

Events and Conditions

- Two further important **synchronisation primitives** in concurrent systems are the *event* and the *condition*.

Events

- An **event** is defined as the **brief**, or **pulsed occurrence** of anything which is **significant** to a system and may affect its **behaviour** in some way.
- As an example, consider what happens when we, as pedestrians, make use of a **pedestrian crossing** when attempting to cross a busy main road such as Broadway.
- When we press the **button** on the pedestrian crossing, this represents the brief occurrence of an *event* which is **significant** to the **PLC** or **computer** that is controlling the lights. In response to this event, the controller alter the traffic lights to allow pedestrians to cross safely.
- In effect the pedestrian control system is designed to **WAIT** or **synchronise** itself to the **occurrence** of such an event.

- For example, the main control loop within the pedestrian crossing controller might have been written using pseudocode like this

```
Do_Forever()  {  
    Wait for pedestrian to press button // wait or synchronise to an event  
    Turn Vehicle Traffic lights to Red  
    Turn Pedestrian lights to Go  
    Turn Pedestrian lights to Red  
    Turn Vehicle Traffic lights to Go  
}
```

- Here the system controller is implemented as a **continuous loop** that alternates between giving priority to traffic or to pedestrians.
- What **stops it continuously cycling through this loop** is the action of **waiting for the button** to be pressed by a pedestrian.
- In effect the process or system is **synchronising** itself **to**, or **waiting** for the **occurrence** of this event.
- If no pedestrians are waiting then the button will never be pressed and thus priority will always given to traffic. Furthermore if several pedestrians are waiting and all of them press the button randomly, the event will be **not queued** up by the system, thus only one event per cycle of the loop is permitted.
- Other events will be **lost** if the system is not actively **waiting** for the event as an event is brief and **disappears immediately** it has been signalled.
- In effect an event can be thought of as a **transitory occurrence** of **pulse** of **infinitely short period** which is then **reset automatically**, so that only those processes/threads that are **actively waiting** for it when it occurs can synchronise themselves to it. If they arrive **after** the event occurs then they will have to wait until its next occurrence.

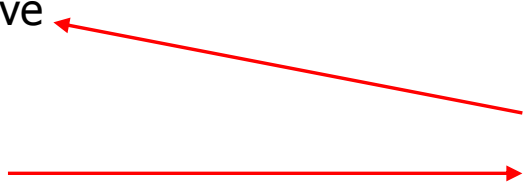
- As another example consider a manufacturing facility which is assembling Cars on a production line.
- Here, a **robot** might actively be **waiting** for a **body shell** to come along a **conveyor belt** before it can assemble say the engine. In essence the **robot** is synchronising itself to an occurrence of a specific **event**, namely **the arrival of the chassis**.
- The **production line** could then synchronise itself to another event namely that **the robot has completed it's assembly**.
- This could trigger the production line to move on and fetch the next chassis. The pseudo code might look something like this.

Robot

```
Do Forever {  
    Wait for Chassis to arrive  
    Assembling car  
    .....  
    Signal Robot finished  
}
```

Production Line

```
Do Forever {  
    Move chassis into position  
    .....  
    Signal Chassis has arrived  
    Wait for Robot to finish  
}
```



- Be careful when using events in a handshaking arrangement like this as it is possible for *races* to occur between threads, where one thread may *not* yet be waiting on the event when it is signalled by the other, resulting in a lost event (they are not queued up and stored) and a *deadlocked system* with each thread is still waiting for the other.
- Thus it may be more desirable to use semaphores like a producer/consumer arrangement.

Implementing Event Synchronisation under Win32

- Events come in two flavours in Win32, namely
 - **Single** thread release events and
 - **Multiple** thread release events.
- Both types of event are encapsulated by the class 'CEvent'.
- A named event may be **created** and threads/processes **may chose to wait** for the event to be '**signalled**' before they are released to continue processing.
- After an event is signalled both types are **Automatically Reset** to the **non-signalled (or blocking) state**.
- By default both types of event are created in the **non-signalled**, (or blocking) state, meaning that processes/threads waiting on the event will be blocked and will have to wait until it is explicitly signalled by a process/thread, although you can create events in the **signalled** state if you want (see rt.h and rt.cpp for examples).
- Basically the CEvent class provides functionality to
 - **Create** a Event.
 - **Wait** for an Event to be signalled, i.e. to become true.
 - **Signal** an Event.
- Events are demonstrated in Tutorial Q6

Single Thread Release Events

- With **Single thread** release events, when the event occurs, **at most one waiting thread will be released**.
- If there is more than one thread waiting for the event, then only the **first in the queue will be released**,
- All other waiting events will have to wait for **subsequent signalling** of the event.
- If **no threads are waiting** when the event is signalled, then the event will still be reset back to the **non-signalled state**, even though no threads were released, so it is possible for an event to be missed by a thread/process due to race and timing problems between threads.

Multiple Thread Release Events

- With multiple thread release events, when the event is signalled, **ALL waiting threads are released** before it is reset back to the blocking, **non-signalled state**.
- If there are **no waiting threads**, then the event is still reset back to the **non-signalled state** blocking those threads that arrive after it was signalled.

Example Production Line code

```
CEvent ChassisReady("ChassisInPosition", SINGLE_RELEASE); // create a non-signalled event
CEvent RobotFinished("RobotDone", SINGLE_RELEASE); // create a non-signalled event

UINT __stdcall ProductionLine(void *args) // thread to represent production line
{
    while(1) { // do forever
        SLEEP(5000); // simulate time delay due to chassis moving into position
        printf("Production Line: New Chassis Arrived\n");
        ChassisReady.Signal(); // signal event occurrence
        RobotFinished.Wait(); // wait for robot to assemble car
    }
    return 0;
}

UINT __stdcall Robot(void *args)
{
    while(1) { // do forever
        ChassisReady.Wait(); // wait for next chassis to arrive
        printf("Robot: Assembling Chassis\n");
        SLEEP(5000); // simulate time delay due to assembling
        RobotFinished.Signal(); // signal robot has assembled car
    }
    return 0;
}

int main()
{
    CThread TheRobot(Robot); // create an active child thread
    CThread TheProductionLine(ProductionLine); // create an active child thread
    ...
    ...
    return 0;
}
```

Conditions

- A **condition** object is a kind of **'event'** object with slightly different behaviour.
- You can think of a **condition** as being in one of two states, **true** or **false**, (**signalled** or **non-signalled** if you prefer).
- Unlike an Event, which is **brief** or **transitory** by nature (i.e. an event occurs and resets itself in **zero time**), a **condition exists** for a **finite** or **measurable period** of time.
- In addition, whereas an event will automatically **reset** itself to the non-signalled state after each occurrence of its **Signal()** operation, a condition will remain in the signalled state until it is **manually reset** under software control.
- Conditions have been encapsulated into the **CCondition** class in the **rt.cpp** library.
- Basically the **CCondition** class provides functionality to
 - **Create** a Condition.
 - **Wait** for a Condition to be signalled, i.e. wait for condition to become true.
 - **Test** if a Condition is signalled, or true.
 - **Signal** a Condition is true, i.e. non-blocking state.
 - **Reset** a Condition to false, i.e. the blocking state.
- Conditions are demonstrated in Tutorial Q6

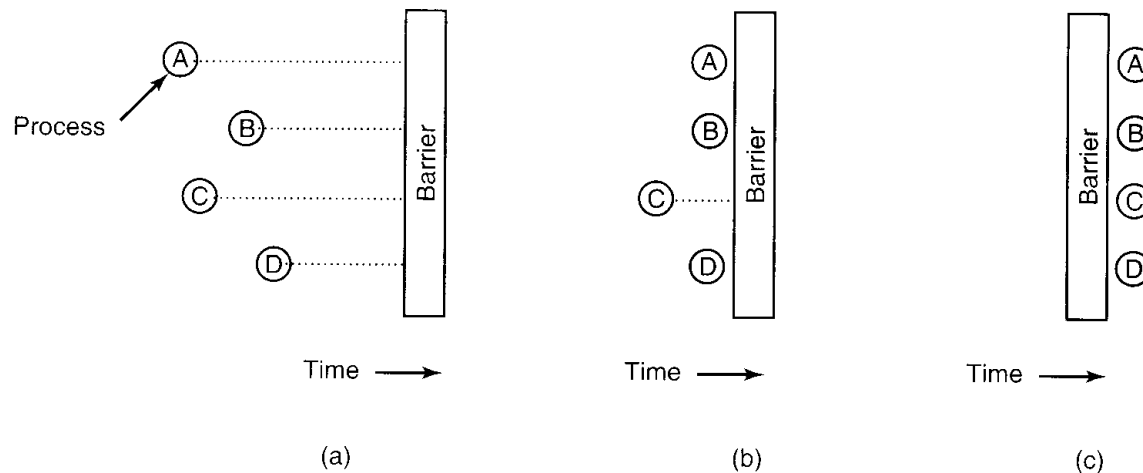
Conditions and their Applications

- To explain the application of conditions, it is best to use an analogy, so let's go back to the **pedestrian crossing** scenario we discussed earlier when we looked at events.
- As pedestrians, when we wish to cross a road, we must **wait** or **synchronise** ourselves to the **Go/Stop** signs facing us.
- If we did not, then simply marching straight into the road would have likely dire consequences to our health and well being
- When pedestrians arrive at a crossing, they can **all cross** as long as the condition '**Go**' is **true** or must **all wait** if it is **false**.
- As pedestrians therefore we are programmed (if only by *natural selection*) to **wait** for some **condition** to occur before proceeding to cross the road, namely that the **Go** signal is illuminated indicating that it is now safe to cross the road.
- It's the same situation with cars and trucks. They can all proceed through an intersection if their light is showing **green**, but must wait if the light is showing **red**

[Question: How is a condition different to a multiple release event ?]

The Rendezvous or Barrier

- Another common synchronisation problem in concurrent system involves the use of a **rendezvous**, or **barrier** as it is sometimes called.
- Here, several **process/threads** must synchronise themselves **to each other** at a point in time where they all agree to meet. That is **all participating threads** are designed to **wait for the arrival of all other threads** before **all of them** proceed together.
- Unlike an event, no one single thread is responsible for detecting the arrival of all others or signalling/releasing them. The simple act of all participating processes/threads **turning up at the rendezvous** and **waiting** is sufficient to trigger it and allow **them all to continue**.
- The diagram below illustrates the idea. Here in figure (a), we see four threads **A, B, C** and **D** all participating in the rendezvous. They arrive at the rendezvous in their own time. In figure (b) we see that **A, B** and **D** have already arrived and are already waiting when **C** arrives.
- As all participating threads have now arrived, the rendezvous is triggered and all four threads resume.



- As a practical example of the application of a rendezvous, imagine these 4 processes represent 3 robots and 1 conveyor belt controller involved in assembling a car on a production line.
- Obviously we must arrange for the 3 robots and the conveyor belt to wait or synchronise themselves with each other after each has finished its respective operation.
 - We wouldn't want a robot to try and assemble the same car twice, for example, just because it had finished and had nothing else to do.
 - Neither would we want the conveyor belt to try to move the car along the production line until the 3 robots had finished assembling it.
- To solve such a problem, we would arrange for all 4 processes to synchronise themselves with a rendezvous.
- When the conveyor belt deposits a new car to be assembled, it 'waits' at the prearranged rendezvous.
- Likewise as each robot completes its assembly of the car, it too 'waits' at the rendezvous.
- It does not matter which process arrives at the rendezvous first or in what order they arrive, the idea is simply to wait until all 4 have finished.
- Once the 3 robots and the conveyor belt are all waiting at the rendezvous, that is the car has been assembled, then the conveyor belt process can proceed to move the car along to the next phase in its assembly and similarly the robots can begin all over again with a new car. (Of course they may have to wait for a new car to move into position, but that is a separate synchronisation issue requiring an 'Event' as the solution)
- An implementation of a Rendezvous is shown below in the simplified C++ class CRendezvous. A complete, more detailed implementation of this class can be found in the 'rt.h' header file.

A Basic Outline of a Rendezvous Class Implementation

```
class CRendezvous
{
    int          NumThread;           // number of threads participating in rendezvous.
    int          NumberWaiting ;      // number of threads still yet to arrive. Initially set to
                                     // value of NumThread above

    CEvent       RendezvousEvent(...) ; // a Multi-thread release event

public:
    void Wait() {                     // operation must be protected by a mutex
        if(--(NumberWaiting) == 0) {  // if all threads arrived
            NumberWaiting = NumThreads ; // reset the count
            RendezvousEvent.Signal() ;   // release all waiting threads
        }
        else
            RendezvousEvent.Wait() ;    // and wait for the other threads
    }

    CRendezvous(int NumberParticipating) { // Constructor
        NumThread = NumberWaiting = NumberParticipating ;
    }
};
```

- The implementation of a rendezvous is easy to understand. All threads waiting to synchronise themselves perform a wait() operation on the rendezvous object which in turn performs a wait on a multi-thread release **event inside the object**. When the last thread arrives it detects it is the last one to arrive (i.e. NumberWaiting is 0) and thus signals the event releasing both it and the waiting threads. The program below shows how the rendezvous would typically be used.

Example use of a Rendezvous with 5 Thread

```
UINT __stdcall RendezvousThread(void *args)           // A thread to use a rendezvous, args points to an int to identify the thread number
{
    CRendezvous  r1("MyRendezvous", 5);              // a rendezvous object involving 5 threads

    int x = *(int*)(args);                             // get my thread number given to me by my parent (main() )
                                                        // this is given to the thread by the parent thread during the call

    for(int i = 0; i < 10; i++){
        r1.Wait();                                     // wait at the rendezvous point
        printf("%d", x) ;                             // do something after being released, i.e. print my threadnumber
        SLEEP(1000 * x);                               // sleep for an amount of time based on my thread number
    }                                                  // go back at wait again (10 times)
    return 0 ;                                         // terminate thread
}

int main()
{
    int a[] = {0,1,2,3,4};                             // array of thread numbers

    CThread t1(RendezvousThread, ACTIVE, &a[0]);       // create 5 threads and pass them a pointer to a number from the array
    CThread t2(RendezvousThread, ACTIVE, &a[1]);
    CThread t3(RendezvousThread, ACTIVE, &a[2]);
    CThread t4(RendezvousThread, ACTIVE, &a[3]);
    CThread t5(RendezvousThread, ACTIVE, &a[4]);
    ...
    ...
    ...

    return 0 ;
}
```