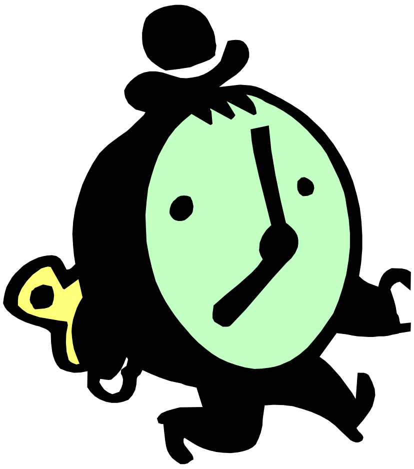
Real-Time Systems Programming

Summer-Semester 2002
Lecture 19
20 June 2002



Synchronization and
Communication
Part I

The 5 Minute Review Session

1) *What is concurrency?*
2) *Why concurrency?*
3) *How can we do „multiple things at the same time“? (Or at least pretend to do so ...)*
4) *What is a cyclic executive? What are the advantages and disadvantages?*
5) *What are the aspects of a concurrent process model?*

Overview

1) Coordination = communication + synchronization
2) Semaphores
3) Conditional critical regions
4) Monitors

These lecture notes are based on slides kindly provided by Burns and Wellings

Where are we?

1) ***Coordination = communication + synchronization***
   - ***Mutual exclusion and condition synchronization***
   - ***Busy waiting***
   - ***Suspend and resume***
2) Semaphores
3) Conditional critical regions
4) Monitors

These lecture notes are based on slides kindly provided by Burns and Wellings

Synchronisation and Communication

- ***Synchronisation***:
  - Satisfies constraints on interleaving of actions of processes
  - E.g. action by process A occurs after action by process B
- ***Communication***:
  - Passing of information from one process to another
  - Usually based upon either shared variables or message passing
- Concepts are ***linked***:
  - Communication requires synchronisation
  - Synchronisation = contentless communication
- Synchronization and communication are ***essential for correct behavior*** of a concurrent program

Coordination

- Coordination mechanisms in general:
  - Message Passing
  - Shared Memory
  - Semaphores (binary and counting)
  - Mutexes and Condition Variables
  - Readers/Writers Locks
  - Tasking and Rendezvous
  - Event Flags

Shared Variable Communication

- ***Examples***:
  - Busy waiting
  - Semaphores
  - 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

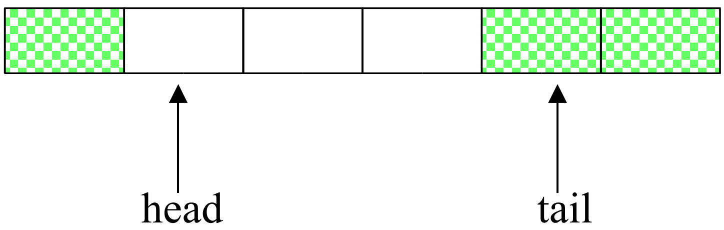
Mutual Exclusion

- ***Critical section***:
  - Sequence of statements that must appear to be executed indivisibly
- ***Mutual exclusion***:
  - The synchronisation required to protect a critical section (Dijkstra 1965)
- ***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***
  - Process wants to perform operation *A*
  - *A* is safe/sensible only if another process has taken some other action *B*
- ***Example***: ***bounded buffer***
  - ***Producer*** processes must block if buffer full
  - ***Consumer*** processes must block if buffer empty



Is mutual exclusion necessary?

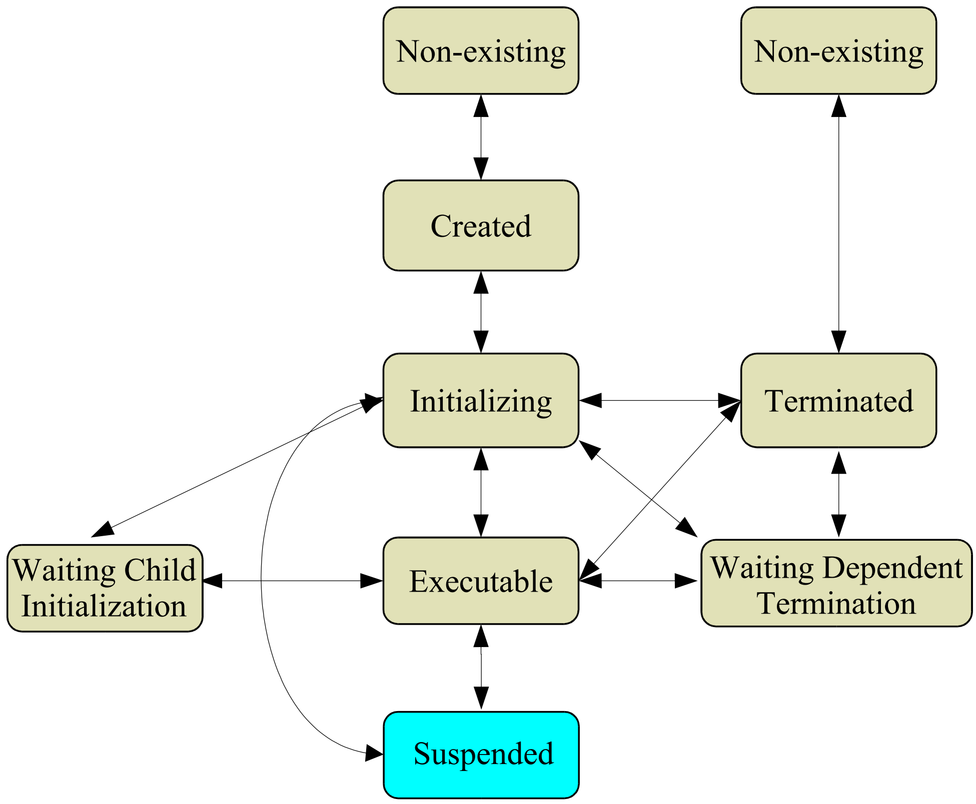
Busy Waiting

- For synchronisation, processes may ***set and check shared variables*** that are acting as flags (***spin-locks***)
- Works well for ***condition synchronisation***
- However:
  - No simple method for ***mutual exclusion***
  - Queuing discipline (***fairness***) difficult to ensure
  - ***Correctness*** difficult to prove
  - Misuse of shared variables by rogue tasks may ***corrupt*** entire system

Suspend and Resume

- Busy wait algorithms are in general ***inefficient***
  - Processes use processing cycles when they cannot perform useful work
  - On multiprocessor systems, they can give rise to excessive traffic on the memory bus or network
- ***Alternative***:
  - Remove a process from set of runnable processes if the condition for which it is waiting does not hold (process ***suspension***)

Process States



Java's **suspend()** *and* **resume()**

```
boolean flag;
final boolean up = true;
final boolean down = false;

class FirstT extends Thread {
  public void run() {
    ...
    if (flag == down) {
      suspend();
    };
    flag = down;
    ...
  }
}
```

```
class SecondT extends Thread {
  FirstT T1;

  public SecondT(FstT T) {
    super();
    T1 = T;
  }

  public void run() {
  ...
  flag = up;
  T1.resume();
  ...
  }
}
```

- ***The problem:*** testing and suspension are not atomic
  - ***Race condition*** may occur
- Java has therefore made these methods ***obsolete***

Safe Suspension

- Solutions to race condition problem use a two-stage suspend operation:
  - P1 announces intent to suspend
  - Until suspension of P1, resume operation will be deferred
- *Ada* provides safe version as part of Real-Time Annex

```
with Ada.Synchronous_TaskControl;
use Ada.Synchronous_TaskControl;
...
Flag: Suspension_Object;
...
task body P1 is
begin
  ...
  Suspend_Until_True(Flag);
  ...
end P1;
```

```
task body P2 is
begin
  ...
  Set_True(Flag);
  ...
end P2;
```

Where are we?

1) Coordination = communication + synchronization
2) ***Semaphores***
   - ***Review of operation***
   - ***Ada, POSIX, LegOS***
   - ***Criticisms***
3) Conditional critical regions
4) Monitors

These lecture notes are based on slides kindly provided by Burns and Wellings

Semaphores

- Operations on ***Semaphores***
  - **INIT**(*S*, *Value*)
    - Initialize *S* to *Value*
  - **WAIT**(*S*), or **P**(*S*):
    - ***If S > 0:***
      - Decrement *S* by 1
    - ***Otherwise:***
      - Delay process until $S > 0$
      - Then decrement *S* by 1
  - **SIGNAL**(*S*), or **V**(*S*):
    - Increment *S* by 1

Concurrency and Semaphores

- All semaphore operations are ***atomic***
- Two processes executing P or V operations on the same semaphore:
  - Cannot interfere with each other
  - Cannot fail during 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 Y1
      statement Y2
   signal (mutex)
   statement Z
end P1
```

```
process P2
   statement A
   wait (mutex)
      statement B1
      statement B2
   signal (mutex)
   statement C
end P2
```

In what order will the statements execute ?

Bounded Buffer with Semaphores

```
sem_init(sem-free, MAX);
sem_init(sem-avail, 0);
sem_init(sem-mutex, 1);
in = out = 0;
```

```
Producer() {
  while (1) {
    item = produce();
    wait(sem-free);
    wait(sem-mutex);
    buffer[in] = item;
    in = (in + 1) % MAX;
    signal(sem-mutex);
    signal(sem-avail);
  }
}
```

```
Consumer() {
  while (1) {
    wait(sem-avail);
    wait(sem-mutex);
    item = buffer[out];
    out = (out + 1) % MAX;
    signal(sem-mutex);
    signal(sem-free);
    consume(item);
  }
}
```

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;
```

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 (⇨ *Homework*)
- 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)

Example semaphore programs in Ada

- ***Recall:*** *the essence of abstract data types is that they can be used without knowledge of their implementation*

```
package Semaphore_Package is
  type Semaphore(Initial : Natural) is limited private;
  procedure Wait (S : Semaphore);
  procedure signal (S : Semaphore);
private
  type Semaphore ...
end Semaphore_Package;
```

- Ada does not directly support semaphores
  - But can construct wait and signal procedures from Ada synchronisation primitives

The Bounded Buffer in Ada

```
package Buffer is
  procedure Append (I : Integer);
  procedure Take (I : out Integer);
end Buffer;

package body Buffer is
  Size : constant Natural := 32;
  type Buffer_Range is mod Size;
  Buf : array (Buffer_Range) of Integer;
  Top, Base : Buffer_Range := 0;
  Mutex : Semaphore(1);
  Item_Available : Semaphore(0);
  Space_Available : Semaphore(Size);

  procedure Append (I : Integer) is separate;
  procedure Take (I : out Integer) is separate;
end Buffer;
```

The Bounded Buffer in Ada cont.

```
procedure Append(I : Integer) is
begin
  Wait(Space_Available);
  Wait(Mutex);
    Buf(Top) := I;
    Top := Top+1
  Signal(Mutex);
  Signal(Item_Available);
end Append;
```

```
procedure Take(I : out Integer) is
begin
  Wait(Item_Available);
  Wait(Mutex);
    I := BUF(base);
    Base := Base+1;
  Signal(Mutex);
  Signal(Space_Available);
end Take;
```

Semaphores in C/POSIX

- Few modern programming languages support semaphores directly – but many OSs do
- ***POSIX*** provides ***counting semaphores*** for communication between processes or threads

```
#include <time.h> typedef ... sem_t;

int sem_init(sem_t *sem, int pshared, unsigned int value)
int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abstime);

int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *value);
```


`pshared` is 1 *iff* the semaphore can be used between processes; otherwise, can only be used between threads of the same process

legOS Counting Semaphores

- Are analogous to POSIX counting semaphores:

```
// The pshared argument is there only for
// backwards-compatibility and can be ignored
int sem_init(sem_t *sem, int pshared, unsigned int value);

int sem_wait(sem_t *sem);

int sem_trywait(sem_t *sem);

int sem_post(sem_t *sem);
```

Criticisms of semaphores

- Semaphores are an ***elegant*** low-level synchronisation primitive (and historically important)
- However, their use is ***error-prone***
  - If a semaphore is omitted or misplaced, the entire program may collapse
  - ***Mutual exclusion*** may not be assured and ***deadlock*** may appear just when the software is dealing with a rare but critical event
- A ***more structured*** synchronisation primitive is required for the RT domain
- No high-level concurrent programming language relies entirely on semaphores

Where are we?

1) Coordination = communication + synchronization
2) Semaphores
3) ***Conditional critical regions***
4) Monitors

These lecture notes are based on slides kindly provided by Burns and Wellings

Conditional Critical Regions (CCR)

- ***Critical region:***
  - A section of code that is guaranteed to be executed in mutual exclusion
- ***Shared variables*** are grouped together into named regions and are tagged as being ***resources***
- Processes are prohibited from entering a region in which another process is already active
- Condition synchronisation is provided by ***guards***
  - When a process wishes to enter a critical region it evaluates the guard (under mutual exclusion)
  - if the guard evaluates true it may enter
  - if it is false the process is delayed
- As with semaphores, ***no guarenteed access order***

The Bounded Buffer I

```
program buffer_eg;
  type buffer_t is record
    slots      : array(1..N) of character;
    size       : integer range 0..N;
    head, tail : integer range 1..N;
  end record;

  buffer : buffer_t;
  resource buf : buffer;

  process producer is separate;
  process consumer is separate;
end.
```

The Bounded Buffer II

```
process producer;
  loop
    region buf when buffer.size < N do
      -- place char in buffer etc
    end region
  end loop;
end producer

process consumer;
  loop
    region buf when buffer.size > 0 do
      -- take char from buffer etc
    end region
  end loop;
end consumer
```

A Problem

- A version of CCRs has been implemented in ***Edison***
- One problem with CCRs:
  - Processes must re-evaluate their guards every time a CCR naming that resource is left
  - A suspended process must become executable again in order to test the guard
    - If guard is still false, process must return to the suspended state

- ***Edison*** is a language intended for embedded applications, implemented on multiprocessor systems
  - Each processor only executes a single process so it may continually evaluate its guards if necessary

Where are we?

1) Coordination = communication + synchronization
2) Semaphores
3) Conditional critical regions
4) ***Monitors***
   - ***Condition variables (WAIT + SIGNAL)***
   - ***POSIX mutexes and condition variables***
   - ***Nested monitor calls***

These lecture notes are based on slides kindly provided by Burns and Wellings

Monitors

- Another problem with CCRs:
  - Can be dispersed throughout the program
- ***Monitors*** provide ***encapsulation***, and ***efficient condition synchronisation***
- The critical regions are written as procedures and are encapsulated together into a single module:
  - All variables that must be accessed under mutual exclusion are hidden
  - All procedure calls into the module are guaranteed to be mutually exclusive
  - Only the operations are visible outside the monitor
- Monitors have been implemented in ***Modula-1*** and ***Concurrent Pascal***

The Bounded Buffer I

```
 monitor buffer;
   export append, take;
    var (*declare necessary vars*)

    procedure append (I : integer);
      ...
    end;

    procedure take (var I : integer);
      ...
    end;
begin
  (* initialisation *)
 end;
```

How do we get condition synchronisation?

Condition Variables

- Different semantics exist
- In Hoare's monitors:
  - A condition variable is acted upon by two semaphore-like operators **WAIT** and **SIGNAL**
- When a process issues a WAIT:
  - Process is blocked (suspended) and placed on a queue associated with the condition variable
  - ***Note***: a wait on a condition variable always blocks unlike a wait on a semaphore
- A blocked process releases its hold on the monitor
  - Allows another process to enter
- A SIGNAL releases one blocked process
  - If no process is blocked then the signal has ***no effect***

- Note that a signal on a semaphore *always* has an effect on the semaphore
- The semantics of *wait* and *signal* is more aking to *suspend* and *resume*

The Bounded Buffer II

```
monitor buffer;
  export append, take;

  var BUF : array[ . . . ] of integer;
  top, base : 0..size-1;  NumberInBuffer : integer;

  spaceavailable, itemavailable : condition;

  procedure append (I : integer);
  begin
    if NumberInBuffer = size then
      wait(spaceavailable);
    end if;
    BUF[top] := I;
    NumberInBuffer := NumberInBuffer+1;
    top := (top+1) mod size;
    signal(itemavailable)
  end append;
```

The Bounded Buffer III

```
  procedure take (var I : integer);
  begin
    if NumberInBuffer = 0 then
      wait(itemavailable);
    end if;
    I := BUF[base];
    base := (base+1) mod size;
    NumberInBuffer := NumberInBuffer-1;
    signal(spaceavailable);
  end take;

begin (* initialisation *)
  NumberInBuffer := 0;
  top := 0; base := 0
end;
```

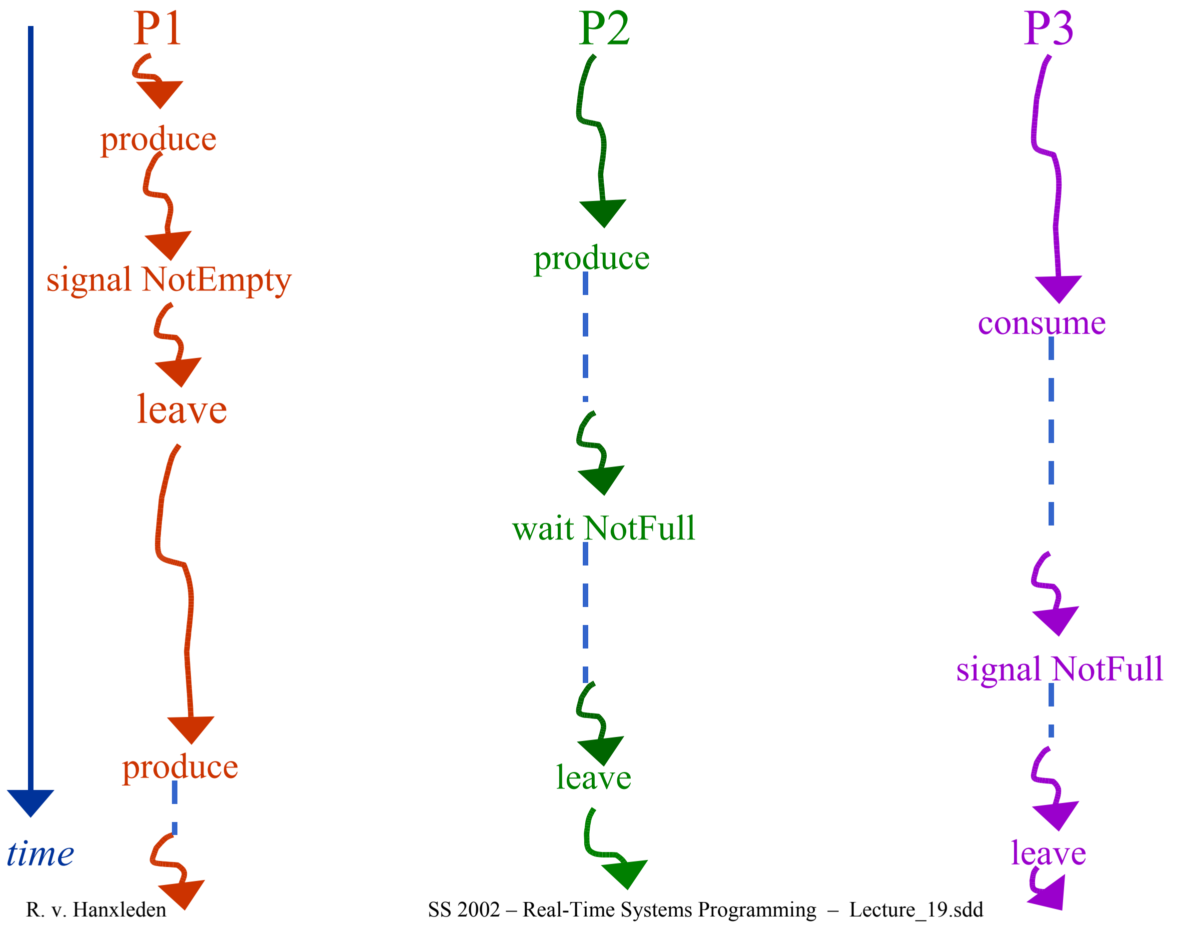
If a process calls **`take`** *when there is nothing in the buffer then it will become* ***suspended*** *on* **`itemavailable`***.*

A process appending an item will, however, ***signal*** *this suspended process when an item does become available.*

The Semantics of SIGNAL

- How to assure ***mutual exclusion*** between the signalling process and the process that is restarted?
- Different options:
  1) A signal is allowed only as the ***last action*** of a process before it leaves the monitor
  2) A signal operation has the side-effect of executing a ***return*** statement, i.e. the process is forced to leave
  3) A signal operation which unblocks another process has the effect of ***blocking itself***; this process will only execute again when the monitor is free (Hoare 1974)
  4) A signal operation which unblocks a process ***does not block*** the caller. The unblocked process must gain access to the monitor again

SIGNAL – Example



POSIX Mutexes and Condition Variables

- POSIX Mutexes and Condition Variables:
  - Equivalent to ***monitor*** for communication and synchronisation ***between threads***
  - Provide functionality of monitor, with procedural interface
- Require ***same address space***
  - Not applicable across process boundaries
- Are a more ***structured*** alternative to semaphores

POSIX Mutexes and Condition Variables

- Mutexes and condition variables have associated attribute objects
- ***Example attributes:***
  - set the semantics for a thread trying to lock a mutex that it already has locked
  - 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;
```


Here we will use default attributes only

POSIX Interface I

```
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 certain attributes        */

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

POSIX Interface II

```
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 */
```

POSIX Interface III

```
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 */
```

POSIX Interface IV

```
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

POSIX Bounded Buffer I

```
#define BUFF_SIZE 10

typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t buffer_not_full;
  pthread_cond_t buffer_not_empty;
  int count, first, last;
  int buf[BUFF_SIZE];
} buffer;


int append(int item, buffer *B ) {
  PTHREAD_MUTEX_LOCK(&B->mutex);

  while(B->count == BUFF_SIZE) {
    PTHREAD_COND_WAIT(&B->buffer_not_full, &B->mutex);
  }

  /* put data in the buffer and update count and last */
  PTHREAD_MUTEX_UNLOCK(&B->mutex);
  PTHREAD_COND_SIGNAL(&B->buffer_not_empty);
  return 0;
}
```

POSIX Bounded Buffer II

```
int take(int *item, buffer *B ) {
  PTHREAD_MUTEX_LOCK(&B->mutex);

  while(B->count == 0) {
    PTHREAD_COND_WAIT(&B->buffer_not_empty, &B->mutex);
  }

  /* get data from the buffer and update count and first */
  PTHREAD_MUTEX_UNLOCK(&B->mutex);
  PTHREAD_COND_SIGNAL(&B->buffer_not_full);
  return 0;
}

int initialize(buffer *B) {
  /* set the attribute objects and initialize the */
  /* mutexes and condition variable */
}
```

Nested Monitor Calls

- What to do if a process having made a nested monitor call is suspended in another monitor?
  - The mutual exclusion in the last monitor call will be relinquished by the process (semantics of wait)
  - However, mutual **exclusion** will not be relinquished by processes in monitors from which the nested calls have been made; processes that attempt to invoke procedures in these monitors will become ***blocked***
- ***Approaches:***
  - Maintain the lock: e.g. ***POSIX*, *Java***
  - Prohibit nested procedure calls altogether: e.g. ***Modula-1***
  - Provide constructs to let a monitor procedure release its mutual exclusion lock during remote calls

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

Summary I

- ***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
- ***Indefinite postponement*** — a process being unable to proceed as resources are not made available

Summary II

- ***Semaphore*** — a non-negative integer that can only be acted upon by ***WAIT*** and ***SIGNAL*** atomic procedures
- Two more structured primitives are:
  - ***Conditional critical regions***
  - ***Monitors***
- Suspension in a monitor is achieved using ***condition variable***

To Go Further

- ☞ [Burns and Wellings 2001] – Chapter 8
- ☞ [Gallmeister 1995] - Chapter 4