

A monitor allows only one process at a time to execute the monitor's subprograms

- Calls are queued if the monitor is busy at the time of call

Shared data and access methods reside in the monitor, not in the client program

Mutually exclusive access to shared data is built in

## Shared Buffer with Monitors

```
type  databuf = monitor
const bufsize = 100;
var buf : array[1..bufsize] of integer;
    next_in, next_out : 1..bufsize;
    filled : 0..bufsize;
    sender_q, receiver_q : queue;

procedure entry deposit(item : integer);
 begin
  if filled = bufsize then delay(sender_q);
  buf[next_in] := item;
  next_in := (next_in mod bufsize) + 1;
  filled := filled + 1;
  continue(receiver_q);
 end;
```

## Shared Buffer with Monitors

**`procedure entry`** **-** only one can be executing at any given time

**`delay`** **-** places process that calls it in the specified queue and removes its exclusive access rights to the monitor

**`continue`** – disconnect process that calls it from the monitor and check specified queue for processes suspended by a **`delay`** operation

## Shared Buffer with Monitors

Initialization code for **`databuf`**:

```
begin
  filled := 0;
  next_in := 1;
  next_out := 1;
end;
```

## Shared Buffer with Monitors

```
type producer =
  process(databuf:db)
 cycle
   -- produce value nv
   buffer.deposit(nv)
 end;
end producer;
```

```
type consumer =
  process(databuf : db)
 cycle
   buffer.fetch(sv)
   -- consume value sv
 end;
end consumer;
```

## Shared Buffer with Monitors

Program that uses shared buffer:

```
var a_producer : producer;
  a_consumer : consumer;
  a_buffer : databuf;
begin
 init
   a_buffer, a_producer(a_buffer),
   a_consumer(a_buffer);
end;
```

## Evaluation of Monitors

- The monitor ADT avoids competition synchronization problems that can occur with semaphores
- Co-operation has the same problems as semaphores
- Java threads are based on the idea of monitors