Synchronization and Communication

Making processes/threads work together

Computadores II / 2004-2005

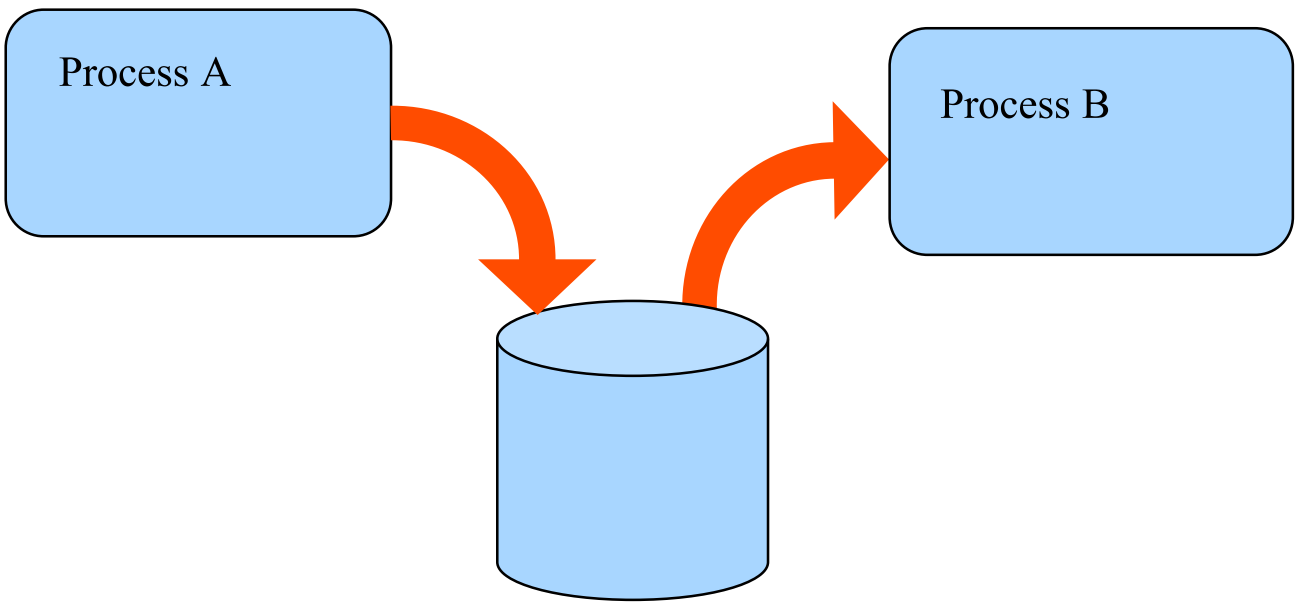
Objectives

- To understand the requirements for communication and synchronisation based on shared variables
- To briefly review semaphores, monitors and conditional critical regions
- To understand various alternatives like POSIX mutexes, Java synchronized methods or Ada 95 protected objects

Process Cooperation

- The correct behaviour of a concurrent program depends on **synchronisation** and **communication** between its processes
- **Synchronisation**: the satisfaction of constraints on the interleaving of the actions of processes (e.g. an action by one process only occurring after an action by another)
- **Communication**: the passing of information from one process to another
  - Concepts are linked since communication requires synchronisation, and synchronisation can be considered as contentless communication.
  - Data communication is usually based upon either shared variables or message passing.

Shared Variable Communication



Shared Variable Communication

- Examples: busy waiting, semaphores and monitors
- Unrestricted use of shared variables is unreliable and unsafe due to multiple update problems
- Consider two processes updating a shared variable, X, with the assignment: X:= X+1
  - load the value of X into some register
  - increment the value in the register by 1 and
  - store the value in the register back to X
- As the three operations are not indivisible, two processes simultaneously updating the variable could follow an interleaving that would produce an incorrect result

Shared Resource Communication

```
type Coordinates is
  record
    X : Integer;
    Y : Integer;
  end record;
Shared_Cordinate: Coordinates;
```

```
task body Helicopter is
  Next: Coordinates;
begin
  loop
   Compute_New_Cordinates(Next);
    Shared_Cordinates  := Next;
  end loop
end;
```

```
task body Police_Car is
begin
  loop
    Plot(Shared_Cordinates);
  end loop;
end;
```

Shared Resource Communication

X = 5

Y = 4

5

4

```
1,1
2,2
3,3
4,4
5,5
6,6
...
```

Villain's Escape

Route

(seen by helicopter)

Villain Escapes!

1,1

2.2

3,3

4,4

4,5

Police Car’s

Pursuit Route

Avoiding Interference

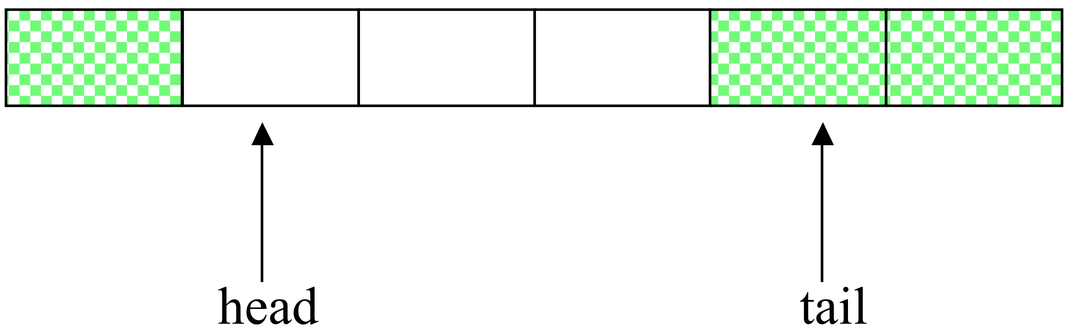
- The parts of a process that access shared variables must be **executed indivisibly** with respect to each other
- These parts are called **critical sections**
- The required protection is called **mutual exclusion**

Mutual Exclusion

- A sequence of statements that must appear to be executed indivisibly is called a critical section
- The synchronisation required to protect a critical section is known as mutual exclusion
- Atomicity is assumed to be present at the memory level. If one process is executing X:= 5, simultaneously with another executing X:= 6, the result will be either 5 or 6 (not some other value)
- If two processes are updating a structured object, this atomicity will only apply at the single word element level

Condition Synchronisation

- Condition synchronisation is needed when a process wishes to perform an operation that can only sensibly, or safely, be performed if another process has itself taken some action or is in some defined state
- E.g. a bounded buffer has 2 condition synchronisation:
  - the producer processes must not attempt to deposit data onto the buffer if the buffer is full
  - the consumer processes cannot be allowed to extract objects from the buffer if the buffer is empty



Is mutual exclusion necessary?

Busy Waiting

- One way to implement synchronisation is to have processes set and check shared variables that are acting as flags (**spinlocks**)
- This approach works well for condition synchronisation but no simple method for mutual exclusion exists
- Some possibilities
  - One flag (fails)
  - Two flags (fails)
  - Peterson’s algoritm

Problems with busy waiting

- Busy wait algorithms are in general inefficient; they involve processes using up processing cycles when they cannot perform useful work
- Even on a multiprocessor system they can give rise to excessive traffic on the memory bus or network (if distributed)
- If not properly done:
  - Can fail to provide mutual exclusion
  - Can produce **livelocks**

Simple algorithm

```
process P1
  loop
   flag1 = up;
    while flag2 = up do
      null
    end;
    <critical section>
    Flag1:= down
    <non-critical section>
  end
end P1;
```

Peterson's algorithm

```
process P1
  loop
    flag1:=up;
    turn:=2;
    while (flag2 = up and turn = 2) do
      null
    end;
    <critical section>
    Flag1:=down
    <non-critical section>
  end
end P1
```

Semaphores

- A semaphore is a non-negative integer variable that apart from initialization can only be acted upon by two procedures P (or WAIT) and V (or SIGNAL)
- WAIT(S)   If the value of S > 0 then decrement its value by one; otherwise delay the process until S > 0 (and then decrement its value).
- SIGNAL(S)   Increment the value of S by one.
- WAIT and SIGNAL are atomic (indivisible). Two processes both executing WAIT operations on the same semaphore cannot interfere with each other and cannot fail during the execution of a semaphore operation

Condition synchronisation

```
var consyn : semaphore (* init 0 *)
```

```
process P1;
   (* waiting process *)
   statement X;
   wait (consyn)
   statement Y;
end P1;
```

```
process P2;
   (* signalling proc *)
   statement A;
   signal (consyn)
   statement B;
end  P2;
```

In what order will the statements execute?

Mutual Exclusion

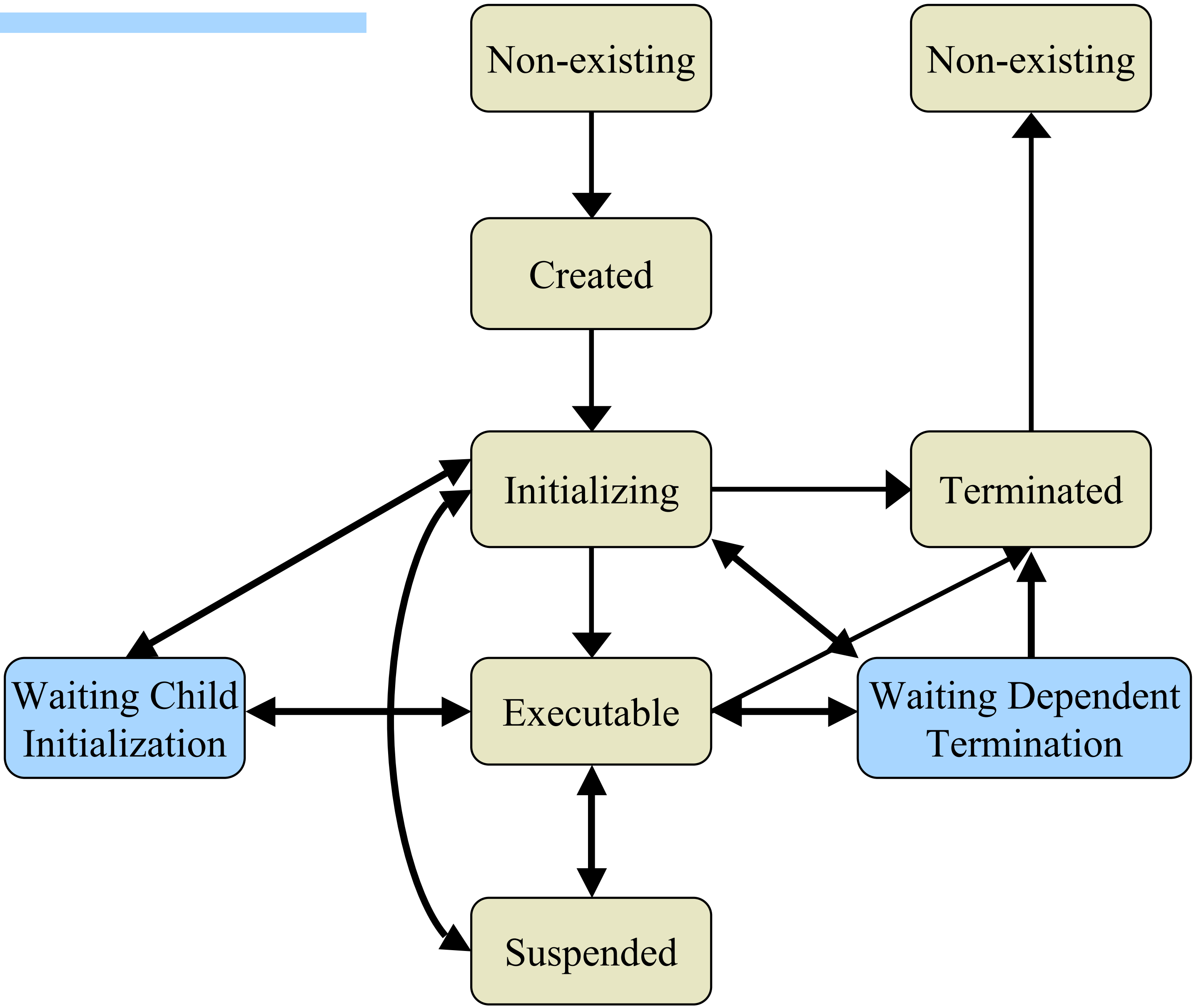
```
(* mutual exclusion *)
var mutex : semaphore; (* initially 1 *)
```

```
process P1;
  statement X
  wait (mutex);
    statement Y
  signal (mutex);
  statement Z
end P1;
```

```
process P2;
  statement A;
  wait (mutex);
    statement B;
  signal (mutex);
  statement C;
end P2;
```

In what order will the statements execute?

Process States



Deadlock

- Two processes are deadlocked if each is holding a resource while waiting for a resource held by the other

```
type Sem is ...;
X : Sem := 1; Y : Sem := 1;
```

```
task A;
task body A is
begin
      ...
      Wait(X);
      Wait(Y);
      ...
end A;
```

```
task B;
task body B is
begin
      ...
      Wait(Y);
      Wait(X);
      ...
end B;
```

Livelock

- Two processes are livelocked if each is executing but neither is able to make progress.

```
type Flag is (Up, Down);
Flag1 : Flag := Up;
```

```
task A;
task body A is
begin
   ...
   while Flag1 = Up loop
     null;
   end loop;
   ...
end A;
```

```
task B;
task body B is
begin
   ...
   while Flag1 = Up loop
     null;
   end loop;
   ...
end A;
```

Starvation

- **Indefinite postponement** (also called starvation or lockout) happens when a set of processes does not have livelocks nor deadlocks but there are processes that never gain access to some resources (typically due to scarcity and priority policies)
- This is not a very hard problem (adding more resources solves the problem)

Liveness

- A system is said to have the **liveness** property if it does not have deadlocks, livelocks nor lockouts
- In a system that possess liveness, any process that wants to perform some action will eventually perform it
- In particular, access to any critical section is guaranteed in finite time

Binary and quantity semaphores

- A general semaphore is a non-negative integer; its value can rise to any supported positive number
- A binary semaphore only takes the value 0 and 1; the signalling of a semaphore which has the value 1 has no effect - the semaphore retains the value 1
- A general semaphore can be implemented by two binary semaphores and an integer. Try it!
- With a quantity semaphore the amount to be decremented by WAIT (and incremented by SIGNAL) is given as a parameter; e.g. WAIT (S, i)

Criticisms of semaphores

- Semaphore are an elegant low-level synchronisation primitive, however, their use is error-prone
- If a semaphore is omitted or misplaced, the entire program is in way to collapse. Mutual exclusion may not be assured and deadlocks may appear just when the software is dealing with a rare but critical event
- A more structured synchronisation primitive is required
- No high-level concurrent programming language relies entirely on semaphores; they are important historically but are arguably not adequate for the real-time domain

Monitors

- A more sophisticated coordination structure are **monitors**
- Monitors provide encapsulation, and efficient condition synchronisation by **condition variables**
- The critical sections are written as procedures and are encapsulated together into a single module
- All variables that must be accessed under mutual exclusion are hidden inside the module
- All procedure calls into the module are guaranteed to be mutually exclusive
- Only the operations are visible outside the monitor

POSIX Semaphores and Mutexes

- Provide the equivalent of a semaphores and monitors for communication and synchronisation between processes/threads
- Mutexes and condition variables have associated attribute objects
- Example attributes:
  - allow sharing of mutexes and condition variables between processes
  - set/get priority ceiling
  - set/get the clock used for timeouts

```
typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;
```

```c
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
  /* initialises a mutex with certain attributes */

int pthread_mutex_destroy(pthread_mutex_t *mutex);
  /* destroys a mutex */
  /* undefined  behaviour if the mutex is locked  */


int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
  /* initialises a condition variable with attributes */

int pthread_cond_destroy(pthread_cond_t *cond);
  /* destroys a condition variable */
  /* undefined, if threads are waiting on the variable  */
```

```c
int pthread_mutex_lock(pthread_mutex_t *mutex);
  /* lock the mutex; if locked already suspend calling thread */
  /* the owner of the mutex is the thread which locked it */

int pthread_mutex_trylock(pthread_mutex_t *mutex);
  /* as lock but gives an error if mutex is already locked */

int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            const struct timespec *abstime);
  /* as lock but gives an error if mutex cannot be obtained */
  /* by the timeout  */


int pthread_mutex_unlock(pthread_mutex_t *mutex);
  /* unlocks the mutex if called by the owning thread */
  /* undefined behaviour if calling thread is not the owner  */
  /* undefined behaviour if the mutex is not locked } */
  /* when successful, a blocked thread is released */
```

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
  /* called by thread which owns a locked mutex */
  /* undefined behaviour if the mutex is not locked */
  /* atomically blocks the caller on the cond variable and */
  /* releases the lock on mutex */
  /* a successful return indicates the mutex has been locked */

int pthread_cond_timedwait(pthread_cond_t *cond,
       pthread_mutex_t *mutex, const struct timespec *abstime);
  /* the same as pthread_cond_wait, except that a error is */
 /* returned if the timeout expires */
```

```
int pthread_cond_signal(pthread_cond_t *cond);
  /* unblocks at least one blocked thread */
  /* no effect if no threads are blocked */


int pthread_cond_broadcast(pthread_cond_t *cond);
  /* unblocks all blocked threads */
  /* no effect if no threads are blocked */


  /*all  unblocked threads automatically contend for */
  /* the associated mutex */
```

All functions return 0 if successful

(as usual in C/POSIX)

Criticisms of Monitors

- The monitor gives a structured and elegant solution to mutual exclusion problems such as the bounded buffer
- It does not, however, deal well with condition synchronization — requiring low-level condition variables
- All the criticisms surrounding the use of semaphores apply equally to condition variables

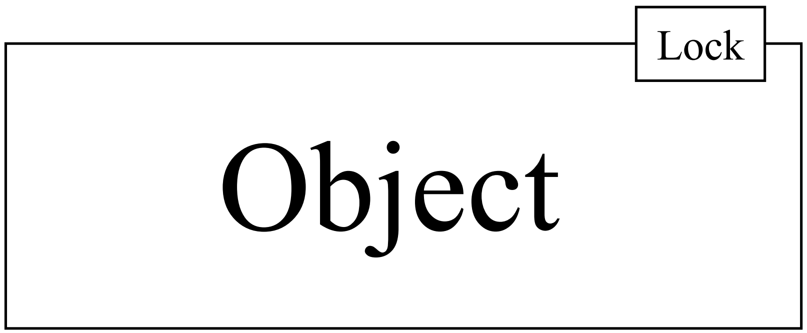
Ada Protected Objects

- Mechanism for monitor implementation in Ada
- Data and operations are encapsulated
- Operations have automatic mutual exclusion
- Guards can be placed on operations for condition synchronization

Synchronized Methods ad Blocks

- **Java** provides a mechanism by which monitors can be implemented in the context of classes and objects
- There is a lock associated with each object which cannot be accessed directly by the application

- There are two mechanisms to use the lock by means of the word **`synchronized`** :
  - as method modifier
  - in block synchronization



Synchronized methods

- When a method is labeled with the **`synchronized`** modifier, access to the method can only proceed once the lock associated with the object has been obtained
- Hence synchronized methods have mutually exclusive access to the data encapsulated by the object, if that data is only accessed by other synchronized methods
- Non-synchronized methods do not require the lock and, therefore, can be called at any time

Example

```java
public class SharedInteger
{
  private int theData;

  public SharedInteger(int initialValue)
  { theData = initialValue;}

  public synchronized int read()
  { return theData; };

  public synchronized void write(int newValue)
  { theData = newValue; };

  public synchronized void incrementBy(int by)
  { theData = theData + by };
}

SharedInteger myData = new SharedInteger(42);
```

Block Synchronization

- Provides the second mechanism for synchronization
- Any block can be labeled as synchronized

```
syncronyzed(HeliOne) {
    motor.start();
    radio.start();
}
```

- The synchronized keyword takes as a parameter an object whose lock it needs to obtain before it can continue

Identity of mechanisms

- Hence synchronized methods are effectively implementable as:

```
public int read()
  {
    synchronized(this) {
      return theData;
    }
  }
```

- Where **`this`** is the Java mechanism for obtaining the current object

Warning

- Used in its full generality, the synchronized block can undermine one of the advantages of monitor-like mechanisms, that of encapsulating synchronization constraints associate with an object into a single place in the program
- This is because it is not possible to understand the synchronization associated with a particular object by just looking at the object itself when other objects can name that object in a synchronized statement.
- However with careful use, this facility augments the basic model and allows more expressive synchronization constraints to be programmed

Waiting and Notifying

- To obtain conditional synchronization requires the methods provided in the predefined object class:

```
public void wait() throws InterruptedException;
           // also throws IllegalMonitorStateException
public void notify();
           // throws IllegalMonitorStateException
public void notifyAll();
          // throws IllegalMonitorStateException
```

- These methods should be used only from within methods which hold the object lock
- If called without the lock, the exception `IllegalMonitor-StateException` is thrown

Waiting and Notifying

- The `wait` method always blocks the calling thread and releases the lock associated with the object
- The `notify` method wakes up one waiting thread; the one woken is not defined by the Java language
- `Notify` does not release the lock; hence the woken thread must wait until it can obtain the lock before proceeding
- To wake up all waiting threads requires use of the `notifyAll` method
- If no thread is waiting, then `notify` and `notifyAll` have no effect

Thread Interruption

- A waiting thread can also be awoken if it is interrupted by another thread
- In this case the InterruptedException is thrown

Summary

- critical section — code that must be executed under mutual exclusion
- producer-consumer system — two or more processes exchanging data via a finite buffer
- busy waiting — a process continually checking a condition to see if it is now able to proceed
- livelock — an error condition in which one or more processes are prohibited from progressing whilst using up processing cycles
- deadlock — a collection of suspended processes that cannot proceed
- starvation — a process being unable to proceed as resources are not made available

Summary

- semaphore — a non-negative integer that can only be acted upon by WAIT and SIGNAL atomic procedures
- A more structured primitive are monitors
- Suspension in a monitor is achieved using condition variable
- POSIX mutexes and condition variables give monitors with a procedural interface
- Ada's protected objects give structured mutual exclusion and high-level synchronization via barriers
- Java's synchronized methods provide monitors within an object-oriented framework